

Observing, Predicting, and Enforcing Properties of Interactions in Data Centers

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
vorgelegte Dissertation von Patrick Jahnke aus Heidelberg
Tag der Einreichung: April 6, 2021, Tag der Prüfung: May 19, 2021

1. Gutachten: Prof. Max Mühlhäuser
 2. Gutachten: Prof. Patrick Eugster
 3. Gutachten: Prof. Sonia Fahmy
- Darmstadt – D17



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department
Distributed Systems
Programming

Observing, Predicting, and Enforcing Properties of Interactions in Data Centers

Doctoral thesis by Patrick Jahnke

1. Review: Prof. Max Mühlhäuser
2. Review: Prof. Patrick Eugster
3. Review: Prof. Sonia Fahmy

Date of submission: April 6, 2021

Date of thesis defense: May 19, 2021

Darmstadt – D17

I dedicate my dissertation work to my family. The love of my live, Kerstin, who fully supported me during my studies with all the accompanied restrictions. My student days were not only very fruitful in an academic sense; I was blessed with the birth of my first son Paul during writing my Bachelor thesis, during writing my Master thesis with the birth of my second son Hendrick and during my PhD studies with the birth of my third son Leon. My three Musketeers: I want you to know, that you are the center of gravity in my life, you are everything for me, and I love you. A strong gratitude to my parents for all the support and love especially in times with difficult circumstances that we as a family went through. Thanks to my brother Oliver and his family. I know that I can always count on you like you can count on me!

I also want to thank some extraordinary colleagues I had the honor to work with during my professional career. Harald Grünberger for giving me a lot confidence in my own capabilities, Adalbert Scherer and Claude Eisenmann who motivated me during our time together at HamaTech and STP to start my studies.

During my time at TU Darmstadt I also had the opportunity to work with great people. Many thanks to Simon Hofmann for the great cooperation and conversations during the Bachelor and Master programs; to Immanuel Schweitzer who connected me to the DSP group, and to the whole MAKI-Team; I'm very thankful for having been a part of it. Thanks to my colleagues within the DSP group Seema Kumar, Marcel Blöcher, and Malte Viering for the great conversations and discussions. Very special thanks to Pavel Chuprikov and Pierre-Louis Roman from USI and Prof. Gerhard Neumann from KIT for all your efforts, support and help.

Last but not least, I do like to thank my advisor Prof. Patrick Eugster. First and foremost for not giving up on me :). It was and will be always a pleasure working with you. I feel deeply honored that you were my advisor and were willing to spend so much effort and share your profound expertise with me. You fulfilled the role of a *Doktorvater* more than I could have ever expected.

I know that a lot of people are not mentioned here who were part of my journey but I also want to take the opportunity to **thank you** with these words!

Erklärungen laut Promotionsordnung

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, April 6, 2021

P. Jahnke

Zusammenfassung

Durch das Wachstum von Cloud-Infrastruktur-Anbietern in der letzten Dekade und dem damit einhergehenden Bedarf an Internet-Verbindungsbandbreite und Berechnungskapazität, hat sich die Konzipierung von Anwendungen stark verändert. Heutzutage werden die meisten Anwendungen als (Cloud native) verteilte Anwendungen entwickelt, bei denen sich der Kommunikationsaufwand zwischen den einzelnen Instanzen eines verteilten Systems erhöht. Für Cloud-Infrastruktur-Anbieter ist somit eine zielgerichtete und effiziente Ausnutzung der vorhandenen Rechenzentrennetzwerkressourcen für verteilte Anwendungen kritisch. Durch das aktuelle Nichtbeachten der Semantik von Kommunikationsmustern in verteilten Anwendungen, wird eine bestmögliche Ausnutzung schwer bis unmöglich. Durch das Erstellen eines Rechenzentrumsnetzwerks, welches sich der Semantik von verteilten Anwendungen bewusst ist, würde sich die Effizienz der entsprechenden Komponenten dramatisch steigern.

In dieser Doktorarbeit werden neue Ansätze in drei wichtigen Bereichen vorgestellt, die den Status quo verbessern. Es wird gezeigt, dass je präziser Eigenschaften eines Rechenzentrumsnetzwerks beobachtet werden, genaue Vorhersagen zur Bandbreitennutzung von Kommunikationssystemen möglich sind und strikte obere Schranken der Latenz für spezifische Interaktionen in verteilten Systemen bestimmt werden können.

Abstract

In the last decade, the rise of cloud computing combined with a tremendous increase in Internet connectivity and the need for more computational performance changed the way we conceive applications. Today, most software developers and architects design new applications as (cloud-native) distributed systems, where the demand for interactions between individual components of a given distributed system increases. Consequently, a demand-specific and efficient utilization of existing data center communication resources for distributed systems is crucial for cloud infrastructure providers. However, the main challenge in terms of efficiency is that distributed system and data center components are still very agnostic to the actual semantics of interactions. By building communication systems that are more cognizant of distributed system semantics, the performance for the corresponding components would increase tremendously.

In this thesis, three fundamental challenges are tackled to improve the state-of-the-art. We show that the more precisely properties of a communication system can be observed, the better the bandwidth usage can be predicted, and bounded communication latency for specific distributed system interactions can be enforced.

Contents

List of Figures	xiv
List of Tables	xvii
Glossary	xxi
1 INTRODUCTION	1
1.1 FARM	3
1.2 CLAIRE	5
1.3 X-LANE	6
1.4 Declaration of Originality.	7
2 FARM	9
2.1 Introduction	10
2.2 Related Work	13
2.3 M&M Framework	15
2.3.1 Synopsis	15
2.3.2 Switch-local Components	17
2.3.3 Centralized Components	18
2.3.4 Switch-local Execution	18
2.3.5 Seeder-Soil Communication	19
2.4 M&M Seed Programming Model	20
2.4.1 Language Overview	20
2.4.2 Illustration	23
2.5 M&M Seed Examples	25
2.5.1 Hierarchical Heavy Hitter	25
2.5.2 Distributed Denial of Service	28
2.5.3 New TCP Connections	30
2.5.4 TCP SYN Flood	30
2.5.5 TCP Incomplete Flood	32
2.5.6 Slowloris	33

2.5.7	Link Failure Detection	34
2.5.8	Traffic Change Detection	35
2.5.9	Flow Size Distribution	36
2.5.10	Superspreader	37
2.5.11	SSH Brute Force	38
2.5.12	Port Scan	39
2.5.13	DNS Reflection	40
2.5.14	Entropy Estimation	42
2.5.15	FloodDefender	43
2.6	M&M Seed Placement	46
2.6.1	Rationale and Notation	46
2.6.2	Placement Quality	46
2.6.3	Constraints	48
2.6.4	Placement Optimization Algorithm	49
2.7	Implementation	49
2.7.1	FARM Components	49
2.7.2	Placement Optimization	51
2.8	Evaluation	51
2.8.1	Setup	52
2.8.2	Scalability	53
2.8.3	FARM's Accuracy vs CPU Load	55
2.8.4	Global Seed Placement Optimization	56
2.8.5	Implementation Microbenchmarks	58
2.9	Conclusions	59
3	CLAIRE	61
3.1	Introduction	62
3.2	Background	65
3.2.1	Short-Time Fourier Transform	65
3.2.2	K -Means Clustering	66
3.2.3	Hidden Markov Model	67
3.2.4	Kalman Filter	68
3.2.5	Kernel Trick	70
3.2.6	Reproducing Kernels	71
3.2.7	Embedding Distributions in a Reproducing Kernel Hilbert Space	72
3.3	Related Work	75
3.4	CLAIRE Design Overview	77
3.4.1	Technical Challenges	77



3.4.2	Design Overview	77
3.4.3	Preprocessing	79
3.4.4	Adapting the Kalman Filter	80
3.5	Prediction and Learning Implementation	87
3.5.1	Complexity	87
3.5.2	Challenges in Monitoring and Management	89
3.6	Evaluation	90
3.6.1	Synopsis	90
3.6.2	Experimental Data	90
3.6.3	Prediction Results	96
3.7	Conclusions	105
4	X-LANE	107
4.1	Introduction	108
4.2	Related Work	110
4.3	X-LANE Design Overview	112
4.3.1	Communication in the X-LANE	112
4.3.2	X-LANE Controller Overview	113
4.3.3	Overview of Jitter Sources	115
4.3.4	X-LANE (Based) Services	116
4.4	Traffic Engineering for Tunnel Trees	117
4.4.1	Overview	117
4.4.2	Network Model	118
4.4.3	Two-Phase Tunnel Allocation	118
4.4.4	Optimization Problem	119
4.4.5	Traffic Engineering Proofs	122
4.4.6	Resource Monitoring and Tuning	125
4.5	Overcoming Interference in Data Centers	126
4.5.1	Bit Flips Errors in Links	126
4.5.2	Buffer Overflows and Jitter in Switches	127
4.5.3	Jitter in Endhost Commodity Hardware	128
4.5.4	Jitter in Endhost Specialized Hardware	133
4.6	Example Services Exploiting X-LANE	133
4.6.1	Failure Detector X-FD	133
4.6.2	Fast State Machine Replication X-Raft	134
4.7	Evaluation	136
4.7.1	Hardware Setup	136
4.7.2	Timing Observations	136

4.7.3	Towards Bounded Communication	139
4.7.4	Fast Fourier Transform Metric	139
4.7.5	Failure Detector Service X-FD	142
4.7.6	Fast State Machine Replication X-Raft	142
4.8	Conclusions	144
5	CONCLUSIONS	145
6	FUTURE WORK	147

List of Figures

2.1	FARM workflow overview. Monitoring and management tasks described in Almanac, possibly by different users, are sent to the seeder. The seeder translates the descriptions into executable Seeds and deploys them on switches in a network-wide optimized manner. At runtime, Seeds (re)act locally and may provide information to their respective harvester if global coordination is needed for the corresponding task.	11
2.2	FARM's architecture overview. Seeds interact via their soil with their harvester and other Seeds.	16
2.3	Core Almanac syntax. \bar{x} represents several instances of x , $[x]$ means that x is optional. Blue highlighted keywords represent standard state machine constructs; orange highlights Seed-specific primitives.	21
2.6	HH detection time with FARM, sFlow, Sonata, and specialized link utilization monitoring systems Planck and Helios.	52
2.4	Network load of FARM with 1 and 10% HH ratios, the sFlow collector with 1 and 10 ms accuracies, and similarly Sonata.	52
2.5	Switch CPU load of FARM and sFlow for HH detection, 10 ms accuracy. . .	52
2.6	CPU load of FARM for an HH and CLAIRE seeds.	55
2.7	FARM's global Seed placement optimization algorithm (cf. Equation 2.1) is close in quality to Gurobi with 10 min timeout and as fast as Gurobi with 1 s timeout.	56
2.8	The PCIe bus easily congests compared to the ASIC bus, calling for polling aggregation.	57
2.9	Soil's CPU load showing the cost of Seed requests' aggregation when Seeds are threads vs processes.	57
2.10	Shared buffer vs gRPC communication latency between seeds being threads or processes and soil.	57
3.1	Segment of single high-volume TCP flow: sampled with a period of 1 s (top) and zoomed-in fine-grained structure of first peak sampled at 10 ms (below)	63
3.2	Prediction of a given network flow.	64

3.3	CLAIRE overview. Training data and measured data are preprocessed (yellow boxes) by transposition to frequency space and complexity reduction via PCA. Subsequently the (1) training data and (2) measured data are handled differently. (1) is used to optimize the hyperparameters and train the system and the observation models (green or green shaded boxes). (2) is used to make an innovation and prediction update and project back to original state space (blue/blue shaded boxes).	78
3.4	Characteristics of UniDCTraces flows.	91
3.5	Characteristics of SAPDCTraces flows.	91
3.6	Data series in time vs frequency domain.	93
3.7	Sample prediction details across approaches.	97
3.8	Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 1	99
3.9	Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 2	99
3.10	Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 3	99
3.11	Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 4	100
3.12	Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 5	100
3.13	Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 6	100
3.14	Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 7	101
3.15	Polling accuracy and CPU load of monitoring with different numbers of monitoring instances.	102
3.16	Evaluation on TE by leveraging CLAIRE predictions.	104
4.1	Overview with dedicated resources for X-LANE (green) and the regular (blue) lane running on the same actual physical network devices and endhosts.	109
4.2	Separating the traffic of the X-LANE and regular communication on switches to prioritize packets and prevent losses in the former. An SDN controller sets switches' rules to adapt buffer allocation and processing priority. The X-LANE is interfaced to the regular system via the gate.	115
4.3	Core logic of the X-ADJ algorithm for PPTE-ADJ. The adjusted tunnel is $T \sim \blacksquare$ sharing queues with $T_a \sim \boxtimes$, $T_b \sim \blacksquare$, and $T_c \sim \boxtimes$. Height of the queue at (x, y) is proportional to $\text{bw}(x, y)$; hence, packet length is proportional to the transmission delay. Note, order of packets is not important.	120
4.4	Overview of packet reception on commodity hardware.	128
4.5	X-LANE is pinned to a dedicated core on which the sources of preemption (cuboids) are entirely (grey) or partly (green) disabled. The regular system is running on all other cores with all the side effects.	129

4.6	Overview of 10,000 packet latency (in μs) on the three X-LANE variants, QJump and DPDK. Note y -axes greatly vary.	135
4.7	Distribution of X-LANE's packet jitter δ (in ns, log scale for data > 1). A jitter of 0 corresponds to the packet(s) with minimum latency λ_{\min} within a dataset. Boxes are 25 th /75 th percentiles, black bars are medians, whiskers are 1 st /99 th percentiles, further data points are grayed out.	138
4.8	Tail latencies at different percentiles (different numbers of "nines") observed over 21 days.	140
4.9	Accuracy of X-FD on all configurations showing the latency threshold needed to reach 100% accuracy.	140
4.10	Overview of (a) latency (in μs) and (b) amplitude of FFT (log scale) of the three X-LANE variants vs QJump and DPDK. QJump and DPDK' latencies look falsely stable due to their very large timescales (y -axes).	141
4.11	Write latency and throughput of X-Raft, etcd Raft, and Redis stand-alone vs with X-Raft. Mean values are plotted with min-max vertical bars. . . .	143

List of Tables

2.1	Features of generic network M&M solutions.	14
2.2	Control messages from seeder to soil.	20
2.3	16 well-known network monitoring and attack examples implemented in FARM with numbers of lines of code. The numbers include all code, e.g., abstracted functions. Most seed codes are in § 2.5.	23
2.4	Elements and notation of optimization model.	46
2.5	Functions and variables of optimization model.	47
3.1	CLAIRE algorithm parameters and types: <u>d</u> ata, <u>R</u> KHS, <u>p</u> rediction parameters, and <u>h</u> yper-parameters.	85
3.2	Observation parameters and values used.	90
3.3	UniDCTraces prediction experiments. While our CLAIRE achieves -9.17% prediction error on average over all flow clusters, ARIMA and GARCH exhibit at best(!) -77.3% and -95.2% prediction error respectively. NNs perform seemingly a bit better than those but can not predict trajectories.	96
3.4	SAPDCTraces prediction experiments. While CLAIRE achieves 8.86% prediction error on average, ARIMA and GARCH exhibit at best(!) -60.24% and -73.80% respectively; NNs perform even worse.	97
3.5	CPU cycles for one prediction.	103
4.1	Number of lines of code for each XLK component.	134
4.2	Summary of X-LANE's timings showing 0 th , 50 th , 99 th , 100 th latency λ percentiles (in μ s), maximum jitter δ_{\max} (in ns) from λ_{\min} , and a metric based on probability bound (i.i.d. assumption) for $10 \times \lambda_{99\text{th}}$ violation over next 100,000 packets. Replacing our Arista 7280CR-48 by an Arista 7150 could in theory reduce all latencies by 2.6 μ s (cf. § 4.7.1).	137
4.3	Mean FFT amplitude for each configuration.	142

Glossary

Symbols

CLAIRE clustered frequency-based kernel Kalman filter xv, xvi, xix, 5–7, 53, 55, 64, 65, 70, 73, 76–78, 82, 83, 85–90, 94–98, 101–105, 145, 147

FARM framework for network M&M xv, xix, 3–5, 7, 11–20, 23, 46, 49–56, 58, 59, 89, 145, 147

X-LANE express-lane xi, xiii, xvi, xvii, xix, xxv, 6–8, 107–119, 125–145, 147, 148

A

AB aggregation benefits 46–48

ACPI Advanced Configuration and Power Interface 129

Almanac automata language for network management and monitoring code xv, 4, 7, 11, 12, 15, 17, 20–23, 25, 49, 51, 54, 59, 147

ANFIS adaptive neuro-fuzzy inference system 75

API application programming interface 114, 116

ARIMA autoregressive integrated moving average 75, 90, 96, 97, 101

ARMA autoregressive moving average 75

ASIC application-specific integrated circuit xv, 17, 19, 34, 50–52, 57, 58, 89, 127

B

BER bit error rate 126, 127

BLRNN bilinear recurrent neural network 75, 94, 101

C

CNN convolutional neural network 90, 94, 101, 102

CPU central processing unit xvi, 102, 103, 117, 128–131

CRC cyclic redundancy code 126, 127

D

DC data center ix, 1–3, 5–7, 10, 13, 17, 51, 53, 54, 59, 62, 65, 87, 91, 92, 98, 102, 108–111, 115, 126, 136, 139, 145, 147

DDoS distributed denial of service 11, 17, 28

DMA direct memory access 116, 131, 132

DNS domain name system xii, 9, 23, 40, 41

DoS denial of service 1, 10, 13, 17, 28, 43

DPDK data plane development kit xvii, 111, 132, 135–139, 141, 142

DS distributed system ix, 1, 3, 5–7, 108, 110, 138, 144, 145

DSL domain specific language 4, 12, 15

E

EDF earliest deadline first 112, 128

EM expectation-maximization 76

F

FCS frame check sequence 126, 127

FCT flow completion time 2, 3, 5, 6, 62, 145

FD failure detector 8, 110, 111, 133, 136, 142, 144

FFT fast Fourier transform xvii, xix, 6, 65, 66, 78–80, 136, 139–142

FT Fourier transform 94–96

G

GARCH generalized autoregressive conditional heteroskedasticity 75, 90, 96, 97, 101

H

harvester M&M centralized task-specific component xv, 4, 11, 12, 14, 16–18, 22, 24–26, 28, 30, 34, 36, 39, 42, 50, 89, 147

HD Hamming distance 127

HH heavy hitter xv, 1, 10, 12–15, 17, 20, 23–26, 52–56, 59

HHH hierarchical heavy hitter 17, 20, 25, 26

HMM hidden Markov model 64, 67, 68

HTTP hypertext transfer protocol 92, 97, 98, 103, 105

I

IPI inter-processor interrupt 130

IRQ interrupt request 129, 130, 132

K

KF Kalman filter 6, 65, 68–70, 72, 76–79, 81–83
KKF kernel Kalman filter xxiii, xxv, 67, 70, 79, 81, 85, 87, 94, 95
KKF-CEO KKF based on the conditional embedding operator 76

L

LSTM long short-term memory 90, 94, 101, 102

M

M2M machine-to-machine 91, 92, 101, 105
M&M management and monitoring xix, xxii, xxiv, 1, 3, 4, 11–15, 17–20, 22, 46, 48–51, 53, 56, 58, 59
MC migration costs 46–48
MdKP multi-dimensional knapsack problem 49
MILP mixed-integer linear program 14, 49, 51, 56
ML machine learning 53, 56, 58
MLP multilayer perceptron 75, 94, 101
MM Markov model 67
MTTFPA mean time to fault packet acceptance 126, 127
MTU maximum transmission unit 126
MU monitoring utility 46, 47

N

NIC network interface card 111, 116, 117, 127, 128, 131, 132, 134, 136
NMI non-maskable interrupt 130, 132
NN neural network xix, 75, 94, 96, 97, 101
NT network telemetry 14
NUMA non-uniform memory access 131, 132

O

ONL Open Network Linux 50, 52, 89, 102
OS operating system 4, 13, 50, 51, 89, 103, 112, 128, 137

P

PC principal component 80

PCA principal component analysis xvi, 6, 65, 78–80, 94, 95
PCIe peripheral component interconnect express xv, 17, 50, 57, 58, 116, 131, 132
PQ placement quality 46, 47

Q

QPI QuickPath interconnect 131, 132

R


RAM random-access memory 52, 142
RBF radial basis function 71
RCU read, copy, update 130, 131
RFS receive flow steering 132
RKHS reproducing kernel Hilbert space 72, 73, 76–79, 81, 82
RNN recurrent neural network 94, 101
RT real-time 112

S

SDN software-defined networking xvi, 17, 25, 43, 48, 55, 115, 117
seed M&M decentralized instance xv, 4, 11–13, 16–22, 24–42, 45–51, 53–59, 89, 103, 145, 147
seeder M&M centralized control instance xv, xix, 11, 18–20, 22, 50, 51, 59
SLO service-level objective 139
SMI system management interrupt 131
SMM system management mode 131
SMR state machine replication 110, 133, 134, 136, 137, 144
SNMP simple network management protocol 91
soil M&M seed foundation layer xv, xix, 16–20, 50, 51, 57, 58
SSH secure shell protocol xii, 9, 23, 38
STFT short-time Fourier transform 65, 66, 94
SVM support vector machine 70, 75

T

TCAM ternary content-addressable memory 4, 10, 12, 15, 17, 19, 20, 22–24, 28, 38, 39, 58
TCP transmission control protocol xi, xv, 2, 9, 23, 30–33, 40, 63, 75, 76, 92, 148



TE traffic engineering xvi, 1, 3, 6, 7, 10, 62, 65, 76, 104, 105, 110, 113, 118, 127, 145, 147, 148

TO traffic optimization 2, 3, 10, 62, 93, 94, 98, 101

TS-KKF time series KKF 76

U

UDP user datagram protocol 76, 92

X

XLK X-LANE Linux kernel module xix, 113, 116, 133, 134

1 INTRODUCTION

In the last decade, the rise of cloud computing combined with a tremendous increase in Internet connectivity and the need for more computational performance changed the way we conceive applications. Today, most software developers and architects design new applications as (cloud-native) distributed systems (DSs), where the demand for interactions between individual components of a given distributed system (DS) increases. Consequently, the demand for cloud/data center (DC) infrastructure also increased dramatically. Communities like the open compute project [140] and open19 [139] have contributed open-source and impressive solutions to optimize data centers (DCs) from a physical perspective. Softwarization of all components within a DC at all levels of granularity is important to connect them and make them manageable in order to treat the infrastructure efficiently. However, the main challenge in terms of performance is that DS and DC components are still agnostic to the actual semantics of interactions. By building communication systems that are more cognizant of DS semantics, the performance for the corresponding components would increase tremendously. To get more awareness of DS semantics, we investigate crucial properties of interaction in DC communication systems. Therefore, the overall and fundamental idea in this thesis is: *the more precisely properties of a communication system can be observed, the better the bandwidth usage can be predicted, and bounded communication latency for specific DS interactions can be enforced*. In this work, three crucial challenges are tackled to improve the state-of-the-art:

Observing properties of a communication system with the needed accuracy;

Predicting bandwidth usage of interactions;

Enforcing bounded communication latency for certain types of DS's interactions.

Observing properties of interaction. Observing/monitoring a communication system with the required precision is the foundation to exert precise management actions on the communication system. Therefore, management and monitoring (M&M) tasks have a very close connection to each other. Currently, individual M&M tasks are running simultaneously to cope with different types of tasks of the communication system (e.g., heavy hitter (HH), denial of service (DoS), super-spreaders, quality of service violations, traffic engineering

(TE), and traffic optimization (TO)). However, the problem with existing monitoring solutions usually concerned with observing properties of a communication system is that they stop at observation only. Due to the separation between management and monitoring solutions, delays are introduced, which can have massive effects and lead to a situation where management actions are performed (too) late. Another challenge is that a global state is beneficial or even required for some of the mentioned tasks. Therefore, current monitoring approaches are *collection-centric*: collecting as much information (raw samples and simple statistics) as possible through lightweight agents executing on switching devices, forwarding it all directly to a logically centralized collector that computes a global picture of the communication system by filtering and analyzing data sent by all agents (e.g., sFlow [147], IPFIX [42]). While centralizing all information may seem to yield the most complete picture of the network state and is simple from a programming perspective, it is hard to scale to, even today's most efficient industrial DC designs [59]. In addition, information can be outdated by the time it is processed due to the transport latency. Existing approaches exhibit many limitations that affect their semantics, scalability, and responsiveness, dubbed recently the “*resource efficiency and full accuracy dilemma*” [85].

Predicting bandwidth usage of interactions. One typical problem observed in networks that have to deal with large amounts of traffic is *congestion* – when a network device receives more data packets than it can process, individual packets are delayed or even dropped, inevitably lowering applications and services' interactions and, thus, their performance. The reason for congestion is the highly dynamic *bursty* nature of individual interactions combined with their large number transmitted on the same physical link producing high peaks simultaneously. By increasing the physical link bandwidth, nowadays up to 400 Gb/s, the problem is not avoided. Many so-called elephant flows [144] in low bandwidth networks, in fact, result from peaks which are “flattened”; in high bandwidth networks, these retain their original bursty nature. Thus, flow completion times (FCTs) are unlikely to be reduced (linearly) with increased bandwidth in the presence of congestion when relying on reactive congestion control mechanisms (e.g., TCP, DCTCP [4], PCC [51], RCP [55], or XCP [99]), as these incur further communication between sender and receiver and introduce delays in reacting [97]. Hence new approaches are needed for better exploitation of available network bandwidth [2], [97]. Ideally, the network can be managed and monitored in a very fine-grained manner to enable proactive prevention mechanisms. An overutilization would be recognized early enough and, thus, preventive actions as re-route flows can be executed before packet loss or congestion occur [33].

Enforcing bounded communication latency. While predicting network bandwidth would optimize properties like congestion and FCTs, due to proactive re-routing of flows,

one crucial issue that occurs in DSs remains: the longstanding problem of the unpredictability of transmission times for individual packets and, in particular *asynchronous* behavior of commodity networks and hosts [159]. Today, the design of (cloud-native) DSs treat the underlying infrastructure as a generic communication system, striving for an approach that is independent of the network architecture. Consequently, many DSs assume a fully asynchronous system, where packets may be arbitrarily delayed (as well as reordered in transit or even dropped) and unbounded processing times. Some communication patterns can cope with asynchronous behavior without delivery guarantees due to their focus on throughput and thus highly benefit from a proactive TE/TO as discussed before. Particular types of interactions of a DSs benefit strongly from — or even require — synchronous behavior for specific coordination tasks [157], implemented via various primitives such as group membership [32, 101], atomic commit [73], or notably consensus [65]. The use of such systems is very broad. Types of systems using ZooKeeper [86] with Paxos [111] or Raft [138] by default or as option for coordination/fault tolerance include resource management (e.g., Mesos [79], YARN [174]), key-value and wide-column stores (e.g., Accumulo [8], HBase [11], etcd [58], TiKV [171]), data analytics (e.g., Hadoop [9], Spark [186]), or distributed file systems (e.g., HDFS [10]) to only name few. Mitigating interference by minimizing latency and jitter and reducing bounds for practical purposes would be very beneficial for such systems. Thus, there exist two contradictory requirements within (currently) one flow. While the majority of packets within a given flow are still throughput-oriented, a small but crucial part for the coordination of DSs is latency/jitter critical. As discussed above, to increase the throughput-oriented traffic, the available bandwidth in a network needs to be exploited by predicting its interactions. In contrast, to increase the performance of DS coordination tasks, the bounds on latency and jitter for coordination traffic need to be enforced.

In this work, we elaborate on the three challenges discussed above to improve the performance of crucial DS and DC components. We provide an M&M system with unmatched observation accuracy together with an integrated and responsive support for executing actions. On top of such an M&M system, an accurate and fine-grained proactive approach can predict the evolution of flows and their needed bandwidth to prevent congestion/packet loss and improve FCT. To increase the performance of DS coordination tasks, latency and jitter bounds are enforced for particular types of interactions.

1.1 FARM

FARM, a framework for network M&M, is a comprehensive M&M solution that allows to observe various properties of interactions and execute actions immediately. FARM is

selection-centric as opposed to collection-centric. It supports expressive decentralized reasoning through so-called *seeds* deployable directly on a large range of hardware and software network platforms. Seeds accurately poll traffic statistics, probe packets, and perform (re)actions *locally* on these network devices; they execute in a lightweight manner and interact among each other and to a global analyzer *only in specific, well-defined states*, if at all. More precisely, the features and contributions of FARM presented in chapter 2 are:

Decentralized architecture A key idea in the FARM approach is to run tasks and *perform actions where they belong*. FARM is designed for network management beyond simple monitoring. It efficiently use resources available on network devices for accurate and efficient M&M. Seeds are executed on the switch level to get *select* information which is as timely and accurate as possible and to immediately *perform required reactions directly*. A seed can nonetheless communicate with a M&M centralized task-specific component (harvester) to achieve a global perspective and take global decisions if needed but does so much more efficiently since the information is prefiltered locally.

Expressive model FARM uses a domain specific language (DSL) called automata language for network management and monitoring code (Almanac) to describe M&M tasks by leveraging the intuitive abstraction of state machines to be executed as seeds. State machines are an expressive vehicle, already well-known from literature, to capture network policies concisely and precisely in a way cognizant of dynamics and amenable to verification [103]. Almanac makes it easy to succinctly describe M&M tasks as executable entities (seeds) without knowledge of network topology or resources. It is specialized to define communication patterns, resource utilization levels, placement policies, and local (re)actions. Almanac captures a broad spectrum of use-cases where seeds can analyze switch-wide *statistics*, *packet payloads*, but also ternary content-addressable memory (TCAM) rules.

Optimized task co-deployment FARM’s runtime system enables *dynamic deployments and relocations* of seeds across devices without disruptions, which facilitates holistic resource optimization — continuous in time and space — of seed placement for co-existing M&M tasks. To that end, FARM uses a novel, specialized optimization algorithm that considers network device resources, various costs (e.g., seed migration), and *beneficial aggregation factors* from (re)using collected data for multiple M&M tasks deployed side-by-side.

Commodity hardware & software support FARM’s implementation allows M&M tasks to be implemented on a wide variety of platforms. It is built on Stratum [166], an open-source framework that supports *hardware and software platforms of major vendors*. FARM runs on commodity hardware and on two switch operating systems (OSs): Open

Network Linux [141] and Arista EOS [15]. FARM's software components are independent from other frameworks to maximize interoperability.

By fulfilling the presented features, FARM is the underlying approach to predict and enforce interactions in DCs.

1.2 CLAIRE

The *clustered frequency-based kernel Kalman filter*, or CLAIRE for short, is a novel prediction technique to anticipate DS's interaction behavior and needed network bandwidth to reduce congestion and FCT. The intuition behind CLAIRE is the following: based on an observation over a given interval, the prediction consists of multiple points at high resolution (e.g., every 10 ms). After considering the computation time of the prediction, there needs to be enough time after the prediction for preventive measures and actions. Therefore, the focus of CLAIRE, presented in chapter 3, is to show the feasibility of such fine-grained prediction, with the following properties:

Individual flow prediction CLAIRE aims at preventing congestions due to overlapping peaks of different flow trajectories. Therefore, predicting at a level down to individual flows is mandatory to enable adequate and individual adaptations to bursts, especially under congestion.

High resolution CLAIRE performs predictions with a high resolution, i.e., consisting of multiple points within a small period, at a high rate predicting more than a mere total or aggregate, but an actual trajectory. A high resolution is needed to capture individual peaks since peaks are flattened over long intervals with a low number of high peaks and other communication which is comparatively lower.

Sufficient prediction lead time Sufficient lead time in order to have a large window gives CLAIRE the opportunity for taking appropriate corrective actions.

Accurate predictions Predictions with high accuracy (close to the eventual truth) is a hard constraint in order to avoid reacting inadequately [126].

Scalable approach Provide a scalable solution to enable a large number of flow predictions simultaneously. Therefore, CLAIRE's model is optimized to be as small and effective as possible to be scalable but still achieve accurate prediction.

Commodity hardware support CLAIRE's implementation allows prediction on a wide variety of platforms. The focus is on commodity switches without additional specialized hardware.

To fulfill the mentioned properties CLAIRE harnesses the following techniques: the history of a given flow is transformed by a fast Fourier transform (FFT) into frequency space, where flows show a more distinctive pattern, especially by considering particular frequency components. The principal component analysis (PCA) [96] extracts key characteristics (key frequency components). For accurately learned models and to enable CLAIRE to run on commodity DC network switches with limited hardware resources, individual flows are grouped via k -means into clusters with their learned CLAIRE models. The models relying on a kernel-based variant of the Kalman filter, which captures the high-dimensional, non-linear state space. In combination with CLAIRE a simple TE approach can exploit available resources to improve the FCT by reducing congestion and packet loss.

1.3 X-LANE

The research question underlying the present chapter is whether DC components can mitigate interference and enforce communication bounds for specific DS interactions, which benefit from minimizing latency and jitter, thus reducing bounds for practical purposes. Concretely, a fundamental issue for DSs is unbounded latency, where it is unable to distinguish a failed process from a slow one. From a theoretical stance, most useful coordination problems such as group membership [32, 101], atomic commit [73], or notably consensus [65] are unsolvable in environments with unbounded communication delays. Neutralizing interference sources of commodity systems and enforcing communication bounds are beneficial for DS coordination interactions. These interference sources arise as a result of mainly two causes: (a) packet losses and (b) jitter in packet transmission and processing latencies — manifesting in different ways in the three infrastructure element types that are endhosts/servers, switches, and links. Therefore, we introduce in chapter 4 an express-lane — X-LANE for short — that strives first and foremost to minimize jitter, and in the process, also achieves unprecedented and bounded low latency, with the following properties:

Reliable interaction enforcement Providing an exclusive execution environment on end-hosts to reduce latency and jitter, and a traffic engineering algorithm considering residual jitter and queuing delay to perform packet-level latency analysis and reliability in the X-LANE.

Generic system design A system design that is extendable with services that can take advantage of X-LANE’s existence. Furthermore, a simple usage of X-LANE services for existing applications.

Commodity hardware support Current DCs are usually a mix of commodity and newest generations of hardware. Therefore, X-LANE needs to be executable atop commodity hardware & software, as well as on intelligent network devices.

X-LANE takes advantage of dedicated physical resources tuned to become interference-free to enforce bounds on latency and jitter for DS coordination interactions.


In this thesis, three approaches will be presented in the same order as mentioned above. Every chapter starts with an introduction. While in FARM, and X-LANE related work are discussed thereafter, CLAIRE starts with a background section, where particular concepts are introduced that will be discussed in the related work section, which comes right behind the background section. All the chapters then introduce their system design and additional concepts. Based on the system design and additional approach-specific concepts, the implementation details for the individual approaches are discussed, followed by the corresponding evaluation. While an approach specific conclusion complements the individual chapters an overall conclusion together with a future work discussion completes this thesis.

1.4 Declaration of Originality.

All ideas, models, algorithms, and implementation details described are the results of my work under the supervision of Prof. Patrick Eugster. All systems presented in this work (FARM, CLAIRE, and X-LANE) are built from scratch. However, I was fortunate to cooperate with colleagues and students during my studies. During discussions, the corresponding colleagues and students contributed to the models and algorithms, and will hence become co-authors of future publications which are based on individual chapters of this work.

In particular, the switch components of FARM were prototyped and evaluated during David Gengenbach's Masterthesis [68]. The placement optimization heuristic was developed under the lead of Pavel Chuprikov, and the programming language Almanac was refined in cooperation with Pierre-Louis Roman.

The first prototype of the kernel-based Kalman Filter of CLAIRE was discussed with Emmanuel Stapf in [165] in cooperation with Prof. Gerhard Neumann and extended by the k -means clustering approach in Jonas Mieseler's Bachelor Thesis [128]. A comparison with existing neural network approaches was discussed in the Master Thesis of Jonas Mieseler [129]. The TE approach used in CLAIRE was developed in cooperation with Pavel Chuprikov.



The X-LANE failure detector atop commodity hardware & software was prototyped and evaluated in Vincent Riesop's Bachelor Thesis [156]. The traffic engineering algorithm of X-LANE was developed in cooperation with Pavel Chuprikov, and the formalism of the failure detector (FD) and the X-Raft was improved with support from Pierre-Louis Roman.

2 FARM

2.1	Introduction	10
2.2	Related Work	13
2.3	M&M Framework	15
2.3.1	Synopsis	15
2.3.2	Switch-local Components	17
2.3.3	Centralized Components	18
2.3.4	Switch-local Execution	18
2.3.5	Seeder-Soil Communication	19
2.4	M&M Seed Programming Model	20
2.4.1	Language Overview	20
2.4.2	Illustration	23
2.5	M&M Seed Examples	25
2.5.1	Hierarchical Heavy Hitter	25
2.5.2	Distributed Denial of Service	28
2.5.3	New TCP Connections	30
2.5.4	TCP SYN Flood	30
2.5.5	TCP Incomplete Flood	32
2.5.6	Slowloris	33
2.5.7	Link Failure Detection	34
2.5.8	Traffic Change Detection	35
2.5.9	Flow Size Distribution	36
2.5.10	Superspreader	37
2.5.11	SSH Brute Force	38
2.5.12	Port Scan	39
2.5.13	DNS Reflection	40
2.5.14	Entropy Estimation	42
2.5.15	FloodDefender	43

2.6	M&M Seed Placement	46
2.6.1	Rationale and Notation	46
2.6.2	Placement Quality	46
2.6.3	Constraints	48
2.6.4	Placement Optimization Algorithm	49
2.7	Implementation	49
2.7.1	FARM Components	49
2.7.2	Placement Optimization	51
2.8	Evaluation	51
2.8.1	Setup	52
2.8.2	Scalability	53
2.8.3	FARM's Accuracy vs CPU Load	55
2.8.4	Global Seed Placement Optimization	56
2.8.5	Implementation Microbenchmarks	58
2.9	Conclusions	59

2.1 Introduction

To maintain DC networks, administrators need to continuously observe properties of interactions to detect exceptional behavior. Therefore, many tasks are run simultaneously to detect different types of anomalies (e.g., HH, DoS, super-spreaders, quality of service violations, TE, and TO), and corresponding mitigating actions performed. Existing monitoring systems exhibit many limitations that affect their semantics, scalability, and responsiveness, dubbed recently the *resource efficiency and full accuracy dilemma* [85].

Based on rigid and (looking back) constrained and closed switch designs, early monitoring approaches were *collection-centric*: collecting as much information (raw samples and simple statistics) as possible through lightweight agents executing on switches, forwarding it all straightforwardly to a logically centralized collector that computes a global picture of the network state by filtering and analyzing data sent by all agents (e.g., sFlow [147], IPFIX [42]).

More recent monitoring approaches exploit the increasing programmability of network devices to obtain yet more information, in particular from packet payload. However, sending raw packets and statistics from hundreds or thousands of switches to a collector can quickly congest network links and overwhelm the collector, even if implemented in a streaming fashion (e.g., Marple [134], Sonata [74], Newton [194]).

In addition, monitoring tasks in existing systems only pull information from switches but do not support any ensuing *local (re)actions* [98] like adding a TCAM rule or P4 [26]

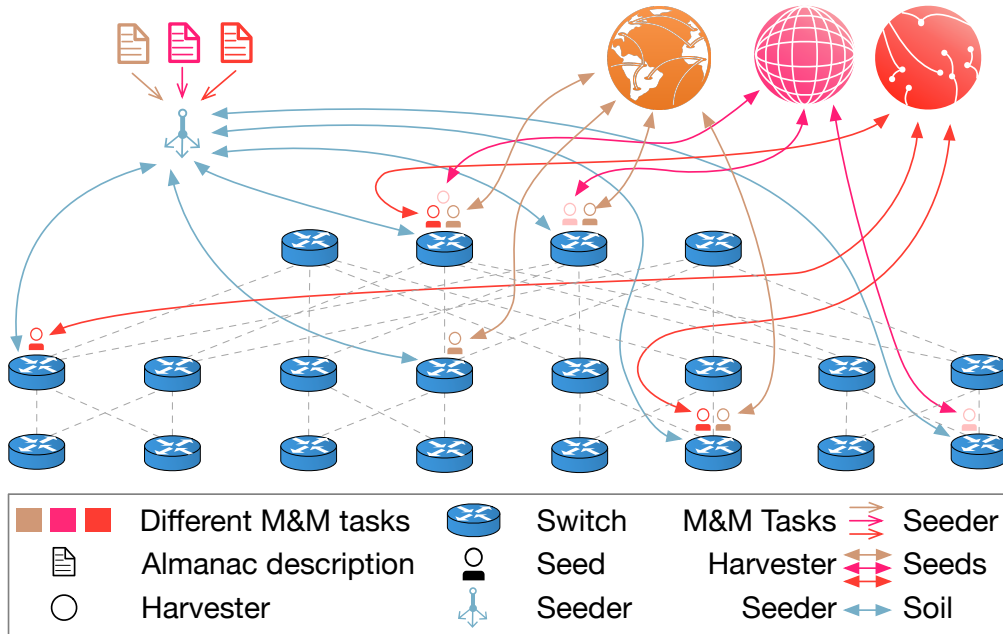


Figure 2.1: FARM workflow overview. Monitoring and management tasks described in Almanac, possibly by different users, are sent to the seeder. The seeder translates the descriptions into executable Seeds and deploys them on switches in a network-wide optimized manner. At run-time, Seeds (re)act locally and may provide information to their respective harvester if global coordination is needed for the corresponding task.

table entries to quench distributed denial of service (DDoS) attacks [130]. Triggering reactions on appliances requires additional mechanisms, incurring a high latency that may be unacceptable when fast reactions are necessary (e.g., DDoS attacks).

As mentioned, many monitoring tasks are needed to enforce correct behavior of the network. Naïvely running several tasks independently side-by-side can lead to transmitting and processing the same data multiple times exacerbating bottlenecks and introducing operational costs. Yet existing solutions provide no opportunities for globally optimizing resource usage across the network *and* concurrently running monitoring tasks.

Finally, many solutions turn to specialized hardware or software platforms [152, 95, 133, 183, 74, 194] to mitigate performance hurdles induced by design limitations mentioned before. These dependencies limit their deployment in most settings.

In this chapter we present a novel management and monitoring (M&M) system called

FARM to address the needs of accurate, efficient, scalable and semantically rich monitoring as well as management. FARM is *selection-* centric as opposed to collection-centric. It supports expressive decentralized reasoning through so-called *M&M seed* — or just *seeds* for short — deployable directly on a large range of hardware and software network platforms. Seeds accurately poll traffic statistics, probe packets, and perform (re)actions *locally* to these network devices; they execute in a lightweight manner and interact among each other and with their *harvester* (i.e., a global analyzer) *only in specific, well-defined states*, if at all. Fig. 2.1 shows the workflow of an M&M task in FARM from its description to its execution. After pinpointing limitations of existing work in § 2.2, we present the features and contributions of our novel solution:

Decentralized architecture [DEC] (§ 2.3): A key idea in the FARM approach is to run tasks and *perform actions where they belong*. FARM is designed for network management beyond simple monitoring. It exploits semantic knowledge to efficiently use resources available on network devices for accurate and efficient monitoring. Seeds are executed on the switch level to get *select* information which is as timely accurate as possible and to immediately *perform required reactions directly*. A seed can nonetheless communicate with a harvester to achieve a global perspective and take global decisions if needed, but does so much more efficiently since the information is prefiltered locally.

Expressive model [EXPR] (§ 2.4): FARM uses a DSL called Almanac¹ to describe M&M tasks by leveraging the intuitive abstraction of state machine to be executed as seeds. State machines are an expressive vehicle, already well-known from literature, to capture network policies concisely and precisely in a way cognizant of dynamics and amenable to verification [103]. Almanac makes it easy to succinctly describe M&M tasks as executable entities (seeds) without knowledge of network topology or resources. It is specialized to define communication patterns, resource utilization levels, placement policies, and local (re)actions. Almanac captures a broad spectrum of use-cases where seeds can analyze switch-wide *statistics*, *packet payloads*, but also TCAM rules. To the matter of presentation and comparability to existing work we use in § 2.4 the HH detection example — identification of flows beyond a threshold size — a very well known example. As it is unable to demonstrate the full potential of FARM, we present a wide variety of M&M tasks using Almanac in § 2.5.

Optimized task co-deployment [OPTIM] (§ 2.6): FARM’s runtime system enables *dynamic deployments and relocations* of seeds across devices without disruptions, which facilitates holistic resource optimization — continuous in time and space — of seed placement

¹An almanac is a calendar with climate data and seasonal advices for farmers.

for co-existing M&M tasks. To that end, FARM uses a novel, specialized optimization algorithm that considers network device resources, various costs (e.g., seed migration), and *beneficial aggregation factors* from (re)using collected data for multiple M&M tasks deployed side-by-side.

Platform-independent implementation [INDEP] (§ 2.7): FARM’s implementation allows M&M tasks to be implemented on a wide variety of platforms. It is built on Stratum [166], an open-source framework that supports *hardware and software platforms of major vendors*. FARM runs on commodity hardware and on two switch OSs: Open Network Linux [141] and Arista EOS [15]. FARM’s software components are independent from other frameworks to maximize interoperability.

§ 2.8 provides an empirical evaluation of FARM in SAP SE production DC using different switch OSs and varying numbers of switches. Our experiments show: (a) FARM experiences significant gains in responsiveness (up to $3427\times$ faster over recent generic approaches and $4\times$ faster over highly specialized solutions) and precision, and savings in network bandwidth consumption (up to $10000\times$) and computational effort over the state-of-the-art; (b) commodity switches can execute dozens of (even CPU-intensive) seeds with FARM; (c) FARM’s global optimizer is scalable, efficient, and fast, capable of optimizing up to 10200 Seeds across 1040 switches. We conclude in § 2.9.

2.2 Related Work

Literature is rich on works on (pure) monitoring (e.g., [42, 182, 197, 193]). This section discusses closest related work on *generic* monitoring systems. From the many specialized solutions introduced for specific monitoring scenarios (e.g., HH detection [164, 124, 61, 152], DoS attack detection [163], or link utilization [60]), we refer to a few later for comparison (e.g., Helios [61] and Planck [152]) or for having influenced the design of FARM. The shortcomings compared to FARM concerning features and requirements introduced in § 2.1 are summarized in Tab. 2.1. Note that we view dynamic deployment (migration) as a prerequisite to global optimization and thus report it as subcategory there, although no prior work allowing such deployment attempts optimization across concurrent monitoring tasks. (Thus FARM is also the only system to aggregate information across monitoring tasks.)

sFlow [147] is a standard technology for monitoring network traffic encompassing: I. *sFlow agents* implementing traffic sampling mechanisms; II. a centralized *sFlow collector* analyzing samples or statistics. *sFlow* uses minimal switch-local processing or triage, performing all analysis on II. This hampers latency as all statistical data has to be transferred

Table 2.1: Features of generic network M&M solutions.

System	Local action	Local reaction	Stats poll.	Payload poll.	Dyn. deploy.	Poll. aggreg.	HW	SW
	[DEC]	[EXPR]	[OPTIM]	[INDEP]				
sFlow [147]	○	○	○	○	●	○	●	○
Sonata [74]	●	○	●	●	●	○	○	○
Newton [194]	●	○	●	●	●	○	○	○
OmniMon [85]	●	◐	●	●	●	○	○	○
BeauCoup [36]	●	○	●	○	◐	○	●	○
Marple [134]	●	○	●	○	◐	○	○	○
PathDump [168]	○	○	●	○	○	○	●	○
FARM (this work)	●	●	●	●	●	●	●	●

there, and limits scalability. Though sFlow is not an IETF standard (cf. RFC 3176) it is widely deployed on many switch types of many vendors; thus we use it in our evaluation (cf. § 2.8).

Sonata [74] emphasizes “stream processing-like” network telemetry [137]. *Sonata* is implemented via P4 [26] in the packet processing pipeline. It mitigates the collector bottleneck by using Spark Streaming [187], but the information processed in the end is not reduced. In contrast, FARM only sends prefiltered information to a harvester, if any at all. Besides not leveraging on-switch resources outside the data plane, *Sonata* does not support merging of streams from several sources (switches), and thus can not be used in many standard scenarios like HH detection.² *Sonata* optimizes placement via a mixed-integer linear program (MILP) using Gurobi [75]; as we shall show this limits scalability.

Newton [194] inherits *Sonata*’s streaming network telemetry [137] system design. Like *Sonata*, *Newton* is implemented via P4 [26] and runs complex operations at Spark Streaming [187]. *Newton* can additionally deploy monitoring tasks dynamically and update queries without rebooting switches. *Newton* can also merge streams from several switches, yet despite some ideas to reduce streaming overhead, the logically centralized processing remains, leading to scalability and responsiveness similar to *Sonata*.

OmniMon [85] tries to solve the collector bottleneck by separating tasks on end hosts and

²Several of the authors propose a separate system for HH [76] where they state that *Sonata* can detect HHs “only on a single switch”; they propose to adapt their work in the future “to detect network-wide heavy hitters [...] for inclusion in such a general network telemetry system.”

switches directly. Nevertheless, a centralized controller has to synchronize all end hosts and switches and share a global state. OmniMon does not optimize resource utilization across monitoring tasks. Furthermore, OmniMon has no generic abstraction and all evaluated tasks are individually designed.

BeauCoup [36] abstracts hardware design similarly to FARM. The authors implement a memory-efficient and performant “coupon” system upon the TCAM over queries similar to Sonata, Newton, and OmniMon. BeauCoup focuses on monitoring tasks that are solvable with a probabilistic distinct-counting, limiting the approach in terms of generality.

Marple [134] pioneered the stream-based monitoring approach, but, unlike other systems, supports data aggregation directly on switches by using local state. However, this support comes with a very limited set of aggregation primitives, which suffice only for basic statistics (e.g., counting out-of-order packets or aggregated packet latency) but not for advanced scenarios like HHs. In addition, Marple relies on a specific key-value store design being implemented in hardware on switches. *PathDump* [168] tracks packet trajectories (stored at edge devices), answering queries expressed in a specialized language. PathDump uses commodity hardware and adds little communication overhead, but cannot reach the breadth and responsiveness we aim for. E.g., response times are around 100 ms [168].

2.3 M&M Framework

We first present the complete high-level architecture of our framework for network M&M (FARM) and its various M&M components, and then detail some of the main aspects of component execution and communication.

2.3.1 Synopsis

FARM builds upon the idea of using switch-local support for execution of monitoring tasks (used in Marple [134] or Sonata [74]) and extends this idea further to switch-local *management* tasks including *reactions*. Unlike pure monitoring, management decisions often cannot be made without any centralized coordination. One of the key features of the FARM design is that both monitoring and management functionalities can be decomposed into switch-local (distributed) components and centralized components. The former can take advantage of their proximity to the data source to monitor and actuate, while the latter (if needed) enjoy a global view of system state. Communication among the two types of components follows a well-defined pattern expressed through a novel DSL called Almanac.

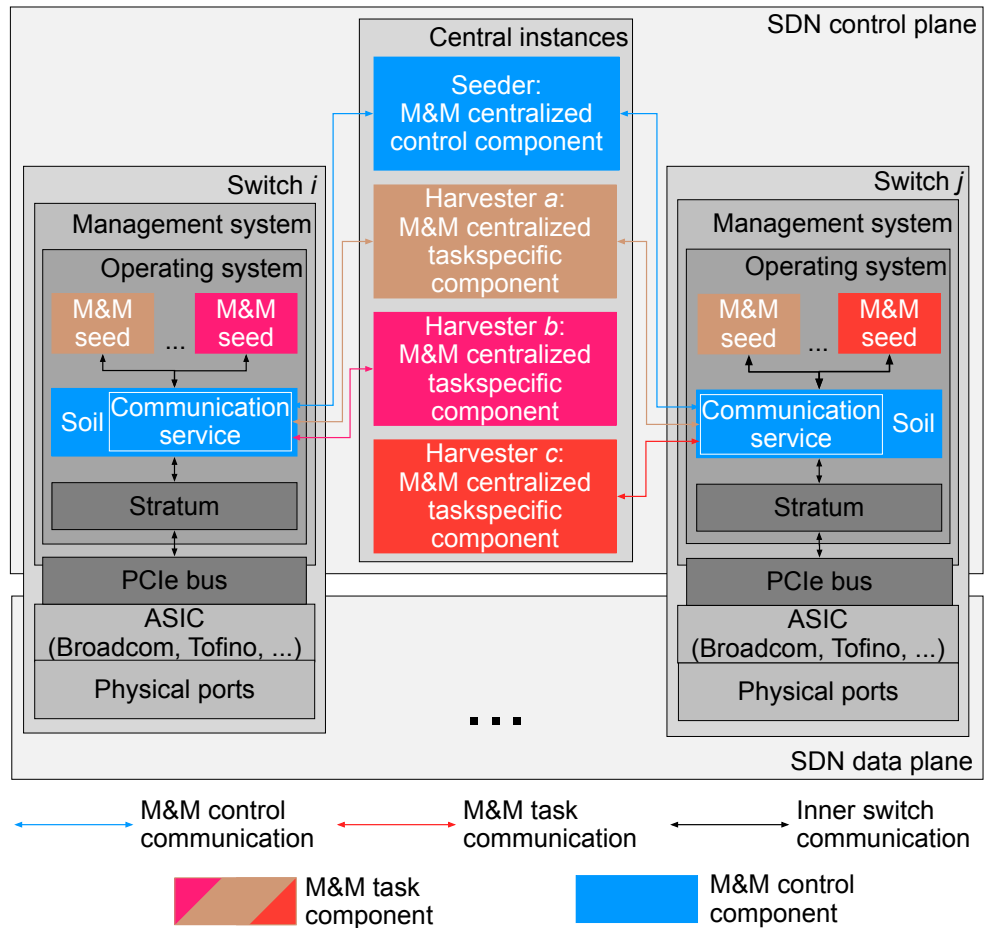


Figure 2.2: FARM's architecture overview. Seeds interact via their soil with their harvester and other Seeds.

Most reactions involve the software-defined networking (SDN) control plane, hence, local reactions entail implementing distributed FARM components *inside the switch-local control plane*. FARM components are designed for generality and seamless deployment across a variety of underlying hardware upon a well-defined abstraction [166] of switch vendor specifics. For superior performance, switch-local components take advantage of hardware resources of the switch as available, providing the best accuracy of monitoring information, with lowest possible delay, as it is crucial for a variety of time-critical measurement tasks (e.g. HH, hierarchical heavy hitter (HHH), DoS, DDoS, and super-spreaders, quality of service violations). Placement of FARM's decentralized components is globally optimized through a unique heuristic.

The general FARM design distinguishes between two types of components, as shown in Fig. 2.2: M&M task components that execute the core logic of M&M applications, and M&M control components that manage the deployment and execution of task components.

2.3.2 Switch-local Components

State-of-the-art DC switching devices have two main processing domains: (i) a management system with a common CPU, and (ii) an application-specific integrated circuit (ASIC) for packet processing. While the ASIC is optimized for fast packet forwarding, the management system is responsible for communication with control devices, e.g., an SDN controller, and, following either local or global management decisions, updating the forwarding rules of the ASIC (i.e., reacting). FARM components run at the management system, but they continuously poll packet processing statistics, including from P4 programs, or sample packets from the ASIC. To optimize the usage of shared switch resources, including polling bandwidth, by multiple M&M tasks, we introduce two types of switch-local components, one of which represents an execution unit, and the other represents a “hypervisor”.

Seed: The M&M seeds — seeds for short — of an M&M task collect, filter, and analyze monitoring data with the goal of performing local management (re)actions (e.g., updating TCAM rules or deploy new P4 table entries) as soon as possible and without requiring remote intervention. Seeds may interact with other seeds and with their harvester (a centralized M&M task-specific coordinator, cf. Fig. 2.1 and § 2.3.3) if and when needed for the M&M task. Seeds, written in Almanac (cf. § 2.4), have their own inner states and run as lightweight instances (processes or threads) in the switch control plane (cf. § 2.7.1).

Soil: The M&M seed foundation layer (soil) manages the execution of the seeds, tracks switch resource usage, and optimizes/aggregates communication with the ASIC over a peripheral component interconnect express (PCIe) bus serving as abstraction layer

between seeds and the switch. The M&M seed foundation layer (soil) employs its own communication service to establish and optimize its communication with local seeds components executed on other devices (i.e., centralized components or other switches' soil).

We detail the seed programming abstraction in § 2.4, but we would like to highlight here that this abstraction is much more generic than query operations used in several prior approaches (e.g., Sonata [74], Marple [134]). This allows FARM to support altering local behavior, e.g., *reacting* to some stimuli, quickly, without the need to involve a centralized entity for decision making and/or seed redeployment.

2.3.3 Centralized Components

Switch-local components may still require centralized components to (i) partake in M&M tasks by taking centralized decisions — when needed — based on data received from distributed seeds, and (ii) coordinate the placement and maximize joint utility of deployed seeds. FARM thus uses:

Harvester: Each M&M task possesses its own harvester, that collects (or *harvests*) events sent by the seeds of the same task and takes global management actions for this task when seed-local decision-making is insufficient.

Seeder: The M&M centralized control instance, called M&M centralized control instance (seeder), optimizes the resource utilization of all M&M tasks co-deployed over the network. It can dynamically (un-)install and (re-)position the seeds following a global placement optimization algorithm (see § 2.6). The seeder also establishes the communication interface for seeds to communicate with each other either over the seeder or directly by requesting the network location of a seed.

2.3.4 Switch-local Execution

Seed.

A seed definition includes an abstract description of where the corresponding seed instance(s) will execute in the network. To make efficient use of limited computational resources at the switch, seeds are made *reactive* in nature, i.e., they perform actions only in response to specific events. Correspondence between events and actions is captured also in the seed definition and that forms the core of an actual M&M algorithm. As the seed behavior itself may need to be changed as part of local reaction, we introduce explicit states to the seed definition, where every state may listen to events and perform actions of its own choosing and have its own polling period (see § 2.4).

After a seed collects monitoring data (e.g., sampled packets, statistics), it analyzes the data, checks whether its inner state has to change, and, whether actions have to be taken. Because of the high dynamics of FARM, a seed can change its polling rate dynamically to reduce the switch resources required, which is also important to optimize M&M tasks globally between different M&M tasks (see § 2.6).

Soil.

The soil manages the execution of seeds and their communication to components executed on other devices, and the local ASIC. In particular, the soil oversees seeds' usage of the switch resources. These resources include, besides CPU and memory, two ASIC-specific types: a) *bus bandwidth* for packet probing and statistics polling and b) *TCAM space* for tracking specific flows and/or implementing various forms of local reaction. The soil synchronizes and aggregates data polling made by different seeds to minimize communication to the ASIC and avoid contention (cf. § 2.8.5).

The main polling pattern the soil can leverage for polling aggregation is when multiple seeds that execute different M&M tasks poll the *same data* from the switch. In such case, it is usually possible to poll the data only once for all these seeds. Such opportunities are statically analyzed and leveraged for aggregation benefits (see § 2.6) by the seeder.

In addition, the soil carefully divides the ASICs' TCAM between (1) monitoring and (2) packet forwarding such that the routing/switching behavior is not affected when rearranging the TCAM memory due to FARM operation. This approach draws inspiration from iSTAMP's [124] TCAM division, used for fine-grained monitoring, and extends it with an accurate polling mechanism between TCAM and seeds.

2.3.5 Seeder-Soil Communication

Since the centralized seeder (un)installs and (re)positions the seeds, it interacts with the communication service of the soil, deployed on all switching devices (see Fig. 2.2). Tab. 2.2 presents the main messages used in the process.

New seeds are added to a switch with `ADD_SEED`. The seeder generates and manages unique IDs for every seed instance. The unique IDs are important for establishing communication among FARM's components (e.g., between different seeds of the same M&M task). An existing seed can be changed with `MODIFY_SEED`. `MON_START` and `MON_STOP` messages are used to start and stop seeds respectively. The current state of a seed is set or queried by `MON_STATUS`. The state is stored at the seed, allowing FARM to migrate seeds between different switches.

For placement decisions the seeder needs resource usage information for subordinate switches. It may derive CPU and memory data by leveraging global view on the services

Table 2.2: Control messages from seeder to soil.

Message	Description
CNFG_TCAM_TIMER	Set TCAM utilization check period
SET_TCAM_THRES	Set threshold on TCAM memory
TCAM_THRES_EXCEED	TCAM threshold set exceeded
GET_TCAM_STATUS	Fetch TCAM status
<ADD MODIFY>_SEED	Add new seed/modify seed definition
MON_PARAM_CHNG	Change seed polling period
MON_<START STOP>	Start seed/stop seed
MON_STATUS	Get or set status (incl. state) of a seed

deployed in the control plane. In contrast, availability of TCAM space depends on the dynamic behavior of network protocols; hence, such information must come directly from network devices. The seeder receives the current status of the TCAM with `GET_TCAM_STATUS` and sets thresholds on it with `SET_TCAM_THRES` to prevent FARM from occupying TCAM space needed for packet forwarding rules. The thresholds are observed by the soil running on the switch. The period for checking TCAM thresholds is defined with `CNFG_TCAM_TIMER`. The soil then checks the available resources of the switch. If the management system has no resources left to execute the seeds or the TCAM threshold is exceeded, the seeder (1) receives a `TCAM_THRES_EXCEED` message from the soil, and (2) starts a new optimization run to achieve better global M&M resource utilization (see § 2.6).

2.4 M&M Seed Programming Model

This section introduces our automata language for network management and monitoring code (Almanac) and illustrates it through the examples of HH and HHH detection.

2.4.1 Language Overview

Almanac is centered around the concept of seeds, which are patterned after the well-known state machine abstraction. Almanac draws inspiration from a variety of more generic languages and models (e.g., Esterel [25], IO-Automata [66]) based on state machines due to programmers' familiarity with that abstraction in the space of networking (cf. [103]), adding features and actions specific to M&M (e.g., TCAM modification). Fig. 2.3 presents a subset of the syntax used to express state machines in Almanac. In the following \bar{x} represents several instances of x , and $[x]$ means that x is optional. Blue highlighted

```

machine    ::= machine mname [extends mname] mdef
mdef       ::= {vardec; state}
vardec     ::= ttype tvar [= expr] | type var [= expr]
ttype      ::= time | probe
type       ::= bool | int | long | list | packet | action | ...
state      ::= state sname sdef
sdef       ::= {place placement; vardec; event}
placement  ::= switchID | range | all
range      ::= [sender | receiver] [pollSubject]
              range compOp val
pollSubject ::= TCAM prefixPatt | port val | ...
event      ::= when (trigger) do {action;}
trigger     ::= reception | tvar | enter | exit
reception  ::= recv msgPatt from mname[@dest]
cond       ::= expr compOp expr | cond boolOp cond | not cond
compOp     ::= < | > | =< | >= | == | <>
boolOp     ::= and | or
expr       ::= expr binOp expr | val | tvar | var | get prefixPatt | ...
binOp      ::= + | - | * | /
action     ::= tvar = expr | var = expr | transit sname
              | if (cond) then {action;} [else {action;}]
              | while (cond) {action;} | TCAMact
              | send val to mname[@dest] | exec val
TCAMact    ::= add prefixPatt rule | delete prefixPatt

```

Figure 2.3: Core Almanac syntax. \bar{x} represents several instances of x , $[x]$ means that x is optional. Blue highlighted keywords represent standard state machine constructs; orange highlights Seed-specific primitives.

keywords represent standard state machine constructs; orange highlights seed-specific primitives.

Machines. A state machine `machine` has a name *mname*. This name describes the generic type of a seed and not a single seed instance. A state machine further includes a set of declarations *vardec* of variables *var*, and a set of declarations of explicit states *state* for the machine. The former represents the local variables of a machine, and are reminiscent of fields in object-oriented languages. A machine optionally `extends` another machine. Almanac currently implements a simple form of single inheritance, where *states* can be overridden in child machines; variables can not be overridden or shadowed, though. (More advanced mechanisms, e.g. [39], are under investigation.) Note that *trigger variables* *tvar* are special kinds of variables used for triggering events, either periodically in time, or every given number of packets. These variables are assigned two respective types *ttype* — `time` and `probe`.

States. Machines must declare their different possible discrete `states`, each having a name *sname* and a definition *sdef* including local variables *vardec*, placement constraints *placement*, and a set of events *event* that can affect the machine in the given state. Placement constraints consist in a set of switch IDs, or a *range* on the number of hops that the machine must be away from the network edge. Such constraints can be of any nature (*compOp*), e.g., upper or lower bounds, exact number, and can be defined with respect to senders (`sender`), receivers (`receiver`) or both (default), with respect to specific TCAM rules (`TCAM`) or to a port (`port`). Absence of placement constraints is denoted by `all`.

Note that as syntactic sugar (not shown in the abstract syntax for brevity), a placement policy can also be described at the level of a machine, which means that it applies to every single state. The same goes for the events described next. (Such global definitions are also subject to overriding.)

Events. Asynchronous events are used to affect the state of the machine. Each event `event` is defined by a *trigger* for executing the event and a set of actions *action* performed in response. The trigger can be entering or exiting the state (`enter`, `exit`), the reception (`recv`) of a message from another machine (instance) or the harvester, or a trigger variable reaching its triggering condition (e.g., a time-lapse). Receptions include pattern matching on messages and can constrain the source of a message to a given machine *mname* at a given *dest*, which can be a seed, a group of seeds (e.g., grouped by M&M task), other switches, the seeder, or a harvester. We omit the details of pattern matching for brevity. A simple and common pattern is a formal argument; if the received message has the

Table 2.3: 16 well-known network monitoring and attack examples implemented in FARM with numbers of lines of code. The numbers include all code, e.g., abstracted functions. Most seed codes are in §2.5.

Use case	Seed Harv.		Use case	Seed Harv.	
Heavy hitter (HH)	29	12	Link failure [197]	31	8
Hier. HH [192]	21	26	Traffic change [160]	7	5
(inherited)			Flow size distr. [53]	30	15
Hier. HH [192]	38	26	Superspreader [183]	58	21
DDoS [130]	71	30	SSH brute force [93]	34	9
New TCP conn. [185]	19	5	Port scan [89]	44	23
TCP SYN flood [185]	63	18	DNS reflection [109]	83	22
Partial TCP flow [185]	73	18	Entropy estim. [131]	67	15
Slowloris [161]	44	29	FloodDefender [163]	126	35

same type as the argument, the corresponding value(s) will be assigned implicitly. Basic conditions include constraints on expressions *expr*, including variables *var* of the machine or the state of the TCAM (*get*).

Actions. The body of an event handler includes a sequence of actions; these are: assignments of expressions to trigger variables (e.g., to modify polling rates) or regular variables, explicit transitions (*transit*) to states, common control structures (*if*, *while*), modifications *TCAMact* of the routing information base in the TCAM (rule addition *add*, removal *delete*; *get* returns a rule for a given prefix), and sending of messages (*send*) to another machine *mname* at a given host *dest* or broadcast to all hosts (no *dest*).

Logic without state machine-related operations can also be modularized into common auxiliary functions (omitted in the syntax for brevity), e.g. to operate on lists, filter TCAM rules, or match regular expressions. Finally, *exec* allows external code to be executed.

2.4.2 Illustration

Tab. 2.3 gives an overview of scenarios implemented with Almanac with numbers of lines of code. We detail the example of heavy hitter (HH) detection (see List. 2.1) to illustrate Almanac in this section all other examples from Tab. 2.3 can be found at §2.5. HHs are flows whose size is larger than a given threshold. The example has two states *observe* and *HHdetected*. In the *observe* state, none of the observed ports is identified as an HH; as soon as the number of transmitted bytes of a port reach the defined *threshold*, a state

```

1 machine HH {
2   place all;
3   time pollingStats = 10;
4   long threshold;
5   action hitterAction;
6   list hitters;
7   state observe {
8     when (pollingStats) do {
9       hitters = getHH(threshold);
10      if (not hitters.empty()) then {
11        transit HHdetected;
12      }
13    }
14  }
15  state HHdetected {
16    when (enter) do {
17      send hitters to Hit@Harvester;
18      setHitterRules(hitters, hitterAction);
19      transit observe;
20    }
21  }
22  when (recv long newTh from Hit@Harvester)
23  do { threshold = newTh; }
24  when (recv action hitAct from Hit@Harvester)
25  do { hitterAction = hitAct; }
26 }

```

List. 2.1: Heavy hitter (HH) seed example.

transition to `HHdetected` occurs. In the `HHdetected` state, the current port list will be sent to the centralized HH harvester instance (`Hit@Harvester`). With this information, the centralized HH harvester instance can react to the HH in the network. In addition, local reaction will be performed that installs TCAM rules through auxiliary functions (abstracted inside the `setHitterRules` procedure) for the detected flows altering QoS policy for respective packets.

The two events for receiving messages and the placement policy are defined outside of the states; as mentioned, this syntactic sugar denotes that they apply to all states. In this example, the harvester sets up the threshold for an HH and can dynamically change the threshold related to the overall traffic load in the network. If the network policy changes, the harvester can also modify the action that seeds apply locally to detected HHs. Auxiliary function `getHH` uses common programming constructs for determining which flows are HHs and is abstracted (thus *italicized*) for brevity.

2.5 M&M Seed Examples

We illustrate seed programming in Almanac with further well-known monitoring examples listed in Tab. 2.3.

Domain specific code is not discussed in the following examples but marked as *italic* (as mentioned it is obviously included in the number of lines of code reported in Tab. 2.3). The communication schema with the corresponding harvester is part of the examples. The harvesters are running on an SDN controller implementation (e.g., Ryu) and are not part of the following examples (cf. § 2.7). Many of the following examples are in the area of attack detection and prevention; for simplicity we omit “attack” from their names.

2.5.1 Hierarchical Heavy Hitter

hierarchical heavy hitters (HHHs) offer finer-grained HH source detection in a hierarchical topology — the traffic of a leaf switch (which was detected as a HH) is not taken into account when calculating the traffic of the spine switch, thus avoiding wrongly tagging a spine switch as an HH. Hierarchical heavy hitters (HHHs) are natural candidates for demonstrating inheritance (from HH). The `HHH` machine (List. 2.2) shows an example of HHH seed implementation that overrides the `observe` state from the `HH` machine (List. 2.1). Inheritance, in this case, reduces the number of LoC from 38 to 27. In short, the HHH harvester determines global HHs and periodically sends these to seeds, that incorporate those (`hitters`) when locally determining HHs. List. 2.4 shows a possible HHH detection implementation without inheritance (compared to List. 2.2). It has two states `observe` and `HHHdetected`. In the former state no HHH is detected among the observed ports; as soon as a HHH is detected a state transition to the latter state occurs. In that latter state, first, the current port list is sent to the centralized HHH monitoring instance (`HHH@Harvester`). With this information the centralized HHH monitoring instance tells the affected seeds to update their lists of `subHHHs`. The list of detected HHH is updated in `HHHdetected`; if it changes a new message is sent to centralized HHH instance. If there is no HHH detected anymore, the centralized HHH instance is informed and the state transits to `observe`. `place all` specifies that the definition has to be installed on all switches. In this example the harvester (List. 2.3) is responsible for setting up the threshold for a HHH and distributing the list of detected HHHs; the list is important because HHs of child switches will not be considered by their parent switches for their HHH calculation.

```

1 machine HHH extends HH {
2     list newHitters;
3     state observe {
4         when (pollingStats) do {
5             newHitters = getHHH(hitters, threshold);
6             if (newHitters <> hitters) then {
7                 hitters = newHitters;
8                 transit HHdetected;
9             }
10        }
11    }
12    when (recv list newHs from Hit@Harvester) do
13        { hitters = newHs; }
14 }

```

List. 2.2: Hierarchical HH (HHH) seed inherited from the HH seed implementation in List. 2.1.

```

1 import ...
2 import HarvesterLib;
3
4 ...
5 def receiveMsg(sender, earg):
6     HHHList[sender.ID] = earg.newHHHList
7     harvester.send("HHHList",
8         getParents(sender.ID),
9         HHHList)
10
11 def calcThreshold():
12     threshold = getNetworkThreshold()
13     harvester.send("threshold", threshold)
14
15 HHHList = [[], []]
16 harvester = HarvesterLib.Harvester("Hit")
17 harvester.receive += receiveMsg
18 threading.Timer(getThresholdPeriod(),
19     calcThreshold)

```

List. 2.3: Excerpt of the Hitter harvester in Python handling communication with HHH seeds.

```

1 machine HHH {
2     place all;
3     time pollingStats = 10;
4     long threshold;
5     list subHHH;
6     list HHHList;
7     list newHHHList;
8     state observe {
9         when (pollingStats) do {
10             newHHHList = getHHH(subHHH, threshold);
11             if (newHHHList <> HHHList) then {
12                 HHHList = newHHHList;
13                 transit HHHdetected;
14             }
15         }
16     }
17     state HHHdetected {
18         when (enter) do {
19             send HHHList to Hit@Harvester;
20             transit observe;
21         }
22     }
23     when (recv long newTh
24         from Hit@Harvester) do {
25         threshold = newTh;
26         transit observe;
27     }
28     when (recv list nuHitrs
29         from Hit@Harvester) do
30         { subHHH = nuHitrs; }
31 }

```

List. 2.4: HHH seed example, expanded vs List. 2.1 + List. 2.2.

2.5.2 Distributed Denial of Service

In a DDoS scenario [130] an attack is not initiated by one attacker, as in a DoS attack; many compromised nodes start contributing to an attack to a target. Coordination is thus crucial to prevent a DDoS attack; often it can just be detected by the number of connection requests from different nodes. In such a (worst) case the network elements closer to the target are able to identify the attack and have to immediately forward this information to others to enact countermeasures.

A DDoS detection system, as presented in the seed example in List. 2.5, usually has four states: (1) In the `normal` state no DDoS attack (anomaly) is suspected. (2) In the second `anomaly` state a switch has identified an anomaly but cannot be sure whether it is just a short term peak. If such an anomaly is observed for a given time duration, a transition to `suspect` occurs. (3) In that `suspect` state all participants in the DDoS attack prevention have to be informed. (4) If a defined number of local instances corroborate the observation then the harvester informs all seeds to change their state to `attack`. In this example the statistics polling accuracy is increased at the `anomaly` state. In the `suspect` state, flow rules against a DDoS attack with limits on individual flows are installed. The `attack` state can only be reached if a message from the DDoS harvester is received. If the attack ends only the DDoS harvester can transit states back to `normal` by broadcasting an `UndoAttack` message. In this example a certain IP range `IPRangeX` is observed. At any time the placement policy and the TCAM rules which will be installed in a certain state (`attack`, `suspect`) can be changed during the execution of the monitoring algorithm.

```
1 machine DDoS {
2   TCAM IPRangeX range -1;
3   place IPRangeX;
4   time pollNormal = 10;
5   time pollAnomaly = 1;
6   int obsTime;
7   int obsTimer;
8   int oldState;
9   list tcamRuleAttack;
10  list tcamRuleSuspect;
11  string placementPolicy;
12  bool suspectSent = false;
13  state normal {
14    when (pollNormal) do {
15      if (not isAnomaly()) then {
16        send "pollNormal"
17          to DDoS@Harvester;
18        obsTime = 0;
19        transit anomaly;
```

```

20     } } }
21     state anomaly {
22         when (pollAnomaly) do {
23             if (isNormal()) then
24                 { transit normal; }
25             if (obsTimer >= obsTime) then
26                 { transit suspect; }
27             obsTime = obsTime + 1;
28         } }
29     state suspect {
30         when (enter) do {
31             suspectSent = false;
32             setSuspectedTCAMRule(ruleSuspect);
33         }
34         when (pollAnomaly) do {
35             if (isNormal()) then
36                 { transit normal; }
37             if (not suspectSent) then {
38                 suspectSent = true;
39                 send "suspect" to DDoS@Harvester;
40             } } }
41     state attack {
42         when (enter) do {
43             setAttackTCAMRule(ruleAttack);
44             send "attack" to DDoS@Harvester;
45         } }
46     when (recv list newTCAMRuleAttack from DDoS@Harvester) do
47         { ruleAttack = newTCAMRuleAttack; }
48     when (recv list newTCAMRuleSuspect from DDoS@Harvester) do
49         { ruleSuspect = newTCAMRuleSuspect; }
50     when (recv list newPlacement from DDoS@Harvester) do
51         { setPlacementPolicy(newPlacement); }
52     when (recv int newObsTimer from DDoS@Harvester) do
53         { obsTimer = newObsTimer; }
54     when (recv "attackMessage" from DDoS@Harvester) do {
55         oldState = currentState;
56         transit attack;
57     }
58     when (recv "undoAttack" from DDoS@Harvester) do
59         { transit oldState; }
60 }

```

List. 2.5: DDoS seed example.

2.5.3 New TCP Connections

The code in List. 2.6 checks and saves the number of new TCP connections (using the SYN flag [185]) created since the last `resetConn` message received from the responsible harvester. If the harvester sends a request, the seed replies back with the number of stored connections.

```
1 machine NewTCPConnections {
2   place range -1;
3   time pollingStats = 10;
4   long conns;
5   state newConn {
6     when (pollingStats)
7       do { conns = conns + getNewConn(); }
8   }
9   when recv request
10    from NewTCPConns@Harvester do
11    { send conns to NewTCPConns@Harvester; }
12  when recv resetConn
13    from NewTCPConns@Harvester do
14    { conns = 0; }
15 }
```

List. 2.6: New TCP connections seed example.

2.5.4 TCP SYN Flood

Compared to new TCP connection detection (cf. List. 2.6), a TCP SYN flood attack can be detected by counting the number of incomplete TCP handshakes in a time interval [185]. The corresponding seed implementation in List. 2.7 contains three states. The `normal` state considers an incomplete TCP handshake to be a packet trace consisting of a SYN packet and a SYNACK packet, with corresponding sequence number and acknowledge number, but without a subsequent ACK packet to complete the handshake. At the `normal` state first global flows are checked, and if a certain threshold is reached, the global flow are split into individual flows and checked. If the individual flows are sending a certain amount of SYN packets a transition to the `suspect` state is executed. In the `suspect` state the bandwidth of the suspected connections gets limited while in the `TCP SYN Flood` state the connections is interrupted.

```

1 machine TCPSYNFlood {
2   place all;
3   time pollingStats = 10;
4   time pollAnomaly = 10;
5   list conns;
6   int switchThresh;
7   int indivThresh;
8   state normal {
9     when (pollingStats) do {
10      conns = getIncompleteHandshakes();
11      if (checkIndivHandshakes(conns,
12        indivThresh))
13      then { transit suspect; }
14      if (checkGlobalConn(conns, switchThresh))
15      then { setIndividualTCAMRules(conns); }
16    }
17  }
18  state suspect {
19    when (enter) do
20      { setSuspectedTCAMRule(); }
21    when (pollingStats) do {
22      if (isNormal()) then { transit normal; }
23      send "suspect" to TCPSYNFlood@Harvester;
24      transit TCPSYNFlood;
25    }
26  }
27  state TCPSYNFlood {
28    when (enter) do {
29      setAttackTCAMRule();
30      send "attack" to TCPSYNFlood@Harvester;
31    }
32  }
33  when (recv int newSwitchThresh from TCPSYNFlood@Harvester) do
34    { switchThresh = newSwitchThresh; }
35  when (recv int newIndivThresh from TCPSYNFlood@Harvester) do
36    { indivThresh = newIndivThresh; }
37  when (recv list undoAttack from TCPSYNFlood@Harvester) do {
38    setTCAMRule(undoAttack);
39    transit normal;
40  }
41 }

```

List. 2.7: TCP SYN flood seed example.

2.5.5 TCP Incomplete Flood

A TCP incomplete flood attack [185] (cf. List. 2.8) is quite similar to a TCP SYN flood attack (cf. List. 2.7), but instead of suffering from an incomplete TCP handshake, the TCP flow as a whole is never completed. As with the TCP SYN flood attack, the TCP incomplete flood attack relies on finer details compared to the new TCP connections (cf. List. 2.6) example.

```
1 machine TCPIncompleteFlood {
2   place all;
3   time pollingStats = 10;
4   time pollAnomaly = 10;
5   list conns;
6   int switchThresh;
7   int indivThresh;
8   state normal {
9     when (enter) do { setTCAMRule(); }
10    when (pollingStats) do {
11      conns = getIncompleteFlows();
12      if (checkIndivFlows(conns, indivThresh) then
13        { transit suspect; }
14      if (checkGlobalFlows(conns, switchThresh)) then
15        { setIndividualTCAMRules(conns); }
16    } }
17   state suspect {
18     when (enter) do { setSuspectedTCAMRule(); }
19     when (pollingStats) do {
20       if (isNormal()) then { transit normal; }
21       send "suspect" to TCPIncomp@Harvester;
22       transit TCPIncomplete;
23     } }
24   state TCPIncomplete {
25     when (enter) do {
26       setAttackTCAMRule();
27       send "attack" to TCPIncomp@Harvester;
28     } }
29   when (recv int newSwitchThresh from TCPIncomp@Harvester) do
30     { switchThresh = newSwitchThresh; }
31   when (recv int newIndivThresh from TCPIncomp@Harvester) do
32     { indivThresh = newIndivThresh; }
33   when (recv list undoAttack from TCPIncomp@Harvester) do {
34     setTCAMRule(undoAttack);
35     transit normal;
36   } }
```

List. 2.8: TCP incomplete flood seed example.

2.5.6 Slowloris

A Slowloris attack [161], as implemented in List. 2.9, is related to the TCP incomplete flow attack. But in contrast to the TCP incomplete flow attack, a Slowloris attacker sends partial headers at a very slow rate (less than the idle connection timeout value on the server), yet never completes the request. The headers are periodically sent to keep sockets from closing, thereby keeping the server resources occupied. Due to the similarity to the TCP incomplete flow attack the description of the seed looks similar and has the same states.

```
1 machine Slowloris {
2   place all;
3   time pollingStats = 10;
4   time pollAnomaly = 10;
5   list conns;
6   int threshold;
7   state normal {
8     when (enter) do { setTCAMRule(); }
9     when (pollingStats) do {
10       conns = getIncompleteFlows();
11       if (checkSlowloris(conns, threshold))
12         then { setIndividualTCAMRules(conns); }
13     } }
14   state suspect {
15     when (enter) do { setSuspectedTCAMRule(); }
16     when (pollingStats) do {
17       if (isNormal()) then { transit normal; }
18       send "suspect" to Slowloris@Harvester;
19       transit SlowlorisAttack;
20     } }
21   state SlowlorisAttack {
22     when (enter) do {
23       setAttackTCAMRule();
24       send "attack" to Slowloris@Harvester;
25     } }
26   when (recv int newThreshold from Slowloris@Harvester) do
27     { threshold = newThreshold; }
28   when (recv list undoAttack from Slowloris@Harvester) do {
29     setTCAMRule(undoAttack);
30     transit normal;
31   } }
```

List. 2.9: Slowloris seed example.

2.5.7 Link Failure Detection

To detect link failures, the ASIC offers a link status. The seed implementation in List. 2.10 takes advantage of the existing status messages and poll them. Once a change is detected (link recovers or link fails), an internal list gets updated, and the seed sends the updated list to the harvester and transits back to the observation state.

```
1 machine LinksStatus {
2   place all;
3   time pollingStats = 1;
4   list newRecos;
5   list newFails;
6   state observe {
7     when (pollingStats) do {
8       newRecos = getRecoveries();
9       newFails = getFailures();
10      if (not newRecos.empty() or
11         not newFails.empty()) then {
12        transit changeDetected;
13      }
14    }
15  }
16  state changeDetected {
17    when (enter) do {
18      if (not newRecos.empty()) then
19        { send newRecos to RecoLinks@Harvester; }
20      if (not newFails.empty()) then
21        { send newFails to FailLinks@Harvester; }
22      transit observe;
23    }
24  }
25 }
```

List. 2.10: Link status monitor seed example.

2.5.8 Traffic Change Detection

The traffic change detector is an external algorithm discussed in [160]. It detects changes in network traffic patterns (e.g., volume, number of connections). Depending on the results, resources have to be adjusted. The seed implementation in List. 2.11 simply executes at a given period the external traffic change detection function which returns the result that has to be set.

```
1 machine TrafficChangeDetector {
2   TCAM flowRules;
3   TCAM TCAMRule = detectorRules();
4   time pollingStats = 10;
5   state normal {
6     when (pollingStats) do {
7       flowRules = exec TraffChangeDetection();
8       setTCAMRule(flowRules);
9     }
10  }
11  when (recv list newFlowRules
12    from TrafficChangeDetector@Harvester) do
13    { flowRules = newFlowRules; }
14 }
```

List. 2.11: Traffic change detection seed example.

2.5.9 Flow Size Distribution

Duffield et al. [53] set out to understand detailed flow statistics of Internet traffic on the basis of flow statistics compiled from sampled packet streams. Increasingly, only sampled flow statistics are available: inference is required to determine the flow characteristics of the original unsampled traffic and propose two inference methods. The scaling method codified the heuristic that when sampling 1 out of N packets, since sampled flows have roughly $1/N$ of their packets sampled, the length of the original flow should be N times the sampled flow. The approach is implemented in List. 2.12 and only requires one state with two different polling rates to deliver (a) the statistics to the moment based estimator and (b) probe packets and start a maximum likelihood estimator. The past results are sent to the harvester.

```
1 machine FlowSizeDistribution {
2   TCAM TCAMRule = distributionRules();
3   time MBEPollingStats = 10;
4   time MLEPollingRate = 100;
5   time SendingPollingRate = 1000;
6   list MBERResults;
7   list MLERResults;
8   list SYNFlowsResults;
9   packet samplePacket;
10  state normal {
11    when (MBEPollingStats) do
12      { MBERResults = MBERResults +
13        MomentBasedEstimator(); }
14    when (MLEPollingRate) do {
15      samplePacket = probe(TCAMRule);
16      SYNFlowsResults = SYNFlows(samplePacket);
17      MLERResults = MLERResults +
18        MaximumLikelihoodEstimator(MBERResults,
19          SYNFlowsResults);
20    }
21  }
22  when (SendingPollingRate) do {
23    send MLERResults
24    to FlowSizeDistribution@Harvester;
25  }
26 }
```

List. 2.12: Flow size distribution seed example.

2.5.10 Superspreader

List. 2.13 depicts superspreader detection [183]: sources that send traffic to a large number of distinct destinations. The seed counts source-destination pairs and classifies a source as superspreader once the number of pairs for that source crosses a given threshold, which depends on the network and its state. Once a superspreader is detected, the number of its connections can be limited and it can even be completely locked out.

```
1 machine Superspreader {
2   place all;
3   time pollingStats = 10;
4   time pollAnomaly = 10;
5   list conns;
6   int switchThresh;
7   int indivThresh;
8   state normal {
9     when (pollingStats) do {
10      conns = getConnections();
11      if (checkIndivConn(conns, indivThresh)
12      then { transit suspect; }
13      if (checkGlobalConn(conns, switchThresh)
14      then { setIndividualTCAMRules(conns); }
15    }
16    state suspect {
17      when (enter) do { setSuspectedTCAMRule(); }
18      when (pollingStats) do {
19        if (isNormal()) then { transit normal; }
20        send "suspect" to Superspr@Harvester;
21        transit superSpreader;
22      }
23      state superSpreader {
24        when (enter) do {
25          setAttackTCAMRule();
26          send "attack" to Superspr@Harvester;
27        }
28        when (recv int newSwTh from Superspr@Harvester) do {
29          switchThresh = newSwTh; }
30        when (recv int newInTh from Superspr@Harvester) do {
31          indivThresh = newInTh; }
32        when (recv list undoAttack from Superspr@Harvester) do {
33          setTCAMRule(undoAttack);
34          transit normal;
35        }
36      }
37    }
38  }
```

List. 2.13: Superspreader seed example.

2.5.11 SSH Brute Force

The code in List. 2.14 detects SSH brute force attacks [93]: repeated SSH login attempts by an attacker willing to gain shell access to one or multiple hosts. The seed first observes all SSH connections and, when an SSH brute force attack epoch is identified, switches to the `AttackParticipantAnalyzer` state. The seed then classifies the hosts appearing during the detected epochs as participants or non-participants of the attack, based on individual past history and “coordination glue”, i.e., the degree to which a given host manifests patterns of probing similar to that of other hosts during the epoch. Participants of an attack will be excluded via TCAM rules.

```
1 machine SSHBruteForce {
2   place range -1;
3   time pollingStats = 10;
4   long globalFailIndic;
5   long threshold;
6   state AggregateSiteAnalyzer {
7     when (pollingStats) do {
8       globalFailIndic = globalFailIndic
9         + getSSHConn();
10      if (globalFailIndic >= threshold) then
11        { transit AttackParticipantAnalyzer; }
12    }
13  }
14  state AttackParticipantAnalyzer {
15    when (enter)
16    do { setTCAMRules(globalFailIndic); }
17    when (pollingStats) do {
18      calcAndRemoveAttacker();
19      transit AggregateSiteAnalyzer;
20    }
21  }
22  when (recv long ctrlTh from SSHBFA@Harvester)
23  do { threshold = ctrlTh; }
24 }
```

List. 2.14: SSH brute force seed example.

2.5.12 Port Scan

The seed in List. 2.15 implements a threshold random walk algorithm [89] to rapidly detect portscanners based on observations (*getPortScans* function) of whether a given remote host connects successfully to newly-visited local addresses. The algorithm is motivated by the empirically-observed disparity between the frequency with which such connections are successful for benign hosts vs. known-to-be malicious hosts (*getDeltaY* function). With the result of the *getDeltaY* function a transition to corresponding state is executed, where necessary TCAM rules will be deployed. The thresholds for the approach are set by the harvester.

```
1 machine ThresholdRandomWalk {
2   place range -1;
3   time pollingStats = 10;
4   list Y;
5   long deltaY;
6   long n1;
7   state init {
8     when (pollingStats) do {
9       Y = Y + getPortScans();
10      deltaY = getDeltaY(Y);
11      if (deltaY >= n1) then
12        { transit scanner; }
13      if (deltaY <= n1) then
14        { transit benign; }
15    }
16  }
17  state scanner {
18    when (enter) do {
19      setExcludeScannerTCAMRules(Y);
20      transit init;
21    }
22  }
23  state benign {
24    when (enter) do {
25      setIncludeBenignTCAMRules(Y);
26      transit init;
27    }
28  }
29  when (recv long n1Update
30    from PortScan@Harvester) do
31    { n1 = n1Update; }
32 }
```

List. 2.15: Port scan seed example.

2.5.13 DNS Reflection

With a DNS reflection attack [109], systems running a certain TCP stack can be abused to amplify TCP traffic by a factor of $20\times$ or higher. To prevent the attack, DNS calls from the inner system to suspected individual hosts have to be checked. The seed implemented in List. 2.16 reflects four typical attack prevention states (`normal`, `anomaly`, `suspect`, `attack`). In the `normal` and `anomaly` states, polling and observing resources will be increased. In the `suspect` state, network resources will be reduced until an attack is identified (`attack` state) and the resources will be cut off for the attacker.

```
1 machine DNSReflection {
2   TCAM IPRangeX range 0;
3   place IPRangeX;
4   time pollNormal = 10;
5   time pollAnomaly = 10;
6   int obsTime;
7   int obsTimer;
8   int DNSThreshold;
9   list ruleSpct;
10  list ruleAtck;
11  state normal {
12    when (pollNormal) do {
13      if (not isAnomaly()) then {
14        send pollNormal
15          to DNSReflectionAttack@Harvester;
16        obsTime = 0;
17        transit anomaly;
18      }
19    }
20  }
21  state anomaly {
22    when (pollAnomaly) do {
23      if (isNormal()) then
24        { transit normal; }
25      if (obsTimer >= obsTime) then
26        { transit suspect; }
27      obsTime = obsTime + 1;
28    }
29  }
30  state suspect {
31    when (enter) do {
32      setSuspectedTCAMRule(ruleSpct);
```

```

33     suspectSent = false;
34 }
35 when (pollAnomaly) do {
36     if (isNormal()) then
37         { transit normal; }
38     if (not suspectSent) do {
39         suspectSent = true;
40         send "suspect"
41         to DNSReflectionAttack@Harvester;
42     }
43 }
44 }
45 state attack {
46     when (enter) do {
47         setAttackTCAMRule(ruleAtck);
48         send "attack"
49         to DNSReflectionAttack@Harvester;
50     }
51 }
52 when (recv int newObsTimer from DNSReflectionAttack@Harvester) do
53     { obsTimer = newObsTimer; }
54 when (recv list newRuleSpct from DNSReflectionAttack@Harvester) do
55     { ruleSpct = newRuleSpct; }
56 when (recv list newRuleAtck from DNSReflectionAttack@Harvester) do
57     { ruleAtck = newRuleAtck; }
58 }
```

List. 2.16: DNS reflection seed example.

2.5.14 Entropy Estimation

To estimate the entropy of an unknown flow of packets, multiple probing techniques are necessary depending on the given flow and its state. Three different probing techniques have been discussed in the literature [131] to achieve good results. The seed implementation in List. 2.17 has four states. The `normal` state finds the best probing technique and transits to the state implementing the corresponding technique (states `linearProbing`, `balancedAllocation`, and `bloomFilter`). After computing the latest information in every state, the seed transits to the optimal probing schema. A summary of the past observations is sent to the harvester at the given `entropyStats` period.

```
1 machine EntropyEstimation {
2   TCAM rule = entropyRules();
3   time pollingStats = 10;
4   time entropyStats = 1000;
5   list pastObs;
6   list EntropyValues;
7   state normal {
8     when (enter) do {
9       transit findOptimalProb(rule);
10    } }
11  state linearProbing {
12    when (pollingStats) do {
13      pastObs = pastObs +
14        getLinearProbingEntropy(rule);
15      transit findOptimalProb(pastObs);
16    } }
17  state balancedAllocation {
18    when (pollingStats) do {
19      pastObs = pastObs +
20        getBalancedAllocEntropy(rule);
21      transit findOptimalProb(pastObs);
22    } }
23  state bloomFilter {
24    when (pollingStats) do {
25      pastObs = pastObs +
26        getBloomFilterEntropy(rule);
27      transit findOptimalProb(pastObs);
28    } }
29  when (entropyStats) do {
30    send pastObs
31    to EntropyEstimation@Harvester;
32  }
```

List. 2.17: Entropy estimation seed example.

2.5.15 FloodDefender

SDN-aimed DoS attacks can paralyze OpenFlow networks by exhausting the bandwidth, computational resources, and flow table spaces. FloodDefender [163] is a system to protect OpenFlow networks against SDN-aimed DoS attacks based on four modules: attack detection, table-miss engineering, packet filter, and flow table management. FloodDefender uses a queueing delay model to analyze how many neighbor switches should be used in the table-miss engineering. We propose a seed implementation in List. 2.18, made up of four different machines corresponding to the mentioned modules which also communicate with each other, a feature that is only shown in this example.

```
1 machine AttackDetectionModule {
2   place all;
3   time pollingStats = 10;
4   state checkTableMiss {
5     when (pollingStats) do {
6       if (getNumTableMiss() > threshold)
7         then { transit splitTableMiss; }
8       transit stopAttack;
9     }
10  }
11  state splitTableMiss {
12    when (enter) do {
13      deploySplitTableMiss();
14      send "start" to FloodDefender@Modules;
15      send "tableMiss" to FloodDefender@Harvester;
16    }
17  }
18  state stopAttack {
19    when (enter) do {
20      send "stopAttack" to FloodDefender@Modules;
21    }
22  }
23 }
24
```

```

25
26 machine PacketFilterModule {
27     list threshold;
28     place all;
29     state startModule {
30         when (enter) do {
31             observePacketFilterParameter();
32             setThreshold();
33         }
34     }
35     state stopModule {
36         when (enter) do { reset(); }
37     }
38     when (recv list newThreshold from FloodDefender@Module) do {
39         threshold = newThreshold;
40         transit startModule;
41     }
42     when (recv "stopAttack" from FloodDefender@Module) do
43         { transit stopModule; }
44 }
45
46
47 machine TableMissEngineeringModule {
48     list schema;
49     place all;
50     state startEngine {
51         when (enter) do {
52             installProtectionRules(schema);
53             send "currentSchema" to FloodDefender@Neighbors;
54         }
55     }
56     state stopEngine {
57         when (enter) do {
58             uninstallProtectionRules(schema);
59             send "uninstallSchema" to FloodDefender@Neighbors;
60         }
61     }
62     when (recv list newSchema from FloodDefender@Neighbors) do {
63         schema = newSchema;
64         installProtectionRules(schema);
65     }
66     when (recv "uninstallSchema" from FloodDefender@Neighbors) do
67         { uninstallProtectionRules(schema); }
68 }
69
70

```

```
71 machine FlowTableManagement {
72     list rules;
73     place all;
74     state startModule {
75         when (enter) do { manageRules(rules); }
76     }
77     state stopModule {
78         when (enter) do { manageRules(rules); }
79     }
80     when (recv list newRules from FloodDefender@Module) do {
81         rules = newRules;
82         transit startModule;
83     }
84     when (recv "stopAttack" from FloodDefender@Module) do
85         { transit stopModule; }
86 }
```

List. 2.18: FloodDefender seed example.

Table 2.4: Elements and notation of optimization model.

Description	Element Set		Description	Element Set	
Seed aggregation groups	g	G	Seeds	m	M
			Resource types	r	R
M&M task	t	T	Switches	s	S

2.6 M&M Seed Placement

This section formulates FARM’s seed placement problem and discusses its hardness and solution.

2.6.1 Rationale and Notation

As introduced in § 2.3 and § 2.4, FARM enables the description of switch-local M&M tasks and their deployment in a distributed manner as seeds on switches. Multiple seeds of the same M&M task may be positioned on different switches, say along a flow, based on resource availability. A seed may also be migrated; either due to placement constraint for the seed not being satisfied anymore (e.g., after a routing change) or due to a new seed with a higher utility needing to use the same switch. Migration induces a migration cost as the seed’s own inner state must be also migrated. Moreover, the polling periods of a seed can be adjusted to save resources on a switch. Also, certain seeds can benefit from aggregation (see § 2.3.4) as they consider the same data. Considering the most accurate and best performance for M&M tasks and the many parameters and options available, we propose an algorithm to optimize seed placement in FARM.

Tab. 2.4 summarizes notation of the involved entities. Lowercase letters denote elements of respective kinds, while the set of all elements of a kind is denoted by the corresponding uppercase letter. E.g., s denotes an individual switch while the set of all switches is represented by S . To run a seed on a switch, different types *resource types* (e.g., CPU, memory) are required. We cluster seeds into *aggregation groups* to formally capture which seeds lead to aggregation benefits, by analyzing the same data.

2.6.2 Placement Quality

The goal of optimization is to *maximize* the *placement quality* (PQ), which is defined as a sum of *monitoring utility* (MU) and *aggregation benefits* (AB) less the *migration costs* (MC)

Table 2.5: Functions and variables of optimization model.

Optimization <i>input</i> description	Function
Returns a set of seeds that belong to t	$\text{seedS}(t)$
Utility of giving seed m 1 unit of type r resource	$\text{util}(m, r)$
Returns a set of switches where m can be placed	$\text{plctS}(m)$
Migration costs when migrating m to s	$\text{migr}(m, s)$
Aggregation benefit for g	$\text{agg}(g)$
Minimum resources of type r used by m	$\text{res}_{\min}(m, r)$
Maximum resources of type r used by m	$\text{res}_{\max}(m, r)$
Available resources of type r on s	$\text{ares}(s, r)$
Optimization <i>variable</i> description	Function
Returns 1 if all t 's seeds are placed, else 0	$\text{tplc}(t)$
Returns 1 if m is placed on s , else 0	$\text{plc}(m, s)$
Amount of res. of type r assigned to m at s	$\text{res}(m, s, r)$

of migrating seeds between switches:

$$\text{maximize (PQ)} = (\text{MU}) + (\text{AB}) - (\text{MC}) \quad (2.1)$$

The three terms (MU, AB, and MC) are explained below. Tab. 2.5 summarizes the helper functions and variables used.

(MU) Monitoring utility. The monitoring utility captures the benefits of assigning resources to seeds in terms of the quality of monitoring for the entire system. Thus, a sum is computed — over all seeds m in M , all switches s in S , and all resource types r in R — of the product of the per-resource utility of a seed and its assigned resources $\text{res}(m, s, r)$:

$$\text{MU} = \sum_{r \in R} \sum_{m \in M} \sum_{s \in S} \text{util}(m, r) \cdot \text{res}(m, s, r)$$

(AB) Aggregation benefits. Aggregating seeds at the same switch can reduce data polling cost, as described in § 2.3.4. We compute its benefits by multiplying the aggregation benefits $\text{agg}(g)$ of an aggregation group g by the number of seeds of the aggregation group g that are placed on switch s . Naturally, aggregation benefits are only obtained if more than one seed is placed on a switch s . $\text{plc}(m, s)$ denotes whether a seed is placed on a switch s or not (returning 1 or 0, respectively). Thus, the number of seeds of an aggregation group g on a switch reduced by 1 is expressed as $\max(0, \sum_{m \in g} \text{plc}(m, s) - 1)$. Total aggregation benefits is a sum over all switches s and all aggregation groups g :

$$AB = \sum_{g \in G} \text{agg}(g) \cdot \sum_{s \in S} \max(0, \sum_{m \in M} \text{plc}(m, s) - 1)$$

(MC) Migration costs. While seed migration can generally optimize the seed layout, it incurs costs that must be considered. Migrating a seed consists of installing its description on the target switch and transferring its state over from the source switch. As the state is being transferred, and before it is deleted on the source switch, the seed resource utilization is temporarily doubled. Furthermore, the network load of the SDN control plane increases during state migration. $\text{migr}(m, s)$ returns the cost of migrating seed m to switch s , its result varies between different seeds, depending on the complexity of the seed definition. In particular, there is no cost associated with the migration of seed m to switch s (i.e., $\text{migr}(m, s) = 0$) if s is already executing another instance of m , m is instead locally duplicated. Migration costs are summed over all seeds m in M and switches s in S :

$$MC = \sum_{m \in M} \sum_{s \in S} \text{migr}(m, s) \cdot \text{plc}(m, s)$$

2.6.3 Constraints

Several constraints have to be taken into account when optimizing monitoring seed placement. For instance, we must guarantee that every seed of every M&M task is placed on *some* switch, or, if there are not enough resources, that none of this task's seeds are counted toward utility. In the following, we describe all six constraints (C1)–(C6) that have to be fulfilled for valid seed placement.

(C1) Every seed is placed at one switch at most. This constraint guarantees that for any M&M task $t \in T$ if one of t 's seeds is placed, then every t 's seed $m \in M$ is placed at most one switch $s \in S$ only:

$$\sum_{s \in S} \text{plc}(m, s) = \text{tplc}(t) \quad \forall t \in T, m \in \text{seedS}(t)$$

(C2) Respecting placement constraints. As described in § 2.4, a seed states placements constraints, as part of its definition. Optimization captures these as follows:

$$\text{plc}(m, s) \leq [s \in \text{plctS}(m)] \quad \forall m \in M, s \in S$$

(C3) Maximally assigned resources for placed seeds. The amount of type r resources assigned to m on s , $\text{res}(m, s, r)$, shall not exceed the defined maximally required resources $\text{res}_{\max}(m, r)$ and can only be non-zero if m is placed at s :

$$\text{res}(m, s, r) \leq \text{res}_{\max}(m, r) \cdot \text{plc}(m, s) \quad \forall s \in S, m \in M, r \in R$$

(C4) Minimally assigned resources. If m is placed on s , it has to be assigned sufficient resources of type r . The minimum requirements of m for type r are $\text{res}_{\min}(m, r)$, thus:

$$\text{res}(m, s, r) \geq \text{res}_{\min}(m, r) \cdot \text{plc}(m, s) \quad \forall s \in S, m \in M, r \in R$$

(C5) Switch resource limit for all seeds. The total of type r resources assigned to seeds at s cannot exceed the available resources of type r at s :

$$\sum_{m \in M} \text{res}(m, s, r) \leq \text{ares}(s, r) \quad \forall s \in S, r \in R$$

§ 2.7.2 outlines our implementation of seed placement under consideration of this MILP problem.

2.6.4 Placement Optimization Algorithm

The hardest part of placement optimization is assigning seeds to switches. After assignment, local optimization becomes a network flow problem and is efficiently solvable.

Getting the assignment right is actually an intractable optimization problem. If we ignore M&M tasks and aggregation benefits, and even consider only one switch, the remaining constraints (C2)–(C5) would still be at least as expressive as the multi-dimensional knapsack problem (MdKP). Thus, the placement optimization problem is *NP-hard in the strong sense*, just like MdKP [102], even for two resource types. Note that the hardness introduced by the multiplicity of resources is unavoidable; as we show in § 2.8.5 switches have several bottlenecks. The time to compute a MILP solution highly depends on the specific problem instance.

To address possible scalability issues in larger deployments, we propose a simple heuristic in Alg. 1.

2.7 Implementation

In this section we elaborate on several aspects of the implementations of FARM’s components and Almanac, and of the seed placement optimization algorithm.

2.7.1 FARM Components

We shed light on four main parts of FARM’s implementation: (1) integration with existing switches’ HW & SW, (2) switch-local and (3) centralized components, and (4) Almanac.

Algorithm 1 FARM seed placement optimization heuristic.

1. Sort M&M tasks T in the order of decreasing minimum utility less minimum migration cost getting t_1, t_2, \dots, t_n .
 2. For every $t \in t_1, t_2, \dots, t_n$
 - a) Repeat while possible: among $m \in \text{seedS}(t)$ choose and *place* such m that adds the most to the current quality.
 - b) If there remains $m \notin \text{seedS}(t)$, remove all $\text{seedS}(t)$ from the current placement.
 3. Redistribute resources using network flow formulation.
-

Switch integration. FARM implements two drivers responsible for the communication between the CPU and the ASIC via the PCIe bus: one driver for Stratum [166] and one for Arista’s EOS SDK [57]. Stratum is an OS module, available for Open Network Linux (ONL) among others, that abstracts the hardware layer of ASICs from major vendors (e.g., Barefoot, Broadcom, Mellanox) to provide common interfaces (e.g., P4Runtime, OpenConfig). As such, FARM is deployable on all ASICs supported by Stratum and Arista EOS switches.

We ensured that communication over the PCIe bus between the (i) CPU running the soil and seeds and (ii) ASIC can be scheduled to fully exploit the bus’ capabilities.

Switch-local components. The seeds and the soil are optimized to be executed directly on switching devices. Seeds can run as isolated processes or as threads of the soil process.

Communication between seeds and the soil is done over a generic interface that supports two communication schemes — one using gRPC [72] and one using a tailor-fitted shared memory buffer usable when seeds are implemented as threads of the soil. gRPC’s poor performance motivated the development of the second scheme. We evaluate both seed execution models and communication schemes in § 2.8.5.

Centralized components. While FARM aims at leveraging as much switch resources as possible, it introduces also centralized components, i.e., the seeder and the harvesters, to support switch-local components. Both seeder and harvesters are implemented in Python and contain a communication service used to interact with the communication service of the soils to exchange data with both soils and the seeds they support. Communication between seeder/harvesters and soils (e.g. Tab. 2.2) is performed via RabbitMQ messages [150].

Almanac. With Almanac, M&M applications can be described irrespective of details of network topology and devices (HW and SW). State machines for seeds are described in Almanac, compiled by the seeder into XML, and transformed from XML to one or more seeds by each switch’s soil. XML is used for interoperability and portability across OSs. The `ttypes` in Almanac (cf. § 2.4) are used by the soil to optimize communication with the ASIC over all running seeds.

2.7.2 Placement Optimization

The M&M placement function is in charge of optimizing the resource utility of the network following heuristics defined in § 2.6. This function takes as inputs (1) the list of switches using FARM, (2) their topology, (3) their local resource consumption, (4) the current seed placement, and finally (5) the resource consumption of each seed. The function outputs the new placement of the seeds and the allocated resources, i.e., period for probing packets and polling statistics. We developed the optimization algorithm in Rust [125] and used a MILP library [123] supporting multiple solvers. The performance of FARM’s placement optimization heuristic is compared to Gurobi [75] and a greedy algorithm in § 2.8.4.

The seeder calls the placement optimization algorithm, every time one of the input parameters of the M&M placement function changes, e.g., when a switch’s soil notifies the seeder that its resources are depleted. The seeder takes the actions necessary to realize the optimizer’s output, e.g., by migrating seeds. When migrating a seed, the seeder first deploys the description of the relocated seeds to its new location, then transfers its state there. The seed resumes execution once the state is migrated.

2.8 Evaluation

In this section, we present the evaluation of FARM in a production DC of SAP SE articulated around the following questions:

- (§ 2.8.2) How does FARM fare against state-of-the-art solutions in terms of responsiveness (i.e., reaction time, mitigation time), network load, and switch CPU load?
- (§ 2.8.3) How does FARM’s monitoring accuracy (which affects responsiveness) scale with a large number of — possibly CPU-intensive — seeds executed concurrently?
- (§ 2.8.4) How does FARM’s placement optimization algorithm scale in terms of placement quality (cf. Equation 2.1) and runtime?
- (§ 2.8.5) How efficient is FARM’s implementation?

Tab. 2.6: HH detection time with FARM, sFlow, Sonata, and specialized link utilization monitoring systems Planck and Helios.

System	Type	Time
FARM	Generic	1 ms
Planck 10 Gbps [152]	Specific	4 ms
Helios [61]	Specific	77 ms
sFlow [147]	Generic	100 ms
Sonata [74]	Generic	3427 ms

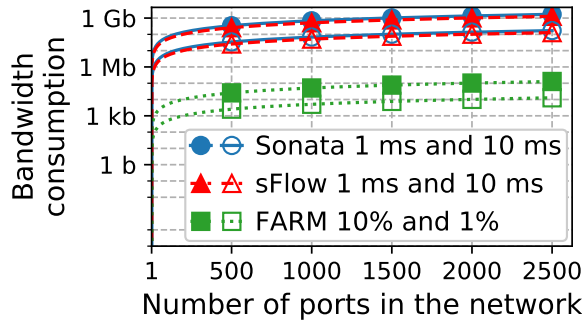


Figure 2.4: Network load of FARM with 1 and 10% HH ratios, the sFlow collector with 1 and 10 ms accuracies, and similarly Sonata.

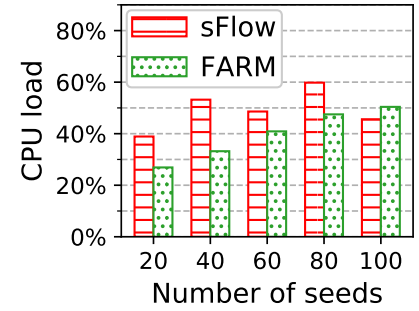


Figure 2.5: Switch CPU load of FARM and sFlow for HH detection, 10 ms accuracy.

2.8.1 Setup

HW & SW. We used APS BF2556X-1T, Accton AS5712, Accton AS7712 and Arista 7280QRA-C36S switches. APS BF2556X-1T run ONL with a 2.0 Tbps Intel Tofino ASIC and an Intel Xeon 8-core 2.6 GHz x86 processor with 32 GB SO-DIMM DDR4 RAM with ECC. Accton AS5712 run ONL and have an Intel Atom C2538 quad-core 2.4 GHz x86 processor with 8 GB SO-DIMM DDR3 RAM with ECC. Accton AS7712 have the same OS and CPU but twice the amount of RAM. Arista 7280QRA-C36S run EOS and have an AMD GX-424CC SOC quad-core 2.4 GHz with 8 GB DRAM.

Topology. We deployed FARM on a cluster with a spine-leaf topology (cf. Fig. 2.1) in a production DC of SAP SE. As FARM is undergoing a long-term evaluation period before being globally rolled out we report performance results on 20 switches.

HH task. We investigate the HH detection task presented in § 2.4.2 by deploying one of such seed per port on all switches.

ML task. Leveraging machine learning (ML) for prediction is a budding field in networks, with neural networks being used in various settings [145, 17, 54]. Additional support for prediction is an often stated need for monitoring [158].

We thus investigate FARM with a CPU-intensive task using ML to react to switch-local events directly on switches. The ML-based tasks CLAIRE chapter 3 instances, predicting the evolution of individual flows, using matrix-matrix multiplications with 1000×1000 matrices. The matrix represents a model characterizing network flows' burstiness, long-range dependence, self-similarity, and periodicity. The algorithm decomposes a flow in a multi-scale manner into a set of linear and stable representations. The Python implementation is executed externally by a seed via `exec`, parameterized by the polled statistics. These seeds incur no network load as they do not interact with others.

2.8.2 Scalability

Identifying HHs is useful for many purposes (e.g., flow-size aware routing, DoS detection, and traffic engineering). Therefore, we use the HH task to compare FARM to other solutions, including ones specialized for HH detection, focusing on responsiveness, network load, and CPU load.

While HH detection does not show the full potential of FARM, its wide-spread use in literature allows the most meaningful comparison against existing approaches. We present a wide variety of M&M tasks supported by FARM in § 2.5.

We evaluate in detail against sFlow [147] and Sonata [74], two generic solutions representatives of collector- and stream-based approaches. Other recent approaches like Newton [194] show promising results, but have the same conceptual limitations as sFlow or Sonata, and are not publicly available.

Responsiveness. Tab. 2.6 compares the time needed to recognize an HH with FARM also to the more specialized Planck [152] (leveraging specialized hardware) and Helios [61] systems. FARM shows great speedups while being at least as generic (sFlow) or more

generic (Planck, Helios, Sonata). Transitively FARM also greatly reduces mitigation time. Note that Sonata only computes a switch-local HH instead of a global one (cf. § 2.2).

FARM achieves such speedups by analyzing traffic directly on switches while the other solutions send raw statistics and data to centralized instances. Another advantage of FARM is the ability to *react* on a switch when recognizing an HH. E.g., to install a rate limit for HHs, a *TCAMaction* can be described with Almanac for the seed in the `HHdetected` state. Both HH recognition *and* mitigation can happen within 1 ms.

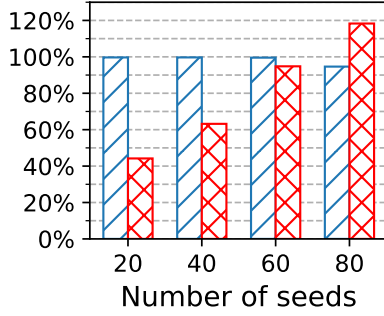
Network load. Fig. 2.4 depicts network load for HH detection and highlights FARM’s benefits over sFlow and Sonata. We chose HH parameters based on observations in our production DC — HHs usually affect 1% of the network ports, 10% at worst, and the HH ratio changes up to once a minute.

sFlow periodically sends packets to probe every port in the network. We thus run sFlow with a 1 ms probing period to achieve a similar detection time as that of FARM, as well as with a 10 ms probing period to reduce load since the load of collector-based solutions increases linearly with the network size. Assuming Sonata could aggregate over several switches to compute HHs, the raw statistics issued by the switches to the Spark system deployed by Sonata would still create further network load. We run Sonata assuming an aggregation factor of 75%, which is the best that Sonata could achieve with an HH ratio changing up to once a minute. Further decentralizing aggregation by using more aggregation levels would only generate yet more network traffic. In comparison, FARM’s bandwidth consumption increases by only 1 packet per minute if the network increases by 100 ports.

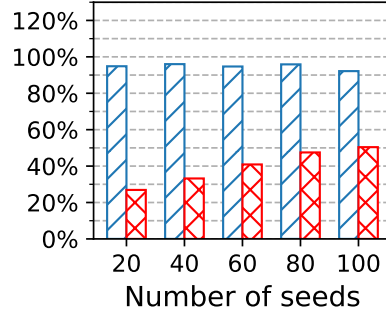
This yields also a linear gradient as shown in Fig. 2.4 (note the log *y* axis). The total amount and the slope is much lower for FARM than for the sFlow collector. Besides using less bandwidth, the computational effort of the collector centralized instance is much higher than FARM’s ($> 1000\times$). Moreover, if the HH ratio changes more often, it is important to recognize the change immediately, which FARM enables.

CPU load. Fig. 2.5 depicts FARM’s and sFlow’s CPU loads as they poll statistics from multiple flow rules with equal monitoring accuracy. We do not compare against Sonata because it mirrors the traffic and thus its bottleneck is the sampling rate of the PCIe bus (cf. § 2.8.5). Its number of individual instances is not meaningful due to the lack of samples.

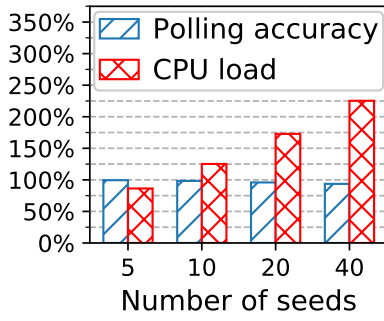
sFlow’s CPU load is higher than FARM’s in all cases except with 100 flows. sFlow’s CPU load is stable since it is a (locally) light approach that samples packets and forwards them to its centralized collector without filtering. On the other hand, FARM analyzes the data



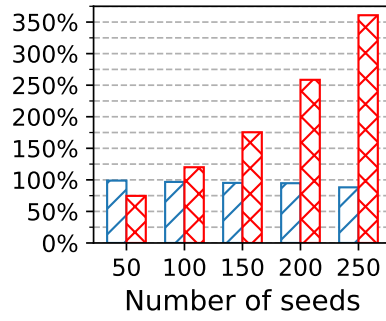
(a) HH seeds: 1 ms accuracy.



(b) HH seeds: 10 ms accuracy.



(c) CLAIRE seeds: 1 ms accuracy, 1 iteration.



(d) CLAIRE seeds: 10 ms accuracy, 10 iterations.

Figure 2.6: CPU load of FARM for an HH and CLAIRE seeds.

and manages its own state, thus CPU load increases with the number of monitored ports. Yet, as long as not all ports are affected, the SDN control plane is not congested with FARM as it is with sFlow (cf. Fig. 2.4).

2.8.3 FARM's Accuracy vs CPU Load

We evaluate the effect of running many collocated seeds on the same switch, specifically from the angle of monitoring accuracy (i.e., polling period) and its impact on CPU load.

HH task. Fig. 2.6a and Fig. 2.6b show CPU load with various numbers of seeds with every seed polling statistics from multiple flow rules every 1 and 10 ms respectively. The HH task incurs only light CPU load and easily scales to more than a hundred of seeds per

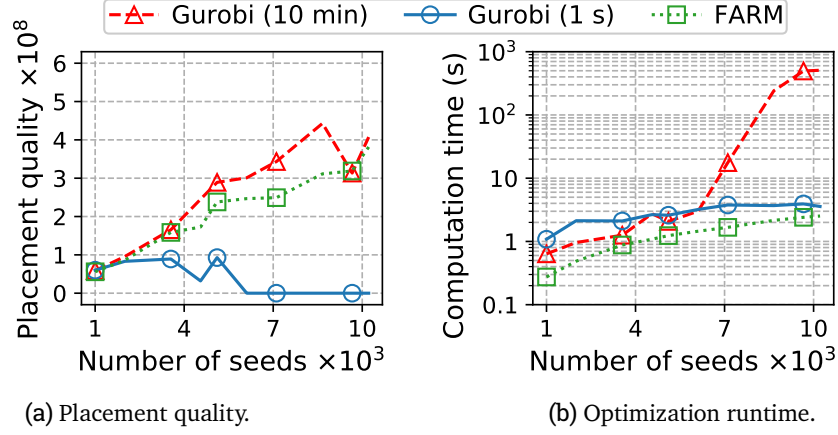


Figure 2.7: FARM's global Seed placement optimization algorithm (cf. Equation 2.1) is close in quality to Gurobi with 10 min timeout and as fast as Gurobi with 1 s timeout.

switch with a 10 ms accuracy.

ML task. Because of their complexity we run ML seeds with a 1 ms accuracy in parallel (cf. Fig. 2.6c), and (2) a 10 ms accuracy with statistics polling once but executing 10 iterations of the algorithm (cf. Fig. 2.6d) thus dividing by 10 the number of seeds executed in parallel. For comparison we use the same statistics polling periods as for the HH task.

Fig. 2.6c shows the CPU load is $\approx 150\%$ higher for the ML task with a 1 ms accuracy than for the HH task. This leads to the situation where the CPU is unable to handle all seeds in parallel due to the many context switches. By dividing the seed into partitions (cf. Fig. 2.6d), the CPU load decreases and the system scales well up to 250 seeds of this ML task.

2.8.4 Global Seed Placement Optimization

Another critical factor for scalability is FARM's placement optimization of different M&M tasks over distributed network resources. For this evaluation, we compare FARM's placement optimization algorithm against a commodity MILP solver using Gurobi [75] (used by Sonata). Two timeouts are used for Gurobi: 1 s to get runtime similar to FARM, and 10 min as an absolute practical upper bound. We test all approaches with up to 10 different tasks (cf. Tab. 2.3) comprising up to 10200 seeds and deploy them on 1040 switches. For every number of seeds to deploy, we execute 10 runs with different resource and placement requirements for tasks.

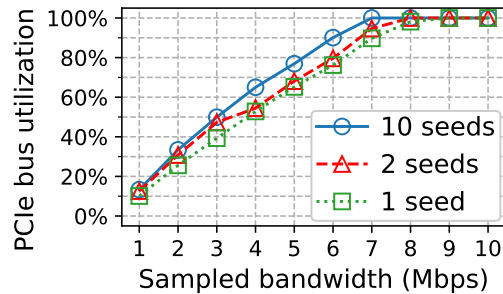


Figure 2.8: The PCIe bus easily congests compared to the ASIC bus, calling for polling aggregation.

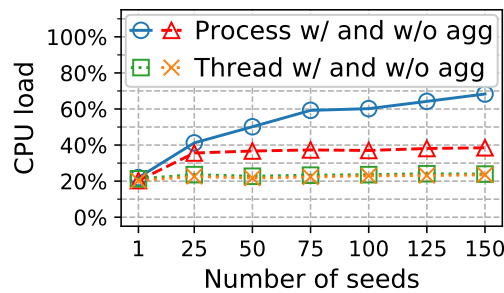


Figure 2.9: Soil's CPU load showing the cost of Seed requests' aggregation when Seeds are threads vs processes.

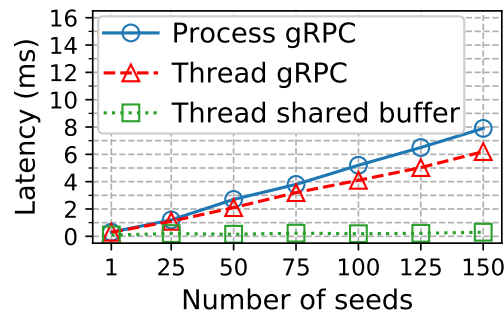


Figure 2.10: Shared buffer vs gRPC communication latency between seeds being threads or processes and soil.

Fig. 2.7a shows the average placement quality (cf. Equation 2.1) and Fig. 2.7b the average time needed to find a corresponding solution. FARM’s placement optimization algorithm achieves similar quality compared to Gurobi but much faster, which is crucial with a large number of tasks and seeds to deploy. Gurobi’s runtime is greater than its timeouts since converting a problem instance to a format Gurobi supports takes time.

2.8.5 Implementation Microbenchmarks

We show via a series of microbenchmarks the need for the optimizations implemented in FARM (cf. § 2.7.1). In particular, we show that FARM performs best with seeds executing as threads within the soil process and using a shared buffer for soil-seeds communication. We used this implementation for the rest of FARM’s evaluation. We deploy the ML task to benchmark switch hardware utilization and identify hardware bottlenecks. We use Accton AS5712 and Accton AS7712 switches for the benchmarks and plot the averaged (similar) results.

PCIe bus capacity. Fig. 2.8 shows that the primal bottleneck for most M&M tasks is the PCIe bus. It rapidly congests as seeds poll the ASIC’s TCAM. The PCIe bus capacity is limited to 8 Mbps on both switches while their ASICs support 100 Gbps, showing a 1:12500 ratio between the two capacities. To circumvent the PCIe bus bottleneck, FARM enables, in addition to data sample polling, the soil to aggregate the seeds’ requests before sending them over the PCIe bus.

Aggregation cost. Aggregating seeds’ requests requires computation by the soil, thus trading PCIe bandwidth for CPU consumption. Fig. 2.9 shows CPU load for aggregation is only noticeable when seeds run as processes, and that thread-based seeds (within the soil) perform equally well regardless of aggregation, even with more than 100 seeds.

Latency overhead. Since Stratum relies on gRPC for ASIC-soil communication, we initially also used gRPC for soil-seed communication. However, Fig. 2.10 shows that gRPC scales linearly with the number of deployed seeds (i.e., connections) and performs so poorly that gRPC becomes the latency bottleneck. As a fix we implemented a soil-seed communication scheme based on a shared buffer that is usable when seeds are threads within the soil. Fig. 2.10 shows a negligible latency overhead of the shared buffer scheme even with 150 seeds.

2.9 Conclusions

We introduced FARM, a novel network management and monitoring (M&M) system for large-scale DCs. The Almanac is a programming language/framework for describing autonomous seeds that co-execute M&M tasks directly on switches. The seeder optimizes their placement across switches with FARM’s specific global optimization algorithm. We evaluated the accuracy and scalability of FARM against existing generic systems (e.g., sFlow, Sonata) and specialized link utilization/HH detectors (e.g., Planck) showing how FARM reduces bandwidth requirements of centralized instances compared to collector-based approaches, and showing further potential to react directly to anomalies on the switch with predefined actions. FARM’s placement optimization algorithm is a scalable heuristic that globally optimizes utility of concurrent M&M tasks with different resource demands by considering heterogeneous hardware resources, aggregation benefits, and migration costs.

3 CLAIRE

3.1	Introduction	62
3.2	Background	65
3.2.1	Short-Time Fourier Transform	65
3.2.2	K -Means Clustering	66
3.2.3	Hidden Markov Model	67
3.2.4	Kalman Filter	68
3.2.5	Kernel Trick	70
3.2.6	Reproducing Kernels	71
3.2.7	Embedding Distributions in a Reproducing Kernel Hilbert Space . .	72
3.3	Related Work	75
3.4	CLAIRE Design Overview	77
3.4.1	Technical Challenges	77
3.4.2	Design Overview	77
3.4.3	Preprocessing	79
3.4.4	Adapting the Kalman Filter	80
3.5	Prediction and Learning Implementation	87
3.5.1	Complexity	87
3.5.2	Challenges in Monitoring and Management	89
3.6	Evaluation	90
3.6.1	Synopsis	90
3.6.2	Experimental Data	90
3.6.3	Prediction Results	96
3.7	Conclusions	105

3.1 Introduction

A fundamental issue which hampers communication system exploiting available resources is fine-grained structure of network flows consists of multiple *peaks* with a high variance [24]. These peaks are identified in many recent works as major source of performance deterioration [190, 162, 35], as they lead to buffer overflow and packet loss on switches with shared, limited resources. With the increase of network link bandwidth, up to 400 Gb/s, the problem is exacerbated, as, due to cost and performance constraints, buffer capacity grows slower than the bandwidth [170]. Thus even shorter peaks may overflow buffers. In fact, many elephant flows [144] in low bandwidth networks result from peaks which are “flattened”; in high bandwidth networks these retain their original bursty nature.

Reaction vs proaction. Reacting to peaks in time to avoid overflows and losses [190], as done by current traffic optimization works (e.g., on flow scheduling [60, 40, 6, 18, 34, 41, 189]), congestion control [120, 196, 5, 19, 4, 51, 55, 99, 97], or traffic engineering [3, 23]), becomes hard if not impossible in highly dynamic data center (DC) environments but, it would have an enormous impact on flow completion time (FCT) [97]. Ideally, existing approaches of a network would *predict* the *evolution* of traffic flow characteristics such as bandwidth — flow *trajectories* — so an impending over-utilization would be recognized early enough to take corrective measures *proactively*, thus enabling better bandwidth utilization with less packet loss in highly dynamic settings [33]. This capability would bring network management closer to the self-driving car analogy [62], where it is absolutely crucial to take corrective actions *before* an incident occurs.

Prediction primitives. Machine learning methods and prediction in particular have been successfully applied to a variety of problems in networks, e.g., congestion control [94], wireless network topology (re-)configuration [178, 180, 78], switch failure mitigation [191], and flow scheduling [54]. While most of them represent from-scratch solutions for specific problems, some are, in fact, more general primitives (e.g., providing look-ahead knowledge of switch failures [191] or flow sizes [54]). The fine-grained flow trajectory prediction is a perfect example of the latter kind of primitives bringing re-usability (i.e., improving a variety of specific solutions) and facilitating interpretability by limiting black-box-like prediction methods to a well-defined subcomponent.

Existing works on prediction of flow characteristics however consider traffic *aggregated in time and space*, i.e., consider flows over long intervals of time, combine many such flows, and predict for the same (aggregated) flows [177, 7]. As most congestion is caused by interaction of short traffic burst from elephant flows, these *coarse-grained* methods cannot

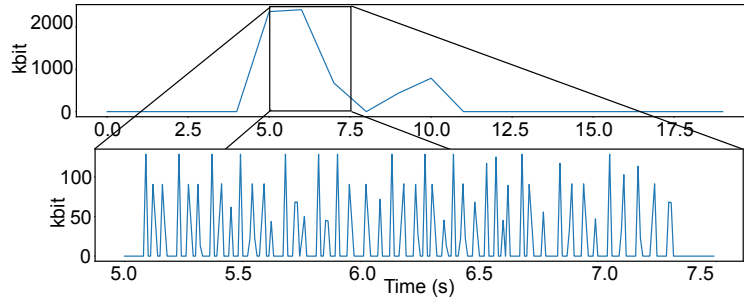


Figure 3.1: Segment of single high-volume TCP flow: sampled with a period of 1 s (top) and zoomed-in fine-grained structure of first peak sampled at 10 ms (below)

usefully predict congestion; without finer resolution in the order of buffer sizes (mind the growing bandwidth) these approaches simply will not foresee signs of congestion.

Consider Fig. 3.1, which shows a segment of a single high-volume TCP flow. When sampled with a 1 s period, the flow is seemingly non-bursty with two larger peaks, while at 10 ms we are able to see the true bursty nature of the flow.

Fine-grained trajectory prediction. Fig. 3.2 shows the intuition behind a fine-grained prediction: based on an observation $\Omega_{[\tau_1^a, \tau_1^b]}$ over *observation interval* $[\tau_1^a, \tau_1^b]$, the prediction consists of multiple points at high resolution (e.g., at 10 ms) within *prediction interval* $[\tau_1^d, \tau_1^e]$ (in green). The computation of that prediction starts at τ_1^b and finishes at τ_1^e , leaving $(\tau_1^d - \tau_1^e)$ *lead time* for preventive measures and actions. For uninterrupted prediction, two consecutive prediction intervals (cf. orange prediction) need to be overlapped by the “compute time” + “lead time”.

In this chapter we are thus interested in the feasibility of such fine-grained trajectory prediction, with the following properties:

Individual flow prediction [INDIV] at a level down to *individual* flows to enable adequate adaptations to bursts especially under congestion;

High resolution [RESOL], i.e., consisting of multiple points at small period, predicting an actual trajectory;

Sufficient prediction lead time [LEAD] in order to have a large window of opportunity for taking corrective actions;

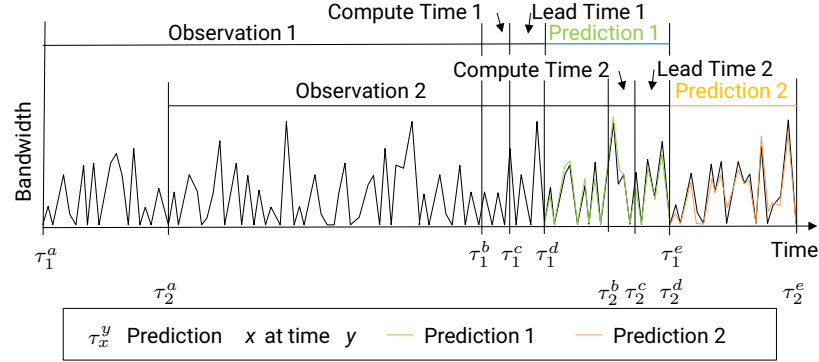


Figure 3.2: Prediction of a given network flow.

Accurate predictions [ACCUR] (close to the eventual truth) in order to avoid reacting inadequately [126];

Scalable approach [SCALE] to enable prediction of large numbers of flows simultaneously;

Commodity hardware support [IMPL] to be implementable on *commodity switches*.

To the best of our knowledge, no existing approach can cope with all requirements stated above. In fact, flow characteristics and its high-variance bursty nature (cf. Fig. 3.1) have been considered non-predictable: “*the traffic patterns inside a data center are highly divergent ..., and they change rapidly and unpredictably.*” [70]. A recent manifesto [28] reiterated the brittleness of existing “*demand estimation and workload prediction methods*”, leaving it as an open question whether “*Machine Learning (ML) and Artificial Intelligence (AI) methods could fully address this shortcoming*”.

CLAIRE. We present a specialized method, which we believe, answers the question of feasibility of fine-grained network flow prediction affirmatively. We view a network with its flows as a *non-linear system* and traffic load time series as observations. Several latent factors contribute to the true very complex *hidden* state of the networked system, e.g., types of programs generating the flows, workloads, user behavior, drivers on hosts, network components like interface cards and switches. Considering this, we introduce a novel prediction technique dubbed *clustered frequency-based kernel Kalman filter*, or CLAIRE for short, which uses the powerful concept of hidden Markov models for modeling the system behavior.

Our approach has several key points: the history of a given flow is transformed by a FFT into frequency space, where flows show a more distinctive pattern espacially by considering particular frequency components. The PCA [96] extracts key characteristics (key frequency components). For accurate learned models and to enable CLAIRE to run on commodity DC network switches with limited hardware resources, individual flows are grouped via k -means into clusters with their learned CLAIRE models. The models relying on a kernel-based variant of the Kalman filter, which captures the high-dimensional, non-linear state space.

We demonstrate the effectiveness of our approach by evaluating the CLAIRE on 20 GB of recorded header data from SAP SE. In summary CLAIRE predicts individual flows's trajectories ([INDIV]) over 500 ms at a (resolution) period of 10 ms ([RESOL]) with an average prediction error of -8.86% ([ACCUR]), which is sufficient for many traffic engineering [90, 80, 60, 46] and congestion control [120, 196, 5, 19, 4, 51, 55, 99, 97] approaches to act proactively. Similar improvements have been measured with 1.2 GB collected as part of a study conducted by Benson et al. aimed at analyzing the characteristics of university DC network traffic [22] but are emitted due to space limitation.

A simple TE approach in combination with the CLAIRE is able to reduce overall packet loss by 78.9%, and increase the throughput by 40%. The computation time for one prediction is 1.89 ms on average for a single flow.

Contributions and roadmap. This chapter contributes: (i) a novel approach for trajectory prediction of individual (and yet unseen) flows (§ 3.4); (ii) a system design of CLAIRE for *commodity* DC switches (§ 3.5) confirming the feasibility of our approach in practice; and (iii) a preliminary evaluation (§ 3.6) of our approach showing its high accuracy and scalability compared to state-of-the-art coarse-grained and deep learning approaches. Related prior work is summarized in § 3.3. We draw conclusions and discuss future work in § 3.7.

3.2 Background

We provide here background information for our CLAIRE and its components.

3.2.1 Short-Time Fourier Transform

The short-time Fourier transform (STFT) is a method from the Fourier analysis to represent the temporal change of the frequency spectrum of a signal. While the Fourier transform

does not provide information about the temporal variation of the spectrum, the short-time Fourier transform (STFT) is also suitable for non-stationary signals whose frequency characteristics change over time.

For transformation, the time signal is subdivided into individual time segments with the aid of a window function, and these individual time segments are converted into individual spectral ranges. The temporal stringing together of the resulting spectral ranges represents the STFT.

There are two types of the short-time Fourier transformation, a *time-continuous* transformation and a *time-discrete* transformation, where the latter can be applied in digital signal processing and hence is relevant here.

For the time-discrete STFT the signal is present as a sequence of individual samples, which is subdivided into sections by a discrete window function. The time axis is generally expressed by an integer index n . The discrete STFT is given as

$$STFT\{x[n]\}(m, \omega) \equiv X(m, \omega) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-j\omega n} \quad (3.1)$$

with signal $x[n]$ and window $w[n]$. In applications, the calculation of the transformation is realized by a FFT.

An essential feature of the STFT transformation is the uncertainty principle. This relation describes a relationship between the resolution in the time domain and the resolution in the frequency domain, where the product of time and frequency represents a constant value. If the highest possible resolution in the time domain is desired, for example, to determine the time when a particular signal switches on or off, then this results in a blurred resolution in the frequency domain. If a high resolution in the frequency range is necessary to be able to determine the frequency accurately, then it follows a blur in the time domain, and the exact time when a particular signal switches on or off can only be determined vaguely. Therefore, the size of the time segments provided by the window function has to be adjusted according to the application.

3.2.2 *K*-Means Clustering

K-means [77] is a popular clustering technique for efficiently partitioning a set of observations into a specific number k of clusters. Each observation belongs to the cluster with the nearest mean that serves as prototype of the corresponding cluster. The problem can be formulated as

$$\operatorname{argmin}_S \sum_{i=1}^k \sum_{x_j \in S_i} \|x_j - \mu_i\|^2 \quad (3.2)$$

where (x_1, \dots, x_N) is the set of observations D , k the desired number of clusters, $S = \{S_1, \dots, S_k\}$ the cluster sets containing the data points such that $\bigcup_{i=1}^k S_i = D$ and $S_i \cap S_j = \emptyset$ for $i \neq j$. μ_i denotes the mean of observations in S_i .

Solving the problem exactly is NP-hard therefore k -means approaches the problem by iteratively refining a solution which finally converges to a local optimum. Given an initial set of cluster centers $S^0 = \{\mu_1^0, \dots, \mu_k^0\}$, which are typically randomly-chosen observations from D , the algorithm alternates between two steps:

1. **Assignment step:** Each observation is assigned to the closest cluster center yielding new cluster assignments:

$$S_i^t = \left\{ x_p \mid \|x_p - \mu_i^{t-1}\|^2 \leq \|x_p - \mu_j^{t-1}\|^2 \forall j \right\}, \forall i = 1, \dots, k \quad (3.3)$$

2. **Update step:** For each cluster a new mean is computed based on the data points from the assignment step:

$$\mu_i^t = \frac{1}{|S_i^t|} \sum_{x \in S_i^t} x, \forall i = 1, \dots, k \quad (3.4)$$

The steps are repeated until assignments no longer change.

The reason to use k -means clustering is the following: the clusters divide the huge hidden state of the system into multiple sub-spaces, where every sub-space has its corresponding system and observation models, which are used for further calculations and a fine-grained prediction of the kernel Kalman filter (KKF). Clustering reduces the kernel size and improves runtime performance. For a given flow, the input to the clustering is a histogram of the frequencies of a 10s s long history of that flow. The number of clusters depends highly on the variance, complexity, and diversity of the trajectories of the flows.

3.2.3 Hidden Markov Model

A Markov model (MM) can be used to model stochastic changing systems. The Markov model (MM) assumes that possible future states of a system are only dependent on its current state and not on events preceding the current state. This assumption is called the Markov property and enables reasoning and computation with the model that would otherwise be intractable. The underlying system emitting the time series is autonomous and the state of the whole system is just partially observable, since only the time series is observable. Therefore, a hidden Markov model (HMM) needs to be used to model the system.

Given a stochastic changing system consisting of two time-discrete stochastic processes $\{X_t\}_{t \in \mathbb{N}}$ and $\{Y_t\}_{t \in \mathbb{N}}$, of which only the latter is observable, the Markov Approach intends

to draw conclusions about the course of the first process by observing the second. In the particular case considered in this work $\{X_t\}_{t \in \mathbb{N}}$ would be the process of the underlying system and $\{Y_t\}_{t \in \mathbb{N}}$ would be the observable time series the system emits. This approach depends on the assumption of the two Markov properties:

1. Markov property: The current state of the first process depends solely on its last state:

$$\begin{aligned} \forall t \in \mathbb{N} : P(X_t = x_t | X_1 = x_1; \dots; X_{t-1} = x_{t-1}; X_t = x_t | \\ Y_1 = y_1; \dots; Y_{t-1} = y_{t-1}) = P(X_t = x_t | X_1 = x_1) \end{aligned} \quad (3.5)$$

2. Markov property: The current state of the second process depends solely on the last state of the first process:

$$\begin{aligned} \forall t \in \mathbb{N} : P(Y_t = y_t | X_1 = x_1; \dots; X_{t-1} = x_{t-1}; X_t = x_t | \\ Y_1 = y_1; \dots; Y_{t-1} = y_{t-1}) = P(Y_t = y_t | X_1 = x_1) \end{aligned} \quad (3.6)$$

A hidden Markov model (HMM) can now be defined as a 5-tuple $\lambda = (S; V; A; B; \pi)$, where $S = \{s_1; \dots; s_n\}$ denotes the set of all states, meaning all values the random variable X_t can assume, and $V = \{v_1; \dots; v_n\}$ denotes the set of all observations, meaning all emission Y_t the system can output. $A \in \mathbb{R}^{n \times n}$ is the state transition matrix, where a_{ij} indicates the probability of the transition from state s_i to state s_j . $B \in \mathbb{R}^{n \times n}$ is the observation matrix, where $b_i(v_j)$ indicates the probability to observe v_j in state s_i . Lastly, $\pi \in \mathbb{R}^n$ denotes the distribution for the initial state, where $\pi_i = P(X_1 = s_i)$ indicates the probability that s_i is the initial state.

3.2.4 Kalman Filter

The Kalman filter (KF) is based on linear dynamical systems discretized in the time domain. It can be used for state estimation problems in Bayesian models with a continuous state space. The Kalman filter (KF) can be viewed as an extension to the HMM, where the Markov process over hidden variables (which take values in a continuous state space as opposed to a discrete state space) is a linear dynamical system, with a linear relationship among related variables and where all hidden and observed variables follow a Gaussian distribution.

Similar to the HMM the KF is used with a state transition model represented by state transition matrix \mathbf{F} and an observation model represented by observation matrix \mathbf{H} . Since

the system addressed in this paper is autonomous, the simplified system dynamics for the KF are given by:

$$x_k = \mathbf{F}_k x_{k-1} + w_k \quad (3.7)$$

$$z_k = \mathbf{H}_k x_k + v_k \quad (3.8)$$

The model assumes x_k is the true state of the system at time k and is evolved only from the previous state a $k - 1$, whereas z_k is the observation emitted from the system at time k . The process noise is denoted as w_k and the observation noise as v_k and both are assumed to be drawn from a zero mean multivariate normal distribution, with covariances \mathbf{Q}_k and \mathbf{R}_k respectively: $w_k \sim \mathcal{N}(0, \mathbf{Q}_k)$ and $v_k \sim \mathcal{N}(0, \mathbf{R}_k)$. \mathbf{F} is assumed to describe the dynamics of the system and is used to transition from the previous state to the next and \mathbf{H} describes how an emitted observation depends on the state of the system.

With the system dynamics defined, the model can now be used for filtering problems. In a filtering problem one wants to draw conclusions regarding the true hidden state of the system by observing its outputs. Since the sought-after true hidden state of the system is always normally distributed, due to the precondition that w and v are normally distributed, it can be described by a mean and a covariance. One solution to this problem is the estimation of the mean μ_k and the covariance Σ_k of the state $\hat{\mathbf{X}}_k \sim \mathcal{N}(\mu_k, \Sigma_k)$ with the aid of information extracted from the measurement series z_k, z_{k-1}, \dots, z_1 . The estimation of the state should be based on the all previous observations to minimize the error. For longer growing measurement series this requirement quickly results in a mathematical optimization problem that is computational intractable, due to the fact that for every new state estimate the whole measurement series has to re-evaluated. The underlying concept of the KF is to formulate the state estimate at point k as a linear combination of the previous estimate and the new observation z_k . This is possible since the estimate a point $k - 1$ contains the information of measurement series $z_{k-1}, z_{k-2}, \dots, z_1$ which facilitates an efficient computational solution of the problem.

The KF algorithm consists of two update rules. First, the prediction update where an a priori estimate of the state (mean) $\mu_{k|k-1}$ and the covariance $\Sigma_{k|k-1}$ is calculated, and second, the innovation update which utilizes the new emitted observation z_k and the Kalman gain Matrix $\hat{\mathbf{K}}_k$ to correct the a priori estimate of the state and the covariance to obtain an a posteriori estimate for the state μ_k and the covariance Σ_k .

Prediction update:

$$\mu_{k|k-1} = \mathbf{F}_{k-1} \mu_{k-1} \quad (3.9)$$

$$\Sigma_{k|k-1} = \mathbf{F}_{k-1} \Sigma_{k-1} \mathbf{F}_{k-1}^T + \mathbf{Q}_{k-1} \quad (3.10)$$

Innovation update:

$$\mu_k = \mu_{k|k-1} + \hat{\mathbf{K}}_k(z_k - \mathbf{H}_k\mu_{k|k-1}) \quad (3.11)$$

$$\Sigma_k = \Sigma_{k|k-1} - \Sigma_{k|k-1}\mathbf{H}_k^T\hat{\mathbf{K}}_k \quad (3.12)$$

With the Kalman gain:

$$\hat{\mathbf{K}}_k = \Sigma_{k|k-1}\mathbf{H}_k^T(\mathbf{H}_k\Sigma_{k|k-1}\mathbf{H}_k^T + \mathbf{R}_k)^{-1} \quad (3.13)$$

The Kalman gain can be seen as an assessment of how much of an impact the new observation z_k should have in the innovation update of μ_k . Innovations of measurements, which are afflicted by a greater uncertainty as their estimate, should have less of an impact in the correction as those, for which the opposite is the case.

Usually, prediction and innovation updates are alternated, but if observations are irregularly multiple prediction updates can be executed in a row before executing an innovation update. As stated previously, the KF assumes linear system dynamics which limits its usability substantially. Another, for the context of this work more promising, group of KF variants is the one of KKF's to which the CLAIRE belongs.

3.2.5 Kernel Trick

Kernel methods are a class of algorithms in the field of machine learning which are used for pattern analysis. Patterns found in a data sets can be utilized for a wide variety of tasks like clustering, classification, or regression analysis. Two well-known algorithms utilizing kernel methods are support vector machine (SVM) and Gaussian processes.

Kernels are functions used to implicitly map the used data from its original space into a possibly infinite-dimensional feature space, where a better separation of the data points is possible. This is referred to as the *kernel trick* since it allows the user to apply linear pattern analysis methods to non-linear separable data points.

Be X the input space, a function $k : X \times X \rightarrow \mathbb{R}$ is called kernel, if a inner product space $(F, \langle \cdot, \cdot \rangle)$ and a mapping function $\varphi : X \rightarrow F$ exist for which the following holds: $\forall x, y \in X : k(x, y) = \langle \varphi(x), \varphi(y) \rangle_F$. F is then called *feature space* and φ *feature mapping*.

In practice explicitly calculating an inner product in the feature space can become computational intractable, since φ could map into an infinite-dimensional feature space. That is why kernel functions are needed to implicitly calculate the inner product in the feature space without explicitly performing the mapping.

One example for a function used in feature mapping is the radial basis function (RBF)

$$\phi(x) = \exp\left(-\frac{\|x - c\|^2}{2\sigma^2}\right) \quad (3.14)$$

Where x denotes a data point in the original space and $\phi(x)$ its mapping to one dimension of the feature space. The characteristics of the RBF are defined by its bandwidth σ and its center c . If φ consists of d ϕ 's with different characteristics, $\varphi(x)$ can map any $x \in X$ to a d -dimensional feature space. The calculation of an inner product of two data points explicitly mapped into the feature space $\varphi(x)^T \varphi(y) = [\phi_1(x), \phi_2(x), \dots, \phi_d(x)] \cdot [\phi_1(y), \phi_2(y), \dots, \phi_d(y)]^T$, would be computationally very expensive and only possible for a feature space with a finite number of dimension d . Therefore, the RBF-Kernel can be defined to calculate this inner product without having to actually do the mapping:

$$k(x, y) = \langle \varphi(x), \varphi(y) \rangle \quad (3.15)$$

$$= \varphi(x)^T \varphi(y) \quad (3.16)$$

$$= \exp\left(-\frac{\|x - y\|^2}{2\sigma^2}\right) \quad (3.17)$$

In both cases a scalar is returned which can be seen as the similarity measure between the two data points. To calculate the similarity measures for a whole data set $X = \{x_1, x_2, \dots, x_n\}$ the kernel matrix $\mathbf{K}_{xx} \in \mathbb{R}^{n \times n}$, also called *Gram matrix*, is formulated:

$$\begin{aligned} \mathbf{K}_{xx} = \mathbf{\Upsilon}_x^T \mathbf{\Upsilon}_x &= \begin{bmatrix} \varphi(x_1)^T \\ \vdots \\ \varphi(x_n)^T \end{bmatrix} \times [\varphi(x_1) \quad \dots \quad \varphi(x_n)] = \\ &= \begin{bmatrix} \langle \varphi(x_1), \varphi(x_1) \rangle & \dots & \langle \varphi(x_1), \varphi(x_n) \rangle \\ \vdots & \ddots & \vdots \\ \langle \varphi(x_n), \varphi(x_1) \rangle & \dots & \langle \varphi(x_n), \varphi(x_n) \rangle \end{bmatrix} \end{aligned} \quad (3.18)$$

An explicit representation for φ is not necessary, since F is an inner product space and all valid kernel functions k satisfy *Mercer's condition* [64].

3.2.6 Reproducing Kernels

The inner products of the Gram matrix of any kernel function $k(x, y)$ on $X \times X$ for some data set $X = \{x_1, x_2, \dots, x_n\}$ that satisfies Mercer's condition form an inner product space

H . If H contains the inner product of all possible data point pairs, H is complete and called a *Hilbert space*. It can also be seen as a set of functions defined on the data set X and each element can be seen as a function of the form $f(x) = k(x, \cdot)$ producing the inner product and is approximated with the equation

$$f(x) = \sum_{i=1}^n k(x, y_i) c_i, \quad (3.19)$$

which represents a weighted sum of kernel evaluations with the weights $c_i \in \mathbb{R}$.

A function k is called a *reproducing kernel* of such a Hilbert space if the following two conditions are satisfied:

- (1) $k(x, y) \in H$ for any $y \in X$
- (2) $\langle f(\cdot), k(\cdot, y) \rangle = f(y)$ for all $f \in H$ (reproducing property)

By definition of H Equation 3.5 is satisfied since H contains the inner product of all possible data point pairs $k(x, y)$. It can now be shown with the help of Equation 3.19 that

$$\begin{aligned} \langle f(x), k(x, y) \rangle &= \left\langle \sum_{i=1}^n k(x, y_i) c_i, k(x, y) \right\rangle \\ &= \sum_{i=1}^n k(y, y_i) c_i \\ &= f(y), \quad \forall y \in X. \end{aligned} \quad (3.20)$$

Thus Equation 3.6 is also satisfied and H is a reproducing kernel Hilbert space (RKHS) and k is its reproducing kernel.

As stated by the Moore–Aronszajn theorem [16], every symmetric positive definite kernel possesses the reproducing property and its associated Hilbert space is a reproducing kernel Hilbert space (RKHS).

3.2.7 Embedding Distributions in a Reproducing Kernel Hilbert Space

One foundation for embedding the formulations of the KF in a RKHS is the capability of embedding probability densities. 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 § 3.2.5 and § 3.2.6. Usually, the underlying distribution is not known, but a set of samples from it. The embedding of the distribution,

also called *empirical mean embedding*, can then be estimated through a finite-sample average calculated as

$$\hat{\mu}_X = \frac{1}{m} \sum_{i=1}^m \varphi(x_i) = \frac{1}{m} \mathbf{\Upsilon}_x^T. \quad (3.21)$$

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 empirically estimated using a finite number of samples from both distributions with

$$\hat{C}_{XY} = \frac{1}{m} \sum_{i=1}^m \varphi(x_i) \otimes \varphi(y_i). \quad (3.22)$$

Another form of distribution embedding needed for the CLAIRe is the embedding of a condition density defined as $\mu_{Y|x} := \mathbb{E}_{Y|x}[\varphi(Y)]$. 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

$$\hat{C}_{Y|X} = \mathbf{\Upsilon}_y(\mathbf{K}_{xx} + \lambda \mathbf{I}_m)^{-1} \mathbf{\Upsilon}_x^T, \quad (3.23)$$

where $\mathbf{K}_{xx} = \mathbf{\Upsilon}_x^T \mathbf{\Upsilon}_x$ is the Gram matrix and $\mathbf{\Upsilon}_x = [\varphi(x_1), \dots, \varphi(x_m)]$ the feature mappings of all states, as already defined in Equation 3.18. 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 $p(x) = \int_Y p(x|y)p(y)dy = \mathbb{E}_Y[p(x|y)]$. When embedding the distribution $p(x)$ by using the embedding rule for conditional distributions we end up at

$$\begin{aligned}
\mu_X &= \mathbb{E}_X[\varphi(X)] \\
&= \mathbb{E}_Y \mathbb{E}_{X|Y}[\varphi(X)] \\
&= \mathbb{E}_Y[C_{X|Y}\varphi(Y)] \\
&= C_{X|Y}\mathbb{E}_Y[\varphi(Y)] \\
&= C_{X|Y}\mu_Y,
\end{aligned} \tag{3.24}$$

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:

$$\begin{aligned}
C_{XX} &= \mathbb{E}_X[\varphi(X) \otimes \varphi(X)] \\
&= \mathbb{E}_Y \mathbb{E}_{X|Y}[\varphi(X) \otimes \varphi(X)] \\
&= \mathbb{E}_Y[C_{XX|Y}\varphi(Y)] \\
&= C_{XX|Y}\mu_Y.
\end{aligned} \tag{3.25}$$

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

$$\begin{aligned}
C_{XY} &= \mathbb{E}_Y[\mathbb{E}_{X|Y}[\varphi(X)] \otimes \varphi(Y)] \\
&= C_{X|Y}\mathbb{E}_Y[\varphi(Y) \otimes \varphi(Y)] \\
&= C_{X|Y}C_{YY},
\end{aligned} \tag{3.26}$$

where 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

$$\begin{aligned}
\mu_{Y|x} &= C_{Y|X}\varphi(X) \\
&= C_{XY}C_{XX}^{-1}\varphi(X) \\
&= \frac{(C_{X|Y}C_{YY})^T\varphi(X)}{C_{(XX)|Y}\mu_Y}.
\end{aligned} \tag{3.27}$$

3.3 Related Work

Many recent works predict specific processes in the broader context of networks. We focus on works closest related to prediction of network flow trajectories, and our approach.

Network flow evolution prediction. The general topic of network traffic prediction was already subject of several research works [100]. One of the first approaches to model a time series of traffic data is the autoregressive moving average (ARMA) model. ARMA 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; it was assessed for network traffic prediction mostly with single applications, e.g., BitTorrent [84], FTP [83]. As ARMA is only applicable for time series produced by stationary stochastic processes, the autoregressive integrated moving average (ARIMA) model was developed. ARIMA and variants of it were used for traffic prediction in, e.g., 3G mobile [184], or public safety networks [177]. Like ARMA, ARIMA is only applicable for linear time series with constant variance. generalized autoregressive conditional heteroskedasticity (GARCH) [7] captures non-linear time series with a time-dependent variance. The authors show that the model performs better than ARIMA in capturing the bursty nature of Internet traffic whose variance changes over time.

Non-linear time series can also be modeled by neural networks (NNs), and many different types of NNs were thus explored for predicting network traffic. E.g., Park and Woo [145] apply dynamic bilinear recurrent neural networks (BLRNNs), showing superior performance compared to static BLRNNs or classic NNs like the multilayer perceptrons (MLPs) previously also used for prediction [31, 43]. Other works combine NNs with linear approaches like ARIMA under the assumption that a traffic time series consists of linear and non-linear components [17]. Other approaches connect NNs with further methodologies, e.g., with fuzzy systems in the adaptive neuro-fuzzy inference system (ANFIS) [30]. SVMs were also used for predicting network traffic, e.g., by Liang et al. [121] using an ant colony optimization algorithm. A genetic algorithm is used in [21] to find the best combination of functions to approximate time series accounting for TCP throughput. However, none of the above or related approaches are fine-grained – the used *data was always aggregated* in two ways (cf. [INDIV]):

Time: The data set consisted of traffic data collected during a time span of days, weeks, or even months, aggregated *on a temporal scale* leading to sampling intervals for the traffic load between 1s and 1h. Only [37] used an interval of 100ms. Aggregated traffic data becomes less complex and much easier to learn. However, as the characteristics of individual flows are not present anymore, they cannot be predicted. This is problematic

as already rather short traffic bursts with very high loads can cause congestion [24]. In order to predict *individual* flows, small sampling intervals are needed as otherwise short high-volume flows are represented by only few data points ([RESOL]; cf. Fig. 3.1).

Space: The selected data was also aggregated *in terms of flows*, i.e., all socket-to-socket connections either between same hosts or same network elements were combined to one big flow, even across different protocols (e.g., TCP, user datagram protocol (UDP)). Again, this hides the bursty characteristics of the traffic and leads to the assumption that peaks are very rare.

Aforementioned works also split flows into training and testing parts, learning the prediction model for a flow entirely from the flow itself; this is very difficult for short high-volume flows. Our evaluation in § 3.6 shows how these limitations yield poor prediction accuracy. Other most recent work is coarse-grained in that it only predicts *total* flow sizes [54].

Several traffic engineering works also touch upon prediction. MicroTE [23] distinguishes between unpredictable and predictable traffic, identifying the former by comparing differences between past measured traffic matrices to thresholds. No actual predictions are made. Similarly, Pathbooks [114] leaves actual prediction for future work. Lastly, Valadarsky et al. [172] do not attempt to predict actual traffic demands but an efficient (coarse-grained) routing strategy.

Techniques related to CLAIRE . Related techniques are the (1) time series KKF (TS-KKF) [151] and (2) KKF based on the conditional embedding operator (KKF-CEO) [195]. (1) is a variant of the KF where observations, system states, and update equations are brought to a feature space by using a kernel function. However, in contrast to CLAIRE, a kernelized version of the system model is only derived for the transition model but not the observation model. Thus observations are computed from the states simply by adding a noise term which can lead to wrong assumptions about the covariance of the prediction (cf. [ACCUR]). Another difference to CLAIRE is that KF formulations are only embedded in a *sub-space* of the feature space and hence, the approach is not fully exploiting the infinite-dimensionality nature of the feature space. (2) embeds the formulations of the KF in a RKHS by using the conditional embedding operator explained in more detail along with the RKHS in § 3.4.4. In contrast to (1), (2) formulates the KF in the full feature space provided by the kernel. Moreover, the transition model does not have to be learned using the expectation-maximization (EM) algorithm, as with (1), but can be computed from training data. However, as for (1), the observation model is not formulated in the RKHS and the observations again are 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 standpoint as the observations were not generated by a Markov process, and leads to update equations that are far away from the original KF equations compared to CLAIRE.

3.4 CLAIRE Design Overview

This section presents our solution to fine-grained network flow trajectory prediction. We define the trajectory of a network flow as a time series of kbit values representing data quantities transmitted per sampling period Δ_S (cf. Tab. 3.2). The smaller the sampling period, the more detailed flow characteristic with its variance can be observed. Fig. 3.1 gives an intuition of the impact on a smaller sampling period. It shows two trajectories with different sampling periods.

3.4.1 Technical Challenges

Applications, end-host control logic, and network behavior influence each other and represent a complex system, whose true state we cannot possibly learn. Flow trajectories can be seen as high-dimensional observations Ω_I of this non-linear system with hidden states and unknown dynamics. Thus, once we are able to predict the real (hidden) state in a fine-grained manner we can achieve the goals of § 3.1. We identified several challenges of such prediction, not all of which are orthogonal, some even conflicting:

- (1) Reduce input data (features) for *timely* prediction (~ 10 ms).
- (2) Represent input data in a way capturing the system state and its evolution, including *non-linear bursty* behavior.
- (3) Reduce system state space to work with *limited resources*.
- (4) Predict flows based on the learned past of *other* flows.

3.4.2 Design Overview

In short CLAIRE uses Kalman filtering for state estimation and further prediction. However, as opposed to standard KFs, which only work in linear systems, CLAIRE uses a hidden finite state representation that is embedded in a reproducing kernel Hilbert space (RKHS). This space represents a non-linear transformation of the original state space into a possibly infinite-dimensional kernel space. While the system and observation models for the systems

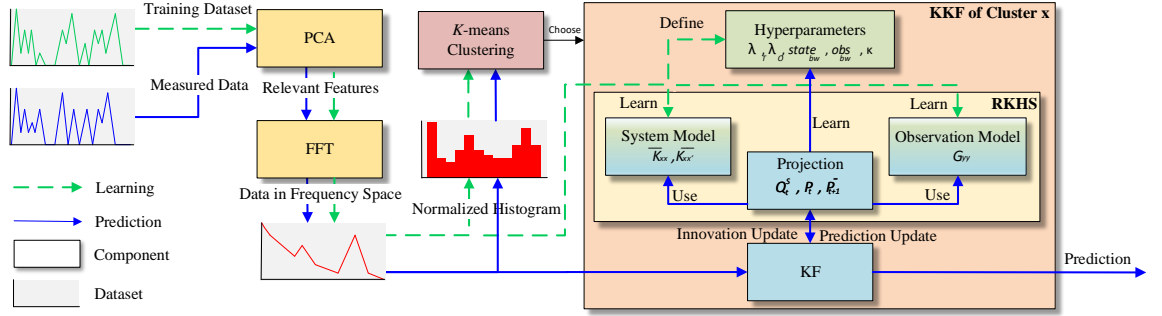


Figure 3.3: CLAIRE overview. Training data and measured data are preprocessed (yellow boxes) by transposition to frequency space and complexity reduction via PCA. Subsequently the (1) training data and (2) measured data are handled differently. (1) is used to optimize the hyperparameters and train the system and the observation models (green or green shaded boxes). (2) is used to make an innovation and prediction update and project back to original state space (blue/blue shaded boxes).

of interest are typically non-linear in the original state space, they can be approximated efficiently in the finite RKHS by linear models [67]. In addition, CLAIRE transforms its observations into the frequency domain, which emphasizes the difference between the observations, especially in the high-frequency domain, which influences the characteristics of the trajectory in time-space massively. Clusters of flows depending on the histogram of the frequencies with own learned models split the complexity for CLAIRE.

In more detail CLAIRE, combines, inherits from, and augments several known techniques, and can be viewed as addressing the above challenges (1)-(4) in a highly simplified manner as follows:

FFT transforms the input time series to frequency space [179], as bursty traffic often looks arbitrary in time space but has noticeable characteristics in the frequency space (2),(3).

PCA reduces dimensionality of our observation space and thus the input vector size without losing entropy [96] (1).

KF performs outlier-resistant Bayesian state estimation for linear models with a continuous state space (4).

RKHS (or reproducing kernel Hilbert space) transforms the KF to be able to estimate states of non-linear models (2).

Subspace KKF reduces the size of a kernel matrix for the RKHS making model representation more compact (3).

In Fig. 4.1 the green dashed lines show the learning phase for a given training data set. First, a defined interval of single observations in a given data sampling period (cf. Tab. 3.2) is transposed into frequency space. Next, the frequency components of the FFT are transformed to a normalized histogram and use the k -means clustering to find the best fitting center. Every center has its own KKF models (system model, observation model) which are trained after using the PCA to extract relevant features. In the learning phase, several hyperparameters (see Tab. 3.1) are *a priori* defined to regularize the system and observation models, and to scale the bandwidth of the kernel function. Solid blue lines show the prediction phase where the measured data (observation) is also transformed into frequency space. As in the training phase, a normalized histogram is created from the frequency components of the FFT and k -means is used to find the best fitting center. After filtering the relevant features via PCA, the learned KKF models are used for the innovation and prediction updates of the KF. Since the system and observation models are linear in the RKHS, a projection between the RKHS and the original space is needed.

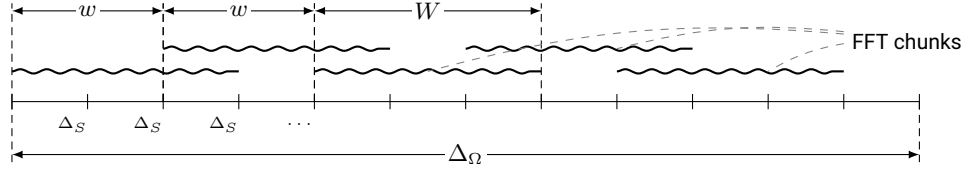
Next we elaborate on feature extraction (§ 3.4.3) and the KKF formulation (§ 3.4.4) which uses a sparse representation of the RKHS to maintain [SCALE].

3.4.3 Preprocessing

The preprocessing (or feature extraction) takes place during both the learning and the prediction phases, and includes two steps: FFT and PCA. Since the details on PCA, and FFT already discussed in § 3.2 they are abstracted out of the mathematics underlying KKF.

The raw traffic time series is represented by observations $\Omega_I \in \mathbb{R}^{n_\Omega}$ over respective intervals I of length Δ_Ω (e.g., $[\tau_1^a, \tau_1^b]$ in Fig. 3.2). Each Ω_I has $n_\Omega = \lfloor \frac{\Delta_\Omega}{\Delta_S} \rfloor$ components, and j th component is equal to the amount of data (in kbits) transmitted during $[\tau + (j - 1) \cdot \Delta_S, +\Delta_S]$ for $I = [\tau, +\Delta_\Omega]$, where $[\tau, +x]$ is a shorthand for $[\tau, \tau + x]$

FFT. We first transform the observation $\Omega = \Omega_I$ to the frequency domain by means of the FFT so that the flow structure is more understandable for the KKF algorithm. Nonetheless, we do not want to lose time information entirely, so first we split the observation of length n_Ω into smaller chunks of length W . The starting points of the single chunks at which they are sampled from the original signal are defined by w :



Parameters Δ_S , w , and W thus decide how many chunks the original signal is split into and whether—and if so by how much—they overlap. The selection of an appropriate overlap length can reduce artifacts in the frequency domain. The FFT step produces $\Omega^{\text{FT}} \in \mathbb{R}^{n_{\text{FT}} \times n_C}$ representing $n_{\text{FT}} = \lfloor \frac{n_{\Omega} - W}{w} \rfloor$ chunks, each having $n_C \approx \frac{2s}{\Delta_S}$ frequency components.

PCA. To reduce complexity and save computation time for better [SCALE], the PCA reduces the dimensions for every Ω^{FT} , yet in frequency space producing $\Omega^{\text{PCA}} \in \mathbb{R}^{n_{\text{PCA}}}$. The PCA uses an orthogonal transformation to compute the principal components of the data set that are ordered in such a way that first components explain more variance of the data set than the later. The assumption is that some frequencies might account for such a small amount of variance that omitting them does not noticeably distort the flow’s characteristics. Note, PCA is conducted on the training set, and computed principal components are later used for learning.

3.4.4 Adapting the Kalman Filter

Kalman filtering is a Bayesian inference techniques that aims at predicting the behavior of partially-observable systems. Bayesian inference operates not with the concrete predicted states and observations, but rather with their beliefs, which implies a high level of noise robustness. We assume to receive an observation $\vec{y}_t = \Omega_{I_{y(t)}}^{\text{PCA}}$ from the system at *logical* time t whose true hidden state is $\vec{x}_t = \Omega_{I_{x(t)}}^{\text{PCA}}$. The real time difference between t and $t + 1$ is the *look-ahead interval* Δ_{LA} with length equal to Δ_P plus lead time plus compute time, e.g., $\Delta_{LA} = \tau_k^e - \tau_i^b$ in Fig. 3.2. Thus, the observation intervals are $I_{y(t)} = [\tau_t^a, \tau_t^b] = [\tau_t^a, +\Delta\Omega]$ and $I_{x(t)} = [\tau_{t-1}^b, +\Delta\Omega]$, where $\tau_t^a = (t - 1)\Delta_{LA}$ as in Fig. 3.2. The way the intervals overlap may seem counterintuitive, but it provides tighter coupling between successive states making the transition model more robust.

Kalman filtering. The probabilistic model captures: *prior* belief $p(\vec{x}_{t+1}|\vec{y}_{1:t}) \sim \mathcal{D}_x(\theta_{x,t+1}^{\text{pri}})$ over current system state conditioned on previous observations; belief $p(\vec{y}_{t+1}|\vec{x}_{t+1}) \sim \mathcal{D}_y(\theta_{y,t+1}^{\text{obs}})$ over an *observation* conditioned on the system state; and *posterior* belief

$p(\vec{x}_{t+1}|\vec{y}_{1:t}, \vec{y}_{t+1}) \sim \mathcal{D}_x(\theta_{x,t+1}^{pos})$ over the current system state incorporating a new observation. \mathcal{D}_x and \mathcal{D}_y are fixed parametric distributions. The dynamic behavior is governed by a transition function $\mathbb{F}_{x'|x}$ in a *prediction update* $\theta_{x,t+1}^{pri} = \mathbb{F}_{x'|x}(\theta_{x,t}^{pos})$, while the observation function $\mathbb{F}_{y|x}$ links the system state with the observation, $\theta_{y,t}^{obs} = \mathbb{F}_{y|x}(\theta_{x,t}^{pri})$. The essence of Kalman filtering lies in its ability to refine the belief over a current state \vec{x}_t by incorporating a current observation y_t with an *innovation update* that accounts for the uncertainty of \vec{x}_t and the noise of y_t : $\theta_{x,t+1}^{pos} = \text{KF}(\theta_{x,t+1}^{pri}, y_t; \mathbb{F}_{y|x})$; KF is a fixed function.

The major limitation of plain Kalman filtering is that the system's behavior, i.e. $\mathbb{F}_{x'|x}$ and $\mathbb{F}_{y|x}$, must be linear, making it hard to capture complex behavior needed for [ACCUR].

Kernel Kalman filtering. Machine learning routinely applies non-linear feature mapping to addresses complex behaviors, i.e., replacing \vec{x}_t with $\vec{\varphi}_t = \vec{\varphi}(\vec{x}_t)$ and y_t with $\vec{\phi}_t = \vec{\phi}(y_t)$, where $\vec{\varphi}$ and $\vec{\phi}$ are non-linear functions. If both φ and ϕ are embeddings into \mathbb{R}^d , then the entire Kalman filtering machinery can be transferred to a non-linear setting, producing prior belief $\mathcal{D}_\varphi(\theta_{\varphi,t+1}^{pri})$, observation belief $\mathcal{D}_\phi(\theta_{\phi,t}^{obs})$, and the posterior belief $\mathcal{D}_\varphi(\theta_{\varphi,t+1}^{pos})$, while keeping both the transition function $\mathbb{F}_{\varphi'|\varphi}$ and observation function $\mathbb{F}_{\phi|\varphi}$ linear. The caveat is that now, $\mathcal{D}_\varphi(\theta_{\varphi,t+1}^{pos})$ is not a belief over hidden state, but rather over an embedding, and we need a new *state reconstruction* function $\mathbb{F}_{x|\varphi}$ to get back the original state belief: $\theta_{x,t}^{pos} = \mathbb{F}_{x|\varphi}(\theta_{\varphi,t}^{pos})$.

Given a potentially higher dimension of $\vec{\phi}$ and $\vec{\varphi}$, it is crucial for [SCALE] that KKF does not actually compute $\vec{\varphi}(\vec{x})$ and $\vec{\phi}(\vec{y})$ explicitly, it only calculates the respective dot products: $\vec{\varphi}(\vec{x})^T \vec{\varphi}(\vec{x}') = k_\varphi(x, x')$ and $\vec{\phi}(\vec{y})^T \vec{\phi}(\vec{y}') = k_\phi(y, y')$ for all train vectors \vec{x} and \vec{y} ; k_φ and k_ϕ are *kernels*, hence the name.

Learning KKF model from finite samples. Naturally, before we can actually apply KKF for prediction, we must *learn* the system parameters represented by the three function $\mathbb{F}_{\varphi'|\varphi}$, $\mathbb{F}_{\phi|\varphi}$, and $\mathbb{F}_{x|\varphi}$, as well as the initial parameters for an *a priori* sub-space embedding $\theta_{\varphi,1}^{pri}$. First, we choose different settings for the execution of the learning, including the training set and the values of several hyperparameters through optimization in the learning phase. The training set consists of (y, \vec{x}, \vec{x}') triples, where $y = \Omega_{[\tau, +\Delta_\Omega]}$, $x = \Omega_{[\tau + \Delta_\Omega - \Delta_{LA}, +\Delta_\Omega]}$, $x' = \Omega_{[\tau + \Delta_\Omega, +\Delta_\Omega]}$, and $\tau \in \mathbb{R}$.

KF updates.

As mentioned above, the KF equations are composed of two updates – (i) the *prediction update*, which maps the current belief state to the next time step, and (ii) the *innovation*

update, which incorporates the current observation in our belief state. The specific distributions KF is based upon are multi-variate normal distributions denoted by \mathcal{N} :

$$\begin{aligned}\mathcal{D}_x(\theta_{x,t}^{pos}) &\equiv \mathcal{N}(x; \vec{\mu}_{x,t}^{pos}, \vec{\Sigma}_{x,t}^{pos}) & \mathcal{D}_\varphi(\theta_{\varphi,t}^{pri}) &\equiv \mathcal{N}(\varphi_t; \vec{\mu}_{\varphi,t}^{pri}, \vec{\Sigma}_{\varphi,t}^{pri}) & \mathcal{D}_\varphi(\theta_{\varphi,t}^{pos}) &\equiv \mathcal{N}(\varphi_t; \vec{\mu}_{\varphi,t}^{pos}, \vec{\Sigma}_{\varphi,t}^{pos}) \\ \mathcal{D}_\phi(\theta_{\phi,t}^{obs}) &\equiv \mathcal{N}(\phi; \mu_{\phi,t}^{obs}, \sigma_{\phi,t}^{obs})\end{aligned}$$

Prediction update. We will start by embedding the prediction update (i) in . In Hilbert space, the mapping of the *a posteriori* mean embedding $\vec{\mu}_{\varphi_t}^{pos}$ of time point t to the prior mean embedding of the next state $\vec{\mu}_{\varphi_{t+1}}^{pri}$ (the hyphen denotes that the mean is an *a priori* belief) is given by a conditional operator $\vec{\mathcal{C}}_{\varphi'|\varphi}$. As our model is unknown, $\vec{\mathcal{C}}_{\varphi'|\varphi}$ needs to be computed using a sample-based estimator. This can be achieved as

$$\vec{\mathcal{C}}_{\varphi'|\varphi} = \vec{\Upsilon}_{x'} (\vec{K}_{xx} + \lambda_T \vec{I}_m)^{-1} \vec{\Upsilon}_x^T, \quad (3.28)$$

where $\vec{K}_{xx} = \vec{\Upsilon}_x^T \vec{\Upsilon}_x$. The matrix $\vec{\Upsilon}_x = [\varphi(x_1), \dots, \varphi(x_{m-1})]$ contains the feature mappings of all states and the matrix $\vec{\Upsilon}_{x'} = [\varphi(x_2), \dots, \varphi(x_m)]$ the mappings of all subsequent states. Thus the conditional operator is able to map the mean embeddings of the current state to the embedding of the next state. Parameter λ_T is used to regularize the observation model (Gram matrix). The prediction update equations of the traditional KF can then be reformulated in a RKHS as

$$\langle \vec{\mu}_{\varphi_{t+1}}^{pri}, \vec{\Sigma}_{\varphi_{t+1}}^{pri} \rangle = \mathbb{F}_{\varphi'|\varphi}(\vec{\mu}_{\varphi_t}^{pos}, \vec{\Sigma}_{\varphi_t}^{pos}) \stackrel{\text{def}}{=} \langle \vec{\mathcal{C}}_{\varphi'|\varphi} \vec{\mu}_{\varphi_t}^{pos}, \vec{\mathcal{C}}_{\varphi'|\varphi} \vec{\Sigma}_{\varphi_t}^{pos} \vec{\mathcal{C}}_{\varphi'|\varphi}^T + \vec{\Upsilon}_{x'} \vec{V} \vec{\Upsilon}_{x'}^T \rangle \quad (3.29)$$

where $\vec{\Upsilon}_{x'} \vec{V} \vec{\Upsilon}_{x'}^T$ is the covariance of the zero-mean Gaussian noise of the transition model which is also learned from the training data.

Innovation update.

For the innovation update (ii), we define an observation operator $\vec{\mathcal{C}}_{\phi|\varphi}$ which maps the state embedding to the observation embedding and therefore, represents the CLAIRE equivalent of the observation matrix in the original KF formulation. The observation operator is estimated using the training data with $\vec{\mathcal{C}}_{\phi|\varphi} = \vec{\Phi}_y (\vec{K}_{xx} + \lambda_O \vec{I}_m)^{-1} \vec{\Upsilon}_x^T$, where $\vec{\Phi}_y = [\phi(y_1), \dots, \phi(y_m)]$, and λ_O is a regularization parameter, getting:

$$\langle \mu_{\phi_t}^{obs}, \sigma_{\phi_t}^{obs} \rangle = \mathbb{F}_{\phi|\varphi}(\vec{\mu}_{\varphi_t}^{pri}, \vec{\Sigma}_{\varphi_t}^{pri}) \stackrel{\text{def}}{=} \langle \vec{\mathcal{C}}_{\phi|\varphi} \vec{\mu}_{\varphi_t}^{pri}, \nu \rangle, \quad (3.30)$$

where ν is the variance of zero-mean Gaussian noise. 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 compute the Kalman gain, which represents the relative importance of the error with respect to the prior belief state estimation. Using the Kalman gain, we finally arrive at the *a posteriori* belief over the state. The Hilbert space equivalent of the innovation update equations are

$$\langle \vec{\mu}_{\varphi_t}^{pos}, \vec{\Sigma}_{\varphi_t}^{pos} \rangle = \text{KF}(\vec{\mu}_{\varphi_t}^{pri}, \vec{\Sigma}_{\varphi_t}^{pri}, y_t, ; \vec{\mathcal{C}}_{\phi|\varphi}) \stackrel{\text{def}}{=} \langle \vec{\mu}_{\varphi_t}^{pri} + \vec{\mathcal{Q}}_t (\phi(y_t) - \vec{\mathcal{C}}_{\phi|\varphi} \vec{\mu}_{\varphi_t}^{pri}), \vec{\Sigma}_{\varphi_t}^{pri} - \vec{\mathcal{Q}}_t \vec{\mathcal{C}}_{\phi|\varphi} \vec{\Sigma}_{\varphi_t}^{pri} \rangle \quad (3.31)$$

where the Kalman gain matrix is computed by

$$\vec{\mathcal{Q}}_t = \vec{\Sigma}_{\varphi_t}^{pri} \vec{\mathcal{C}}_{\phi|\varphi}^T (\vec{\mathcal{C}}_{\phi|\varphi} \vec{\Sigma}_{\varphi_t}^{pri} \vec{\mathcal{C}}_{\phi|\varphi}^T + \kappa \vec{I}_m)^{-1}. \quad (3.32)$$

The zero-mean Gaussian noise of the observation model is estimated as $\kappa \vec{I}_m$.

State reconstruction. All computed means and covariances lie in the Hilbert space. Thus, 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 $\vec{\mathcal{C}}_{x|\varphi}$ is used. Similarly to the already defined conditional operators, it is computed using a sample-based estimator resulting in

$$\vec{\mathcal{C}}_{x|\varphi} = \vec{X} (\vec{K}_{xx} + \lambda_O \vec{I}_m)^{-1} \vec{\Upsilon}_x^T \quad (3.33)$$

with the hidden state matrix $\vec{X} = [x_1, \dots, x_m]$. The reconstruction is now conducted by applying the conditional operator to the mean and covariance embeddings, yielding

$$\langle \vec{\mu}_{x_t}^{pos}, \vec{\Sigma}_{x_t}^{pos} \rangle = \mathbb{F}_{x|\varphi}(\vec{\mu}_{\varphi_t}^{pos}, \vec{\Sigma}_{\varphi_t}^{pos}) \stackrel{\text{def}}{=} \langle \vec{\mathcal{C}}_{x|\varphi} \vec{\mu}_{\varphi_t}^{pos}, \vec{\mathcal{C}}_{x|\varphi} \vec{\Sigma}_{\varphi_t}^{pos} \vec{\mathcal{C}}_{x|\varphi}^T \rangle.$$

Finite-sample RKHS Embedding.

The CLAIRE embeds the state belief in a potentially infinite-dimensional Hilbert space using a non-linear feature map. In this space, non-linear inference can be performed by linear matrix operations as shown above.

As our models are unknown, $\vec{\mu}_{\varphi_t}^{pos}$ and $\vec{\Sigma}_{\varphi_t}^{pos}$ 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 $\vec{m}_t \in \mathbb{R}^{m \times 1}$ and the covariance by a finite-dimensional matrix $\vec{S}_t \in \mathbb{R}^{m \times m}$ through

$$\vec{\mu}_{\varphi_t}^{pos} = \vec{\Upsilon}_{x'} \vec{m}_t^{pos} \quad \vec{\Sigma}_{\varphi_t}^{pos} = \vec{\Upsilon}_{x'} \vec{S}_t^{pos} \vec{\Upsilon}_{x'}^T \quad \text{and} \quad \vec{\mu}_{\varphi_t}^{pri} = \vec{\Upsilon}_{x'} \vec{m}_t^{pri} \quad \vec{\Sigma}_{\varphi_t}^{pri} = \vec{\Upsilon}_{x'} \vec{S}_t^{pri} \vec{\Upsilon}_{x'}^T$$

When inserted into Equation 3.29, we receive a *finite-sample prediction update* formulation where the finite-dimensional *a priori* mean embedding is computed as

$$\vec{m}_{t+1}^{pri} = (\vec{\Upsilon}_{x'})^{-1} \vec{\mu}_{\varphi_{t+1}}^{pri} = \vec{C}_{\varphi'|\varphi} \vec{m}_t^{pos}.$$

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

$$\vec{S}_{t+1}^{pri} = \vec{\Upsilon}_{x'}^{-1} \vec{\Sigma}_{\varphi_{t+1}}^{pri} (\vec{\Upsilon}_{x'}^{-1})^T = \vec{C}_{\varphi'|\varphi} \vec{S}_t^{pri} \vec{C}_{\varphi'|\varphi}^T + \vec{V}.$$

As a next step, the equations for a *finite-sample innovation update* are introduced. The finite-dimensional Kalman gain matrix $\vec{Q}_t \in \mathbb{R}^{m \times m}$, defined from equation $\vec{Q}_t = \vec{\Upsilon}_{x'} \vec{Q}_t \vec{\Phi}_y^T$ is estimated by

$$\vec{Q}_t = \vec{S}_t^{pri} \vec{C}_{\phi|\varphi}^T (\vec{G}_{yy} \vec{C}_{\phi|\varphi} \vec{S}_t^{pri} \vec{C}_{\phi|\varphi}^T + \kappa \vec{I}_m)^{-1}$$

with $\vec{G}_{yy} = \vec{\Phi}_y^T \vec{\Phi}_y$ the Gram matrix of the embedded observations. The learned observation model of the underlying system is given by $\vec{G}_{yy} \vec{C}_{\phi|\varphi}$ with $\vec{C}_{\phi|\varphi} = (\vec{K}_{xx} + \lambda_O \vec{I}_m)^{-1} \vec{K}_{xx'}$. Using \vec{Q}_t , the finite-dimensional *a posteriori* mean embedding is derived as

$$\vec{m}_t^{pos} = \vec{m}_t^{pri} + \vec{Q}_t (\vec{k}_{:y_t} - \vec{G}_{yy} \vec{C}_{\phi|\varphi} \vec{m}_t^{pri}).$$

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

$$\vec{S}_t^{pos} = \vec{S}_t^{pri} - \vec{Q}_t \vec{G}_{yy} \vec{C}_{\phi|\varphi} \vec{S}_t^{pri}.$$

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 and covariance, the derivations are

$$\vec{\mu}_{x_t}^{pos} = \vec{X} \vec{C}_{x|\varphi} \vec{m}_t^{pos} \quad \vec{\Sigma}_{x_t}^{pos} = \vec{X} \vec{C}_{x|\varphi} \vec{S}_t^{pos} \vec{C}_{x|\varphi}^T \vec{X}^T.$$

Table 3.1: CLAIRE algorithm parameters and types: data, RKHS, prediction parameters, and hyper-parameters.

Parameter	Description	T
$data_{train}$	Training set	D
λ_T	Regulation parameter for transition model	
λ_O	Regulation parameter for observation model	H
κ	Observation noise covariance	
$\vec{K}_{xx}, \vec{K}_{xx'}$	Gram matrices of state embeddings	
\vec{G}_{yy}	Observation Gram matrix	
$\vec{C}_{\varphi' \varphi}^S$	Sub-space transition model matrix	
$\vec{C}_{\phi \varphi}^S$	Sub-space observation model matrix	R
\vec{X}	State matrix	
\vec{n}_t^{pos}	Sub-space mean embedding	
\vec{P}_t^{pos}	Covariance embedding	
\vec{Q}_t^S	Sub-space Kalman gain matrix	
$\vec{\mu}_{x_t}^{pos}$	Mean prediction	P
$\vec{\Sigma}_{x_t}^{pos}$	Covariance prediction	

Sub-space feature of CLAIRE.

The KKF possesses almost cubic computational complexity for the number of training samples due to the inversion of the Gram matrix $\vec{K}_{xx} \in \mathbb{R}^{m \times m}$ [67]. Thus, for large training sets the calculations become practically intractable which was the main reason for introducing k -means clustering discussed above. However, the presented CLAIRE 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, while still all samples are used to learn the model. To achieve this, a sub-space feature mapping $\vec{\Gamma}_x = [\varphi(x_1), \dots, \varphi(x_{n-1})]$ which contains the mappings of only $n \ll m$ training samples is defined, such that $\vec{\Gamma}_x \subset \vec{\Upsilon}_x$. The Gram matrix is then calculated as $\vec{K}_{xx} = \vec{\Upsilon}_x^T \vec{\Gamma}_x$ with dimensions $\vec{K}_{xx} \in \mathbb{R}^{m \times n}$ leading to new conditional embedding operators for the model learning which are called *sub-space conditional embedding operators*, introduced in [67].

The formulations for the *prediction update* of the sub-space CLAIRE approach stays the

same as for full-space with

$$\langle \vec{\mu}_{\varphi_{t+1}}^{pri}, \vec{\Sigma}_{\varphi_{t+1}}^{pri} \rangle = \mathbb{F}_{\varphi'|\varphi}^S(\vec{\mu}_{\varphi_t}^{pos}, \vec{\Sigma}_{\varphi_t}^{pos}) \stackrel{\text{def}}{=} \langle \vec{C}_{\varphi'|\varphi}^S \vec{\mu}_{\varphi_t}^{pos}, \vec{C}_{\varphi'|\varphi}^S \vec{\Sigma}_{\varphi_t}^{pos} (\vec{C}_{\varphi'|\varphi}^S)^T + \vec{\Upsilon}_{x'} \vec{V} \vec{\Upsilon}_{x'}^T \rangle \quad (3.34)$$

but the conditional operator now is $\mathcal{C}_{\varphi'|\varphi}^S = \vec{\Upsilon}_{x'} \vec{K}_{xx} \vec{L}_T^S \vec{\Gamma}_x^T$, where $\vec{L}_T^S = ((\vec{K}_{xx})^T \vec{K}_{xx} + \lambda_T \vec{I}_n)^{-1}$.

For the *innovation update* also the formulation of the sub-space approach stays the same as for full-space with the conditional operator mentioned before:

$$\langle \vec{\mu}_{\varphi_t}^{pos}, \vec{\Sigma}_{\varphi_t}^{pos} \rangle = \text{KF}^S(\vec{\mu}_{\varphi_t}^{pri}, \vec{\Sigma}_{\varphi_t}^{pri}, y_t, ; \vec{C}_{\phi|\varphi}^S) \stackrel{\text{def}}{=} \langle \vec{\mu}_{\varphi_t}^{pri} + \vec{Q}_t^S (\phi(y_t) - \vec{C}_{\phi|\varphi}^S \vec{\mu}_{\varphi_t}^{pri}), \vec{\Sigma}_{\varphi_t}^{pri} - \vec{Q}_t^S \vec{C}_{\phi|\varphi}^S \vec{\Sigma}_{\varphi_t}^{pri} \rangle \quad (3.35)$$

Moreover, a sub-space Kalman gain matrix is needed that build for using the sub-space conditional operator. It can be computed as

$$\vec{Q}_t^S = \vec{\Sigma}_{\varphi_t}^{pri} (\vec{C}_{\phi|\varphi}^S)^T (\vec{C}_{\phi|\varphi}^S \vec{\Sigma}_{\varphi_t}^{pri} (\vec{C}_{\phi|\varphi}^S)^T + \kappa \vec{I}_m)^{-1}$$

Again, for the reconstruction of the state distribution a third conditional operator is needed which is calculated by $\vec{C}_{x|\varphi}^S = \vec{X} \vec{K}_{xx} \vec{L}_T^S \vec{\Gamma}_x^T$ and utilized to map the feature embeddings back to the original space with the equations

$$\langle \vec{\mu}_{x_t}^{pos}, \vec{\Sigma}_{x_t}^{pos} \rangle = \mathbb{F}_{x|\varphi}^S(\vec{\mu}_{\varphi_t}^{pos}, \vec{\Sigma}_{\varphi_t}^{pos}) \stackrel{\text{def}}{=} \langle \vec{C}_{x|\varphi}^S \vec{\mu}_{\varphi_t}^{pos}, \vec{C}_{x|\varphi}^S \vec{\Sigma}_{\varphi_t}^{pos} (\vec{C}_{x|\varphi}^S)^T \rangle.$$

As for the full-space CLAIRE approach, 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 $\vec{n}_t^{pos} = \vec{\Gamma}_x \vec{\mu}_{\varphi_t}^{pos}$ and $\vec{P}_t^{pos} = \vec{\Gamma}_x^T \vec{\Sigma}_{\varphi_t}^{pos} \vec{\Gamma}_x$ whereas $\vec{n}_t^{pos} \in \mathbb{R}^{n \times 1}$ and $\vec{P}_t^{pos} \in \mathbb{R}^{n \times n}$. With this estimation we can formulate the finite-sample prediction update equations for the sub-space CLAIRE approach that allows for the computation of a finite-dimensional sub-space a priori mean embedding by $\vec{n}_{t+1}^{pri} = \vec{C}_{\varphi'|\varphi}^S \vec{n}_t^{pos}$ whereas $\vec{C}_{\varphi'|\varphi}^S = (\vec{K}_{xx})^T \vec{K}_{xx} \vec{L}_T^S$ the sub-space transition matrix. For the covariance embedding estimation we end up at

$$\vec{P}_{t+1}^{pri} = \vec{C}_{\varphi'|\varphi}^S \vec{P}_t^{pos} (\vec{C}_{\varphi'|\varphi}^S)^T + \vec{\Gamma}_x^T \vec{\Upsilon}_{x'} \vec{V} \vec{\Upsilon}_{x'}^T \vec{\Gamma}_x$$

For the finite-sample innovation update we start by defining a finite-dimensional sub-space Kalman gain matrix $\vec{Q}_t^S \in \mathbb{R}^{n \times n}$ which is estimated by

$$\vec{Q}_t^S = \vec{P}_t^{pri} \vec{L}_O^S ((\vec{K}_{xx})^T \vec{G}_{yy} \vec{C}_{\phi|\varphi}^S \vec{P}_t^{pri} \vec{L}_O^S + \kappa \vec{I}_n)^{-1} \vec{K}_{xx}^T$$

whereas $\vec{C}_{\phi|\varphi}^S = \vec{K}_{xx} \vec{L}_O^S$ is the observation matrix. The equations for the finite-dimensional sub-space *a posteriori* mean embedding are then given by

$$\vec{n}_t^{pos} = \vec{n}_t^{pri} + \vec{Q}_t^S (\vec{k}_{:y_t} - \vec{G}_{yy} \vec{K}_{xx} \vec{L}_O^S \vec{n}_t^{pri})$$

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

$$\vec{P}_t^{pos} = \vec{P}_t^{pri} - \vec{Q}_t^S \vec{G}_{yy} \vec{K}_{xx} \vec{L}_O^S \vec{P}_t^{pri}$$

The last step is the reconstruction of the state distribution. The derivation for the mean and for the covariance is given by

$$\vec{\mu}_{x_t}^{pos} = \vec{X} \vec{C}_{x|\varphi}^S \vec{n}_t^{pos} \quad \vec{\Sigma}_{x_t}^{pos} = \vec{X} \vec{C}_{x|\varphi}^S \vec{P}_t^{pos} \vec{X} (\vec{C}_{x|\varphi}^S)^T.$$

3.5 Prediction and Learning Implementation

We present our realization of CLAIRE presented in §3.4, shown as pseudocode in Alg. 2, for a better understanding of the code, Tab. 3.1 highlight the CLAIRE algorithm parameters with its types. It is important to mention, that the CLAIRE model can be learned offline so that observations are omitted from the underlying system and used, after preprocessing, in the innovation step only at prediction. Our current implementation in Python allows us to easily predict 1000 flows in parallel on commodity DC switches as we will show in §3.6.3.

Furthermore, we discuss CLAIRE's implementation on commodity data center switches.

3.5.1 Complexity

The KKF possesses almost $O(m^3)$ computational complexity for the number of training samples m (due to the inversion of the $m \times m$ Gram matrix [67]). Thus, for large training sets the calculations become practically intractable. For that reason, our CLAIRE uses a scalable variant of KKF that handles even large training data set. The core idea is to represent the model only by a subset S , $|S| \ll m$ of the training samples (still all samples are used to learn the model). The model uses new conditional embedding operators $\mathbb{F}_{\varphi'|\varphi}^S$, called *sub-space conditional embedding operators* [67].

This has benefits with respect to [SCALE] and [IMPL]. We identified several time-dependent parameters in the KKF equations [67] that can be precomputed before receiving any observations at all, i.e., before making predictions, which allowed to vastly reduce the computation time of the CLAIRE algorithm. The asymptotic complexity of prediction is

Algorithm 2 Core CLAIRE algorithm.

```

function PREPROCESS-DATA( $data$ )
   $chunk \leftarrow \text{FORM-STATE-CHUNK}(data)$ 
   $data \leftarrow \text{STFT}(chunk)$ 
  return  $data$ 

function LEARN( $data_{train}$ )
  while  $data$  in  $data_{train}$  do
    // STFT, PCA, and clustering
     $data \leftarrow \text{PREPROCESS-DATA}(data)$ 
     $\langle \lambda_T, \lambda_O, \kappa \rangle \leftarrow \text{OPTIMIZE-HYPERPARAMS}(data)$ 
     $\langle \vec{C}_{\varphi'|\varphi}^S, \vec{C}_{\phi|\varphi}^S, \vec{X}, \vec{n}_1^{pri}, \vec{P}_1^{pri} \rangle \leftarrow \text{ESTIMATE-MODEL}(\vec{K}_{xx}, \vec{K}_{xx'}, \vec{G}_{yy}, data)$ 
     $\langle \vec{Q}_t^S, \vec{P}_t^{pos}, \vec{P}_{t+1}^{pri} \rangle \leftarrow \text{PROJECT}(\lambda_T, \lambda_O, \kappa, \vec{C}_{\varphi'|\varphi}^S, \vec{C}_{\phi|\varphi}^S, \vec{X}, \vec{n}_1^{pri}, \vec{P}_1^{pri})$ 
  return  $\langle \lambda_T, \lambda_O, \kappa, \vec{C}_{\varphi'|\varphi}^S, \vec{C}_{\phi|\varphi}^S, \vec{X}, \vec{n}_1^{pri}, \vec{P}_1^{pri}, \vec{Q}_t^S, \vec{P}_t^{pos}, \vec{P}_{t+1}^{pri} \rangle$ 

function PROJECT( $learnedParams$ )
   $\vec{Q}_t^S \leftarrow \text{KALMAN-GAIN}(learnedParams)$ 
   $\vec{P}_t^{pos} \leftarrow \text{A-POSTERIORI-COVARIANCE-EMBED}(learnedParams)$ 
   $\vec{P}_{t+1}^{pri} \leftarrow \text{A-PRIORI-COVARIANCE-EMBED}(learnedParams)$ 
  return  $\langle \vec{Q}_t^S, \vec{P}_t^{pos}, \vec{P}_{t+1}^{pri} \rangle$ 

function PREDICT( $data, \vec{Q}_t^S, \vec{P}_t^{pos}, \vec{P}_{t+1}^{pri}$ )
  // STFT, PCA, and clustering
   $data \leftarrow \text{PREPROCESS-DATA}(data)$ 
  // Innovation update
   $\vec{n}_t^{pos} \leftarrow \text{A-POSTERIORI-MEAN-EMBEDDING}(data, \vec{Q}_t^S, \vec{P}_t^{pos}, \vec{P}_{t+1}^{pri})$ 
  // Prediction update
   $\vec{n}_{t+1}^{pri} \leftarrow \text{A-PRIORI-MEAN-EMBEDDING}(data, \vec{Q}_t^S, \vec{P}_t^{pos}, \vec{P}_{t+1}^{pri})$ 
  // Project into state space
   $\vec{\mu}_{x_t}^{pos} \leftarrow \text{MEAN-PREDICT}(data, \vec{n}_t^{pos}, \vec{n}_{t+1}^{pri}, \vec{Q}_t^S, \vec{P}_t^{pos}, \vec{P}_{t+1}^{pri})$ 
   $\vec{\Sigma}_{x_t}^{pos} \leftarrow \text{COVARIANCE-PREDICT}(data, \vec{n}_t^{pos}, \vec{n}_{t+1}^{pri}, \vec{Q}_t^S, \vec{P}_t^{pos}, \vec{P}_{t+1}^{pri})$ 
  return  $\langle \vec{\mu}_{x_t}^{pos}, \vec{\Sigma}_{x_t}^{pos} \rangle$ 

```

$O(n^2)$ where n depends on the number of samples, yet is *independent of the number of flows and of network elements*.

The last step in PREDICT for the innovation and prediction update is the calculation of the *a posteriori* and *a priori* sub-space mean embeddings \vec{n}_t^{pos} and \vec{n}_{t+1}^{pri} , 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.

3.5.2 Challenges in Monitoring and Management

The requirements for fine-grained prediction lead to two immediate consequences when implementing our algorithm:

- The period for sampling individual flows Δ_S has to be small enough to allow the algorithm to learn the fine-grained structure of flows (in our case 10 ms).
- Prediction and further steps have to be done “close” to the forwarding plane to minimize latency. Predicting more in advance is likely to compromise on accuracy, so it is more efficient to optimize the prediction time with respect to accuracy, and avoid additional latency between signal, prediction, and further steps (reaction).

For obtaining statistics from a switch’s ASIC in a fine-grained structure, we used FARM (cf. chapter 2) as the monitoring system on switches.

FARM (cf. chapter 2) was an enabler for CLAIRE as existing systems like sFlow [147] and IPFIX [42] are designed for fetching *raw* data from a switch and sending it to a centralized instance for analysis, which can not accommodate short lead time.

We developed CLAIRE seeds that allowed for complex processing of monitored data at high [RESOL] directly on the switch (cf. Fig. 2.6). The statistics from CLAIRE are not only used for online prediction of the flow directly on the switch, but also for offline re-learning of the flow structure. The prediction algorithm runs directly on the *management system* of the switch, allowing the prediction to be received as quickly as possible, without delay affecting [ACCUR]. The system consists of (a) a *monitoring instance* per flow, and (b) a single global harvester *learning instance* (see § 2.3.3). The monitored data from the ASIC to the corresponding monitoring/CLAIRE instances to predict the evolution of a given flow. CLAIRE runs on white-box switches and two different Linux-based OS: ONL [141] which is open source, and the extensible for [IMPL] Arista EOS [15] (see § 2.7). Once the learning process is completed or updated during execution, the global CLAIRE learning instance is able to update the prediction model of the CLAIRE instance running on the switch.

Table 3.2: Observation parameters and values used.

Parameter	Description	Value
Δ_{Ω}	Observation interval length	2 s
Δ_S	Data sampling period	10 ms
avg Δ_P	Average prediction interval length	500 ms
w	Chunk start period	50 ms / Δ_S
W	Chunk size	500 ms / Δ_S
n_{PCA}	Number of principal components	80

3.6 Evaluation

We assess CLAIRE in terms of its accuracy ([ACCUR]) and its overhead ([SCALE]). The parameters we used in all prediction experiments are summarized in Tab. 3.2.

3.6.1 Synopsis

To configure the parameters of CLAIRE for achieving its full potential, we take a look at the given data and its nature (§ 3.6.2). Subsets of the contained flows are selected as part of *either* training or test sets. The trade-off between complexity and entropy of the data is optimized to run the CLAIRE algorithm on commodity switching hardware, as explained in § 3.4.2 and § 3.4.3, without increasing prediction error (cf. [ACCUR]). We evaluate both prediction and runtime performance (§ 3.6.3). In considering all peaks, the prediction error of CLAIRE is calculated and compared against the following alternative approaches (cf. paragraph 3.3): 1. ARIMA [177, 184]; 2. GARCH [7]; 3. long short-term memory (LSTM) [167, 54]; 4. convolutional neural network (CNN) [108, 145, 31, 43]. The runtime performance measurements on commodity hardware switches (Accton AS5712, Arista 7280QRA-C36S) show how accurately the statistics can be polled and how many flows can be predicted (cf. [SCALE]) without specialized hardware ([IMPL]).

3.6.2 Experimental Data

We analyzed two different data sets with very different characteristics (e.g. number of peaks, peak height, flow duration, flow size, protocol types) to show generality:

■ < 0.1s
 ■ 0.1 - 1s
 ■ 1 - 10s
 ■ 10 - 100s
 ■ > 100s

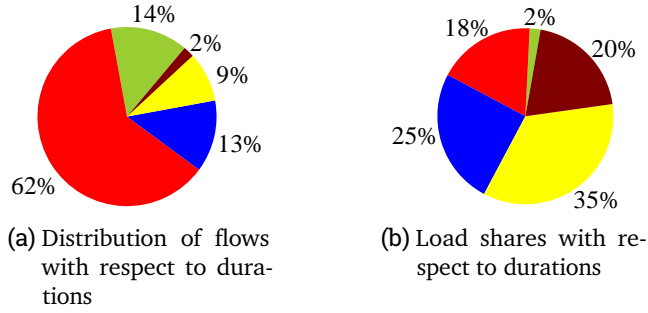


Figure 3.4: Characteristics of UniDCTraces flows.

■ < 5ms
 ■ < 0.1s
 ■ < 1s
 ■ < 10s
 ■ < 1m
 ■ < 5m
 ■ < 10m
■ < 15m
■ < 20m
■ < 25m
■ < 29m
■ < 32m
■ < 35m
■ < 29m

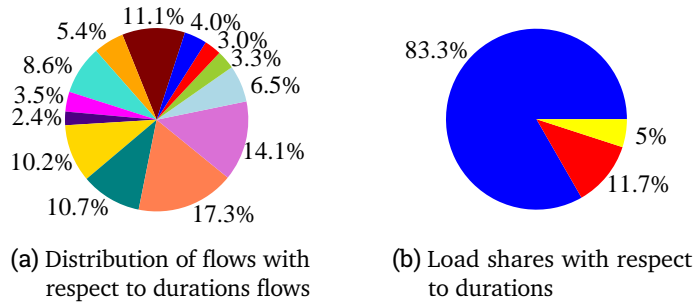


Figure 3.5: Characteristics of SAPDCTraces flows.

UniDCTraces: 1.2 GB collected as part of a study conducted by Benson et al. aimed at analyzing the characteristics of university DC network traffic [22]. During their study, simple network management protocol (SNMP) data, topology information, and packet traces were collected from a university DC for 65min worth of traffic at an edge switch in a DC consisting of 22 network devices and 500 servers.

SAPDCTraces: 20 GB of header data (application layer) within 32 842 flows from an SAP SEDC, with mainly intra-DC machine-to-machine (M2M) communication.

Compared to other data sets (e.g., from Facebook [188]) the chosen data sets contain information about topology without complete anonymization.

Composition. In a first step, the packet traces were analyzed by gathering statistical information about the composition of the traffic data. A flow represents a socket-to-socket connection between two hosts in the DC that starts with a TCP handshake and ends with a TCP teardown. For the UniDCTraces data sets, 87% of the traffic transmitted by the observed network switch, TCP was used as the transport layer protocol, and only 13% was transmitted over UDP. A closer look at the TCP traffic then revealed which protocols were used at the application layer. The hypertext transfer protocol (HTTP) traffic makes up the largest share with 75% of the TCP traffic. An analysis of the HTTP traffic data of the UniDCTraces data set constituting HTTP traffic shows that this traffic is highly human-triggered as the starting times for communication are completely arbitrary. The packet traces contain roughly 130 000 unique HTTP flows. A distribution of the durations of the HTTP flows and the corresponding loads are shown in Fig. 3.4.

In contrast the SAPDCTraces data set contains mostly M2M communication devoid of human involvement. Over 95% of the SAPDCTraces data set is TCP traffic, mainly for network file system protocols. The packet traces of the SAPDCTraces data set contain roughly 32 842 unique long-lasting M2M flows. A distribution of the durations of the SAPDCTraces data set flows and the corresponding loads are shown in Fig. 3.5. While the SAPDCTraces are much larger than the UniDCTraces (20 GB vs 1.2 GB of header information) they contain far fewer flows (32 842 vs 130 000). Even a larger portion of the data in SAPDCTraces is transmitted as part of elephant flows than in the UniDCTraces.

Flow durations. As shown in Fig. 3.4a and Fig. 3.5a, the durations across flows are highly diverse. Most of the flows of the UniDCTraces data set (62%) are very short with a length between 0.1s and 1s, as indicated by the red slice of the pie chart. To fully grasp the characteristics of the flows it is important to know which share of the traffic load is caused by which *duration group* (group of flows with similar durations). This information is given in Fig. 3.4b and together with Fig. 3.4a yields interesting insights. The extremely short flows with a duration of up to 0.1s cause only 2% of the traffic and can therefore be neglected. The same goes for many flows in the duration group of 0.1s to 1s, though not for all of them. The most important group seems to be the one containing flows with a rather short length between 1s and 100s, since only 22% of the flows possess such a length and still they cause 60% of the traffic. As mentioned in paragraph 3.3 these flows are impossible to capture by aggregating over time as done in prior work (cf. §3.6.3), since the flows would only be represented by one or a few data points (cf. [INDIV]). Just the

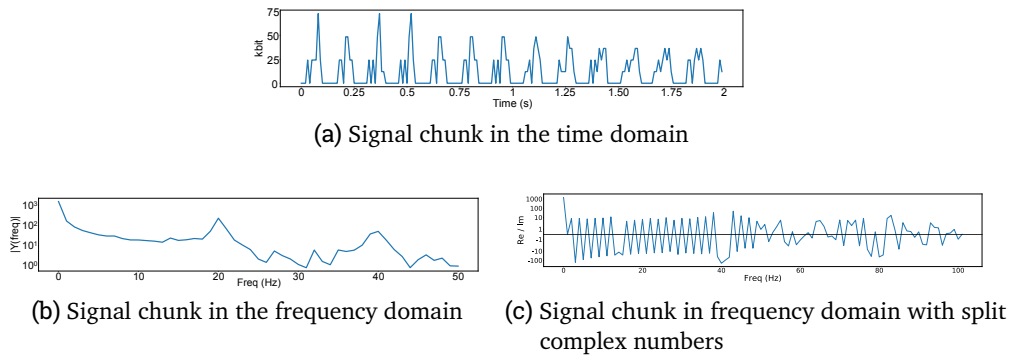


Figure 3.6: Data series in time vs frequency domain.

flows longer than 100s could be captured sufficiently (see § 3.6.3). As Fig. 3.4b indicates, this duration group is responsible for 20% of the traffic.

While the data set from SAP SE is much larger (20 GB to 1.2 GB - only header information) it contains much fewer flows (32 842 to 130 000). By analyzing the 32 842 flows the trend of elephant flows which is shown by the UniDCTraces data set is even more significant. 96% of the flows here are shorter than 35 min and still 93% of the flows are shorter than 32 min (see Fig. 3.5a). But the flows shorter than 32 min cause only 5% of the traffic and the flows shorter than 35 min cause 16.7% of the traffic. This means that 4% of the flows are responsible for 83.3% of the traffic respectively, and 7% of the flows are responsible for 95% of the traffic.

Burstiness. When examining the progression of all flows (high-volume vs. low-volume - long-lasting vs. short-lasting), a typical flow pattern is observed. One example flow is shown in Fig. 3.1; it clearly reveals the extremely bursty nature of the network traffic. Even though an already rather low sampling interval of 1s 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 are insignificant. This shows that even the long flows often just consist of short traffic peaks with long low-volume phases between them. For a TO system that tries to increase the utilization 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. This was already visualized in Fig. 3.1, which was detailing the first peak of the flow, with a

sampling interval of 0.01s vs 1s. Thus a peak itself consists of shorter peaks which we henceforth refer to as (*peak*) *impulses*. At the shortest possible sampling interval these impulses would represent single packets of the flows. In the vast majority of flows, the peaks consist of (1) a rising phase in which the peak impulses gradually increase to a certain maximum kbit value and (2) a peak body where the traffic load remains rather 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 potentially causes congestion. Thus, the goal is a successful prediction of the course of the peak rise given only a small part of it.

As shown in Fig. 3.6a from the UniDCTraces; it clearly reveals the extremely bursty nature of the network traffic, sampled with a period $\Delta_S = 10\text{ ms}$ over a prediction observation $\Delta_\Omega = 2\text{ s}$. Fig. 3.6b shows the less bursty data series as produced by the STFT during preprocessing (see § 3.4.3). 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. To use the STFT results with CLAIRE, the complex numbers have to be split into their real and imaginary parts. The resulting data series consisting alternately of real and imaginary values (see Fig. 3.6c) are sent to the PCA for dimension reduction (cf. Fig. 4.1).

Data Analysis. As shown in Fig. 3.1, the flow structure is extremely bursty when observed at a small sampling interval. As a consequence, the time 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 § 3.4 aggregate the flows on a temporal scale and either are not applicable for linear time series with variable variance (cf. ARIMA) and predict zeros or very low values, or start to oscillate arbitrarily (cf. GARCH) because they can not predict the complex hidden system state. Even state-of-the-art NNs like LSTMs [167] (a specialization of recurrent neural networks (RNNs)) and CNNs [108] (that can be expected to perform better than BLRNNs [145] and MLP [31, 43] used in the past), are unable to learn the complex peak structures and converge to the mean of the trajectory. As we will see in § 3.6.3 this would prevent a TO system from predicting short traffic peaks.

By learning the traffic signal in the frequency instead of the time domain the data sets that the KKF has to deal with becomes less bursty (see § 3.4.3). Fig. 3.6 compares the courses of different data series. In Fig. 3.6a a 2s long flow chunk is shown. When computing the Fourier transform (FT) of the chunk, a data series as shown in Fig. 3.6b 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, $f_N = 0.5 f_S = 0.5 \frac{1}{\Delta_S}$. For reconstructing the time signal from sine waves we therefore only need to save the first N values of the FT results, with $N = f_N + 1$. 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 CLAIRE, 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 Fig. 3.6c which clearly shows that the data series is considerably less bursty than the original time series.

We illustrate the effect of the transition to the frequency domain using an example. We assume that the training set that the KKF should learn from consists of only one short high-volume flow with a duration of 10s. When using a sampling interval $\Delta_S = 0.01$ our training set $data_{train}$ contains a time series of kbit values observed every 0.01s which possesses the dimensionality $data_{train} \in R^{1000 \times 1}$. As mentioned in § 3.4.3, the time signal is now split into single overlapping chunks. When using a sampling interval $\Delta_C = 0.05$ and a chunk length of $w = 1$ the signal is split into 200 unique chunks that all contain 100 data points, where 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 representing 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 $data_{train} \in R^{200 \times 102}$. As we can see, the ability to work with the traffic data needs to be paid with an increase of the observation dimensionality.

The number of dimensions by which the data sets can be reduced with a PCA to reduce the complexity of the training data set and save computation time differs between the clusters of recurrent flows. The third columns of Tab. 3.3 and Tab. 3.4 show the cumulated explained variance over the principal components. As we see in the tables, the influence of the reduction on the flow structure in the time domain when reducing the dimensionality from 102 to 80 is acceptable while reducing the computational complexity by 20%.

As mentioned in § 3.1, the training sets used during the experiments contain observations of kbit values which do not represent the true (hidden) state of the underlying system. Our training sets only contain observations, however, as noted in § 3.4.3, 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,

Table 3.3: UniDCTraces prediction experiments. While our CLAIRE achieves -9.17% prediction error on average over all flow clusters, ARIMA and GARCH exhibit at best(!) -77.3% and -95.2% prediction error respectively. NNs perform seemingly a bit better than those but can not predict trajectories.

GID	Group stats			Prediction error				
	N_f	$\sigma^2(\text{PCA})$	$W_{\text{opt.}}$	CLAIRE	ARIMA	GARCH	LSTM	CNN
1	4	95.9%	0.15 s	-2.0%	-96.3%	-97.9%	-71.19%	-15.41%
2	13	91.7%	0.4 s	-8.4%	-100%	-98.6%	-74.69%	-85.35%
3	6	98.7%	0.55 s	5.6%	-100%	-96.6%	-91.19%	-10.14%
4	24	94.9%	0.2 s	-22.8%	-99.9%	-98.2%	-89.64%	-10.04%
5	5	98.1%	0.2 s	-12.1%	-91.8%	-95.5%	-100%	-100%
6	6	90.6%	0.15 s	-13.2%	-97.7%	-97.5%	-100%	-100%
7	3	89.9%	0.25 s	2.5%	-94.1%	-97.4%	-34.7%	-43.25%
8	10	97.0%	0.3 s	-4.3%	-100%	-97.6%	-15.4%	-32.50%
9	4	99.5%	1.95 s	-4.1%	-99.9%	-95.7%	-100%	-100%
10	4	98.5%	0.5 s	-0.7%	-100%	-99.3%	-10.07%	-10.7%
11	5	99.8%	0.25 s	-0.8%	-99.9%	-99.0%	-100%	-100%
12	18	95.9%	0.95 s	-10.2%	-91.9%	4102.7%	-53.56%	-53.56%
13	5	94.3%	0.55 s	-1.6%	-91.7%	98.7%	-53.52%	-53.56%
14	3	98.1%	0.35 s	-10.2%	-99.7%	7490990.2%	-100%	-100%
15	11	90.6%	0.25 s	-17.0%	-97.7%	-98.9%	-100%	-100%
16	13	93.2%	0.55 s	-40.2%	-99.0%	-95.2%	-12.9%	-13.01%
17	3	91.4%	0.95 s	-45.4%	-90.8%	-99.3%	-50.02%	-46.56%
18	4	98.6%	0.45 s	4.7%	-95.8%	13550.9%	-100%	-100%
19	7	93.4%	0.15 s	-7.3%	-100%	603%	-100%	-100%
20	3	94.8%	0.95 s	2.0%	-77.3%	1533.6%	-28.94%	-100%
21	3	94.8%	0.95 s	2.0%	-77.3%	1533.6%	-100%	-32.07%
Avg:				-9.17%	- 90.05%	375552.30%	-70.75%	-60.67%

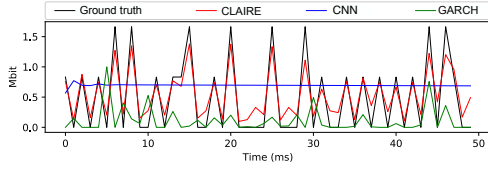
the test flow was formed using the FTs of multiple chunks with increasing length. One possibility would, e.g., be to use the FT of the next s and combine it with the FTs computed over the next 2 and 3 s . This allows us to include some more *long-term* behavior of the flow in the state representation. In such a scenario, an observation would be represented by the FT over the next second of the time signal.

3.6.3 Prediction Results

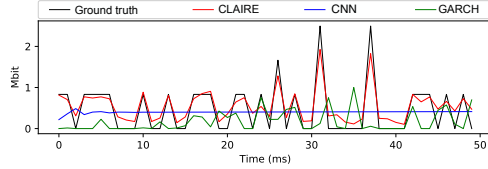
As mentioned in § 3.4.2 and § 3.4.3, the traffic flows that originated from recurring connections are grouped into clusters depending on the normalized histograms of flow chunks in frequency space. In our traffic prediction experiments, we always predict on

Table 3.4: SAPDCTraces prediction experiments. While **CLAIRE** achieves 8.86% prediction error on average, ARIMA and GARCH exhibit at best(!) -60.24% and -73.80% respectively; NNs perform even worse.

Group stats			Prediction error				
GID	N_f	$\sigma^2(\text{PCA})$	CLAIRE	ARIMA	GARCH	LSTM	CNN
1	16085	93.7%	-10.08%	-86.97%	-73.80%	3734.82%	1729.27%
2	3532	91.0%	-10.16%	-81.03%	-89.81%	4881.68%	2328.22%
3	956	95.6%	-6.95%	-77.03%	-94.58%	5006.81%	3082.94%
4	1050	89.1%	-0.47%	-81.72%	-92.03%	3591.98%	2280.79%
5	178	96.3%	-6.87%	-77.54%	-96.54%	3431.94%	2596.40%
6	8	97.5%	-19.22%	-60.24%	-93.52%	3514.34%	2185.67%
7	42	88.6%	-8.28%	-73.35%	-97.99%	3938.41%	3516.55%
Avg:			-8.86%	-76.84%	-91.18%	4014.28%	2531.40%



(a) Comparison of CLAIRE, ARIMA, and GARCH on flow cluster 3



(b) Comparison of CLAIRE, ARIMA, and GARCH on flow cluster 19

Figure 3.7: Sample prediction details across approaches.

different flows than what we train on. For all prediction experiments described here, we used $\Delta_\Omega = 10\text{s}$ as observation interval; a data sample period $\Delta_S = 10\text{ms}$ was used for producing chunks of $W = 500\text{ms}$ and a chunk starting index $w = 50\text{ms}$ (cf. § 3.4.2, Tab. 3.2, and § 3.4.3).

Prediction accuracy (**ACCUR**).

For the UniDCTraces HTTP data, 21 clusters were identified. The prediction results are shown in Tab. 3.3, where the column “# flows” shows how many flows each cluster contained. On average, a cluster consisted of 8 unique flows. The performance of **CLAIRE** was tested on every flow cluster separately. During an experiment, 50% of the flows were selected as the test set. The rest of the flows were used for constructing the training set.

The results of the SAP SE data set are shown in Tab. 3.4. As for the UniDCTraces data set, a given flow is either used for training or for testing. In addition, here, if a flow is

used for training the first 10min of that flow is used; if it is used for testing that starts after 10min. Due to the similarity of the inter-DC traffic, 7 clusters are enough to obtain good accuracy, while 2 clusters can be considered as “outliers” because they contain only few flows.

Bursty nature In § 3.6.2 we discussed that traffic flows often consist of short traffic peaks that in turn can be divided into a rising phase and a phase with a mainly constant load level [23]; with the peak rises thus being the more important parts of flows, we explicitly concentrate our predictions on these parts. For the HTTP we tested the ability of CLAIRE to predict the peak rises after only a few observations from the beginning of the peak. As mentioned above the reason is that the start of the communication seems to be arbitrary because humans are involved. Furthermore, due to the small amount of training and testing data we vary the prediction time, the hidden state can not be fully learned for a fixed time interval because of the low number of training observations. Therefore, the number of chunks (cf. § 3.4.3) for the UniDCTraces data set differs between the flows and ranges between 3 and 60 steps in the frequency domain which corresponds to observed time intervals of 150ms to 3s duration. When the observation stops and the true prediction begins, all flows from the UniDCTraces data set still exhibit a low traffic load which then increases rapidly by around 150% on average during the next s.

Metric. The *prediction error* metric used for both data sets takes the highest single peak (highest kbit value) and its corresponding interval predicted by CLAIRE during a given prediction interval, and compares it to the highest peak value in the same interval of the eventual ground truth data. This metric is motivated by the requirements for a TO algorithm which will be interested in the maximum peak of the predicted window. The difference between both values divided by the actual highest value represents the applied error metric for the prediction (see Tab. 3.3 and Tab. 3.4 *prediction error* column). Note that the difference between (1) under- and (2) overestimation is important, and at the same time we never observed both (1) and (2) for a same cluster, thus we do not use absolute values.

With much more training data being available for the SAPDCTraces data set compared to the UniDCTraces data set, CLAIRE was always used in the former case with a fixed time period for prediction of 500ms, resulting in 10 prediction steps in the frequency domain. Even if the prediction error of Tab. 3.4 does not look more outstanding compared to Tab. 3.3, we have to consider that the SAPDCTraces data set is much more heterogeneous, the peaks are much steeper and higher, making prediction harder. Nevertheless, as shown in this section, the prediction of entire trajectories and their structure from the SAPDCTraces

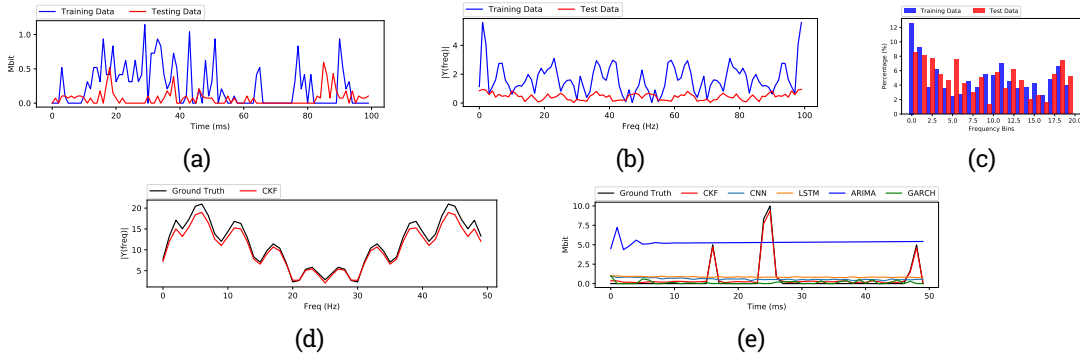


Figure 3.8: Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 1

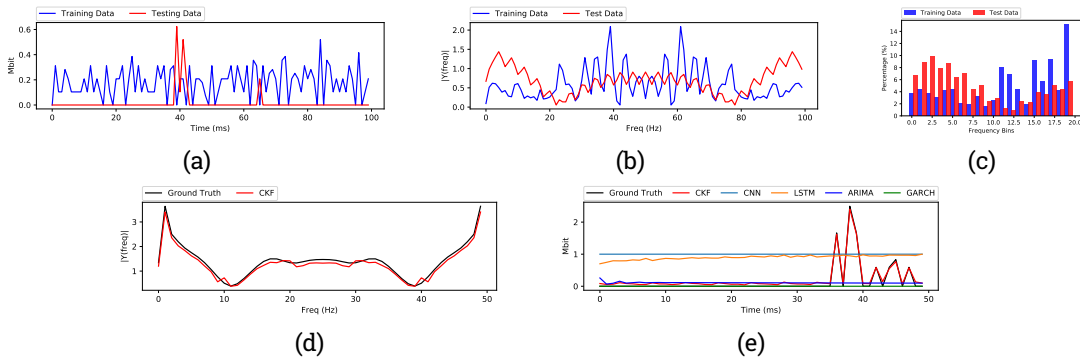


Figure 3.9: Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 2

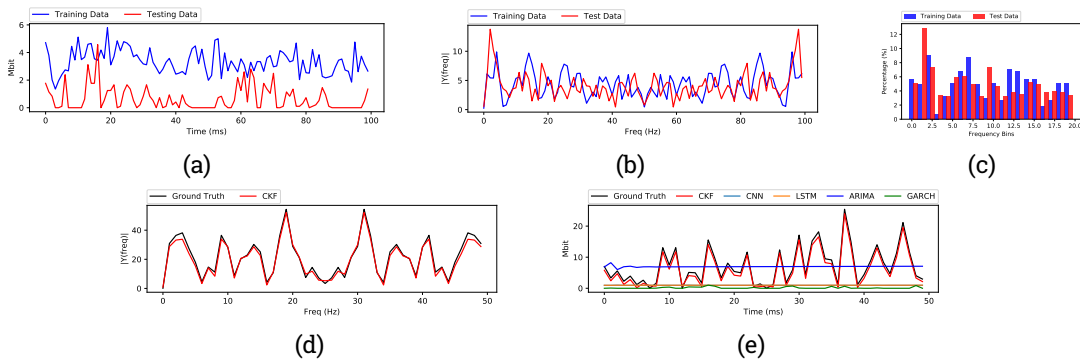


Figure 3.10: Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 3

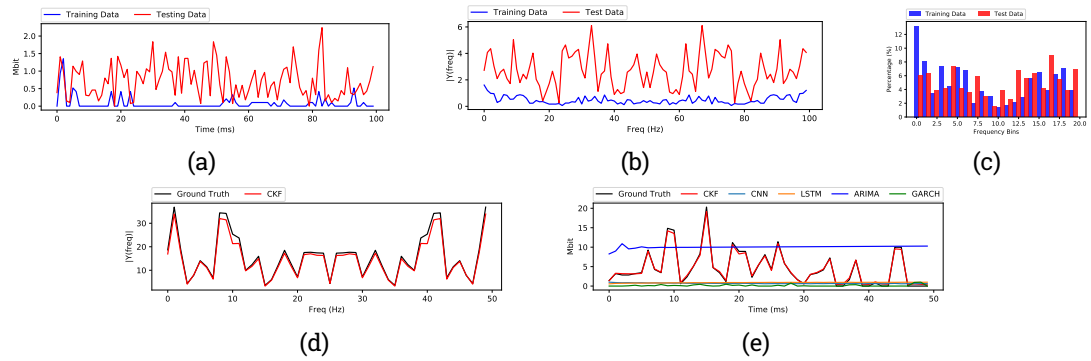


Figure 3.11: Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 4

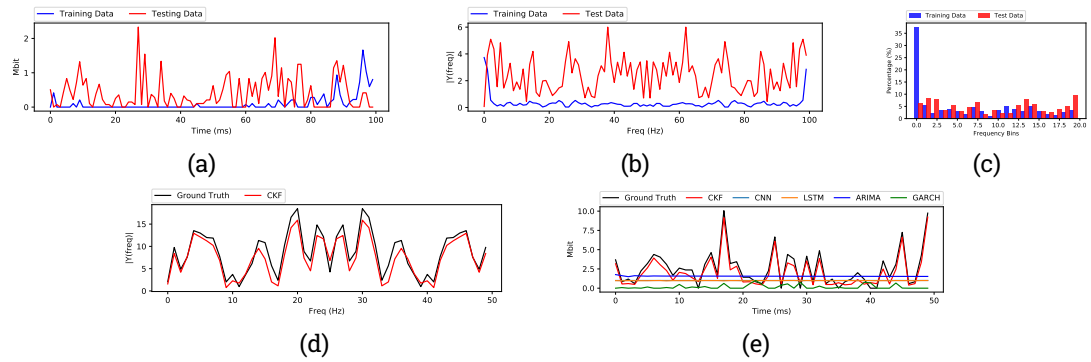


Figure 3.12: Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 5

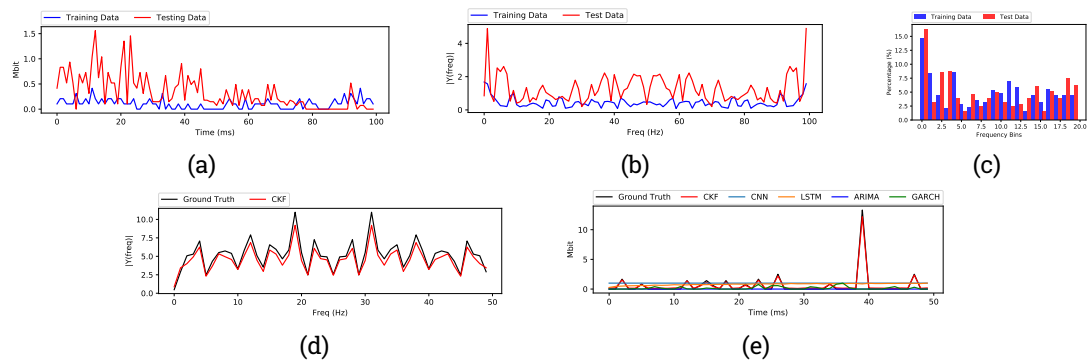


Figure 3.13: Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 6

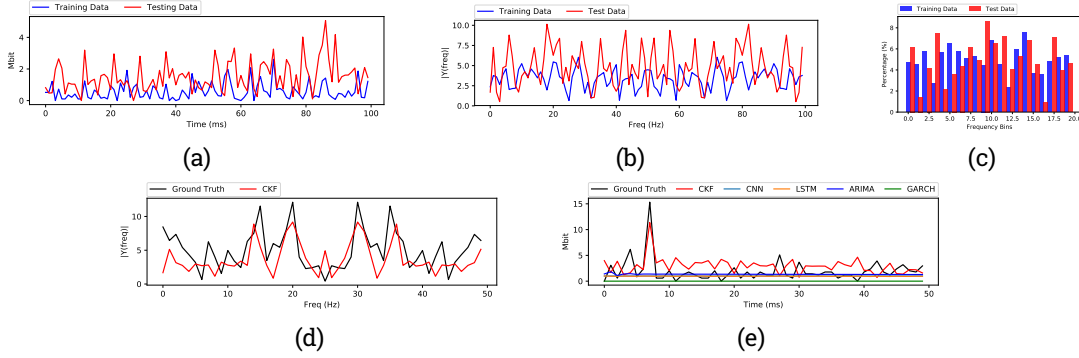


Figure 3.14: Example preprocessing ((a)-(c)) and prediction steps ((d)-(e)) of cluster 7

data set is much more accurate because of the two reasons mentioned above (amount of training data and predictable M2M communication). Another benefit, which comes with the SAPDCTraces data set, is that not only the highest peak and its corresponding interval between two Δ_S can be predicted. With the SAPDCTraces data set we consider all peaks with $peak \geq (highest\ peak\ in\ prediction\ interval \times 60)/100$. This yields more information for a TO algorithm, because now it knows exactly the interval Δ_S and height of peak(s), and even whether an oversubscription may occur in the next prediction interval; due to the time shift between the different flows this would prevent packet loss or congestion.

Alternative approaches.

As mentioned, CLAIRE was compared to the time series approaches ARIMA and GARCH as well as state-of-the-art NNs like LSTMs [167] (a specialization of RNNs) and CNNs [108] (that can be expected to perform better than BLRNNs [145] and MLPs [31, 43] used in the past). As expected, ARIMA and GARCH are not able to learn the hidden state of the system at every chunk length and consider the peaks as outliers (ARIMA), predicting zeros or very low values, or oscillate arbitrarily (GARCH). All NN approaches show the same behavior, by converging to the mean of a given trajectory without really considering the peaks. We tried several optimizations but the behavior was always the same. As the amount of transmitted data is low and basically consists of few small peaks for the UniDCTraces data set, the NN approaches achieve an underestimation and in some cases (with very few peaks) good results. For the SAPDCTraces data sets where we have a huge amount of transmitted data (together with very many small peaks) the NNs hugely overestimate small peaks which explains the huge errors on that data set. Below, examples of flows vs

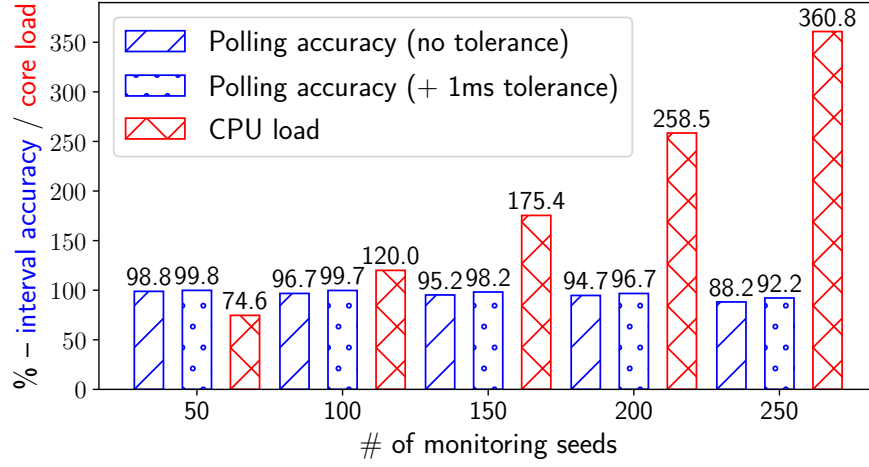


Figure 3.15: Polling accuracy and CPU load of monitoring with different numbers of monitoring instances.

their predictions for all SAPDCTraces clusters are visualized.

Cluster Results In this appendix we give an impression of the preprocessing and prediction steps shown in Fig. 4.1 with one example for each of the 7 clusters of the SAPDCTraces data set. The first figures (Fig. 3.8(a), Fig. 3.9(a), etc.) always show the training/test data window (1s) in time space. The respective second figures (b) show the data window from the first figure in frequency space. The normalized histograms which decide what is clustered are shown in the third figures (c). The forth figures (d) show the prediction and ground truth of the next 0.5s in frequency space. The last figures (e) show the ground truth, the CLAIRE in time space together with ARIMA, GARCH, LSTM, CNN.

Overhead ([SCALE],[IMPL]).

Being able to run our CLAIRE algorithm on commodity DC switch hardware is as important as achieving good prediction results. For the system evaluation we used an Edgecore AS5512-54X 10G [56] switch with OpenSwitch and ONL and Arista 7280QRA-C36S 10G [12] with Arista EOS.

Polling accuracy and CPU load. As discussed in § 3.4 CLAIRE required a new optimized monitoring system to poll the statistics of individual flows periodically at exact points in

Table 3.5: CPU cycles for one prediction.

Approach	CPU cycles avg	CPU cycles min	CPU cycles max
CLAIRE	48033428.7	28985277	164123937
ARIMA	158142741.6	90348180	331880529
GARCH	7063469546.1	3344161590	21315984630
LSTM	283656493.4	14362589	336300311
CNN	980063918.6	933049464	1023699573

time because inaccuracies in data measures obviously carry over to predictions. Fig. 3.15 shows the result of the monitoring polling accuracy with an interval of 10ms. On the x axis different numbers of monitoring instances were tested. The blue bars show the accuracy in polling time without any tolerance. The green bars visualize the accuracy in polling time if we consider a tolerance of 1ms which is still in an acceptable range. The red bars show the CPU load of one core with the given number of monitoring instances running simultaneously. The tested switch has a quad core CPU for the management system, which means that by monitoring 100 individual flows only 50% of one core is occupied. As mentioned in § 3.5.2 the monitoring instance sends a copy of the monitored data to an instance which is responsible for re-classifying and re-learning a flow, if the prediction results of an individual flow are not accurate enough. This step is not time-critical and can be computed on other devices with more computational resources. Therefore, it is omitted.

CPU cycles. As discussed in § 3.5.2 on top of the seeds the CLAIRE instances are predicting given flows. One CLAIRE prediction needs, on a single core, between 1ms and 4ms (1.89 ms average) for running 1000 predictions in different flow clusters. The variance is due to cache misses and context switches of the OS and computational optimization of the learned model. For the HTTP traffic as shown in Tab. 3.3 the optimal prediction length varies between different clusters. The average optimal length is 490ms, which is the interval between two predictions. For the SAPDCTraces data set the chunk length was constant at 500ms. By taking the average processing time of the CLAIRE algorithm into account, more than 800 flows can be predicted on one switch (cf. [SCALE]). Tab. 3.5 shows that CLAIRE uses far fewer central processing unit (CPU) cycles for one prediction on average than approaches compared against.

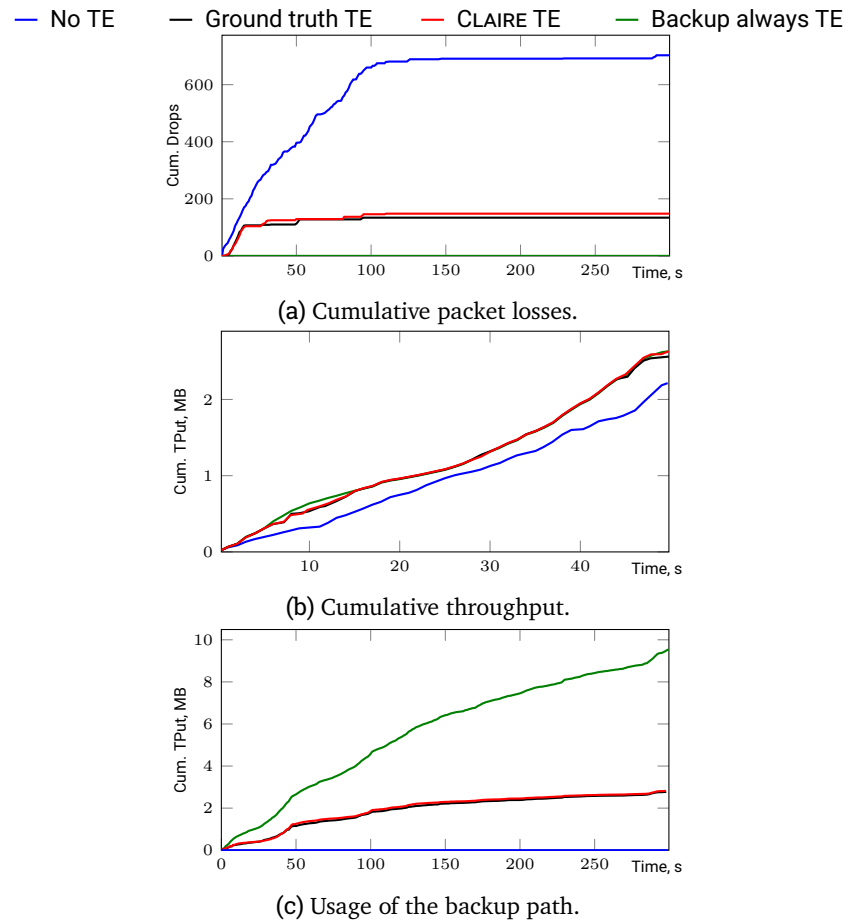


Figure 3.16: Evaluation on TE by leveraging CLAIRE predictions.

Proactive TE

After showing the results on predicting individual flows with CLAIRE and demonstrating the feasibility to monitor the data within the needed sampling period at scale where the prediction of individual flows is achievable, we are now highlighting the benefits and impact for TE algorithms. We used for this experiment a very simple approach where we introduced for each flow a backup path into the network. Whenever we predict an overflow of the buffer, we reroute the flow to prevent packet loss. In the future, we see a lot of potential for further research in this field. We rerun SAPDCTraces sampled flows

for better reproducibility on a well-known network simulator (NS2 [169]). We built a 3 stage Clos topology [59] with nine switches and added a backup path between the first and the last switch. 3.16a shows the cumulative number of packet losses in different scenarios: (a) with no TE, (b) using the ground truth of the next 500 ms, (c) using the results of the CLAIRE prediction, and (d) always using the backup path. As visualized, the number of losses dramatically decreases with the existence of prediction, which has a very positive effect on cumulative throughput ($> 40\%$). Overall the number of packet losses decrease by 78.9%. In Fig. 3.16b, we highlight the benefits of the prediction concerning throughput, which correlates with the benefits of packet loss. In this evaluation, using the backup path would prevent packet loss. So it can be seen as an optimal solution in this scenario (cf. 3.16a). By comparing the cumulative throughput against always choosing the backup path and making decisions on the ground truth data, we see that we are close to the optimal – backup path always – solution mentioned above. As a matter of fact, CLAIRE TE uses the backup path only when CLAIRE predicts an upcoming packet loss. Therefore, the backup path usage is much lower than the backup path always approach, as shown in Fig. 3.16c. Using the ground truth data for (perfect) prediction for TE yields no visible difference compared to CLAIRE-based TE.

3.7 Conclusions

In this chapter we introduced a novel approach which predict the bandwidth usage of interactions by predicting network flow trajectories in a fine-grained manner. More precisely, we focused on the prediction of peak structures of individual flows, the only type of prediction that truly considers the existing bursty nature of the network traffic which is a main cause of network congestion. In the conducted traffic prediction experiments, CLAIRE was used to predict the peak rises in a single flow from recurrent socket-to-socket connections. We evaluated CLAIRE on two different traffic data sets– one centered on human interaction (through HTTP traffic) and another on M2M communication. The overall failure is -8.86% which shows that CLAIRE is capable of fine-grained trajectory prediction.

4 X-LANE

4.1	Introduction	108
4.2	Related Work	110
4.3	X-LANE Design Overview	112
4.3.1	Communication in the X-LANE	112
4.3.2	X-LANE Controller Overview	113
4.3.3	Overview of Jitter Sources	115
4.3.4	X-LANE (Based) Services	116
4.4	Traffic Engineering for Tunnel Trees	117
4.4.1	Overview	117
4.4.2	Network Model	118
4.4.3	Two-Phase Tunnel Allocation	118
4.4.4	Optimization Problem	119
4.4.5	Traffic Engineering Proofs	122
4.4.6	Resource Monitoring and Tuning	125
4.5	Overcoming Interference in Data Centers	126
4.5.1	Bit Flips Errors in Links	126
4.5.2	Buffer Overflows and Jitter in Switches	127
4.5.3	Jitter in Endhost Commodity Hardware	128
4.5.4	Jitter in Endhost Specialized Hardware	133
4.6	Example Services Exploiting X-LANE	133
4.6.1	Failure Detector X-FD	133
4.6.2	Fast State Machine Replication X-Raft	134
4.7	Evaluation	136
4.7.1	Hardware Setup	136
4.7.2	Timing Observations	136
4.7.3	Towards Bounded Communication	139
4.7.4	Fast Fourier Transform Metric	139

4.7.5	Failure Detector Service X-FD	142
4.7.6	Fast State Machine Replication X-Raft	142
4.8	Conclusions	144

4.1 Introduction

In the last decade, a tremendous increase in Internet connectivity and the need for more computational performance changed the way we conceive applications. Today, most new applications are conceived as distributed, and in particular cloud-based, applications. The design of DCs and middleware layers then has to take into account all properties for distributed coordination, including performance, fault-tolerance, and consistency [27] — a hard task.

Interference in DSs. Most DS designs treat the underlying infrastructure as a generic communication system. One of the main issues with that is the longstanding problem of interference of concurrent interactions and thus unpredictable latency of commodity networks and hosts [50]. As a consequence, many DSs suffer from interference, manifesting through packets that may be arbitrarily delayed in the network (as well as reordered in transit or even dropped), and unbounded processing times.

While some applications and components can cope with unreliable communication systems due to their focus on throughput, many others benefit strongly from enforcing communication bounds with ultra low latency/jitter. This is especially the case for coordination tasks [157] whose use is very widespread in practice. Types of systems using the ZooKeeper [86] coordination service based on the popular Paxos [111] protocol by default or as option for coordination/fault tolerance include resource management (e.g., Mesos [79], YARN [174]), key-value and wide-column stores (e.g., Accumulo [8], HBase [11], etcd [58], TiKV [171]), data analytics (e.g., Hadoop [9], Spark [186]), or distributed file systems (e.g., HDFS [10]) to only name few.

X-LANE. The research question underlying the present chapter is whether DC components can mitigate interference and enforce communication bounds for specific DS interactions, which benefit from minimizing latency and jitter, thus reducing bounds for practical purposes. Therefore, we introduce in this chapter an express-lane — X-LANE for short — that strives first and foremost to minimize jitter, and in the process, also achieves unprecedented and bounded low latency, with the following properties:

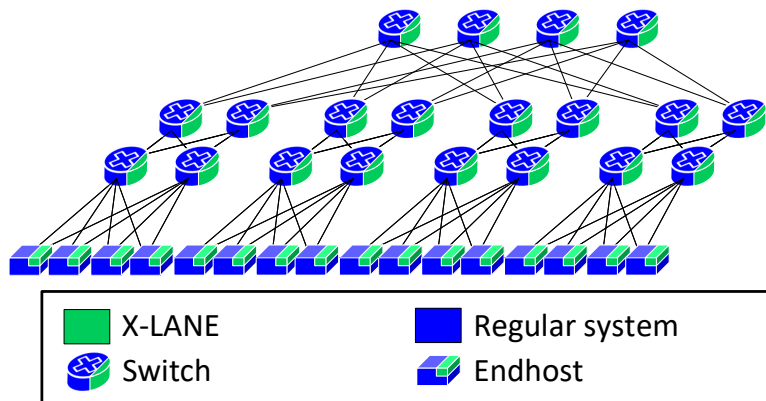


Figure 4.1: Overview with dedicated resources for X-LANE (green) and the regular (blue) lane running on the same actual physical network devices and endhosts.

Reliable traffic enforcement X-LANE provides an exclusive execution environment on endhosts to reduce latency and jitter, and a traffic engineering algorithm considering residual jitter and queuing delay to perform packet-level latency analysis and reliability in the X-LANE.

Generic system design A system design that is extendable with services that can take advantage of an X-LANE existence. Furthermore, a simple usage of X-LANE services for existing applications.

Commodity hardware support Current DCs are usually a mix of commodity and newest generations of hardware. Therefore, X-LANE is designed and implemented to be executable atop commodity hardware & software, as well as on intelligent network devices.

X-LANE takes advantage of dedicated physical resources tuned to become interference-free to fulfill the described properties for traffic that is latency/jitter critical in order to enforce communication bounds.

In contrast to prior works on low latency communication (e.g., [119, 71, 132, 146, 20, 149]), our focus is *not* on reducing common-case or 99th percentile latency, but on reducing maximum *jitter* to a point where it becomes so small relative to an already very low latency, that, in practice, can be assumed to be bounded. Moreover, we include *endhost response times*, and only provide bounded jitter to applications that rely on it (e.g., for coordination). Thus, X-LANE is an interference-free environment for interactions which benefits from ultra low latency in the *single-digit microsecond range* and bounded jitter in *nanosecond range* (cf. Fig. 4.1). The remaining interactions follow common design

principles. While being more generic in design compared to prior work on minimizing average latency, and also considering endhosts, X-LANE delivers significantly tighter bounds for latency and jitter for commodity hardware.

In short, X-LANE neutralizes *sources of interference* inherent to DC applications. These arise as a result of mainly two causes: (a) packet losses and (b) jitter in packet transmission and processing latencies — manifesting in different respective ways in the three infrastructure element types that are endhosts/servers, switches, and links.

We compare the achievements of X-LANE to the existing related work in § 4.2 before we introduce a design of X-LANE atop commodity hardware & software, as well as for intelligent network devices if available (§ 4.3). A TE algorithm incorporating residual jitter and queuing delay to perform packet-level latency analysis in the X-LANE (§ 4.4). Implementation of X-LANE overcoming sources of jitter on top of commodity hardware and software, as well as improvements and simplifications taking advantage of Netronome’s NFP-4000-based smartNICs [136] are discussed in § 4.5. § 4.6 shows the definition and implementation of two example services using the X-LANE: a FD dubbed X-FD, and the X-Raft state machine replication (SMR) protocol adapted from Raft [138]. § 4.7 evaluates X-LANE in a production DC of SAP SE through the deployment of the two services. Furthermore, we measure median latency and maximum jitter of the X-LANE on commodity hardware and software (Linux) (5.130 μ s latency and 655.000 ns jitter) and smartNICs (4.133 μ s latency, 152.000 ns jitter) *with heavy concomitant traffic over the course of 21 days*. Further comparisons display vast improvements over DPDK [52] (1.735 \times lower latency, 81,816 \times lower jitter), and QJump [71] — heavily advertised as providing *bounded* latency — (1.501 \times lower latency, 72,758 \times lower jitter), which greatly affect the coordination of DSs. We also show the applicability of X-LANE by integrating its SMR in the Redis key-value store [154], making it strongly consistent while decreasing latency 18 \times and increasing write throughput 1.5 \times . We draw the conclusions of this work in § 4.8.

4.2 Related Work

Distributed coordination and failure detection. Over the years, several authors have explored the improvements the coordination for DSs but only considering individual components or specific problems. Seminal works like mostly-ordered multicast [148] and unreliable ordered multicast [118] are multicast approaches where the ordering is done at the switches. Both approaches greatly improve the Paxos [111] consensus protocol thanks to in-network ordering. R2P2 [105]-based HovercRaft [104], NetPaxos [48, 47], and Consensus in a Box [88] similarly leverage switches for consensus protocols; like the

Albatross [116] membership service, they do not give guarantees under an overloaded network. Their main goal is to speed up resolution of individual services, via specific switch instrumentation, without considering other instances of the same protocol, other such protocols, or the network as a whole. Additionally, these approaches do not include synchronous interaction to the endhosts' user space required for many jitter-bounded applications (e.g., FDs).

Silo [92] shows feasibility of guarantees without constraining network elements; the guarantees provided are however not strong enough for applications like FDs in terms of jitter and packet loss. Falcon [117] focuses on what the network needs to provide to implement a perfect (reliable) FD, rather than how it can do so, and resorts to program-controlled crashes when the FD falsely suspects processes of being crashed due to missed timeouts, contradicting reliability.

Low latency. In recent years there were numerous proposals for achieving low latency network communication. The introduced approaches typically bound latency at the 99th percentile. The reason for the 99th percentile is that it is hard to deal with the sources of jitter in a complex system (cf. § 4.5.3). Tails of the tail [119], a seminal work in this area, identifies major jitter sources on endhosts, but does not consider the network, and focuses on 99th and 99.9th percentile latency, not 100th. Another path leading work is QJump [71] which proposes to achieve bounded latency on commodity hardware, but focuses on queues's priorities for low latency delivery and does not consider sources of jitter on endhosts (cf. § 4.5.3).

The data plane development kit (DPDK) framework is known for its fast and efficient poll mode drivers and fast packet processing capabilities. It has a wide range of driver implementations for various network interface cards (NICs). The DPDK developers have restructured and implemented a majority of the network device driver code and structure. DPDK operates by polling the network device from the user space application, which allows the programmer to harvest network packets bypassing the kernel network stack completely. As mentioned many works build on DPDK, e.g., Homa [132], Fastpass [146], IX [20], ZygOS [149], Chameleon [173]. These approaches try to optimize utilization and 99th percentile latency. Thus, they could be applied at regular system but as shown in § 4.7 are insufficient for X-LANE.

Time synchronization. Lee *et al.* [115] propose a time synchronization scheme for DCs offloading the synchronization code completely to a netFPGA card. However, time synchronization is not as hard a task as FD, as time synchronization packets can be lost. This approach does not consider network elements and packet loss under high load as

X-LANE does.

Endhost synchrony. Efforts on achieving real-time (RT) guarantees for commodity OSs like Linux are related to the X-LANE. RTLinux [155] is a real-time OS microkernel running the entire Linux OS as a fully preemptive process. RTLinux treats every process as having RT requirements, while X-LANE can treat a process in fair scheduled manner, or with even stronger RT guarantees; traditional RT schedulers, e.g. earliest deadline first (EDF) [122], can actually not guarantee that a specific task is performed by a given deadline, as they can not predict the system environment and are influenced by system service executions.

4.3 X-LANE Design Overview

Current networked systems are mostly designed to maximize overall throughput and utilization, sacrificing (dropping) packets in many scenarios in order to keep a global “good” performance. Existing works (e.g., [132, 119]) put additional emphasis on tail latency minimization recognizing the requirements of latency-sensitive applications and services. While these approaches address the needs of many services, they leave out those that require not only fast but also timely sensitive interaction, i.e., *timely services* exhibiting severe performance degradation upon delayed message delivery.

4.3.1 Communication in the X-LANE

To reconcile the needs of (I) timely sensitive (ultra-low latency and jitter), assured interaction that exhibits stable behavior as long as interconnecting devices function properly, with those of (II) best effort interaction with optimistic latency and reliability guarantees, we introduce an express lane (referred to also as X-LANE) following our original design (outlined in Fig. 4.2) isolated from the “regular system” which follows common design principles. This architecture is reminiscent of earlier models of separate systems [176, 175], yet realizes them *concretely, in a single infrastructure, with commodity hardware and software.*

X-LANE properties. X-LANE’s novelty is characterized by an explicit upper-bound on the latency of all the messages sent by a given process p to another process q , i.e., X-LANE keeps the latency of every such message within $[\lambda_{\min}^{p,q}, \delta_{\max}^{p,q} + \delta_{\max}^{p,q}]$, where $\lambda_{\min}^{p,q}$ is the best-case latency, and $\delta_{\max}^{p,q}$ is its concomitant maximum jitter.

Bounded time communication protocols. We achieve bounded communication properties in X-LANE by implementing a *periodic unicast protocol* where a process p can send a message to a given process q with latency upper bound $\lambda_{\min}^{p,q} + \delta_{\max}^{p,q}$, but under two constraints: p can send only once during every period $\pi^{p,q}$, and the packet size may not exceed $\sigma^{p,q}$. In addition, we specifically address one-to-many communication patterns by a *periodic reliable multicast protocol* that allows a process p to send a message to a set of processes Q with a common latency range $[\lambda_{\min}^{p,Q}, \lambda_{\min}^{p,Q} + \delta_{\max}^{p,Q}]$. A crucial requirement for both our protocols is that *all their parameters become known by the sending process at the protocol setup time*, i.e., before the first use, in order to allow services to adjust their internal timeouts for the best possible performance.

Note that throughput-oriented abstractions are *not* suitable for X-LANE, for they leave message size unspecified, while, clearly, no $\lambda_{\min}^{p,q}$ latency bound would hold uniformly for every message size, and queueing behind an arbitrarily large message leads to unbounded maximum jitter $\delta_{\max}^{p,q}$.

Timely unicast and reliable multicast serve as *backbone for all communication* between processes in the X-LANE. In the following, “periodic protocol” refers to “unicast protocol or reliable multicast protocol”. Bounding latency in the sending process is addressed in § 4.3.4 and detailed in § 4.5.

4.3.2 X-LANE Controller Overview

The properties provided by the two periodic protocols require careful orchestration of network resources in the X-LANE. X-LANE introduces a controller that takes on two main orchestration responsibilities: 1) *resource allocation*, i.e., answering requests from services with the most suitable protocol parameters, subject to network capacity constraints; and 2) *resource optimization*, i.e., keeping overall utilization of the X-LANE. TE techniques that underpin the controller’s operation are presented in § 4.4.

Each endhost runs a client of the X-LANE controller, which is a component of the X-LANE Linux kernel module that serves as a proxy between services and the controller. The client exposes the controller API (cf. List. 4.1) to services forwarding requests and responses in both directions. It is important to note that only the bounded communication over X-LANE is managed by the X-LANE controller. The rest of the communication proceeds as usual and uses the remaining resources in the usual best-effort manner. If no requests are ever made to the X-LANE controller, no network resources are spared or lost.

```

1 // Service request parameters for the
2 // resources from the X-lane
3 struct request {
4     int loadsize;        // max packet size (B)
5     int period;         // packet period ( $\mu$ s)
6     struct {
7         uint32_t ip;      // MCast or UCast IPv4
8         uint16_t port;   // service port
9     } receiver;
10 };
11 // Resources approved by X-LANE controller
12 static const int UNBOUNDED = -1;
13 struct resources {
14     int loadsize;        // max packet size (B)
15     int period;         // approved period ( $\mu$ s)
16     int minLatency;     // minimum latency (ns)
17     int maxJitter;      // maximum jitter (ns)
18 };
19 // Reason for resource modification
20 enum Reason { TE, BW_EXCEEDED, BW_UNUSED };
21 // Downcalls from services to controller
22 ↓ resources requestBandwidth(request req);
23 ↓ void releaseBandwidth();
24 ↓ void changeBandwidth(request req);
25 // Upcalls from controller to applications
26 ↑ void bandwidthChanged(resources res,
27     Reason reason = TE);
28 ↑ void bandwidthTerminated();

```

List. 4.1: Extract of the **X-LANE** controller C application programming interface (API) used for resource allocation. Structure `resources` defines a timely periodic protocol. The first three methods are called by services the next two are upcalls/callbacks.

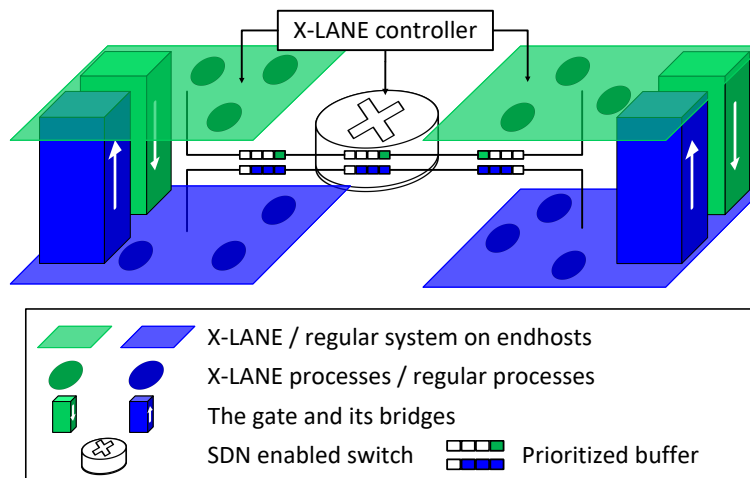


Figure 4.2: Separating the traffic of the x-LANE and regular communication on switches to prioritize packets and prevent losses in the former. An SDN controller sets switches' rules to adapt buffer allocation and processing priority. The x-LANE is interfaced to the regular system via the gate.

4.3.3 Overview of Jitter Sources

To implement an X-LANE usable in practice for time-sensitive tasks, X-LANE is required to mitigate the inherent uncertainties in DC computing. We expose and address numerous jitter sources in § 4.5. In short, we identify the following causes:

- **Packet loss:** Packets can be lost, leading to retransmissions and thus uncertain latency. Besides intentional drops (e.g., for security), packet loss has two well-known causes:
 - **Bit flip errors:** Bits can get flipped in links, leading to packets being marked as corrupted and discarded (§ 4.5.1);
 - **Buffer overflows:** Packets are dropped when the finite resources on processing units are overloaded (§ 4.5.2).
- **Intrinsic jitter:** While common switching devices forward packets with little jitter (§ 4.5.2), endhosts and their commodity components have been becoming more complex, leading to many sources of jitter (§ 4.5.3) and motivating the need for moving the intelligence closer to network devices (§ 4.5.4). The lack of bounds on jitter further makes packet delay hard to distinguish from packet loss.

4.3.4 X-LANE (Based) Services

X-LANE enables processes executing on the regular system to interact with the X-LANE services that may offer *timely* responses thanks to the unique timing guarantees of communications in the X-LANE. There are a few intricacies to X-LANE that developers must take into account when interacting with and/or developing these services. First, applications and services, being in separate lanes, must use a specific interface to exchange data with each other. Second, X-LANE handles communications differently than in the regular system.

Building bridges between lanes. On endhosts, services must communicate with applications which have to deal with shared processor time. This resource sharing introduces unpredictable jitter for those processes while critical interactions need an upper bound on certain tasks. Therefore X-LANE provides the *gate*, that builds on two sets of queues called *bridges*, to establish the interface between processes in the X-LANE and on the regular system.

The bridges are depicted in Fig. 4.2; the express-to-regular (X-R) bridge (green cuboid) grants write access to the X-LANE (green parallelogram) and read access to the regular system (blue parallelogram); inversely for the R-X bridge (blue cuboid).

bridges are adressable using direct memory access (DMA) over PCIe (to minimize jitter, cf. § 4.5.3) but are placed at different locations depending on the endhost hardware configuration.

Using X-LANE services. services are implemented as components of the X-LANE Linux kernel module (XLK) (cf. § 4.6 for the list of already available services), and as thus have direct access to the X-LANE controller (cf. § 4.3.2) and to the XLK component responsible for communication with the NIC. Each service has a dedicated queue in the R-X bridge where it can receive (1) queries from applications wishing to start/stop using that service, and (2) queries and payloads specific to that service API. When an application starts using a service, the service requests network resources from the X-LANE controller and spawns a new queue in the X-R bridge dedicated to messages from this service to that application. Communications between services and the NIC are handled by another XLK component: the NIC bridge. The NIC bridge bundles up all the payloads from a service into packets and sends them over the wire at the allowed periodicity (cf. *period* in List. 4.1), and unpacks payloads on the receiver side. Like drivers, the bridge implementation varies between hardware.

Express communication on commodity Hardware. While commodity NICs rapidly process and copy packets to the main memory, they are not programmable. Procedures to send and receive packets must thus be executed by the CPU.

When handling packets that belong to the X-LANE, guaranteeing minimal response time and tight timing bounds for these procedures is especially challenging on commodity hardware. There is an abundance of sources of jitter within the CPU itself and in the communication path between the CPU and the NIC that prevents a jitter-free streamline flow of packets. As a response, we implemented a series of countermeasures to enable the X-LANE on commodity hardware, greatly improving the time bounds over the regular system. We dedicate § 4.5.3 to the solution due to the amount of details. On commodity hardware, both types of gate bridges are in the main memory.

Express communication on smartNICs. Unlike commodity NICs, new generation NICs — so-called smartNICs — are highly programmable. Tasks can be offloaded from the CPU to the processing engine of a smartNIC, ranging from packet pre-processing to complex programs. The (relative) simplicity of the hardware and software stacks of smartNICs, over those of an endhost operated by a Linux kernel, and their proximity to the physical interface enable for packets to be processed on smartNICs with far lower latency and jitter (cf. § 4.7.2). This makes smartNICs ideal to handle X-LANE services.

Processing for sending and receiving packets over the X-LANE is confined within the smartNIC. This processing is mostly as with commodity hardware, but with direct access to the packet processing pipeline and the ingress and egress buffers on the NIC (cf. § 4.5.4). The gate's X-R bridge is stored in the smartNIC's memory area while the R-X bridge is in the endhost main memory.

4.4 Traffic Engineering for Tunnel Trees

The key underlying mechanism of the controller are *latency-bounded fixed-bandwidth* tunnels, more precisely — tunnel trees (due to multicast), from sender to receiver processes.

4.4.1 Overview

The X-LANE controller relies on SDN for tunnel setup. In particular, by acting as an SDN controller, it gets access to network-wide view in a form of a *network topology graph* G and the means to manage switches. For every link $(u, v) \in G$, the following information is used: bandwidth $\text{bw}(u, v)$, size of an egress queue $\text{qlen}(u, v)$, minimum delay $\lambda(u, v)$, and maximum jitter $\delta(u, v)$. Importantly, $\lambda(u, v)$ and $\delta(u, v)$ need only include processing

and propagation delays, which are stable for switches and are made stable at endhosts by X-LANE’s endhost implementation (see § 4.5).

A resource *allocation* is represented by a set \mathcal{T} of directed *tunnels*, where every $T \in \mathcal{T}$ is a *subtree* of G with a *single source* $\text{src}(T)$ and a set of sinks $\text{dst}(T)$. Tunnels are in one-to-one correspondence with allocated `resources` (see List. 4.1); hence, for every $T \in \mathcal{T}$, we have packet size $\sigma(T)$, period $\pi(T)$, minimum latency $\lambda_{\min}(T)$, and maximum jitter $\delta_{\max}(T)$. X-LANE further employs TE techniques [81, 91, 82] to guarantee channel availability. The particular TE algorithm used for the synchronous lane is similar to B4’s state-of-the-art approach [82] (with worst-case estimation of available throughput) but is built upon a finer-grained network model to allow for packet-level latency bounds.

4.4.2 Network Model

We describe next the underlying system model intuitively and refer to § 4.4.5 for details. The X-LANE controller does not make any explicit resource reservations in the network but instead relies on *rate limiting* at the endhosts, forcing services to adhere to periodic protocol parameters. Thus, the input traffic of a given tunnel T consists of a packet p , $\text{size}(p) = \sigma(T)$, entering the $\text{src}(T)$ node precisely every $\pi(T)$. No relative time constraints are imposed between different tunnels. Once a packet p from T enters a node u , p is either considered delivered, if $u \in \text{dst}(T)$, or p is placed into the u ’s egress queue(s) corresponding to next hop(s) — $\{v : (u, v) \in T\}$, provided there is sufficient buffer space, if not — p is dropped. Switching and/or processing delays at u are incorporated into links leading to u (see below) to capture all delay sources uniformly. At the egress queue, p waits for its turn to be transmitted according to FIFO order, and after $\text{size}(p)/\text{bw}(u, v)$ seconds more p leaves the queue. It takes anywhere between $\lambda(u, v)$ and $\lambda(u, v) + \delta(u, v)$ before p enters the next hop v accounting for the minimum residual jitter remaining after applying techniques described in § 4.5.

TE of the X-LANE accounts for both the intrinsic uncertainties of the system and uncertainties arising from multiple services sharing network resources. Ultimately, TE ensures that every allocation \mathcal{T} is *valid w.r.t.* topology G , which essentially means that no actual system behavior violates $\lambda_{\min}(T)$ and $\delta_{\max}(T)$ for $T \in \mathcal{T}$ (formal definition see in § 4.4.5).

4.4.3 Two-Phase Tunnel Allocation

Resources in the X-LANE are allocated reactively, upon concrete requests by services.

To bootstrap a periodic protocol, a service calls the `requestBandwidth` method of the controller API passing the desired packet size and periodicity in a `request` structure r . The controller handles r as follows: 1) a new tunnel T is allocated between the

sender and receiver(s); 2) switches' meter tables are updated for resource monitoring; 3) parameter adjustments for other affected tunnels in \mathcal{T} are communicated to corresponding services using the `bandwidthChanged` callback; 4) the approved resources with periodicity adjusted according to the allocation are returned to the service. Naturally, the new tunnel T must *match* the request r , i.e., $\sigma(T) = r.\text{loadsize}$, $\text{src}(T) = \text{ORIG}(r)$, $\text{dst}(T) = \text{vs}(r.\text{receiver})$, $\pi(T) \geq r.\text{period}$ (mind the adjustment), where $\text{ORIG}(r)$ is the process that originated r and $\text{vs}(\cdot)$ derives receiving nodes from $r.\text{receiver}$. The returned structure reflects all the T 's parameters of a periodic protocol (cf. paragraph 4.3.1): latency range $[\lambda_{\min}(T), \lambda_{\min}(T) + \delta_{\max}(T)]$, periodicity $\pi(T)$, and load size $\sigma(T)$. The service frees the resources by using `releaseBandwidth`. For the X-LANE properties to be reliable, every `bandwidthChanged` callback invoked by the controller comes with a grace period, during which the service can send messages under the old periodic protocol guarantees.

4.4.4 Optimization Problem

A distinguishing feature of our setting is the inevitable interference between already established tunnels and the new tunnel. Trying to minimize such interference, we arrive to an optimization problem underlying steps 1) and 3) above.

Problem (PPTE). *Given a network G , an allocation \mathcal{T} , and a sequence of service requests r_1, \dots, r_k , find a sequence of new tunnels $\mathcal{T}' = T'_1, \dots, T'_k$ and adjust parameters of \mathcal{T} , s.t., T'_i matches r_i for $1 \leq i \leq k$, $\mathcal{T} \cup \mathcal{T}'$ is valid w.r.t. G , and $\sum_{T \in \mathcal{T} \cup \mathcal{T}'} (\lambda_{\min}(T) + \delta_{\max}(T))$ is minimized.*

Solving PPTE directly is challenging as deriving parameters (or even checking validity) for a general \mathcal{T} is highly non-trivial due to interdependency between arrival times for packets queueing behind each other. Hence to simplify the problem, we split the allocation into two phases: *optimization* and *adjustment*. The *optimization* phase PPTE-OPT takes as input a request sequence and decides on the matching sequence of tunnels, while the *adjustment* phase PPTE-ADJ alters the parameters of all tunnels so they become valid w.r.t. the network G . To make the adjustment phase always successful, we require the set of tunnels after optimization to be \approx -valid, i.e., for any $(u, v) \in G$ it must hold that $\sum_{T \in \mathcal{T}: (u,v) \in T} \sigma(T) \leq \text{qlen}(u, v)$. Thanks to a two-phase approach we can freely choose a heuristic for PPTE-OPT without affecting \mathcal{T} 's validity and, consequently, the reliability of latency bounds.

Problem (PPTE-ADJ). *For a network G and an \approx -valid \mathcal{T} , adjust $\pi(\cdot)$, $\lambda_{\min}(\cdot)$, and $\delta_{\max}(\cdot)$ of \mathcal{T} so \mathcal{T} is valid w.r.t. G .*

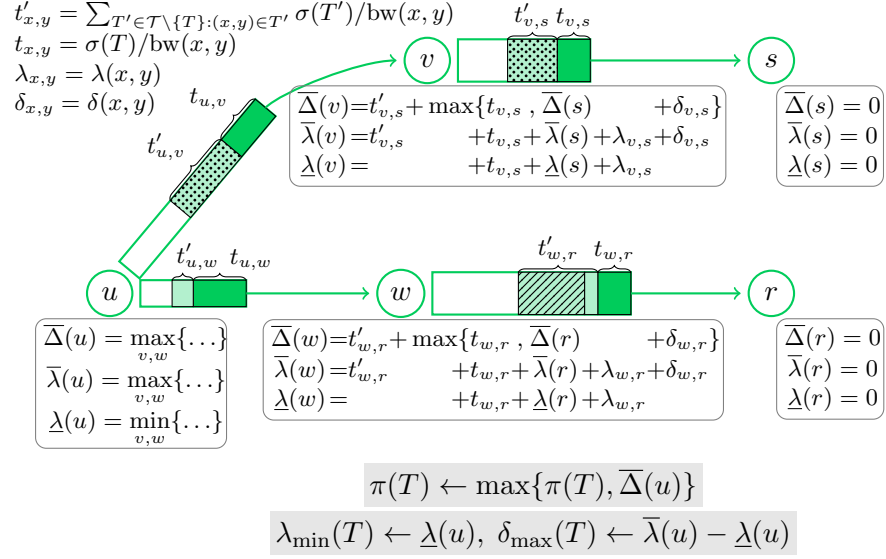


Figure 4.3: Core logic of the x-ADJ algorithm for PPTE-ADJ. The adjusted tunnel is $T \sim \blacksquare$ sharing queues with $T_a \sim \text{green square}$, $T_b \sim \text{blue square}$, and $T_c \sim \text{red square}$. Height of the queue at (x, y) is proportional to $\text{bw}(x, y)$; hence, packet length is proportional to the transmission delay. Note, order of packets is not important.

Our algorithm x-ADJ for PPTE-ADJ is illustrated in Fig. 4.3 (pseudocode is in § 4.4.5, Alg. 3); it has two logical steps. First, we compute minimum $\underline{\lambda}(u)$ and maximum $\bar{\lambda}(u)$ packet latencies from u , assuming at each queue (i) interfering traffic behaves in the worst possible way; (ii) at most one packet is present from each tunnel. Second, we compute a period lower bound $\bar{\Delta}(u)$, which would ensure (ii) indeed holds. The last step may increase the period beyond what was requested.

Theorem 1. *x-ADJ correctly solves PPTE-ADJ.*

Since parameters of \mathcal{T} are set based on a solution to PPTE-ADJ, the algorithm for PPTE-OPT must optimize in accordance with some fixed adjustment algorithm ALG.

Problem (PPTE-OPT). *For a network G , an allocation \mathcal{T} , a sequence r_1, \dots, r_k of requests, and an algorithm ALG for PPTE-ADJ, find a sequence $\mathcal{T}' = T'_1, \dots, T'_k$ of tunnels, s.t. T'_i matches r_i , $\mathcal{T} \cup \mathcal{T}'$ is \approx -valid w.r.t. G , and $\sum_{T \in \mathcal{T}^*} (\lambda_{\min}(T) + \delta_{\max}(T))$ is minimized; $\mathcal{T}^* = \text{ALG}(G, \mathcal{T} \cup \mathcal{T}')$.*

Algorithm 3 A recursive algorithm for PPTE-ADJ

```

1: procedure PPTE-ADJ( $G, \mathcal{T}$ )
2:   for  $T \in \mathcal{T}$  do
3:     PPTE-ADJ-DFS( $\text{src}(T), T, \bar{\Delta}(*), \bar{\lambda}(*), \underline{\lambda}(*))$ 
4:      $\pi(T) \leftarrow \max\{\bar{\Delta}(\text{src}(T)), \pi(T)\}$ 
5:      $\lambda_{\min}(T) \leftarrow \underline{\lambda}(\text{src}(T))$ 
6:      $\delta_{\max}(T) \leftarrow \bar{\lambda}(\text{src}(T)) - \underline{\lambda}(\text{src}(T))$ 
7:   procedure PPTE-ADJ-DFS( $u, T, \bar{\Delta}(*), \bar{\lambda}(*), \underline{\lambda}(*))$ 
8:     for  $v \in T : (u, v) \in T$  do
9:       PPTE-ADJ-DFS( $v, T, \bar{\Delta}(*), \bar{\lambda}(*), \underline{\lambda}(*))$ 
10:     $t'_{u,v} \leftarrow \sum_{T' \in \mathcal{T} \setminus \{T\} : (x,y) \in T'} \frac{\sigma(T')}{\text{bw}(u,v)}, \quad t_{u,v} \leftarrow \frac{\sigma(T)}{\text{bw}(u,v)}$ 
11:    if  $u \in \text{dst}(T)$  then
12:       $\bar{\Delta}(u) \leftarrow 0; \quad \bar{\lambda}(u) \leftarrow 0; \quad \underline{\lambda}(u) \leftarrow 0$ 
13:    return
14:     $\bar{\lambda}(u) \leftarrow \max_{v:(u,v) \in T} \{\bar{\lambda}(v) + \lambda(u, v) + \delta(u, v) + t_{u,v} + t'_{u,v}\}$ 
15:     $\bar{\Delta}(u) \leftarrow \max_{v:(u,v) \in T} \{t'_{u,v} + \max\{t_{u,v}, \{\bar{\Delta}(v) + \delta(u, v)\}\}\}$ 
16:     $\underline{\lambda}(u) \leftarrow \min_{v:(u,v) \in T} \{\underline{\lambda}(v) + \lambda(u, v) + t_{u,v}\}$ 

```

Algorithm 4 A shortest-path-based heuristic for PPTE-OPT

```

1: procedure PPTE-OPT-HEUR( $G, \mathcal{T}, r$ )
2:    $E' \leftarrow \{e \in G : \sum_{x \in \mathcal{T} \cup \{r\}} \sigma(T) \leq \text{qlen}(e)\}$ 
3:   for  $(u, v) \in E'$  do
4:      $w[u, v] \leftarrow \lambda(u, v) + \delta(u, v) + r.\text{loadsize}/\text{bw}(u, v)$ 
5:     for  $T \in \mathcal{T} : (u, v) \in T$  do
6:        $w[u, v] \leftarrow w[u, v] + \sigma(T)/\text{bw}(u, v)$ 
7:    $\tilde{G} \leftarrow (V, E', w)$   $\triangleright$  Weighted graph of non-overflowing edges
8:    $T \leftarrow (\{\text{ORIG}(r)\}, \emptyset)$ 
9:    $\sigma(T) \leftarrow r.\text{loadsize}; \quad \pi(T) \leftarrow r.\text{period}$ 
10:  while  $\text{dst}(T) \neq \text{vs}(r.\text{receiver})$  do
11:    for  $u \in \text{vs}(r.\text{receiver}) \setminus \text{dst}(T)$  do
12:      for  $v \in T$  do
13:         $\triangleright SP_G(u, v)$  finds the shortest  $u \rightsquigarrow v$  path in  $G$ 
14:         $\triangleright LP_T(u)$  finds the longest path from  $u$  in  $T$ 
15:         $\triangleright LP_T^w(u)$  and  $SP_G^w(u, v)$  return respective weight
16:         $c[v] \leftarrow SP_G^w(u, v) + \max\{0, SP_G^w(u, v) - LP_T^w(u)\}$ 
17:       $\text{cost}[u] \leftarrow c[\arg\min_{v \in T} \{c[v]\}]$ 
18:       $\text{path}[u] \leftarrow SP_{\tilde{G}}(u, \arg\min_{v \in T} \{c[v]\})$ 
19:       $T \leftarrow T \cup \text{path}[\arg\max_{u \in \text{vs}(r.\text{receiver}) \setminus \text{dst}(T)} \{\text{cost}[u]\}]$ 
20:  return  $T$ 

```

Irrespective of the adjustment phase, it is hard to even check the existence of a provisionally valid allocation if multiple requests must be answered at once. Furthermore, even for a relatively straightforward x-ADJ, PPTE-OPT is hard even for a single request (see Thm. 3). Thus, we resort to our x-OPT heuristic (see § 4.4.5, Alg. 4). The idea of x-OPT is to build tunnel T for r by gradually attaching shortest paths in a special weighted graph \tilde{G} to the next $v \in \text{vs}(r.\text{receiver})$. \tilde{G} 's weights capture $\bar{\lambda}(\cdot)$ evolution in accordance with x-ADJ.

Theorem 2. *Checking feasibility of PPTE-OPT is NP-hard.*

Theorem 3. *Finding an optimal solution to PPTE-OPT with $\text{ALG} \equiv \text{x-ADJ}$ is NP-hard even for a single request.*

4.4.5 Traffic Engineering Proofs

Given a network G and a sequence of multicast trees $\mathcal{T} = T_1, \dots, T_n$, a run \mathcal{R} of \mathcal{T} over G is a sequence $\mathcal{R} = (P_1, \dots, P_n)$ of packet sequences $P_i = (p_{i,1}, \dots, p_{i,k_i})$, where $\text{size}(p_{i,j}) = \sigma(T_i)$. Let us use $\mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$ as a time domain using ∞ when the packet was dropped. For every packet $p_{i,j}$ and every $v \in T_i$, there are three time variables: *arrival* time $t_{i,j}^+(v) \in \mathbb{R}_\infty$, *transmission start* time $t_{i,j}^\rightarrow(v) \in \mathbb{R}_\infty$, and *departure* time $t_{i,j}^-(v) \in \mathbb{R}_\infty$. The set $\text{IB}_t(u, v)$ of packets residing at time t in the output buffer of the u 's egress port connected to v is derived as $\text{IB}_t(u, v) \equiv \{p_{i,j} : (u, v) \in T_i \text{ and } t_{i,j}^+(u) \leq t \leq t_{i,j}^-(u)\}$. The variables must satisfy the following set of constraints: *periodicity* (PD), *bandwidth* (BW), *delay* (DE₁ and DE₂), *fifo* (FI), *mutex* (ME), *work conservation* (WC), *buffer size* (BS), and *greedyness* (GR).

$$\text{pd } t_{i,j+1}^+(\text{src}(T_i)) = t_{i,j}^+(\text{src}(T_i)) + \pi(T_i).$$

$$\text{bw } (u, v) \in T_i \Rightarrow t_{i,j}^-(v) = t_{i,j}^\rightarrow(v) + \sigma(T_i)/\text{bw}(u, v).$$

$$\text{DE}_1 \ (u, v) \in T_i \Rightarrow t_{i,j}^+(v) \geq t_{i,j}^-(v) + \lambda(u, v).$$

$$\begin{aligned} \text{DE}_2 \ (u, v) \in T_i \text{ and } t_{i,j}^+(v) \neq \infty \Rightarrow \\ \Rightarrow t_{i,j}^+(v) \leq t_{i,j}^-(v) + \lambda(u, v) + \delta(u, v). \end{aligned}$$

$$\text{fi } t_{i,j}^+(v) > t_{i',j'}^+(v) \Rightarrow t_{i,j}^\rightarrow(v) > t_{i',j'}^\rightarrow(v).$$

$$\text{me } (t_{i,j}^\rightarrow(v), t_{i,j}^-(v)) \cap (t_{i',j'}^\rightarrow(v), t_{i',j'}^-(v)) = \emptyset \text{ for } i \neq i' \text{ or } j \neq j'.$$

$$\text{wc } \bigcup_{i,j} [t_{i,j}^+(u), t_{i,j}^\rightarrow(u)] \subseteq \bigcup_{i,j} [t_{i,j}^\rightarrow(u), t_{i,j}^-(u)].$$

$$\text{bs } \sum_{p \in \text{IB}_t(u,v)} \text{size}(p) \leq \text{qlen}(u, v).$$

$$\text{gr } t_{i,j}^+(v) = \infty \Rightarrow \exists t. \lambda(u, v) \leq t - t_{i,j}^-(u) \leq \lambda(u, v) + \delta(u, v), (v, v') \in T_i \text{ and } \sum_{p \in \text{IB}_t(v,v')} \text{size}(p) + \sigma(T_i) > \text{qlen}(v, v').$$

A sequence of of multicast trees $\mathcal{T} = T_1, \dots, T_n$ is *valid* w.r.t. G iff for any run \mathcal{P} of \mathcal{T} over G it holds that for any $l \in \text{dst}(T_i)$, $t_{i,j}^+(l) \geq t_{i,j}^+(\text{src}(T_i)) + \lambda_{\min}(T_i)$ and $t_{i,j}^+(l) \leq t_{i,j}^+(\text{src}(T_i)) + \lambda_{\min}(T_i) + \delta_{\max}(T_i)$.

Proof of Thm. 1. Consider the sequence of tunnels $\mathcal{T} = T_1, \dots, T_n$ for a topology G after adjustments made by Alg. 3. To show the validity of \mathcal{T} w.r.t. G we consider an arbitrary run \mathcal{R} of packets $\{p_{i,j}\}$ and prove that packet arrival times satisfy the parameters of corresponding T_i s. There are two properties essential to that: (i) the period adjusted at Line 4 guarantees that no two packets from the same channel meet at the same queue; (ii) the tunnel parameters set at Line 5 and Line 6 are never violated. While it is (ii) that ultimately implies validity, the proof of (ii) relies on (i). On the account of that, we start with the latter.

(i) The proof goes by contradiction: assume that t^* is the smallest $t = t_{i,j'}^+(u)$ such that $t_{i,j'}^+(u) < t_{i,j}^-(u)$ for some $j < j'$. Let us consider a unique path u_0, \dots, u_k, u_{k+1} in T_i such that $u_0 = \text{src}(T_i)$, $u_k = u$, $u_{k+1} = v$. We prove by induction on $(k - i)$ that $t_{i,j'}^+(u_i) < t_{i,j}^+(u_i) + \bar{\Delta}(u_i)$, where $\bar{\Delta}(\cdot)$ is from the call to PPTE-ADJ-DFS with $T = T_i$.
Base case. Since before t^* there was never more than two packets from any channel simultaneously in a single queue we know that $t_{i,j}^-(u) \leq t_{i,j}^+(u) + t_{u,v} + t'_{u,v} \leq t_{i,j}^+(u) + \bar{\Delta}(u)$, where the last inequality follows from Line 15. Hence using the assumption, we get $t_{i,j'}^+(u) < t_{i,j}^-(u) \leq t_{i,j}^+(u) + \bar{\Delta}(u)$ and the base case.

Inductive case. The induction hypothesis is $t_{i,j'}^+(u_{i+1}) < t_{i,j}^+(u_{i+1}) + \bar{\Delta}(u_{i+1})$. We can easily conclude due to work conservation that $t_{i,j'}^+(u_{i+1}) \geq t_{i,j'}^+(u_i) + t_{u_i, u_{i+1}} + \lambda(u_i, u_{i+1})$. Again, due to minimality of t^* , we know that $t_{i,j}^+(u_{i+1}) \leq t_{i,j}^+(u_i) + t'_{u_i, u_{i+1}} + t_{u_i, u_{i+1}} + \lambda(u_i, u_{i+1}) + \delta(u_i, u_{i+1})$. From Line 15 we also have inequality $\bar{\Delta}(u_i) \geq t'_{u_i, u_{i+1}} + \bar{\Delta}(u_{i+1}) + \delta(u_i, u_{i+1})$, which combined with the previous one gives $t_{i,j}^+(u_{i+1}) + \bar{\Delta}(u_{i+1}) \leq t_{i,j}^+(u_i) + \bar{\Delta}(u_i) + t_{u_i, u_{i+1}} + \lambda(u_i, u_{i+1})$. As a result, the induction hypothesis implies $t_{i,j'}^+(u_i) + t_{u_i, u_{i+1}} + \lambda(u_i, u_{i+1}) < t_{i,j}^+(u_i) + \bar{\Delta}(u_i) + t_{u_i, u_{i+1}} + \lambda(u_i, u_{i+1})$, which after dropping equal parts gives us the induction step: $t_{i,j'}^+(u_i) < t_{i,j}^+(u_i) + \bar{\Delta}(u_i)$

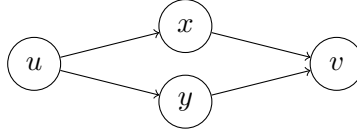
Finally, to get a contradiction we must notice that on the one hand we have $t_{i,j'}^+(\text{src}(T_i)) < t_{i,j}^+(\text{src}(T_i)) + \bar{\Delta}(\text{src}(T_i))$, on the other hand we know from periodicity that $t_{i,j'}^+(\text{src}(T_i)) \geq \pi(T_i) + t_{i,j}^+(\text{src}(T_i))$ and due to Line 4, we know that $\pi(T_i)$ must be at least $\bar{\Delta}(\text{src}(T_i))$ —

a contradiction. And it is easy to see that if $t_{i,j'}^+(u) \geq t_{i,j}^-(u)$ once $j' > j$, now two packets from the same channel can be at the same queue simultaneously.

(ii) First, using Line 16 it is straightforward to see by induction on u 's height in T_i that for any $l \in \text{dst}(T_i)$, $t_{i,j}^+(l) \geq t_{i,j}^+(u) + \underline{\lambda}(u)$. So the first part of validity, namely $t_{i,j}^+(l) \geq t_{i,j}^+(\text{src}(T_i)) + \underline{\lambda}(\text{src}(T_i))$, easily follows. Also, since we have (i) established, it can be easily seen that the queueing delay at any $(u, v) \in \text{src}(T_i)$ cannot exceed $t'_{u,v} + t_{u,v}$, so, again by induction on u 's height in T_i and using Line 14, we can show that for any $l \in \text{dst}(T_i)$, $t_{i,j}^+(l) \leq t_{i,j}^+(u) + \bar{\lambda}(u)$, and applying to $u = \text{src}(T_i)$ we get the second part of validity. \square

Proof of Thm. 2. Consider a NP-hard partition problem where given a set of integers x_1, \dots, x_n , $x_i \in \mathbb{N}$, one must check if this set can be partitioned into two sets of equal sum. We reduce an instance $\{x_i\}_i$, $S = \sum_i x_i$ of such problem to feasibility checking for PPTE-OPT.

For that, we build a four-vertex topology and create n requests r_1, \dots, r_n .



We set $r_i.\text{loadsize} = x_i$, $\text{vs}(r_i.\text{receiver}) = \{v\}$, $\text{orig}(r_i) = u$ and $\text{qlen}(u, x) = \text{qlen}(u, y) = \frac{S}{2}$, while $\text{qlen}(x, v) = \text{qlen}(y, v) = \infty$, all link rates are 1 and all delays are zero. First, it is easy to see that since any tunnel goes either through x or through y any \approx -valid solution allows us to recover partitioning because the total size of the total size of the two queues is exactly S . Conversely, given a partition, we can easily derive \approx -valid tunnels. \square

Proof of Thm. 3. In the directed Steiner problem we are given a weighted (weights are non-negative) directed graph G , a source vertex u and a set of destination vertices U' , and we are asked to find a minimum weight subgraph H of G s.t. every $u' \in U'$ is reachable from u in H .

It is straightforward to see that there always exists an optimal H which is weakly acyclic, i.e., it is acyclic when ignoring edge directions; hence, every such graph can be seen as a tunnel from u to U' .

Given an instance (G, u, U') of the directed Steiner tree problem we construct PPTE-OPT in the following way. The network topology graph \tilde{G} would contain exactly the same edges as G , link delays would be set to zero, link rates are inverses of the corresponding weights in G , and link queue sizes would be assumed infinite. Next, with every edge $(u, v) \in \tilde{G}$

we associate an existing tunnel $T_{u,v}$ with $\text{src}(T_{u,v}) = u$, $\text{dst}(T_{u,v}) = \{v\}$ and setting $\sigma(T_{u,v}) = 0$. Finally, we create a single request r , s.t., $\text{ORIG}(r) = u$, $\text{VS}(r.\text{receiver}) = U'$, and $R.\text{loadsize} = 1$. Let us denote as T the tree matching r in a solution to PPTE-OPT.

The most important thing to notice is that every delay is due to a packet from some $T_{u,v}$ being queued behind a packet from T , because all link delays are zero and all tunnels except T have zero load size. Moreover, there can be at most one such packet causing delay at every (u, v) , and, hence, for every $T_{u,v}$. Thus, the total maximum latency is the sum among all $(u, v) \in T$ of a unit-sized packet queueing delay. Due to the way we set link rates, such delay for (u, v) is exactly the weight of (u, v) in G .

Due to a remark the correspondence between directed Steiner trees and tunnels made earlier on, for every *optimal* Steiner tree we have a tunnel introducing the delay equal to the Steiner tree's weight, and, naturally, every tunnel corresponds to some Steiner tree. \square

4.4.6 Resource Monitoring and Tuning

In addition to its resource allocation task, the X-LANE controller improves resource utilization by monitoring and refining the set of already allocated tunnels.

Controller oversight. For instance, if a service wants to adjust its `loadsize` and/or `period` without disrupting other services, the `requestBandwidth` and `releaseBandwidth` methods force it to establish a new periodic protocol first, migrate all clients there, and only then release the old resources. This two-phase approach incurs artificial delay, adds complexity, and wastes the synchronous `laneresources`. The `changeBandwidth` method of the controller's API shortcuts the process by leveraging the `bandwidthChange` mechanism discussed earlier.

When a service attempts to use more resources than assigned, some of its packets get dropped at a rate limiter. X-LANE can do nothing to maintain timeliness for those packets, and neither should it as the service has violated the protocol. To ensure an already broken interaction does not waste resources, the controller decreases priority of that service's packets right after the drop, voiding their timing guarantees. Then, the jitter reduction is communicated to services sharing queues with the misbehaving one, and the latter is notified by `bandwidthChanged` with `resources.priority` set to `UNBOUNDED` and `reason` to `BW_EXCEEDED`. This service may recover later with `changeBandwidth`. Further, switches' meter tables are used to identify services that behave well but *underutilize* resources. The controller reclaims a portion of their bandwidth through the `bandwidthChanged` callback with higher `period` and `reason` set to `BW_UNUSED`.

In the extreme scenario when a service keeps violating the protocol and/or drives its bandwidth allocation to zero by not utilizing resources, the controller terminates the protocol unilaterally with `bandwidthTerminated`.

Fine-grained jitter control with sub-lanes. Earlier, we saw newly set up tunnels adding jitter to existing ones and vice versa, whose effect we incorporated in periodic protocol parameters. Certain combinations of services require a different approach. A low-traffic jitter-sensitive service (e.g., failure detection, cf. § 4.6.1) and a throughput-oriented one needing a “small enough” latency bound (e.g., replication), affect each other in very unequal ways leading to suboptimal overall performance. The controller addresses this issue through virtual sub-lanes— virtual controller instances that use different priority levels for synchronous communication, isolating services in a higher-priority sub-lane from lower-priority sub-lanes. This separation needs only be reflected at the tunnel setup, where lower-priority tunnels must include jitter from higher-priority ones but not the other way around.

4.5 Overcoming Interference in Data Centers

Comprehensive mitigation of jitter sources due to interference with the rest of the DC (outlined in § 4.3.3) is key to achieving ultra-low latency properties of the X-LANE. In what follows we describe our technique and discuss implementation details.

4.5.1 Bit Flips Errors in Links

Most of the messages transmitted via the X-LANE are expected to be much smaller than the MTU size. To reduce the data transmission overhead X-LANE tries to pack multiple data chunks into a single physical packet. The increased chance of packet loss due to bit flip errors is mitigated by using two custom error correction schemata that provide the same mean time to fault packet acceptance (MTTFPA) as layer 2 headers (i.e., 10^6 years with a bit error rate of 10^{-12}), while supporting either up to 55 chunks of 26 bytes per MTU or up to 40 chunks of 36 bytes (depending on the schema). Both schemata use a specific choice of cyclic redundancy code (CRC) polynomials. Common CRCs allow for checking and correcting transmission errors caused by bit flips in the network’s physical layer. The CRC used by layer 2 headers gives MTTFPA of at least 10^6 years with a bit error rate of 10^{-12} and a pessimistic probability of 4 bit burst of $1e-3$ for the whole packet [63].

Three parameters affect the error correction capability of a CRC: data word length, frame check sequence (FCS), and CRC generator polynomial. All together influence the

Hamming distance (HD) and thus the number of non-detectable errors at a given HD [107]. The FCS is the resulting value of a CRC calculation and is influenced by the CRC implementation, but primarily by the CRC polynomial [153]. Analogously to the CRC of layer 2 headers [45], which gives MTTFPA of at least 10^6 years with a bit error rate of 10^{-12} and a pessimistic probability of 4 bit burst of $1e-3$ for the whole packet [63], we introduce CRC for the X-LANE as follows. Considering the payload of a layer 2 packet with a size of up to 1500 bytes (MTU size), X-LANE splits the payload into smaller chunks each with a dedicated FCS. So if a FCS with HD 6 is used, a single chunk of payload must be no longer than $\lfloor (14/3 \times 5) \rfloor$ bytes or $\lfloor (14/3 \times 7) \rfloor$ bytes if HD is 8.

Based on a comprehensive analysis on 32 bit CRC error correction capabilities [106] X-LANE can be configured to use one of the two following polynomials: 1. The 32 bit polynomial `0xFA567D89` (1, 1, 15, 15) provides HD 8 for up to 274 bits and HD 6 for up to 32736 bits [29]. 2. The 24 bit polynomial `0xbd80de` provides HD 6 for up to 2026 bits [44].

Using chunks with HD 6 each chunk has a size of $\lfloor ((14/3 \times 5) + 3) \rfloor$ bytes hence 55 chunks fit in a MTU (1265 bytes of data total). If we use the 32 bit polynomial, each chunk with HD 8 has a size of $\lfloor ((14/3 \times 7) + 4) \rfloor$ bytes resulting in 40 chunks (1280 bytes). With both schemata X-LANE can transmit up to 1280 bytes net data in each packet with the same MTTFPA of $\geq 10^6$ years as a layer 2 header.

4.5.2 Buffer Overflows and Jitter in Switches

Endhost NICs have a large amount of buffer memory available, allowing them to enqueue large numbers of packets before they are constrained to drop some. In contrast, common switching hardware has a much smaller amount of (shared) buffer memory, that is commonly exceeded in the case of congestion, leading to packet losses ultimately hampering latency and jitter bounded communication. Common switches with an ASIC as forwarding processor can have their shared buffer split in multiple queues that are populated with packets from incoming traffic and are processed following a given scheduling strategy.

X-LANE uses a strict priority scheduler to realize the TE approach introduced in § 4.4, to serve queues in order of priority, i.e., a non-empty queue is chosen over any other queue with lower priority. For each switch handling X-LANE's flows, the X-LANE controller (cf. § 4.3.2) dedicates the switch's highest priority queues to the X-LANE, and adapts the queues' size to the expected load. X-LANE packets are therefore processed as fast as possible, reducing both jitter and the risk of packet drops since packets are processed before the queue is full. Furthermore, common switches are tailor-fitted to forward packets, they thus do so deterministically in the ns range [13].

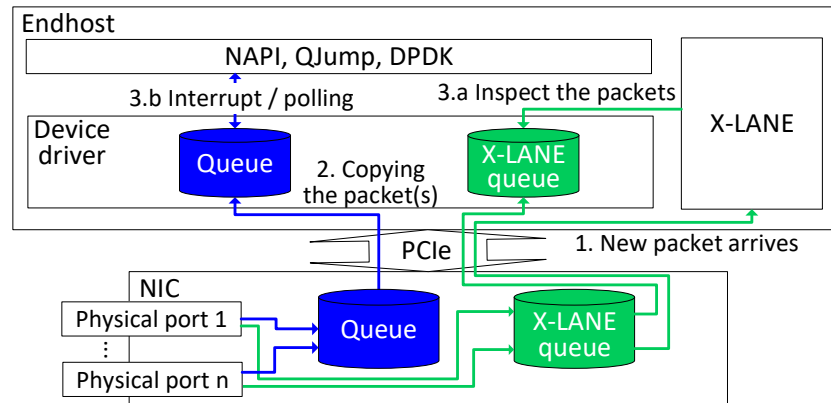


Figure 4.4: Overview of packet reception on commodity hardware.

4.5.3 Jitter in Endhost Commodity Hardware

While the standard network stack built upon endhost commodity hardware can be used for throughput oriented communication, the many sources of jitter it contains preclude X-LANE from using it for bounded communication. Fig. 4.4 depicts how packets are handled when received on the X-LANE (green) compared to the regular communication (blue); X-LANE focuses on timestamping packets as early as possible to minimize stamping jitter, doing so even before their payload is inspected, therefore performing *optimistic timestamping*. In the following, we give an overview of the measures implemented in X-LANE to drastically reduce jitter and latency of transmitting packets atop endhost commodity hardware/software.

First, at least a CPU core must remain available at all times for X-LANE services to promptly send and receive packets to/from other applications running on other endhosts. To do so, X-LANE runs on a dedicated core (§ 4.5.3) and shunts preemption on it to minimize completion time of X-LANE services (§ 4.5.3). Second, packets must be copied between the CPU, for processing, and the NIC, for remote exchange, while avoiding jitter-prone kernel memory management (§ 4.5.3). Fig. 4.5 gives an overview of preemption sources X-LANE has disabled compared to a regular system.

Highly Responsive X-LANE Dedicated CPU Core

Execution slots on CPU cores are managed by the OS kernel scheduler which, typically, distributes these slots in a fair manner across all applications to avoid resource starvation. Timing-sensitive tasks are, therefore, regularly preempted to leave room for other tasks, increasing both latency and jitter for the former. Even EDF schedulers [122] are affected by their jitter-prone environments and cannot guarantee the highest degree of responsiveness

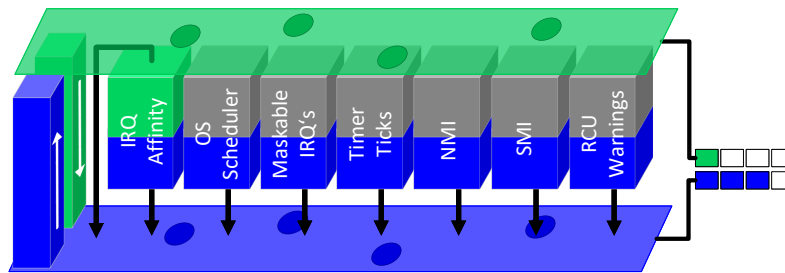


Figure 4.5: X-LANE is pinned to a dedicated core on which the sources of preemption (cuboids) are entirely (grey) or partly (green) disabled. The regular system is running on all other cores with all the side effects.

for such tasks. Furthermore, CPUs can switch between power consumption modes (i.e., C-states defined by the ACPI standard) to save energy when idle but need to wake up from an idle mode to execute a task, hampering response time [49].

X-LANE thus is *pinned* to a core, and isolates it from the scheduler to avoid task preemptions for a better response time. We call this core *X-LANE's core* as it is (almost) exclusively managed by X-LANE. X-LANE's core is isolated by including it in the `isolcpus` kernel boot parameter. To avoid costly wake-ups, X-LANE's core remains in the highest active state by setting the following kernel boot parameters: `cpuidle.off=1`, `powersave=off`, `processor.max_cstate=0`.

X-LANE's Uninterrupted Execution

Interrupt request (IRQ) signals are generated by hardware devices, e.g., I/O devices or CPU, to notify a core of an event to handle. The CPU preempts the task it is running to treat the received IRQ, which in effect increases the task's completion time and completion jitter due to the unpredictability of these IRQs.

We mitigate these delays by shielding X-LANE's core from as many IRQs as possible, as overviewed in Fig. 4.5. Those that cannot be ignored see their impact reduced (e.g., timer ticks).

IRQ affinity. On multi-core systems, IRQs can be distributed among cores (1) statically — IRQs are always routed to the same core, or (2) dynamically — IRQ affinity is set such that IRQs are handled by the core running the lowest priority task.

Most IRQs are routed away from X-LANE's core via a static distribution while other cores use a dynamic distribution, achieved by changing each IRQ's `smp_affinity` file in `/proc`.

IRQ masking. Some IRQs cannot be re-routed by setting their IRQ affinity, such as inter-processor interrupts (IPIs) that target a specific core. These IRQs can however be masked to prevent them from preempting the targeted core.

X-LANE masks IRQs with the `local_irq_save(int state)` kernel function before it executes X-LANE tasks. To re-enable the IRQs and restore the IRQ `state` X-LANE executed the `local_irq_restore(int state)` function with the same parameter. During the execution of the X-LANE tasks, the masked IRQs are routed to other cores, by adapting their affinity, to preserve the correct operation of the system.

NMI watchdog. The Linux kernel integrates a watchdog timer that regularly sends non-maskable interrupts (NMIs) to each core to test for hardware failures; it halts the system if the hardware does not handle the NMI. There exists no standard kernel mechanisms to ignore the watchdog's NMIs.

X-LANE prevents these jitter-inducing NMIs by disabling the watchdog using the `nowatchdog` kernel boot parameter.

Timer ticks. Timer ticks are a special type of IRQs originating from CPU-local timers or external timers. They are used to run routines at a set frequency, typically between 100 and 1000 Hz, as configured in the kernel [142]. In our experiments, we have observed a substantial processing time for each of these interrupts, ranging from 1.5 μ s to 50 μ s.

X-LANE mitigates timer interrupts by configuring the kernel with the `CONFIG_NO_HZ_IDLE=y` option and adding X-LANE's core to the `nohz_full` kernel boot parameter, which sets the given core to adaptive-tick mode. While this mode does not completely oust interrupts, it greatly reduces their frequency to 1 Hz, offering significant timing improvements. For even greater improvements, the X-LANE masks timer interrupts during the execution of its services. Masking these IRQs however will trigger warnings from the read, copy, update (RCU) stall detector that preempt the masked cores.

RCU warnings. The RCU stall detector issues a warning if a core is looping (1) in an RCU read-side critical section or (2) with interrupts and preemptions disabled. The stall detector triggers these warnings, i.e., time-wise unpredictable offloadable callbacks, once its grace period is over.

The RCU stall detector issues warnings to X-LANE's core as a side-effect of masking timer (and other) interrupts on them. X-LANE thus offloads RCU callbacks to other cores by configuring the kernel with the `CONFIG_RCU_NOCB_CPU=y` option and adding X-LANE's core to the `rcu_nocbs` kernel boot parameter. Further, less callbacks are triggered and offloaded

by increasing the grace period of the RCU stall detector set in the `rcu_cpu_stall_timeout` kernel boot parameter.

Unmaskable SMIs. System management interrupts (SMIs) are x86-specific unmaskable interrupts that force *all* cores to switch to system management mode to run safety-related tasks. These thus monopolize all cores for up to milliseconds, creating jitter. Some SMIs are critical to the safety of the system/hardware such as the ones forcing cores throttling to prevent overheating and hardware damage. These SMIs however are rare and typically do not happen in nominal scenarios.

To prevent SMIs and still protect system health, core throttling is disabled in the BIOS and X-LANE manages fans itself.

Packet Transfer Between X-LANE's Core and NIC

Sending and, in particular, receiving packets on an endhost is not a task as straightforward as on a switch. The complexity of this task lies within the memory management and device management modules of the Linux kernel that contain design decisions typically favoring fairness, i.e., reducing overall latencies, over prioritizing accesses for selected applications.

To reduce latency and jitter, X-LANE optimizes (1) how packets are copied between X-LANE's core, that packs outgoing and unpacks incoming packets, and a NIC, that encodes/decodes packets to/from the wire, and how (2) these two devices notify one another that a packet is ready to be handled by the other.

Packet copy. When booting, the NIC's driver initializes a queue on the NIC for outgoing packets waiting to be sent (i.e., TX ring buffer), and two queues for received packets waiting to be processed by a CPU core (i.e., RX ring buffers): one on the NIC and one in the main memory. Queues hosted on the NIC are accessible by every CPU via DMA over PCIe. However, different cores experience different access timings since computer architectures nowadays have non-uniform memory accesss (NUMAs). As such, both CPU and the main memory are split into several NUMA nodes; memory accesses and device accesses via PCIe within the same NUMA node are faster than across nodes as the latter are forced to use the slower QuickPath interconnect (QPI) link.

X-LANE operates its *dedicated RX ring buffers*, one on the NIC and one in the main memory (X-LANE queues in Fig. 4.4), for packets received on the laneto prevent jitter from the regular system packets' head-of-line blocking. The TX ring buffer remains unaffected as there is no risk of head-of-line blocking when the NIC transmits packets. In addition,

X-LANE selects its dedicated core such that it runs on the NUMA node that the NIC's PCIe lanes are connected to, thus avoiding the QPI link when performing a DMA to the NIC to send or receive packets.

Packet notification. While the NIC constantly polls its local TX ring buffer, populated by cores, and thus does not need any extra step to send packets, the NIC driver running on a core must be informed by the NIC that a packet is waiting to be processed in an RX ring buffer. The driver can be notified by: (1) receiving an IRQ sent by the NIC for each received packet, which is fast but inefficient for bursty traffic that creates a lot of IRQ masks, or (2) regularly polling the NIC's RX ring buffer (e.g., DPDK [52]), that fetches packets in batches but incurs a latency penalty for older packets (at the front of the queue) and for low polling frequencies.

X-LANE uses the IRQ-based approach to optimize delivery timing. X-LANE's core is not subject to bursty IRQs as the bandwidth is carefully managed and smoothened by the X-LANE controller (cf. § 4.3.2). As shown in Fig. 4.4, a NIC receiving a packet sends an IRQ to X-LANE's core, set with a fitting IRQ mask, using receive flow steering [143] (step 1). In response, X-LANE's core timestamps the packet, doing it as early as possible to minimize pre-stamping jitter, and copies the packet via DMA from X-LANE's queue in the NIC to X-LANE's queue in the main memory to prepare it for inspection (step 2). X-LANE then shares the packet timestamp with the application via the S-A bridge and only delivers the unpacked payload once it has been inspected (step 3.a), also via the S-A bridge. In comparison, X-LANE does not change how packets are handled on the regular system, e.g., with NAPI, DPDK (step 3.b).

Endhost Implementation Discussion

Additional work *in the kernel* would further improve the readiness of the implementation. For instance, X-LANE is currently limited by the granularity of some kernel boot parameters that affect all cores (e.g., disabling the NMI watchdog) and would benefit from per-core feature selection to better isolate its core. Further, most of these features are statically set at boot time, or even compile time. A dynamic configuration would help X-LANE's adaptation at runtime, reducing its endhost footprint when the X-LANE is unused. Ideally, we would be able to fully isolate cores at runtime to greatly improving X-LANE's efficiency both in terms of endhost resource utilization and implementation effort.

X-LANE currently uses one core but can scale to multiple without introducing delays as long as they are in the same NUMA node. The implementation currently focuses on Intel Xeon architecture, but AMD's EPYC has fewer NUMA nodes yet more cores, different memory management, and PCIe 4 that could improve X-LANE.

4.5.4 Jitter in Endhost Specialized Hardware

As an alternative to endhost commodity hardware, we propose an implementation of the X-LANE on recent intelligent network devices (i.e., smartNICs) that completely avoid kernel-induced jitter since they are not managed by it.

Our implementation supports Netronome’s smartNICs with NFP-4000 network flow processors. The NFP-4000 natively supports programs in microC, a dialect of C, and P4 [26] via a P4-to-microC transpiler. We chose microC to implement X-LANE’s services on the NFP-4000-powered smartNIC as it is more expressive than P4 despite recent developments on the latter, e.g., microC can directly access packet processing, flow processing cores, internal and external memory units.

Following the NFP-4000’s architecture [136], the components of X-LANE are running on a flow processing island that has 12 flow processors and its own memory to buffer packets. The number of flow processors used for X-LANE can be scaled on demand to match the traffic. Unlike the commodity hardware implementation, here X-LANE has direct access to the packet processing pipeline and the ingress/egress buffers closest to the physical interface which greatly reduces the jitter associated to sending/receiving packets on endhosts (cf. § 4.5.3).

4.6 Example Services Exploiting X-LANE

We propose two X-LANE-based services (cf. § 4.3.4): a failure detector service and a state machine replication service. These services are available for regular processes as part of XLK.

4.6.1 Failure Detector X-FD

We leverage a periodic reliable multicast protocol (cf. paragraph 4.3.1) that resides at the core of X-LANE to propose a heartbeat-based FD, X-FD, with a heartbeat period T . Unlike \mathcal{HB} [1] that outputs a vector of message counters to the application, X-FD tracks the state of remote processes in an *alive* table stored in the X-R bridge that can be read by any application.

X-FD operates in three successive steps. First, a user space application increments a timer value in the R-X bridge at least once per period T . Due to the jitter-prone nature of the application, the value update period must be much smaller than T (e.g., $T/3$ in § 4.7.5). Second, X-FD reads the corresponding value once per T from the R-X bridge and uses it for the heartbeat message, which is sent through the X-LANE every period. Finally, when the destination endhost receives the packet at the queue dedicated to X-LANE on

Table 4.1: Number of lines of code for each XLK component.

Core component	#LoC	Service (cf. § 4.6)	#LoC
Controller client	476	X-FD	223
NIC bridge	515	X-Raft	843
SmartNIC bridge	163		

the NIC, X-FD optimistically timestamps the packet (cf. § 4.5.3) and, while the packet’s payload is being analyzed, the alive table is updated with sender IP, port and last alive message timestamp.

4.6.2 Fast State Machine Replication X-Raft

We offer a second service by adapting Raft [138], a popular state machine replication (SMR) protocol [112, 113], to X-LANE in the form of the X-Raft service — a faster version of Raft using the periodic reliable multicast protocol (and Raft’s acks).

We adapted the well known etcd Raft [58] without any structural modifications to the algorithm or to its different phases (i.e., leader election, log replication/recovery, membership).

X-Raft uses the R-X bridge to enable an application to interact with the SMR (e.g., to propose a value) and uses the X-R bridge to notify the application. Leader election and consensus rounds are performed in X-LANE without interacting with the application.

X-Raft uses X-FD to detect process failure and initiate leader reelection if needed. Throughput-oriented log replication packets are sent via a lower-priority sub-lane with a very small period while commit statements are piggybacked on X-FD’s low-jitter periodic messages. In addition, X-Raft batches parallel consensus instances in one packet akin to other consensus protocols [181]. Timeouts are greatly reduced thanks to X-LANE’s low latency.

The log hosted by the leader is a buffer for uncommitted inputs; an input i is removed from the log when all replicas commit to a state that includes i . X-Raft uses a ring buffer for the log that is big enough to store the logs long enough for all replicas to commit a state or fail. The commit state pointer on the ring buffer is updated when replicas commit a new state.

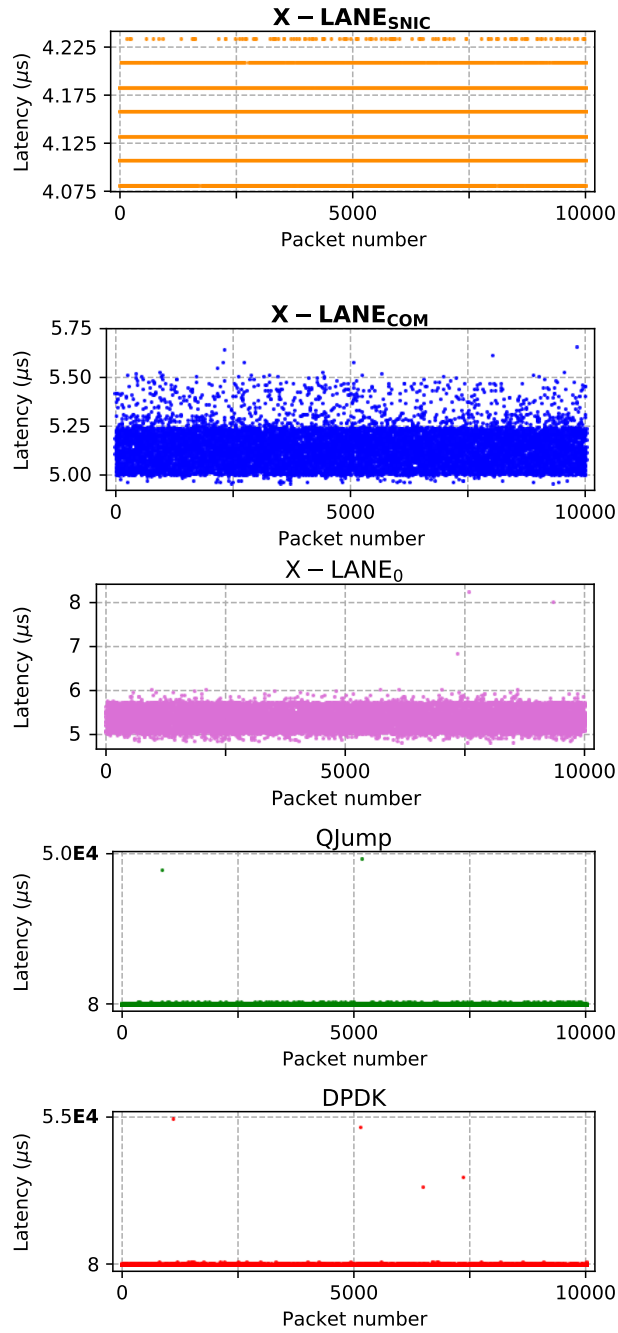


Figure 4.6: Overview of 10,000 packet latency (in μs) on the three x-LANE variants, QJump and DPDK. Note y -axes greatly vary.

4.7 Evaluation

In this section we assess the performance of X-LANE by first evaluating the latency and jitter of the underlying switching hardware (§ 4.7.1), followed by extensive evaluation of X-LANE’s communication timings (§ 4.7.2) and their variability (§ 4.7.3). We then evaluate the X-LANE-enabled services by measuring latency and accuracy of the FD service (§ 4.7.5), and latency and throughput of the SMR service both in isolation and once integrated in the Redis key-value store (§ 4.7.6).

Tab. 4.1 presents an overview of the implementation efforts behind each component of the X-LANE.

4.7.1 Hardware Setup

We ran our evaluation in a DC of major cloud service provider hosting Arista 7280CR-48 [14] switches and 17 servers with Intel Xeon E5-2680 v4 at 2.40GHz (26 cores, 52 threads), 1 TB RAM, Mellanox ConnectX-4 4x10 GbE [127] and Intel XL710 4x10 GbE [87] as commodity NICs, and Netronome Agilio CX 2x10 GbE [136] smartNICs.

Switches’ timing impact. We evaluated the impact of switches on latency and jitter by running multiple benchmarks with varying numbers of switches between endhosts. We observed a stable latency overhead per switch of 3 μ s for unicast and 6 μ s for multicast with no measurable jitter beyond this difference, as expected [69]. We also evaluated the accumulated impact of switches in common DC topologies [59], by running benchmarks up to a 4-hop topology, and only observed an impact on latency, not on jitter. For this reason, we evaluated X-LANE and its services on a 1-hop topology. This topology simulates in-rack computing that represents the majority of communication in optimized systems [59].

Note that the Arista 7280CR-48 switches we used are much slower than, for instance, switches from the Arista 7150 series with processing times of 350 ns according to their data sheet [13]. *Theoretically*, such switches could thus reduce the latency of our setup by at least 2.6 μ s, without affecting jitter.

4.7.2 Timing Observations

Most related works focus on reducing overall latency and maximizing network utilization, this work emphasizes jitter as another, crucial, dimension for many applications and in particular coordination tasks. Hence, we compare latency and jitter (and a fast Fourier transform-based metric in § 4.7.4) of three variants of X-LANE to each other, against QJump [71], and with DPDK [52]. DPDK was used at a lower level by, and

Table 4.2: Summary of X-LANE’s timings showing 0th, 50th, 99th, 100th latency λ percentiles (in μs), maximum jitter δ_{\max} (in ns) from λ_{\min} , and a metric based on probability bound (i.i.d. assumption) for $10 \times \lambda_{99\text{th}}$ violation over next 100,000 packets. Replacing our Arista 7280CR-48 by an Arista 7150 could in theory reduce all latencies by 2.6 μs (cf. §4.7.1).

Approach	λ_{\min}	$\lambda_{50\text{th}}$	$\lambda_{99\text{th}}$	λ_{\max}	δ_{\max}	$P_{10 \times \lambda_{99\text{th}}}^{100,000}$
X-LANE _{SNIC}	4.082	4.133	4.234	4.234	152.0	0.104
X-LANE _{COM}	4.938	5.130	5.446	5.649	655.0	0.301
X-LANE ₀	4.789	5.351	5.823	8.247	3.2E3	0.823
QJump	4.270	7.702	507.714	4.8E4	4.8E7	1.000
DPDK	4.103	8.904	403.256	5.4E4	5.4E7	1.000

thus frames the performances of, many related works on low latency (e.g., Homa [132], Fastpass [146], Chameleon [173]), high performance OSs (e.g., IX [20], Zygos [149]), and high performance SMRs (e.g., Hovercraft [104]) (cf. paragraph 4.2).

Setup. We compare five configurations — DPDK, QJump, and three variants of X-LANE. The two main variants are specific to the used hardware, and the third serves as a baseline:

X-LANE_{SNIC}: X-LANE on intelligent network devices;

X-LANE_{COM}: X-LANE on commodity hardware;

X-LANE₀: X-LANE on commodity hardware without endhost modifications (cf. §4.5.3), only in-network support.

We measured latency and jitter of the periodic unicast protocol on all configurations. We report latency as the time between a process sending a packet and the receiving process timestamping said packet. Sender and receiver processes are co-located on the same server to avoid cross-server clock skew; packets are still sent through the network. Processes sent packets with a 1 s period for QJump and DPDK due to high jitter, and a 10 ms period for X-LANE.

Dataset. DPDK’s and QJump’s runs resulted in a combined 181,440,000 packets on an *idle network of idle endhosts*. Every X-LANE variant’s runs also resulted in 181,440,000 packets. All possible point-to-point connections between servers were included. Contrasting with DPDK and QJump, however, cross-traffic and varying endhost utilization were

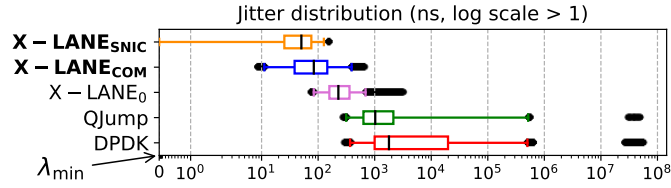


Figure 4.7: Distribution of X-LANE’s packet jitter δ (in ns, log scale for data > 1). A jitter of 0 corresponds to the packet(s) with minimum latency λ_{\min} within a dataset. Boxes are 25th/75th percentiles, black bars are medians, whiskers are 1st/99th percentiles, further data points are grayed out.

also present for X-LANE, setting the bar much higher for X-LANE. The number of packets sent and received for each approach represents 21 days of sampling.

Latency and jitter results. Overall the results reveal: (1) holistic approaches (X-LANE_{SNIC}, X-LANE_{COM}) perform better than network-focused ones (X-LANE₀, QJump) and endhost-focused ones (DPDK), (2) offloading X-LANE to smartNICs (X-LANE_{SNIC}) further improves timings compared to the already efficient commodity hardware approach (X-LANE_{COM}).

Tab. 4.2 overviews the timing measurements while Fig. 4.7 complements the table by exhibiting the main percentiles of the packet jitter distribution of each configuration. Even when running on commodity hardware, X-LANE_{COM} shows great performance benefits compared to QJump and DPDK, e.g., $1.501\times$ and $1.735\times$ lower median latency, and $72,758\times$ and $81,816\times$ lower maximum jitter, respectively. Unsurprisingly, the results indicate that offloading X-LANE to an intelligent network device achieves the best results across the board. Compared to X-LANE_{COM}, X-LANE_{SNIC} achieves $1.241\times$ lower median latency and $4.377\times$ lower maximum jitter. As jitter is the most important factor for coordination tasks in DSs, X-LANE shows its drastic reduction of maximum jitter makes it a prime candidate for such tasks (cf. § 4.7.5, § 4.7.6). The difference in timings between X-LANE₀ and X-LANE_{COM} shows the importance of tuning on endhost commodity hardware (cf. § 4.5.3) to reduce maximum jitter, i.e., tail latencies.

Fig. 4.6 further shows the individual latency of 10,000 packets among the highest outliers. Some packets for QJump and DPDK dramatically increase the jitter implying all the bad side-effects for coordination.

4.7.3 Towards Bounded Communication

We study the stability of the results obtained after 21 days of sampling in § 4.7.2 with the prospect of inching closer towards practical bounded communication in DCs thanks to X-LANE. We first focus on the packets whose latencies are beyond the 99th percentile, then propose an extrapolation using a simple probability-based metric. An additional metric that takes latency, jitter, and the occurrence of outliers into account using a fast Fourier transform can be found in § 4.7.4.

Beyond the 99th percentile. Fig. 4.8 exhibits latency stability by depicting percentiles characteristic of tail latency based on the 181,440,000 packet latencies collected over the course of several weeks. DPDK, which has the highest λ_{avg} , makes one jump at the 99.997th percentile. At the 99.98th percentile, we see once again that as more of QJump’s “outliers” are taken into account, there is a sharp increase in tail latency. All X-LANE variants exhibit a stable behaviour with X-LANE_{SNIC} being the most stable followed by X-LANE_{COM} and X-LANE₀. Another indication that X-LANE fully bounds the communication is the relative jitter defined as $(\lambda_{\text{max}} - \lambda_{\text{min}})/\lambda_{\text{avg}}$. While the relative jitter is ≈ 0.02 for X-LANE_{SNIC}, ≈ 0.13 for X-LANE_{COM}, and ≈ 0.36 for X-LANE₀, the values for DPDK and QJump are orders of magnitude higher: 1,807.18 and 1,113.19, respectively.

Probability-based metric. We consider as a metric the probability of having among the next N packets *at least one* with latency exceeding λ , $\lambda > \lambda_{\text{avg}}$. We cannot get that probability’s true value, so we use instead an upper bound P_{λ}^N under a simplifying assumption that the law of large numbers applies; i.e., packet latencies are independent and identically distributed, and we have performed enough experiments for sample mean λ_{avg} and variance σ^2 to be close to their true values. We derive the probability bound P_{λ}^1 for a single violation from the following tail-bound: $P_{\lambda}^1 \leq \sigma^2/(\lambda - \lambda_{\text{avg}})^2$. By using an independence assumption we further get $P_{\lambda}^n \leq 1 - (1 - P_{\lambda}^1)^n$. P_{λ}^n is a rough bound used only as a metric: the smaller its value is for an approach, the less that approach is prone to outliers.

Tab. 4.2 shows the probability to violate an SLO of $10 \times \lambda_{99\text{th}}$ over 100,000 packets. The results support a greater reliability of measured latency in X-LANE over that of QJump and DPDK.

4.7.4 Fast Fourier Transform Metric

We propose an additional metric to complete § 4.7.2, that considers *both latency and jitter*, as a compact way of comparing solutions’ suitability for bounded communication.

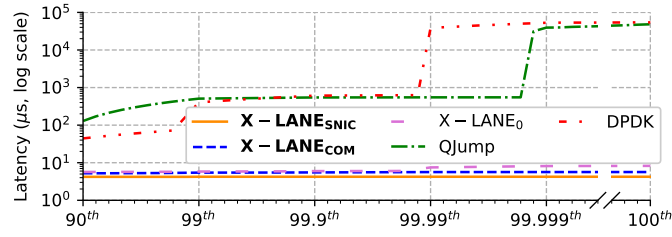


Figure 4.8: Tail latencies at different percentiles (different numbers of “nines”) observed over 21 days.

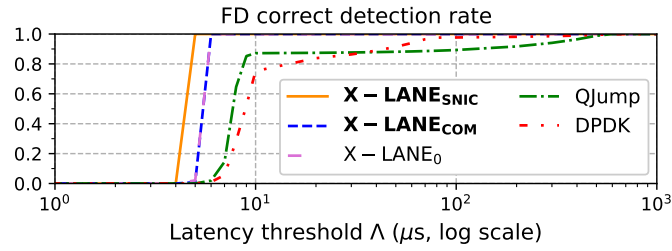


Figure 4.9: Accuracy of X-FD on all configurations showing the latency threshold needed to reach 100% accuracy.

Intuitively low jitter but high latency yields bounded behavior but poor performance; the inverse yields good performance for best-effort interaction but can not be used for timely communication. Moreover, since jitter can lead to cascading deterioration of expected response times, the metric must consider both high latency variability and the *frequency* of such occurrences. The metric must also cover latency since jitter can be artificially reduced by delivering all the packets at once at a given time, hence worsening overall latency.

We propose to input a set of packet latencies into a fast Fourier transform (FFT) [179] to project the latency trajectory into the frequency space and use the amplitude of the result frequencies as a metric. Intuitively, a higher jitter results in a greater frequency amplitude and thus in more unique frequencies that are needed to capture the trajectory. The latency is shown via the amplitude of the smallest frequencies.

Results. Fig. 4.10b transposes the latency of the different configurations shown in Fig. 4.10a (duplicated from Fig. 4.6 in the main paper) into a frequency space of size 250. The mean amplitude for each configuration is reported in Tab. 4.3. High amplitude of the smallest frequency corresponds to a high latency while high amplitudes of other frequencies represent outliers. The three X-LANE variants perform much better with lower

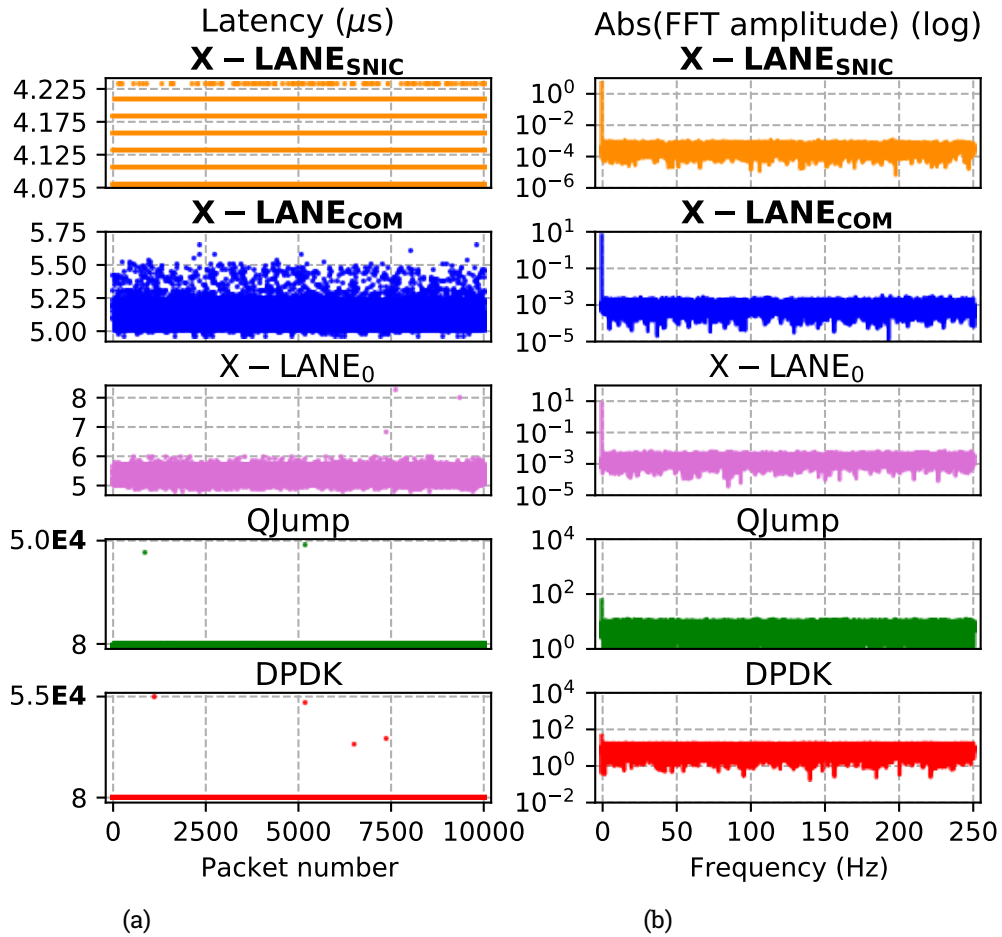


Figure 4.10: Overview of (a) latency (in μs) and (b) amplitude of FFT (log scale) of the three x-LANE variants vs QJump and DPDK. QJump and DPDK' latencies look falsely stable due to their very large timescales (y -axes).

Table 4.3: Mean FFT amplitude for each configuration.

X-LANE _{SNIC}	X-LANE _{COM}	X-LANE ₀	QJump	DPDK
5.891	9.341	15.098	3.0E4	3.9E4

amplitude at the smallest frequency, i.e., low latency, and lower variations overall, i.e., low jitter.

4.7.5 Failure Detector Service X-FD

We implemented the X-FD service (cf. § 4.6.1) atop all five configurations described in § 4.7.2 to compare the accuracy and completeness they provide in practice. We ran X-FD with 17 servers and a heartbeat period T of 1 ms whose value is incremented in an application every $T/3$. We varied the latency threshold Λ after which a process p is suspected of failure by others if no message was received from p in Λ .

Fig. 4.9 shows the rate of correct detection, i.e., accuracy, of the FDs with various threshold Λ , i.e., timeliness of completeness. We omitted T in the computation of the threshold. In practice, X-FD implemented on X-LANE reached a perfect accuracy with practical thresholds well below 8 μ s, and even below 5 μ s for X-LANE_{SNIC}. QJump reaches $\approx 90\%$ accuracy within 10 μ s but struggles for *a few milliseconds* for the remaining 10% needed for perfect accuracy. DPDK takes longer.

These results mean for instance that X-LANE can detect leader failures (e.g., in Raft [138]) orders of magnitude faster than its “low-latency” counterparts. Re-elections can promptly start hence greatly improving liveness.

4.7.6 Fast State Machine Replication X-Raft

We implemented X-Raft (cf. § 4.6.2) using X-LANE_{COM} and evaluated it against etcd Raft [58] by measuring the latency and throughput of write requests (i.e., operations) in groups of 3 to 9 processes, one per server. The configuration was evaluated by having an application send write requests to the group. Latencies were measured as the time between the user space sender emits a request and the time it is available for all user space applications in the group. Accesses to the log, hosted in a RAM disk, were thus not included in the latencies. The sender emits once 10 M write requests whose size follows a truncated normal distribution: min = 1 B, max = 10 MB and observed mean = 25.6 B, standard deviation = 10 B.

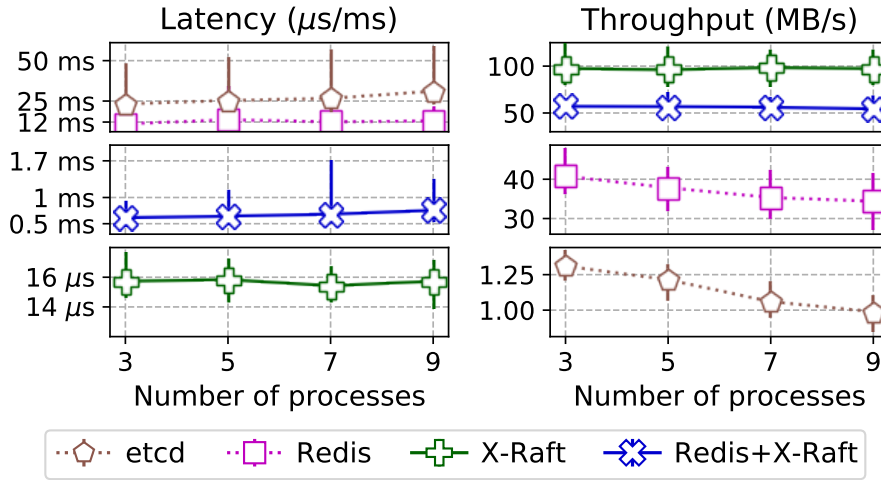


Figure 4.11: Write latency and throughput of X-Raft, etcd Raft, and Redis stand-alone vs with X-Raft. Mean values are plotted with min-max vertical bars.

Fig. 4.11 shows X-Raft performs much better than etcd both in terms of average latency, 15.7 μ s for X-Raft, 26 ms for etcd, and average throughput, 96 MB/s for X-Raft, 1.1 MB/s for etcd. We note that, compared to a unicast connection, X-Raft experiences 3 μ s of added delay due to the switch processing multicast (cf. §4.7.1) and 1.5 μ s for the ϵ safety margin, hampering results. Unlike etcd, X-Raft batches requests before sending them and relies on multicast that scales well with regard to group size etcd’s bandwidth requirement however is linearly proportional to group size.

Treating write requests as operations, with 25.6 B mean request size, X-Raft achieves 3.7 M ops/s mean throughput. In contrast, HovercRaft [104] achieves 1 M ops/s with 24 B requests but uses programmable switches, and NOPaxos [118] achieves 250 k ops/s (unknown size) but centralizes traffic.

Redis integration. To evaluate the genericity of X-LANE, we replaced the default inconsistent replication protocol of the Redis key-value store [154] with X-Raft. The result, a strongly consistent replicated key-value store, only took 26 lines of code of integration. Fig. 4.11 shows latency and write throughput for Redis and Redis+X-Raft with 3-9 servers. X-Raft reduces latency 18 \times on average and increases throughput 1.5 \times .

4.8 Conclusions

X-LANE implements unprecedented low latency and jitter to enforce coordination and control interactions, crucial to many applications in DSs. As this is not needed for all types of distributed interaction, X-LANE confines these bounds to an *express lane*, which is carefully isolated from the regular existing environment for throughput-oriented traffic both in the network and at the endhosts. X-LANE uses an original design leveraging commodity hardware and software, and smartNICs when available. A FD and a SMR protocol adapted from Raft [138] are implemented using X-LANE.

5 CONCLUSIONS


In this thesis, we discussed improvements in observing, exploiting, and enforcing properties of interactions in DCs. We showed that the more precisely properties of interactions are observed, the better the bandwidth usage is predictable, and bounded communication latency for DS coordination interactions are enforced. We introduced FARM for the precise and resource-efficient observation of DC network properties. To achieve the highest possible accuracy, not only at observation, seeds can (re)act locally. Furthermore, FARM builds a generic, scalable and accurate solution for different monitoring and managing tasks with respect to interconnections of DSs. With CLAIRE we presented a specialized method, which we believe answers the question of the feasibility of fine-grained network flow prediction affirmatively. By considering a network with its flows as a *non-linear system* and traffic load time series as observations, several latent factors contribute to the true, very complex *hidden* state of the networked system, e.g., types of programs generating the flows, workloads, user behavior, drivers on hosts, network components like interface cards and switches. CLAIRE exploits the available network resources by taking advantage of FARM via monitoring individual flows at a high resolution and reacting immediately by executing rerouting actions. A simple TE approach in combination with CLAIRE is able to reduce overall packet loss, and increase the FCT. Concluding, X-LANE tackles the longstanding problem of the unpredictability of transmission times for individual packets and, in particular *asynchronous* behavior of commodity networks and hosts. By circumventing existing sources of interference, X-LANE shows that even commodity systems can mitigate interference for tasks that benefit from minimizing latency and jitter, thus reducing bounds for practical purposes. By enforcing communication bounds for coordination tasks, a DS benefits massively from X-LANE.

6 FUTURE WORK

We believe that each chapter of this thesis opens up several avenues for future work. For FARM, the extension to all DC (compute/storage) instances is probably most intuitive future work. With the ability to place seeds on all DC components, FARM becomes a holistic monitoring and management framework for DCs' infrastructure. Since compute/storage devices produce other metrics and capabilities to observe compared to switches, Almanac needs to be extended accordingly which opens new possibilities for FARM's specific optimization heuristic. We are currently investigating several avenues for future work including fault tolerance and extensions to Almanac for ease of use (e.g., advanced inheritance, automated separation into seed and harvester code à-la tireless programming [135]). Also, new programmable hardware opens up new possibilities for monitoring with FARM. For instance Almanac could be compiled (in parts) to P4 [26] and run on new generation switching hardware and directly in the data plane. DCs usually host different generations of switching devices with different capabilities, especially in terms of monitoring. The placement optimization algorithm could address such heterogeneity with more sophisticated utility patterns where a seed can describe its utility with respect to platforms (e.g., Tofino vs Tomahawk). From a data sovereignty perspective, the collected data within seeds could be encrypted. The keys are only available for customers of the corresponding infrastructure. However, with the capabilities of homomorphic encryption, the customer could create functions that the infrastructure provider is able to monitor and manage the infrastructure without creating knowledge about the running applications.

For CLAIRE several extensions of our technique are being explored, e.g., decentralized approaches where several systems simultaneously learn to optimize a global reward, or an adaptive targeting system that dynamically adjusts the set of flows for CLAIRE to track. Autonomous agents could take local TE decisions, but by taking enriched data (from a global perspective) into account, these could optimize against global goals. From a security perspective, it would be interesting to understand how many, and if so, how well CLAIRE can classify and predict attacks and anomaly patterns.

X-LANE opens up many avenues for future research, e.g., which parts of an application best benefit from X-LANE, and how to design and optimize coordination protocols accordingly. Other mechanisms that rely on time bounds, e.g., for security [38], are made



possible. We are exploring extensions and refinements of our work such as expanding the endhost implementation, a clock synchronization service, and link fault tolerance, e.g., using redundant paths in our TE algorithm [110]. Note that to this end most existing coordination protocols focus on process crash failures. While some can handle intermittent message losses inherently without retransmissions, most protocols assume reliable channels (usually implemented via TCP), making it unclear what guarantees they provide under link failures. New hardware generations of smartNICs will enable X-LANE to execute more tasks on the NIC directly and thus circumvent the CPU-NIC interconnect and create tighter bounds closer to the optimal latency and jitter.

Bibliography

- [1] M. K. Aguilera, W. Chen, and S. Toueg. “Heartbeat: A timeout-free failure detector for quiescent reliable communication”. In: Springer Distributed Algorithms. 1997, pp. 126–140.
- [2] M. Alizadeh, J. Crowcroft, L. Eggert, and K. Wehrle. “Network Latency Control in Data Centres (Dagstuhl Seminar 16281)”. In: Dagstuhl Reports. 2016, pp. 15–30.
- [3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and et al. “CONGA: Distributed Congestion-Aware Load Balancing for Datacenters”. In: ACM SIGCOMM CCR. 2014, pp. 503–514.
- [4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. “Data Center TCP (DCTCP)”. In: ACM SIGCOMM. 2010, pp. 63–74.
- [5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. “Less is more: Trading a little bandwidth for ultra-low latency in the data center”. In: USENIX NSDI. 2012, pp. 253–266.
- [6] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. “pFabric: Minimal near-Optimal Datacenter Transport”. In: ACM SIGCOMM CCR. 2013, pp. 435–446.
- [7] N. C. Anand, C. Scoglio, and B. Natarajan. “GARCH - non-linear time series model for traffic modeling and prediction”. In: IEEE NOMS. 2008, pp. 694–697.
- [8] *Apache Accumulo*. <https://accumulo.apache.org>.
- [9] *Apache Hadoop*. <https://hadoop.apache.org>.
- [10] *Apache Hadoop Distributed File System*. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
- [11] *Apache HBase*. <http://hbase.apache.org/>.
- [12] *Arista. 7280R Series Data Center Switch Router*. <https://www.arista.com/assets/data/pdf/Datasheets/7280R-DataSheet.pdf>.

-
- [13] *Arista 7150 Series*. https://www.arista.com/assets/data/pdf/Datasheets/7150S_Datasheet.pdf.
- [14] *Arista 7280R Series*. <https://www.arista.com/assets/data/pdf/Datasheets/7280R-DataSheet.pdf>.
- [15] *Arista EOS*. <https://www.arista.com/en/products/eos>.
- [16] N. Aronszajn. “Theory of reproducing kernels”. In: Transactions of the American mathematical society. 1950, pp. 337–404.
- [17] C. N. Babu and B. E. Reddy. “Performance Comparison of Four New ARIMA-ANN Prediction Models on Internet Traffic Data”. In: Journal of Telecommunications and Information Technology. 2015, pp. 67–75.
- [18] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. “Information-Agnostic Flow Scheduling for Commodity Data Centers”. In: USENIX NSDI. 2015, pp. 455–468.
- [19] W. Bai, L. Chen, K. Chen, and H. Wu. “Enabling ECN in Multi-Service Multi-Queue Data Centers”. In: USENIX NSDI. 2016, pp. 537–549.
- [20] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. “IX: A Protected Dataplane Operating System for High Throughput and Low Latency”. In: USENIX OSDI. 2014, pp. 49–65.
- [21] C. H. Benet, A. Kassler, and E. Zola. “Predicting expected TCP throughput using genetic algorithm”. In: Computer Networks. 2016, pp. 307–322.
- [22] T. Benson, A. Akella, and D. A. Maltz. “Network Traffic Characteristics of Data Centers in the Wild”. In: ACM SIGCOMM IMC. 2010, pp. 267–280.
- [23] T. Benson, A. Anand, A. Akella, and M. Zhang. “MicroTE: Fine Grained Traffic Engineering for Data Centers”. In: ACM CoNEXT. 2011, 8:1–8:12.
- [24] T. Benson, A. Anand, A. Akella, and M. Zhang. “Understanding Data Center Traffic Characteristics”. In: ACM SIGCOMM CCR. 2010, pp. 92–99.
- [25] G. Berry and G. Gonthier. “The Esterel synchronous programming language: Design, semantics, implementation”. In: ACM Science of Computer Programming. 1992, pp. 87–152.
- [26] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. “P4: Programming protocol-independent packet processors”. In: ACM SIGCOMM CCR. 2014, pp. 87–95.

-
- [27] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. E. Rodrigues. “On the use of Clocks to Enforce Consistency in the Cloud”. In: *IEEE Data Engineering Bulletin*. 2015, pp. 18–31.
 - [28] R. Buyya, S. N. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. S. Netto, A. N. Toosi, M. A. Rodriguez, I. M. Llorente, S. D. C. D. Vimercati, P. Samarati, D. Milojevic, C. Varela, R. Bahsoon, M. D. D. Assuncao, O. Rana, W. Zhou, H. Jin, W. Gentzsch, A. Y. Zomaya, and H. Shen. “A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade”. In: *ACM Computing Surveys*. 2018, 105:1–105:38.
 - [29] G. Castagnoli, S. Brauer, and M. Herrmann. “Optimization of cyclic redundancy-check codes with 24 and 32 parity bits”. In: 1993, pp. 883–892.
 - [30] S. Chabaa, A. Zeroual, and J. Antari. “ANFIS method for forecasting internet traffic time series”. In: *IEEE Mediterranean Microwave Symposium*. 2009, pp. 1–4.
 - [31] S. Chabaa, A. Zeroual, and J. Antari. “Identification and Prediction of Internet Traffic Using Artificial Neural Networks”. In: *Intelligent Learning Systems and Applications*. 2010, pp. 147–155.
 - [32] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. “On the Impossibility of Group Membership”. In: *ACM PODC*. 2003, pp. 322–330.
 - [33] K. Chen and L. Huang. “Timely-Throughput Optimal Scheduling with Prediction”. In: *IEEE/ACM TON*. 2017, pp. 2457–2470.
 - [34] L. Chen, K. Chen, W. Bai, and M. Alizadeh. “Scheduling Mix-Flows in Commodity Datacenters with Karuna”. In: *ACM SIGCOMM*. 2016, pp. 174–187.
 - [35] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich. “Catching the Microburst Culprits with Snappy”. In: *ACM SIGCOMM SelfDN*. 2018, pp. 22–28.
 - [36] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford. “BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time”. In: *ACM SIGCOMM*. 2020, pp. 226–239.
 - [37] Y. Chen, B. Yang, and Q. Meng. “Small-time scale network traffic prediction based on flexible neural tree”. In: *Applied Soft Computing*. 2012, pp. 274–279.
 - [38] P.-Y. Chen, S. Yang, and J. A. McCann. “Distributed real-time anomaly detection in networked industrial sensing systems”. In: *IEEE Transactions on Industrial Electronics*. 2015, pp. 3832–3842.
 - [39] B. Chin and T. D. Millstein. “An Extensible State Machine Pattern for Interactive Applications”. In: *Springer ECOOP*. 2008, pp. 566–591.

-
- [40] M. Chowdhury and I. Stoica. “Efficient Coflow Scheduling Without Prior Knowledge”. In: ACM SIGCOMM. 2015, pp. 393–406.
- [41] M. Chowdhury, Y. Zhong, and I. Stoica. “Efficient Coflow Scheduling with Varys”. In: ACM SIGCOMM. 2014, pp. 537–549.
- [42] B. Claise, S. Bryant, S. Leinen, T. Dietz, and B. H. Trammell. “Specification of the ip flow information export (IPFIX) protocol for the exchange of ip traffic flow information (rfc 5101)”. In: IETF RFC5101. 2008.
- [43] P. Cortez, M. Rio, M. Rocha, and P. Sousa. “Internet Traffic Forecasting using Neural Networks”. In: IEEE International Joint Conference on Neural Network Proceedings. 2006, pp. 2635–2642.
- [44] *CRC error correction capabilities*. <https://users.ece.cmu.edu/~koopman/crc/index.html>.
- [45] *CRC Layer 2*. <https://standards.ieee.org/standard/802a-2003.html>.
- [46] A. R. Curtis, W. Kim, and P. Yalagandula. “Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection”. In: IEEE INFOCOM. 2011, pp. 1629–1637.
- [47] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. “Paxos Made Switch-y”. In: ACM SIGCOMM CCR. 2016, pp. 18–24.
- [48] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. “NetPaxos: Consensus at Network Speed”. In: ACM SIGCOMM SOSR. 2015, 5:1–5:7.
- [49] S. Daud, R. B. Ahmad, O. B. Lynn, Z. I. A. Kareem, L. M. Kamaruddin, P. Ehkan, M. N. M. Warip, and R. R. Othman. “The effects of cpu load & idle state on embedded processor energy usage”. In: IEEE ICED. 2014, pp. 30–35.
- [50] J. Dean and L. A. Barroso. “The tail at scale”. In: Communications of the ACM. 2013, pp. 74–80.
- [51] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. “PCC: Re-architecting Congestion Control for Consistent High Performance”. In: USENIX NSDI. 2015, pp. 395–408.
- [52] *DPDK: Data Plane Development Kit*. <https://www.dpdk.org>.
- [53] N. Duffield, C. Lund, and M. Thorup. “Estimating Flow Distributions from Sampled Flow Statistics”. In: ACM SIGCOMM. 2003, pp. 325–336.
- [54] V. Đukić, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla. “Is advance knowledge of flow sizes a plausible assumption?” In: USENIX NSDI. 2019, pp. 565–580.

-
- [55] N. Dukkupati and N. McKeown. “Why Flow-completion Time is the Right Metric for Congestion Control”. In: ACM SIGCOMM CCR. 2006, pp. 59–62.
- [56] EdgeCore. *AS5512-54X 10GbE Data Center Switch Bare-Metal Hardware*. https://www.edge-core.com/_upload/images/AS5512-54X_DS_R03_20180614.pdf.
- [57] *EOSSDK*. <https://github.com/aristanetworks/EosSdk>.
- [58] *etcd*. <https://github.com/etcd-io/etcd/>.
- [59] *F16 - Facebook’s topology*. <https://engineering.fb.com/data-center-engineering/f16-minipack/>.
- [60] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. “Hedera: dynamic flow scheduling for data center networks.” In: USENIX NSDI. 2010, pp. 89–92.
- [61] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. “Helios: a hybrid electrical/optical switch architecture for modular data centers”. In: ACM SIGCOMM. 2011, pp. 339–350.
- [62] N. Feamster and J. Rexford. “Why (and How) Networks Should Run Themselves”. In: ACM ANRW. 2018.
- [63] *FEC tutorial*. http://www.ieee802.org/802_tutorials/06-July/10GBASE-KR_FEC_Tutorial_1407.pdf.
- [64] J. Ferreira and V. Menegatto. “Eigenvalues of integral operators defined by smooth positive definite kernels”. In: Springer Integral Equations and Operator Theory. 2009, pp. 61–81.
- [65] M. J. Fischer, N. A. Lynch, and M. S. Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: Journal of the ACM. 1985, pp. 374–382.
- [66] S. J. Garland and N. Lynch. “Using I/O Automata for Developing Distributed Systems”. In: *Foundations of Component-Based Systems*. Cambridge University Press, 2000, pp. 285–312.
- [67] G. H. W. Gebhardt, A. Kupcsik, and G. Neumann. “The kernel Kalman rule”. In: Springer Machine Learning. 2019, pp. 2113–2157.
- [68] D. Gengenbach. *A Distributed Monitoring and Management Framework for Software-Defined Networks*. 2020.
- [69] P. Gill, N. Jain, and N. Nagappan. “Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications”. In: ACM SIGCOMM. 2011, pp. 350–361.

-
- [70] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. “VL2: a scalable and flexible data center network”. In: ACM SIGCOMM. 2009, pp. 51–62.
- [71] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. “Queues Don’T Matter when You Can JUMP Them!” In: USENIX NSDI. 2015, pp. 1–14.
- [72] *gRPC*. <https://grpc.io/>.
- [73] R. Guerraoui. “Non-blocking atomic commit in asynchronous distributed systems with failure detectors”. In: Springer Distributed Computing. 2002, pp. 17–25.
- [74] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. “Sonata: Query-driven Streaming Network Telemetry”. In: ACM SIGCOMM. 2018, pp. 357–371.
- [75] L. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. URL: <http://www.gurobi.com>.
- [76] R. Harrison, Q. Cai, A. Gupta, and J. Rexford. “Network-Wide Heavy Hitter Detection with Commodity Switches”. In: ACM SOSR. 2018, pp. 1–7.
- [77] J. A. Hartigan and M. A. Wong. “Algorithm AS 136: A k-means clustering algorithm”. In: Journal of the Royal Statistical Society. Series C (Applied Statistics). 1979, pp. 100–108.
- [78] J. Herzen, H. Lundgren, and N. Hegde. “Learning wi-fi performance”. In: IEEE SECON. 2015, pp. 118–126.
- [79] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center”. In: USENIX NSDI. 2011, pp. 295–308.
- [80] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. “Achieving High Utilization with Software-driven WAN”. In: ACM SIGCOMM. 2013, pp. 15–26.
- [81] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. “Achieving High Utilization with Software-driven WAN”. In: ACM SIGCOMM CCR. 2013, pp. 15–26.

-
- [82] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, K. N. B., C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev, S. Padgett, F. Rabe, S. Ray, M. Tewari, M. Tierney, M. Zahn, J. Zolla, J. Ong, and A. Vahdat. “B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google’s Software-defined WAN”. In: ACM SIGCOMM. 2018, pp. 74–87.
- [83] 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: IEEE ICAC. 2011, pp. 1199–1204.
- [84] P. K. Hoong, I. K. T. Tan, and C. Y. Keong. *Bittorrent Network Traffic Forecasting With ARMA*. 2012. arXiv: 1208.1896 [cs.NI].
- [85] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao. “OmniMon: Re-Architecting Network Telemetry with Resource Efficiency and Full Accuracy”. In: ACM SIGCOMM. 2020, pp. 404–421.
- [86] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. “ZooKeeper: Wait-free Coordination for Internet-scale Systems.” In: USENIX ATC. 2010.
- [87] *Intel XL710*. <https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf>.
- [88] Z. István, D. Sidler, G. Alonso, and M. Vukolic. “Consensus in a Box: Inexpensive Coordination in Hardware”. In: USENIX NSDI. 2016, pp. 425–438.
- [89] Jaeyeon Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. “Fast portscan detection using sequential hypothesis testing”. In: IEEE Symposium on Security and Privacy. 2004, pp. 211–225.
- [90] 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: ACM SIGCOMM. 2013, pp. 3–14.
- [91] 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: ACM SIGCOMM. 2013, pp. 3–14.
- [92] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. “Silo: Predictable Message Latency in the Cloud”. In: ACM SIGCOMM CCR. 2015, pp. 435–448.
- [93] M. Javed and V. Paxson. “Detecting Stealthy, Distributed SSH Brute-Forcing”. In: ACM CCS. 2013, pp. 85–96.

-
-
- [94] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar. “A deep reinforcement learning perspective on internet congestion control”. In: International Conference on Machine Learning. 2019, pp. 3050–3059.
 - [95] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. “Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility”. In: ACM SIGCOMM. 2014, pp. 3–14.
 - [96] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.
 - [97] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. McKeown, and S. Katti. “High Speed Networks Need Proactive Congestion Control”. In: ACM HotNets. 2015, 14:1–14:7.
 - [98] L. Jose, M. Yu, and J. Rexford. “Online Measurement of Large Traffic Aggregates on Commodity Switches.” In: ACM Hot-ICE. 2011, pp. 13–13.
 - [99] D. Katabi, M. Handley, and C. Rohrs. “Congestion Control for High Bandwidth-delay Product Networks”. In: ACM SIGCOMM. 2002, pp. 89–102.
 - [100] C. Katris and S. Daskalaki. “Comparing Forecasting Approaches for Internet Traffic”. In: Expert System with Applications. Pergamon Press, Inc., 2015, pp. 8172–8183.
 - [101] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. “A client-server oriented algorithm for virtually synchronous group membership in WANs”. In: IEEE Distributed Computing Systems. 2000, pp. 356–365.
 - [102] H. Kellerer, U. Pferschy, and D. Pisinger. “Multidimensional Knapsack Problems”. In: *Knapsack Problems*. Springer, 2004, pp. 235–283.
 - [103] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. “Kinetic: Verifiable dynamic network control”. In: USENIX NSDI. 2015, pp. 59–72.
 - [104] M. Kogias and E. Bugnion. “HovercRAFT: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services”. In: ACM EuroSys. 2020.
 - [105] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. “R2P2: Making RPCs first-class datacenter citizens”. In: USENIX ATC. 2019, pp. 863–880.
 - [106] P. Koopman. “32-bit cyclic redundancy codes for internet applications”. In: IEEE DSN. 2002, pp. 459–468.
 - [107] P. Koopman and T. Chakravarty. “Cyclic redundancy code (CRC) polynomial selection for embedded networks”. In: IEEE DSN. 2004.
 - [108] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: Advances in neural information processing systems. 2012, pp. 1097–1105.

-
-
- [109] M. Kührer, T. Hupperich, C. Rossow, and T. Holz. “Exit from Hell? Reducing the Impact of Amplification DDoS Attacks”. In: USENIX Security. 2014, pp. 111–125.
 - [110] P. Kumar, Y. Yuan, C. Yu, N. Foster, R. Kleinberg, P. Lapukhov, C. L. Lim, and R. Soulé. “Semi-Oblivious Traffic Engineering: The Road Not Taken”. In: USENIX NSDI. 2018, pp. 157–170.
 - [111] L. Lamport. “The Part-Time Parliament”. In: ACM Transactions on Computer Systems. 1998, pp. 133–169.
 - [112] L. Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: Communications of the ACM. 1978, pp. 558–565.
 - [113] L. Lamport. “Using Time Instead of Timeout for Fault-Tolerant Distributed Systems”. In: ACM Transactions on Programming Languages and Systems. 1984.
 - [114] M. Leconte, A. Destounis, and G. Paschos. “Traffic Engineering with Precomputed Pathbooks”. In: IEEE INFOCOM. 2018, pp. 234–242.
 - [115] K. S. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon. “Globally synchronized time via datacenter networks”. In: ACM SIGCOMM. 2016, pp. 454–467.
 - [116] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. “Taming Uncertainty in Distributed Systems with Help from the Network”. In: ACM EuroSys. 2015, 9:1–9:16.
 - [117] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. “Detecting Failures in Distributed Systems with the Falcon Spy Network”. In: ACM SOSP. 2011, pp. 279–294.
 - [118] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. “Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering”. In: USENIX OSDI. 2016, pp. 467–483.
 - [119] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. “Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency”. In: ACM SoCC. 2014, pp. 1–14.
 - [120] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. “HPCC: High Precision Congestion Control”. In: ACM SIGCOMM. 2019, pp. 44–58.
 - [121] Y. Liang and L. Qiu. “Network Traffic Prediction Based on SVR Improved By Chaos Theory and Ant Colony Optimization”. In: International Journal of Future Generation Communication and Networking. 2015, pp. 69–78.
 - [122] C. L. Liu and J. W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: Journal of ACM. 1973, pp. 46–61.


-
- [123] *lp-modeler*. <https://github.com/jcavat/rust-lp-modeler>.
- [124] M. Malboubi, L. Wang, C. Chuah, and P. Sharma. “Intelligent SDN based traffic (de)Aggregation and Measurement Paradigm (iSTAMP)”. In: IEEE INFOCOM. 2014, pp. 934–942.
- [125] N. D. Matsakis and F. S. Klock. “The rust language”. In: ACM SIGAda Ada Letters. 2014.
- [126] R. Meier, T. Holterbach, S. Keck, M. Stähli, V. Lenders, A. Singla, and L. Vanbever. “(Self) Driving Under the Influence: Intoxicating Adversarial Network Inputs”. In: ACM HotNets. 2019, pp. 34–42.
- [127] *Mellanox ConnectX-4*. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_VPI_Card.pdf.
- [128] J. Mieseler. *Network Traffic Prediction in Data Centers*. 2019.
- [129] J. Mieseler. *The Application of Neural Networks for Traffic Prediction in Data Centers*. 2020.
- [130] J. Mirkovic and P. Reiher. “A Taxonomy of DDoS Attack and DDoS Defense Mechanisms”. In: ACM SIGCOMM CCR. 2004, pp. 39–53.
- [131] M. Mitzenmacher and S. Vadhan. “Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream”. In: ACM SODA. 2008, pp. 746–755.
- [132] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout. “Homa: a receiver-driven low-latency transport protocol using network priorities”. In: ACM SIGCOMM. 2018, pp. 221–235.
- [133] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. “DREAM: Dynamic Resource Allocation for Software-Defined Measurement”. In: ACM SIGCOMM. 2014, pp. 419–430.
- [134] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. “Language-Directed Hardware Design for Network Performance Monitoring”. In: ACM SIGCOMM. 2017, pp. 85–98.
- [135] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. “Tierless Programming and Reasoning for Software-defined Networks”. In: USENIX NSDI. 2014, pp. 519–531.
- [136] *Netronome NFP-4000 network processor*. https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_T00.pdf.
- [137] *Network telemetry*. <https://tools.ietf.org/id/draft-song-opsawg-ntf-02.html>.

-
- [138] D. Ongaro and J. Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: USENIX ATC. 2014, pp. 305–319.
- [139] *Open 19*. <https://www.open19.org/>.
- [140] *Open Compute Project*. <https://www.opencompute.org/>.
- [141] *Open Network Linux (ONL)*. <http://opennetlinux.org/>.
- [142] L. K. Organization. *NO_HZ: Reducing Scheduling-Clock Ticks*. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.
- [143] L. K. Organization. *Scaling in the Linux Networking Stack*. <https://www.kernel.org/doc/Documentation/networking/scaling.txt>.
- [144] K. Papagiannaki, N. Taft, S. Bhattacharyya, P. Thiran, K. Salamatian, and C. Diot. “A Pragmatic Definition of Elephants in Internet Backbone Traffic”. In: ACM SIGCOMM IMW. 2002, pp. 175–176.
- [145] D. C. Park and D. M. Woo. “Prediction of Network Traffic Using Dynamic Bilinear Recurrent Neural Network”. In: IEEE ICNC. 2009, pp. 419–423.
- [146] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. “Fastpass: A Centralized “Zero-queue” Datacenter Network”. In: ACM SIGCOMM. 2014, pp. 307–318.
- [147] P. Phaal, S. Panchen, and N. McKee. “InMon Corporation’s sFlow: A Method for Monitoring Traffic in a Switched and Routed Networks”. In: IETF RFC 3176. 2001.
- [148] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. “Designing Distributed Systems Using Approximate Synchrony in Data Center Networks”. In: USENIX NSDI. 2015, pp. 43–57.
- [149] G. Prekas, M. Kogias, and E. Bugnion. “ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks”. In: ACM SOSP. 2017, pp. 325–341.
- [150] *RabbitMQ*. <https://www.rabbitmq.com/>.
- [151] L. Ralaivola and F. d’Alche-Buc. “Time series filtering, smoothing and learning using the kernel Kalman filter”. In: IEEE International Joint Conference on Neural Networks. 2005, pp. 1449–1454.
- [152] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. “Planck: Millisecond-scale monitoring and control for commodity networks”. In: ACM SIGCOMM. 2014, pp. 407–418.
- [153] J. Ray and P. Koopman. “Efficient high hamming distance CRCs for embedded networks”. In: IEEE DSN. 2006, pp. 3–12.

-
- [154] *Redis*. <https://redis.io/>.
- [155] F. Reghenzani, G. Massari, and W. Fornaciari. “The Real-time Linux Kernel: A Survey On Preempt_RT”. In: ACM Computing Surveys. 2019, pp. 1–36.
- [156] V. Riesop. *Jitter Resilient Linux NetworkStack for Failure Detectors of Distributed Systems*. 2019.
- [157] L. S. Sabel and K. Marzullo. *Election Vs. Consensus in Asynchronous Systems*. Tech. rep. 1995.
- [158] G. Sanjit and C. Ted. “Magic Quadrant for Network Performance Monitoring and Diagnostics”. In: Gartner Research ID G. 2019.
- [159] P. Schnoebelen. “Verifying lossy channel systems has nonprimitive recursive complexity”. In: Elsevier Information Processing Letters. 2002, pp. 251–261.
- [160] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. “Reversible Sketches for Efficient and Accurate Change Detection over Network Data Streams”. In: ACM SIGCOMM IMC. 2004, pp. 207–212.
- [161] D. Senecal. *Slow DoS on the rise*. <https://blogs.akamai.com/2013/09/slow-dos-on-the-rise.html>.
- [162] D. Shan and F. Ren. “ECN Marking With Micro-Burst Traffic: Problem, Analysis, and Improvement”. In: IEEE/ACM Transactions on Networking. 2018, pp. 1533–1546.
- [163] G. Shang, P. Zhe, X. Bin, H. Aiqun, and R. Kui. “FloodDefender: Protecting data and control plane resources under SDN-aimed DoS attacks”. In: IEEE INFOCOM. 2017, pp. 1–9.
- [164] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. “Heavy-Hitter Detection Entirely in the Data Plane”. In: ACM SOSR. 2017, pp. 164–176.
- [165] E. Stapf. *Predicting Traffic Flows for Traffic Engineering in Software-Defined Networks*. 2016.
- [166] *Stratum*. <https://www.opennetworking.org/stratum/>.
- [167] M. Sundermeyer, R. Schlüter, and H. Ney. “LSTM neural networks for language modeling”. In: International Speech Communication Association. 2012.
- [168] P. Tammana, R. Agarwal, and M. Lee. “Simplifying Datacenter Network Debugging with PathDump”. In: USENIX OSDI. 2016, pp. 233–248.
- [169] *The Network Simulator - ns-2*. <https://www.isi.edu/nsnam/ns/>.

-
- [170] T. N. Theis and H.-S. P. Wong. “The end of moore’s law: A new beginning for information technology”. In: IEEE Computing in Science & Engineering. 2017, pp. 41–50.
- [171] TiKV. <https://github.com/tikv/tikv/>.
- [172] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar. “Learning to Route”. In: ACM HotNets. 2017, pp. 185–191.
- [173] A. Van Bemten, N. Đerić, A. Varasteh, S. Schmid, C. Mas-Machuca, A. Blenk, and W. Kellerer. “Chameleon: predictable latency and high utilization with queue-aware and adaptive source routing”. In: ACM CoNEXT. 2020, pp. 451–465.
- [174] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. “Apache Hadoop Yarn: Yet Another Resource Negotiator”. In: ACM SOCC. 2013, pp. 1–16.
- [175] P. Verissimo and A. Casimiro. “The timely computing base model and architecture”. In: IEEE Transactions on Computers. 2002, pp. 916–930.
- [176] P. E. Veri ssimo. “Travelling Through Wormholes: A New Look at Distributed Systems Models”. In: ACM SIGACT News. 2006, pp. 66–81.
- [177] B. Vujicic, H. Chen, and L. Trajkovic. “Prediction of traffic in a public safety network”. In: IEEE International Symposium on Circuits and Systems. 2006, 4 pp.-.
- [178] M. Wang, Y. Cui, S. Xiao, X. Wang, D. Yang, K. Chen, and J. Zhu. “Neural Network Meets DCN: Traffic-driven Topology Adaptation with Deep Learning”. In: ACM Measurement and Analysis of Computing Systems. 2018, 26:1–26:25.
- [179] P. Welch. “The use of fast Fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms”. In: IEEE Transactions on audio and electroacoustics. 1967, pp. 70–73.
- [180] S. Yang, H. Yan, Z. Ge, D. Wang, and J. Xu. “Predictive Impact Analysis for Designing a Resilient Cellular Backhaul Network”. In: ACM SIGMETRICS. 2019, pp. 84–86.
- [181] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. “Hotstuff: Bft consensus with linearity and responsiveness”. In: ACM PODC. 2019, pp. 347–356.
- [182] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha. “FlowSense: Monitoring Network Utilization with Zero Measurement Cost”. In: Springer Conference on Passive and Active Measurement. 2013, pp. 31–41.
- [183] M. Yu, L. Jose, and R. Miao. “Software Defined Traffic Measurement with OpenSketch”. In: USENIX NSDI. 2013, pp. 29–42.

-
-
- [184] Y. Yu, M. Song, Y. Fu, and J. Song. “Traffic prediction in 3G mobile networks based on multifractal exploration”. In: Tsinghua Science and Technology. 2013, pp. 398–405.
- [185] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo. “Quantitative Network Monitoring with NetQRE”. In: ACM SIGCOMM. 2017, pp. 99–112.
- [186] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. “Resilient Distributed Datasets: a Fault-Tolerant Abstraction for In-memory Cluster Computing”. In: USENIX NSDI. 2012.
- [187] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. “Discretized Streams: Fault-tolerant Streaming Computation at Scale”. In: ACM SOSP. 2013, pp. 423–438.
- [188] J. H. Zeng. *Data sharing on traffic pattern inside Facebook’s datacenter network*. <https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/>. 2017.
- [189] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng. “Coda: Toward automatically identifying and scheduling coflows in the dark”. In: ACM SIGCOMM. 2016, pp. 160–173.
- [190] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. “High-Resolution Measurement of Data Center Microbursts”. In: ACM SIGCOMM IMC. 2017, pp. 78–85.
- [191] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, Y. Zhang, Y. Chen, H. Dong, X. Qu, and L. Song. “PreFix: Switch Failure Prediction in Datacenter Networks”. In: ACM Measurement and Analysis of Computing Systems. 2018, 2:1–2:29.
- [192] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund. “Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications”. In: ACM SIGCOMM. 2004, pp. 101–114.
- [193] Y. Zhang. “An Adaptive Flow Counting Method for Anomaly Detection in SDN”. In: ACM CoNEXT. 2013, pp. 25–30.
- [194] Y. Zhou, D. Zhang, K. Gao, C. Sun, J. Cao, Y. Wang, M. Xu, and J. Wu. “Newton: Intent-Driven Network Traffic Monitoring”. In: ACM CoNEXT. 2020, pp. 295–308.
- [195] P. Zhu, B. Chen, and J. C. Principe. “Learning Nonlinear Generative Models of Time Series With a Kalman Filter in RKHS”. In: IEEE Transactions on Signal Processing. 2014, pp. 141–155.

-
- 
-
- [196] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye. “ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY”. In: ACM CoNEXT. 2016, pp. 313–327.
- [197] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. “Packet-Level Telemetry in Large Datacenter Networks”. In: ACM SIGCOMM. 2015, pp. 479–491.