



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

**PROGRAMMING MODELS AND  
EXTENSIVE EVALUATION SUPPORT  
FOR MPTCP SCHEDULING, ADAPTATION  
DECISIONS, AND DASH VIDEO STREAMING**

Am Fachbereich Informatik  
der Technischen Universität Darmstadt  
zur Erlangung des akademischen Grades eines  
Doktor-Ingenieurs (Dr.-Ing.)  
genehmigte Dissertationsschrift

von

**ALEXANDER FRÖMMGEN, M.SC.**  
Geboren am 29. März 1988 in Koblenz

Erstreferent: Prof. Dr.-Ing. Ralf Steinmetz  
Korreferent: Prof. Alejandro Buchmann (Ph.D.)

Tag der Einreichung: 30.04.2018  
Tag der Disputation: 18.06.2018

Hochschulkennziffer D17  
Darmstadt 2018

Frömmgen, Alexander: Programming Models and Extensive Evaluation Support for  
MPTCP Scheduling, Adaptation Decisions, and DASH Video Streaming

Darmstadt, Technische Universität Darmstadt,  
Jahr der Veröffentlichung der Dissertation auf TUPrints: 2018  
Tag der mündlichen Prüfung: 18.06.2018

Veröffentlichung unter CC BY-SA 4.0 International  
<https://creativecommons.org/licenses/>

## ACKNOWLEDGMENTS

Ich möchte mich an dieser Stelle bei einigen Menschen bedanken. Zunächst möchte ich mich bei meinen beiden Betreuern Prof. Buchmann und Prof. Steinmetz für die uneingeschränkte Unterstützung in den vergangenen Jahren bedanken. Beide gaben mir die Möglichkeit und Freiheit, meine Ideen umzusetzen. Ich bin Prof. Steinmetz dankbar dafür, dass ich meine begonnenen Arbeiten mit Sorgfalt und Geduld abschließen durfte.

Ich möchte mich bei Wolfgang und Jörg dafür bedanken, dass den beiden schon immer klar war, dass ich eines Tages promovieren würde.

Des Weiteren möchte ich mich bei all meinen Kollegen  $k \in (\text{DVS} \cup \text{KOMU} \cup \text{MAKI})$  bedanken. Die gemeinsamen Arbeiten mit Björn, Christian, Denny, Julius, Martin, Max, Michael, Robert, Roland, Sabrina, Torsten und Wasiur haben mir viel Freude und spannende Einsichten bereitet. Besonderer Dank gilt Max und Robert, die mich bei meinen ersten akademischen Schritten begleiteten. Ich möchte mich für die tolle Zusammenarbeit mit Michael bedanken – auch wenn deswegen dann einiges nochmal umgeschrieben werden musste. In diesem Zusammenhang möchte ich mich auch bei Roland bedanken. Beide haben erfolgreich ein paar Kanten in meinem Hirn verbunden.

Ich möchte mich bei Denny und Prof. Effelsberg für die Gastfreundschaft im DMS Büro und die vielen spannenden Video-Streaming-Einsichten bedanken. Prof. Effelsberg, vielen Dank für Ihr Vertrauen und Ihre Erwartungen in meine Arbeit. Dies hat mich stets angespornt.

Ich möchte mich bei Boris für die gute Betreuung und Unterstützung bedanken. Danke, dass du mir zur richtigen Zeit Konferenzen jenseits der Kommunikationssysteme aufgezeigt hast. Ich möchte mich bei Amr für das permanente Umformulieren von Absätzen bedanken. Hast ja recht.

Des Weiteren möchte ich mich bei den vielen Studierenden bedanken, welche meine Forschung über die Jahre unterstützt und unter meinen unausgegorenen Ideen zu leiden hatten. Besonderer Dank geht an Andreas, Jan, Max, Nikolas, Sreeram, Stefan, Tobias und Tobias.

Ich möchte mich bei den Feistels für die vielen qualvollen Runden bedanken. Tja, Christopher, jetzt warst du doch (schneller  $\vee$  besser).

Zu guter Letzt möchte ich mich bei meiner Familie für all die Unterstützung bedanken. Danke Mutter. Danke Vater. Danke Schwester. Danke Martin. Danke Noah. Danke Mauz. Danke Krümel.



## ABSTRACT

---

In this dissertation, we identify that the analysis, implementation, and evaluation of communication systems is hindered by two obstacles: *i)* missing abstractions and the resulting implementation complexity and *ii)* the required extensive evaluations for today’s large configuration spaces and heterogeneous network environments. A prominent example is *Multipath TCP* – today’s de facto multipathing transport protocol. Multipath TCP *packet scheduler* innovations are hindered by the implementation complexity of the Linux kernel network stack and the required analysis for a variety of applications and network conditions.

To tackle the first obstacle, we are the first to propose a programming model as abstraction for the design and development of Multipath TCP schedulers. We introduce the *ProgMP* programming model, which provides a powerful specification language and a high-level API to specify executable Multipath TCP schedulers. We show the strength of *ProgMP* by implementing 13 novel general purpose, preference-, and application-aware schedulers tackling diverse objectives. As part of these schedulers, we propose the first *redundant* Multipath TCP scheduler and show that this scheduler significantly reduces latency for applications with tight latency requirements but only moderate throughput needs. We use *ProgMP* for a detailed analysis of design decisions for the incorporation of redundancy to balance latency and throughput. We further propose schedulers that retain fine-grained throughput or latency objectives, or improve the interaction with upper layer protocols such as HTTP/2, while preserving path preferences. Our detailed emulation-based and real world measurements show that *ProgMP* enables timely scheduling decisions and a wide range of executable, novel Multipath TCP schedulers. Besides *ProgMP*, which is our main contribution to overcome the first obstacle of missing abstractions, we further introduce programming models as abstraction for the *adaptation decisions* of adaptive communication systems. Here, we propose to specify the adaptation decision with *event condition action* rules and learn rules for a given utility function with *genetic programming* in extensive network experiments. Finally, we propose a programming model for the specification of *topology adaptations* in communication systems based on *topology graph patterns*.

To overcome the second identified obstacle and foster extensive evaluations, we present the *MACI* framework for the management, scalable execution, and interactive analysis of extensive network experiments. In essence, *MACI* is a combination and integration of established tools to foster rigorous, seamless evaluations throughout the research process. We discuss our *MACI* experiences during *i)* the development and evaluation of our proposed *ProgMP* schedulers, *ii)* the analysis of a distributed topology graph pattern matching protocol, and *iii)* a systematic comparison of DASH video streaming implementations. Our experiences confirm that *MACI* provides support

for the recurring tasks in the evaluation of diverse communication systems and significantly increases research efficiency. The experiments with *MACI*, i. e., the *ProgMP*, the topology graph pattern matching, and the DASH experiments, go beyond an evaluation of *MACI* and significantly contribute to the understanding of these domains.

Overall, this dissertation contributes *i)* three programming models for the domains of Multipath TCP, adaptive communication systems, and topology adaptations in communication systems, *ii)* 13 novel, deployable general purpose, preference-, and application-aware Multipath TCP schedulers, and *iii)* a reusable framework for the seamless execution and analysis of extensive network experiments.

## ZUSAMMENFASSUNG

---

In dieser Dissertation zeigen wir auf, dass die Analyse, die Umsetzung und die Evaluation von Kommunikationssystemen durch *i)* fehlende Abstraktionen und die resultierende Implementierungskomplexität sowie *ii)* die benötigten umfassenden Evaluationen für die Vielzahl an Konfigurationsmöglichkeiten und Netzwerkumgebungen erschwert werden. Multipath TCP, das de facto Transportprotokoll zur Nutzung mehrerer Netzwerkpfade, stellt ein prominentes Beispiel dar. Innovationen im Bereich der *Multipath TCP Paket Scheduler* werden durch die Implementierungskomplexität innerhalb des Linux-Kernels und den benötigten umfassenden Analysen für die Vielzahl an Anwendungen und Netzwerkumgebungen erschwert.

Zur Überwindung des ersten Hindernisses schlagen wir *ProgMP*, das erste Programmiermodell für Multipath TCP Scheduler, als Abstraktion für den Entwurf und die Entwicklung von Multipath TCP Schemulern vor. *ProgMP* beinhaltet eine ausdrucksstarke Spezifikationsprache und eine einfache Programmierschnittstelle zur Spezifikation ausführbarer Multipath TCP Scheduler. Wir zeigen die Stärken von *ProgMP* am Beispiel von 13 neuen Schemulern mit unterschiedlichen Optimierungszielen auf. Als Teil dieser Scheduler schlagen wir den ersten *redundanten* Multipath TCP Scheduler vor und zeigen, dass dieser die Latenz für Anwendungen mit strikten Latenzanforderungen und moderaten Durchsatzanforderungen signifikant reduziert. Wir nutzen *ProgMP* für eine detaillierte Analyse von Entwurfsentscheidungen für die Verwendung von Redundanz zur Abwägung von Latenz und Durchsatz. Des Weiteren schlagen wir Scheduler vor, welche feingranulare Durchsatz- oder Latenzziele einhalten, sowie die Interaktion mit darüber liegenden Protokollen wie beispielsweise HTTP/2 optimieren und dabei Pfadpräferenzen einhalten. Unsere detaillierten Evaluationen mittels Netzwerkemulation sowie Echtweltmessungen zeigen, dass *ProgMP* effiziente Schedulingentscheidungen und eine Vielzahl neuer, ausführbarer Multipath TCP Scheduler ermöglicht. Neben *ProgMP*, unserem Hauptbeitrag zur Überwindung fehlender Abstraktionen, stellen wir des Weiteren Programmiermodelle als Abstraktionen für die Adaptionentscheidung adaptiver Kommunikationssysteme vor. So schlagen wir vor, Adaptionentscheidungen mittels *event condition action* Regeln (Ereignis, Bedingung, Aktion) zu spezifizieren und diese Regeln basierend auf genetischer Programmierung in umfassenden Netzwerkexperimenten automatisch für eine gegebene Nutzenfunktion zu lernen. Schließlich stellen wir ein Programmiermodell für die Spezifikation von Topologie-Adaptionen in Kommunikationssystemen mittels *Graphmustern in der Topologie* vor.

Zur Überwindung des zweiten identifizierten Hindernisses haben wir das *MACI* Framework zur Verwaltung, skalierbaren Ausführung und interaktiven Analyse von umfassenden Netzwerkexperimenten entwickelt. Im Kern ist *MACI* eine Kombination und Integration etablierter Werkzeuge zur Förderung gründlicher, nahtloser Evaluationen während des gesamten Forschungs-

prozesses. Wir diskutieren unsere Erfahrungen mit *MACI* während *i)* der Entwicklung und Evaluation unserer vorgeschlagenen *ProgMP* Scheduler, *ii)* der Analyse eines verteilten Protokolls zur Auffindung von Graphmustern in Topologien, sowie *iii)* eines systematischen Vergleichs von DASH Video Streaming Implementierungen. Unsere Erfahrungen bestätigen, dass *MACI* wiederkehrende Aufgaben in der Evaluation diverser Kommunikationssysteme unterstützt und dadurch die Forschungseffizienz signifikant erhöht. Die Experimente mit *MACI* für *ProgMP*, Topologiemustererkennung und DASH gehen über eine Evaluation von *MACI* hinaus und stellen signifikante Beiträge zum Verständnis der jeweiligen Gebiete dar.

Insgesamt beinhaltet diese Dissertation die folgenden Beiträge: *i)* drei Programmiermodelle für die Bereiche des Multipath TCP Scheduling, der adaptiven Kommunikationssysteme, sowie der Topologie-Adaption in Kommunikationssystemen, *ii)* 13 neue, ausführbare Multipath TCP Scheduler, sowie *iii)* ein wiederverwendbares Framework zur nahtlosen Ausführung und Analyse umfassender Netzwerexperimente.

## AUTHOR'S PUBLICATIONS

---

The author of this dissertation published major parts of the presented content of this dissertation previously in the following publications:

### Major Publications

- [F1] Alexander Frömmgen, Jens Heuschkel, and Boris Koldehofe. “Multipath TCP Scheduling for Thin Streams: Active Probing and One-way Delay-awareness”. In: *Proceedings of the IEEE International Conference on Communications (ICC)*. 2018.
- [F2] Alexander Frömmgen, Denny Stohr, Amr Rizk, and Boris Koldehofe. *Don't Repeat Yourself: Seamless Execution and Analysis of Extensive Network Experiments*. Tech. rep. 2018. URL: <https://maci-research.net>.
- [F3] Michael Stein, Alexander Frömmgen, Roland Kluge, Wang Lin, Augustin Wilberg, Boris Koldehofe, and Max Mühlhäuser. “Scaling Topology Pattern Matching: A Distributed Approach”. In: *Proceedings of the ACM/SIGAPP Symposium on Applied Computing (SAC)*. 2018.
- [F4] Tobias Viernickel, Alexander Frömmgen, Amr Rizk, Boris Koldehofe, and Ralf Steinmetz. “Multipath QUIC: A Deployable Multipath Transport Protocol”. In: *Proceedings of the IEEE International Conference on Communications (ICC)*. 2018.
- [F5] Alexander Frömmgen, Amr Rizk, Tobias Erbschäuber, Max Weller, Boris Koldehofe, Alejandro Buchmann, and Ralf Steinmetz. “A Programming Model for Application-defined Multipath TCP Scheduling”. In: *Proceedings of the ACM/FIP/USENIX Middleware Conference, Best Paper Award*. ACM, 2017, pp. 134–146. URL: <https://progmp.net>.
- [F6] Denny Stohr, Alexander Frömmgen<sup>1</sup>, Amr Rizk, Michael Zink, Ralf Steinmetz, and Wolfgang Effelsberg. “Where are the Sweet Spots?: A Systematic Approach to Reproducible DASH Player Comparisons”. In: *Proceedings of the ACM Conference on Multimedia (MM)*. 2017, pp. 1113–1121. URL: <https://maci-research.net/dash>.
- [F7] Alexander Frömmgen, Tobias Erbschäuber, Torsten Zimmermann, Klaus Wehrle, and Alejandro Buchmann. “ReMP TCP: Low Latency Multipath TCP”. In: *Proceedings of the IEEE International Conference on Communications (ICC)*. Idea proposed in CoNEXT'15 Student Workshop. 2016.

---

<sup>1</sup>The two first authors contributed equally to this work.

- [F8] Michael Stein, Alexander Frömmgen, Roland Kluge, Frank Löffler, Andy Schürr, Alejandro Buchmann, and Max Mühlhäuser. “TARL: Modeling Topology Adaptations for Networking Applications”. In: *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM. 2016, pp. 57–63.
- [F9] Denny Stohr, Alexander Frömmgen, Jan Fornoff, Michael Zink, Alejandro Buchmann, and Wolfgang Effelsberg. “QoE Analysis of DASH Cross-Layer Dependencies by Extensive Network Emulation”. In: *Proceedings of the SIGCOMM Workshop on QoE-based Analysis and Management of Data Communication Networks (Internet-QoE)*. ACM, 2016, pp. 25–30.
- [F10] Alexander Frömmgen, Jens Heuschkel, Patrick Jahnke, Fabio Cuzzo, Immanuel Schweizer, Patrick Eugster, Max Mühlhäuser, and Alejandro Buchmann. “Crowdsourcing Measurements of Mobile Network Performance and Mobility During a Large Scale Event”. In: *Proceedings of the International Conference on Passive and Active Network Measurement (PAM)*. Springer International Publishing. 2016, pp. 70–82.
- [F11] Wasiur R. KhudaBukhsh, Amr Rizk, Alexander Frömmgen, and Heinz Koepl. “Optimizing Stochastic Scheduling in Fork-Join Queuing Models: Bounds and Applications”. In: *Proceedings of the IEEE INFOCOM*. 2017.
- [F12] Alexander Frömmgen, Robert Rehner, Max Lehn, and Alejandro Buchmann. “Fossa: Using Genetic Programming to Learn ECA Rules for Adaptive Networking Applications”. In: *Proceedings of the Local Computer Networks (LCN)*. IEEE. 2015, pp. 197–200.
- [F13] Alexander Frömmgen, Robert Rehner, Max Lehn, and Alejandro Buchmann. “Fossa: Learning ECA Rules for Adaptive Distributed Systems”. In: *Proceedings of the International Conference on Autonomic Computing (ICAC)*. IEEE. 2015, pp. 207–210.
- [F14] Alexander Frömmgen, Björn Richerzhagen, Julius Rückert, David Hausheer, Ralf Steinmetz, and Alejandro Buchmann. “Towards the Description and Execution of Transitions in Networked Systems”. In: *Proceedings of the IFIP International Conference on Autonomous Infrastructure, Management and Security (AIMS)*. Springer International Publishing. 2015, pp. 17–29.

## Additional Publications

- [F15] Alexander Frömmgen and Boris Koldehofe. “Demo: Programming Application-defined Multipath TCP Schedulers”. In: *Proceedings of the ACM/IFIP/USENIX Middleware Conference: Posters and Demos*. ACM, 2017, pp. 13–14.
- [F16] Alexander Frömmgen. *Mininet/Netem Emulation Pitfalls: A Multipath TCP Scheduling Experience*. Tech. rep. 2017. URL: <https://progmp.net/MininetPitfalls.pdf>.
- [F17] Christian Krupitzer, Julian Otto, Fabian Roth, Alexander Frömmgen, and Christian Becker. “Adding Self-improvement to an Autonomic Traffic Management System”. In: *Proceedings of the International Conference on Autonomic Computing (ICAC)*. IEEE. 2017.
- [F18] Alexander Frömmgen, Stefan Haas, Martin Pfannemüller, and Boris Koldehofe. “Switching ZooKeeper’s Consensus Protocol at Runtime”. In: *Proceedings of the International Conference on Autonomic Computing (ICAC) Poster Track*. 2017.
- [F19] Alexander Frömmgen, Denny Stohr, Jan Fornoff, Wolfgang Efelsberg, and Alejandro Buchmann. “Demo: Capture and Replay: Reproducible Network Experiments in Mininet”. In: *Proceedings of the Conference of the Special Interest Group on Data Communication (SIGCOMM)*. ACM. 2016, pp. 621–622.
- [F20] Jens Heuschkel, Alexander Frömmgen, Jon Crowcroft, and Max Mühlhäuser. “VirtualStack: Adaptive Multipath Support through Protocol Stack Virtualization”. In: *Proceedings of the International Network Conference (INC)*. Lulu.com. 2016, p. 73.
- [F21] Alexander Frömmgen, Sreeram Sadasivam, Sabrina Müller, Anja Klein, and Alejandro Buchmann. “Poster: Use Your Senses: A Smooth Multipath TCP WiFi/Mobile Handover”. In: *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM. 2015, pp. 248–250.
- [F22] Alexander Frömmgen, Patrick Wagner, and Alejandro Buchmann. “Simulation-based Retrieval of Adaptation Knowledge”. In: *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT) Student Workshop*. ACM. 2015.
- [F23] Alexander Frömmgen, Stefan Haas, Michael Stein, Robert Rehner, Max Mühlhäuser, and Alejandro Buchmann. “Always the Best: Executing Transitions between Search Overlays”. In: *Proceedings of the European Conference on Software Architecture Workshops*. ACM. 2015, pp. 1–4.

- [F24] Alexander Frömmgen, Max Lehn, and Alejandro Buchmann. “A Property Description Framework for Composable Software”. In: *Proceedings of the European Conference on Software Architecture (ECSA)*. Springer International Publishing, 2014, pp. 267–282.

## AUTHOR'S PATENTS

---

- [1] Alexander Frömmgen. *Method and System for Data Compression*. US Pub. No.: US 2012/0323876. 2011.
- [2] Alexander Frömmgen. *Method and System for Inverted Indexing of a Dataset*. US Pub. No.: US 2012/0323927. 2011.
- [3] Alexander Frömmgen, Sabrina Müller, Sreeram Sadasivam, Anja Klein, Alejandro Buchmann, Max Lehn, and Robert Rehner. *Verfahren zur Aufrechterhaltung der Performance einer Multipath-TCP-Verbindung*. DE102015114164 A1 - Application. 2015.
- [4] Alin Jula, Jan Carstens, and Alexander Frömmgen. *Memory-Aware Scheduling for NUMA Architectures*. US Pub. No.: US 2012/0174117. 2010.



## CONTENTS

---

1	INTRODUCTION	1
1.1	Motivation . . . . .	1
1.2	Problem Statement . . . . .	3
1.3	Approach and Contributions . . . . .	4
1.4	Thesis Organization . . . . .	7
2	BACKGROUND AND RELATED WORK ON MPTCP	9
2.1	Multipath TCP . . . . .	9
2.1.1	Connection Establishment and Wire Protocol Format	10
2.1.2	Path Management . . . . .	12
2.1.3	Implementations . . . . .	13
2.2	Multipath Scheduling . . . . .	14
2.2.1	RFCs on MPTCP Scheduling . . . . .	14
2.2.2	Deployable Multipath TCP Schedulers . . . . .	15
2.2.3	Multipath TCP Schedulers in Simulators . . . . .	17
2.2.4	Multipath TCP Related Schedulers . . . . .	17
2.2.5	Multipathing for SCTP . . . . .	20
2.2.6	Singlepath Scheduling . . . . .	21
2.2.7	MPTCP Scheduling Dependencies . . . . .	21
2.2.8	Discussion . . . . .	21
3	A PROGRAMMING MODEL FOR MPTCP SCHEDULING	23
3.1	Requirement Derivation . . . . .	23
3.2	Programming Models in Communication Systems . . . . .	26
3.3	Programming Model Design . . . . .	28
3.3.1	Model of the Scheduling Environment . . . . .	28
3.3.2	Language Design . . . . .	29
3.3.3	Scheduler Triggering and Execution . . . . .	32
3.3.4	API for Application-aware Scheduling . . . . .	32
3.3.5	Programming Model Discussion and Future Work . . . . .	33
3.4	Execution Environment Implementation . . . . .	36
3.4.1	Scheduler Location and Calling Model . . . . .	36
3.4.2	Runtime Environment . . . . .	37
3.4.3	Runtime Optimizations and Compilation . . . . .	40
3.4.4	API for Application-aware Scheduling . . . . .	41
3.4.5	Testing and Evaluation . . . . .	42
3.4.6	Receiver-Side Packet Handling . . . . .	44
3.4.7	Implementation Discussion and Future Work . . . . .	47
4	ENABLING EXTENSIVE NETWORK EXPERIMENTS	49
4.1	Motivation . . . . .	49
4.2	Observations and Requirement Analysis . . . . .	50
4.3	Experiment-Driven Research Process . . . . .	52
4.3.1	A Single Executable Experiment Instance . . . . .	53
4.3.2	Structuring Experiments . . . . .	54

4.3.3	Interactive Data Analysis for Extensive Experiments	54
4.4	Implementation	55
4.5	Related Work	59
4.6	MACI Experiences	60
4.7	Discussion and Future Work	62
5	DESIGN AND ANALYSIS OF NOVEL MPTCP SCHEDULERS	63
5.1	Evaluation Setup and Scenarios	63
5.2	Revisiting Existing Schedulers	66
5.2.1	Preamble and Reinjection Queue Handling	66
5.2.2	(Default) Minimum RTT Scheduler	66
5.2.3	Round Robin Scheduler	68
5.2.4	Redundant Scheduler	69
5.2.5	Discussion	70
5.3	Active Probing for Thin Streams	71
5.4	Exploring Redundancy	77
5.5	Signaling to Boost Short Flows	83
5.6	Balancing Round-trip Times and Subflow Preferences	86
5.7	Balancing Throughput and Subflow Preference	89
5.8	One-way Delay-Aware Scheduling	93
5.9	Towards HTTP/2-aware Scheduling	98
5.10	Pitfalls in Emulation	102
5.10.1	Using a Fresh Network	102
5.10.2	Impact of Netem on the Network Stack	103
5.10.3	Changing the Network at Runtime	105
5.11	MACI Perspective	106
5.12	ProgMP Perspective	108
5.13	Discussion and Future Work	110
6	A PROGRAMMING MODEL FOR ADAPTATION DECISIONS	111
6.1	Motivation and Approach	111
6.2	Background and Related Work	112
6.3	Specifying Adaptation Decisions with ECA Rules	114
6.4	Learning ECA Rules	114
6.4.1	Exploration Strategy: Genetic Programming	116
6.4.2	Efficiency Improvements	117
6.5	Transition Description and Execution	118
6.6	Evaluation	118
6.7	Discussion and Future Work	120
7	A PROGRAMMING MODEL FOR TOPOLOGY ADAPTATIONS	123
7.1	Motivation and Approach	123
7.2	Topology Adaptation Rule Language TARL	124
7.3	Expressiveness Evaluation	125
7.4	Distributed Topology Pattern Matching	128
7.4.1	Exploring Distributed Topology Pattern Matching	128
7.4.2	MACI Perspective	130
7.5	Discussion and Future Work	131
8	EXTENSIVE DASH VIDEO PLAYER COMPARISON	133

8.1	Motivation and Background . . . . .	133
8.2	Experimental Design Approach . . . . .	134
8.3	DASH Analysis Overview . . . . .	134
8.4	MACI Perspective . . . . .	137
8.5	Discussion and Future Work . . . . .	138
9	DISCUSSION AND FUTURE WORK . . . . .	139
9.1	Programming Models for Communication Systems . . . . .	140
9.2	Extensive Network Experiments . . . . .	141
9.3	Future Work . . . . .	141
	Bibliography . . . . .	145
A	APPENDICES . . . . .	171
A.1	Illustrating MACI Visualizations . . . . .	171
A.2	Additional ProgMP Experiences . . . . .	173
A.3	Packetdrill Testscript for the ProgMP Implementation . . . . .	175
A.4	Presented ProgMP Schedulers . . . . .	177
A.5	ProgMP Language Syntax . . . . .	187
A.6	MPTCP Scheduler Commits . . . . .	192
B	ERKLÄRUNG LAUT §9 DER PROMOTIONSORDNUNG . . . . .	195
C	WISSENSCHAFTLICHER WERDEGANG DES VERFASSERS . . . . .	197



INTRODUCTION

1.1 MOTIVATION

Communication systems have to operate in an increasing variety of network environments and conditions. Applications deployed on the Internet, e. g., have to operate in advanced cellular networks with high throughput and low latency as well as established legacy networks with inferior characteristics. Furthermore, networking applications have to cope with suddenly changing network conditions, e. g., due to network congestion and overload during large scale events [F10]<sup>1</sup>. At the same time, the variety of applications and their individual requirements increases. For example, interactive networking applications expose fundamentally different requirements than on demand video streaming or traditional web browsing on the underlying network. The interaction of the various application requirements and today’s heterogeneous network environments increases the complexity of developing and improving today’s communication protocols and systems.

*The wide range of network environments...*

*...and application requirements...*

*... increases complexity.*

To flexibly fulfill requirements of different applications for varying environments, communication systems are usually composed of components that implement specialized mechanisms and functionality (Figure 1.1). The decomposition of the required functionality proved to be a powerful abstraction for research and development of communication systems. Research and development on these specialized mechanisms, however, remains challenging in the face of complex dependencies between the involved components, the application requirements, and the exposed network conditions. Novel mechanisms and their configurations have to be carefully evaluated for all anticipated applications and network conditions.

*Abstractions...*

*... and systematic evaluations are required to cope with the complexity.*

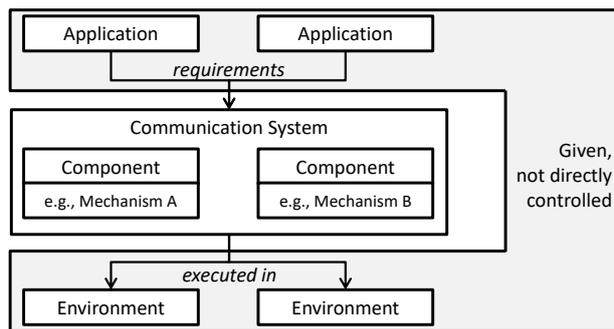


Figure 1.1: Illustration of the dependencies of today’s decoupled, specialized communication system components to fulfill different requirements of various applications in a wide range of different network environments.

<sup>1</sup>Publications of the author of this dissertation are marked with a leading *F*.

*Example:  
Multipath TCP*

To illustrate components and their dependencies in today’s communication systems, we refer to the example of Multipath TCP (MPTCP). Multipath TCP is a recent TCP evolution that overcomes a fundamental limitation of TCP: In general, a TCP connection is restricted to a single network interface per communication partner and a single network path per connection.<sup>2</sup> Today’s networks and networking devices, however, exhibit multiple network interfaces and network paths. Multipath TCP goes beyond TCP and supports the usage of multiple subflows and network paths to increase throughput, reliability, and load balancing in data-centers [147, 148] and mobile scenarios [23, 166]. As Multipath TCP is a transport layer protocol, it is usually implemented in the network stack of the operating system. Multipath TCP implementations can be decomposed into specialized components, e. g., the scheduler maps packets to subflows, whereas the path manager controls the subflow creation (Figure 1.2). These specialized components are further configurable for different application requirements and network environments.

*Multipath TCP  
scheduler research  
and development  
lacks abstractions...*

We note that there is not a single packet scheduler that is optimal for all application scenarios. Instead, the optimal scheduling strategy depends on the requirements of the application, as well as flow and network characteristics. The development of novel, deployable schedulers, however, is hindered by the implementation complexity in the Linux kernel. As a consequence, recent works that propose scheduling optimizations for multipath protocols leave an evaluation within the Multipath TCP Linux kernel open [29, 51, 98, 206] or avoid complex changes in the underlying scheduler [62]. Thus, we identify the need for abstractions for the efficient design and implementation of Multipath TCP schedulers.

*... and extensive  
evaluation support  
for scheduler  
innovations.*

The scheduling strategy influences performance characteristics of upper layer protocols and algorithms, such as the HTTP protocol and DASH adaptation algorithms, and has to consider the experienced network conditions. Thus, novel schedulers require extensive evaluations of their impact on various performance metrics depending on the application requirements and (traffic) characteristics as well as the variety of network environments.

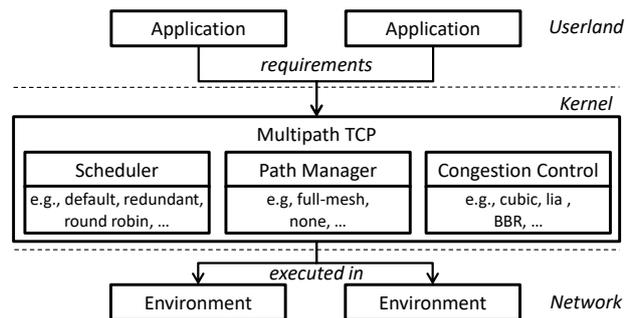


Figure 1.2: Multipath TCP has to fulfil different requirements of various applications in a wide range of different network environments.

<sup>2</sup>More precisely, a TCP connection is identified by and bound to the tuple of (*source IP, source port, destination IP, destination port*). TCP does not provide an explicit notion of multiple paths. Details are discussed in Chapter 2.

## 1.2 PROBLEM STATEMENT

Based on the previous motivation, we identify that the analysis, implementation, and evaluation of communication systems is hindered by *i)* missing abstractions and the resulting implementation complexity, and *ii)* the required extensive evaluations for today's large configuration spaces and heterogeneous network environments. The goal of this dissertation is to overcome these obstacles and develop abstractions for communication system research and development, i. e., abstractions for Multipath TCP scheduling and adaptation decisions, and to enable the seamless execution and analysis of extensive network experiments. This goal results in the following research questions, which are tackled in this dissertation:

*Research Questions*

**RQ I** How can we enable deployable Multipath TCP scheduling innovations?

We note that the MPTCP scheduler design and development lacks fundamental abstractions and requires detailed knowledge about the network stack implementation. The first research question asks for a way to enable MPTCP scheduling innovations and to develop novel general purpose, application-, and preference-aware MPTCP schedulers. The research question asks for a detailed requirement analysis, an appropriate abstraction, and its design, implementation, and evaluation. Our evaluation has to provide evidence that the proposed abstraction enables scheduler innovations, e. g., by presenting and discussing different, novel schedulers.

**RQ II** How can we extensively evaluate and compare the various configurations of communication systems and network protocols in heterogeneous network environments?

This research question emerges from the observation that communication system research *i)* requires systematic network experiment studies with a large number of network experiments, but *ii)* today's network simulators and emulators lack support for an efficient management, execution, and analysis of these extensive network experiments. The evaluation of our proposed solution has to provide evidence that our solution supports the extensive evaluation of a wide range of communication systems and network protocols, e. g., by presenting and discussing various case studies.

**RQ III** What is the design space and what are the opportunities of *i)* general purpose, *ii)* preference-aware, and *iii)* application-aware Multipath TCP scheduling?

We note that the design space of Multipath TCP scheduling is widely unexplored due to *i)* missing abstractions for the design and implementation of Multipath TCP schedulers and *ii)* missing support for systematic extensive network experiments. As we overcome these limitations in the first and the second research question, the third research question asks for the opportunities of *i)* general purpose, *ii)* preference-aware,

and *iii*) application-aware MPTCP scheduling. The answer for this research question has to provide a *comprehensive* set of novel schedulers and their detailed evaluations.

RQ IV What are suitable abstractions for the specification of the adaptation decision for adaptive communication systems?

We note that the specification and derivation of the adaptation decision, i. e., the decision *when* to switch to *which* mechanism, lacks fundamental abstractions. This research question asks for ways to specify adaptation decisions with different abstraction levels. Our evaluation has to provide evidence that our proposed abstractions are suitable for the specification of the adaptation decision, e. g., by applying them on different adaptive communication systems.

### 1.3 APPROACH AND CONTRIBUTIONS

We approach our four research questions with a combination of *i*) an increased level of abstraction by means of bespoke programming models and languages and *ii*) an evaluation methodology that fosters systematic extensive experiments, as illustrated in Figure 1.3. We approach the first research question with *ProgMP*, the *first programming model* and executable specification language for Multipath TCP scheduling. We solve the second research question with *MACI*, the first framework for the management, the scalable execution, and the interactive analysis of a large number of experiments. We solve the third research question and show the strength of our Multipath TCP scheduler programming model *ProgMP* and our evaluation framework *MACI* by designing, implementing, and evaluating various novel general purpose, preference-, and application-aware schedulers tackling diverse objectives. Finally, we provide programming models for the domain of adaptation decisions. Here, we propose to specify the adaptation decisions with *event condition action* (ECA) rules and present the *Fossa* framework to learn these rules in extensive experiments. We further introduce the topology adaptation rule language *TARL* to specify topology adaptations in communication systems. In the following, we present a more detailed overview of our contributions.

#### *RQ I: A Programming Model for Multipath TCP Scheduling*

In this dissertation, we propose the first programming model as abstraction for the design and the implementation of Multipath TCP schedulers to enable a wide range of scheduler innovations [F5]. We present *ProgMP*, the first Multipath TCP scheduler specification language, which enables application developers and researchers to specify Multipath TCP schedulers with a high level of abstraction. We introduce a simple yet powerful scheduling API that enables general purpose, application-, and preference-aware Multipath TCP scheduling. We further provide an efficient runtime environment in the Multipath TCP Linux kernel, closing the gap between the scheduler specification and its execution.

RQ I →  
MPTCP scheduler  
programming model

RQ II →  
Extensive experiment  
framework

RQ III →  
Various novel  
MPTCP schedulers

RQ IV →  
Programming  
models for the  
adaptation decision

The 1<sup>st</sup> executable  
MPTCP scheduler  
specification  
language

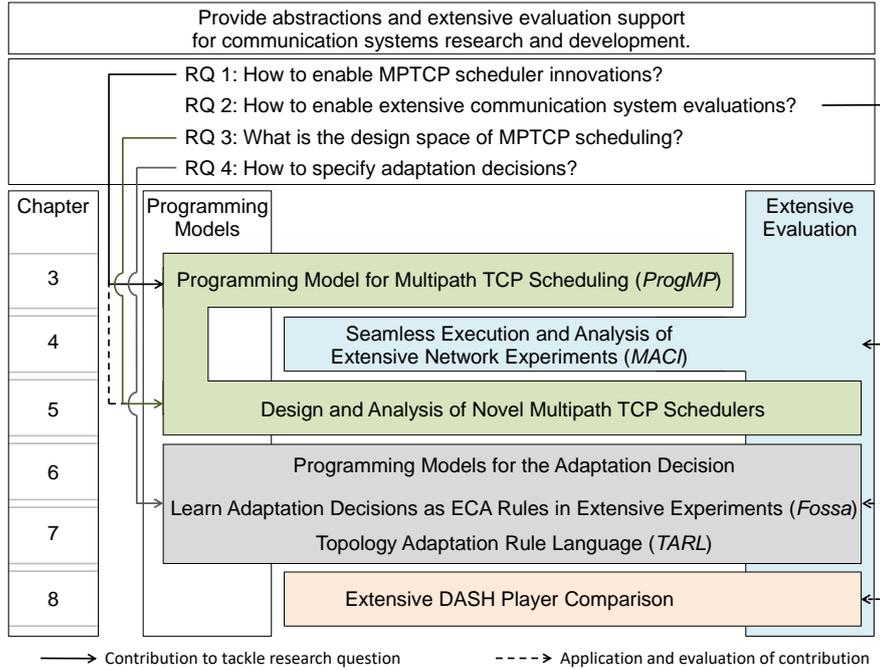


Figure 1.3: Overview and classification of the contributions with regard to the applied concepts of *programming models* and the *extensive evaluations*.

### *RQ II: Seamless Execution and Analysis of Extensive Network Experiments*

We present *MACI*, the first framework for the seamless management, scalable execution, and interactive analysis of network experiments [F2]. *MACI* emerged from our requirements and experiences with a number of experiment driven research efforts, including DASH video streaming configuration studies [F9], the retrieval of ECA rules for adaptive systems [F12], and web performance forecasts for HTTP [F22].

We discuss the advantages of *MACI* and show its significance for research in various communication system domains, such as *i*) the development and evaluation of our proposed novel *ProgMP* schedulers [F1, F5, F15], *ii*) the analysis of a distributed topology graph pattern matching protocol for our topology pattern-based rule language *TARL*, and *iii*) a systematic comparison of DASH video streaming implementations, which goes beyond an evaluation of *MACI* and significantly contributes to the understanding of today's DASH implementations [F6].

### *RQ III: Design and Analysis of Novel Multipath TCP Schedulers*

We use *ProgMP* and *MACI* to derive, specify, and evaluate more than eight novel, deployable Multipath TCP schedulers [F1, F5, F15]. This includes improvements of established schedulers, e. g., by recurrent active probing of unused subflows to obtain fresh round-trip time estimates and by the incorporation of one-way delay estimates. We propose schedulers to balance round-trip times, throughput, and subflow preferences. We propose the first redundant MPTCP scheduler [F7] and discuss design decisions for redundant transmis-

*The 1<sup>st</sup> extensive evaluation framework*

*Extensive distributed TARL analysis*

*Extensive DASH player comparison*

*Novel schedulers...*

*... with active probing.*

*... with one-way delay-awareness.*

*... to achieve performance targets.*

...to balance  
latency and  
throughput.  
...with application-  
awareness.  
...with  
HTTP/2-awareness.  
Advance experiment  
reproducibility

sion, including the choice of new and old packets, the number of duplicates, and the incorporation of backup subflows, to trade throughput for reduced latency. We use application information to improve the flow completion time in heterogeneous environments. Finally, we show the benefits of adaptive, differentiated scheduling during a single connection. Here, we present an HTTP/2-aware scheduler that demonstrates how all building blocks of *ProgMP* interact to improve page load times and reduce the usage of costly subflows. We present the design as well as detailed experiments for all these schedulers.

As part of our extensive *ProgMP* experiments, we additionally contribute support for changing network conditions in the widely used network emulator Mininet [F19] and identify emulation pitfalls [F16].

#### *RQ IV: Programming Models for the Adaptation Decision*

We further contribute abstractions for the adaptation decisions of adaptive networking applications. We apply the concept of *event condition action* (ECA) rules on the specification of the adaptation decision [F13] and propose to learn ECA rules for a user-defined utility function based on extensive experiments [F12]. We further introduce the topology adaptation rule language *TARL* to specify and execute topology adaptations in communication systems based on topology graph patterns.

Learn adaptation  
decisions with  
extensive  
experiments  
ECA and TARL rules  
as adaptation trigger

#### *Overview of the Author's Publications*

Table 1.1 provides a mapping of the contributions of this dissertation to the relevant previous publications of the author of this dissertation.

Table 1.1: Overview of the contributions.

§3	Programming model for Multipath TCP scheduling	→ [F5]
§4	Seamless execution and analysis of extensive network experiments	→ [F2, F6, F9]
§5	Design and evaluation of novel MPTCP schedulers	→ [F1, F5, F7, F15, F16, F19]
Programming models for the adaptation decision		
§6	Learn adaptation decisions as ECA rules in extensive experiments	→ [F12, F13, F14]
§7	Topology adaptation rule language (TARL)	→ [F3, F8]
§8	Extensive DASH player comparison	→ [F6, F9]

Related publications that are not directly reflected in this dissertation are a property description framework for composable software [F24], a concept for a smooth Multipath TCP WiFi to cellular handover [F21], crowdsourc-

ing measurements of mobile network performance and mobility during a large scale event [F10], an experimental evaluation of bounds for stochastic scheduling in fork-join queuing models [F11], and parts of the design of a multipath extension for the QUIC transport protocol [F4].

#### 1.4 THESIS ORGANIZATION

Based on the overview of Figure 1.3, this dissertation is structured as follows. In Chapter 2, we present background and related work on Multipath TCP and Multipath TCP schedulers. Note that we present further background and related work on additional, relevant domains throughout this dissertation in favor of a continuous reading flow. In Chapter 3, we derive the requirements for a Multipath TCP programming model and present our corresponding programming model *ProgMP* and its runtime environment. In Chapter 4, we discuss the recurring requirements and background for extensive network experiments and present our corresponding *MACI* framework. In Chapter 5, we design and evaluate improved general purpose, application-, and preference-aware MPTCP schedulers with *ProgMP* and *MACI*. In Chapter 6, we present background on adaptive systems and learning and search-based software engineering and introduce our *event condition action* rule-based programming model for the adaptation decision. In Chapter 7, we present the topology adaptation rule language *TARL*. In Chapter 8, we apply *MACI* and the concept of extensive network experiments on the application domain of DASH video streaming. Finally, we provide a discussion of our overall contributions and identify future work in Chapter 9.



In this chapter, we provide background on Multipath TCP and present notable related work on Multipath TCP scheduling.<sup>1</sup> This background and related work is essential for the presentation and discussion of the first Multipath TCP scheduler programming model (*RQ I*, Chapter 3) and the design of novel Multipath TCP schedulers (*RQ III*, Chapter 5).

## 2.1 MULTIPATH TCP

Multipath TCP, as specified in [RFC 6824], is a recent TCP evolution, which allows splitting the byte stream of a single transport layer connection over multiple TCP *subflows* and network paths (Figure 2.1). Spreading a single connection on multiple network paths increases throughput, reliability, and load balancing in data-centers [147, 148] and mobile scenarios [F7, 23, 166].

*Multiple subflows  
for multiple paths*

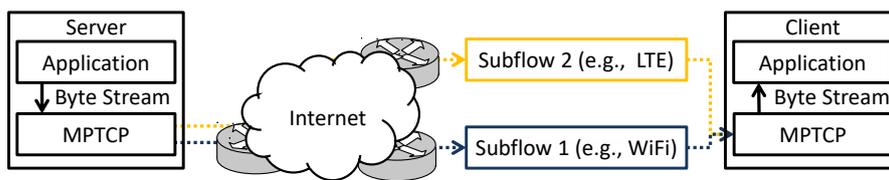


Figure 2.1: Multipath TCP uses multiple subflows and network paths for a single transport layer connection.

Multipath TCP is designed to run in all TCP environments. Therefore, it provides the standard TCP byte stream socket interface dispensing with the need for application modifications. A Multipath TCP connection splits traffic on subflows, where each subflow behaves like a traditional TCP connection to be deployable on the Internet [188]. Multipath TCP overcomes limitations of TCP, e. g., it provides multihoming to enable a mobile handover without additional network support, such as mobile IP [RFC 5944].

*A TCP evolution*

Conceptually, Multipath TCP adds three fundamental aspects to traditional TCP. First, there is a multitude of congestion controls optimized for Multipath TCP to ensure that multiple subflows behave TCP friendly on joint bottlenecks [RFC 6356, 85, 204, 207].<sup>2</sup> Second, MPTCP requires a path management instance to trigger subflow establishments and destructions (see Section 2.1.2). Finally, MPTCP introduces a scheduling decision, i. e., the map-

<sup>1</sup>While scheduling is at the same level as path management and congestion control, we present scheduling in more detail as it is essential for the research questions of this dissertation.

<sup>2</sup>Remarkably, there is a large effort to ensure fairness in mostly disjoint mobile environments with WiFi and cellular networks, whereas browsers typically use multiple TCP connections for web traffic [RFC 2616, 121] and Google uses two concurrent cubic connections for video streaming [100].

ping of packets on subflows (see Section 2.2). The wire protocol format of Multipath TCP includes all primitives to control the multipathing communication and link multiple subflows. While the wire format and various MPTCP enabled congestion controls are specified in RFCs, packet scheduling and path management are not standardized. Many MPTCP implementations, such as today’s implementation in the Linux kernel [131], additionally introduce meta sockets as central abstraction for each connection.

In the following, we sketch the fundamentals of Multipath TCP that are required for this dissertation before discussing related work on Multipath TCP scheduling in Section 2.2. For a comprehensive presentation of Multipath TCP we refer to the corresponding RFCs. For a survey on general network-layer multipathing we refer to Qadir et al. [146].

2.1.1 Connection Establishment and Wire Protocol Format

Establish MPTCP connections

Figure 2.2 illustrates a Multipath TCP connection establishment as specified in [RFC 6824]. Therefore, the initiator (A) and the connection partner (B) include the MP\_CAPABLE option accompanied by additional keys in a traditional TCP three-way handshake.

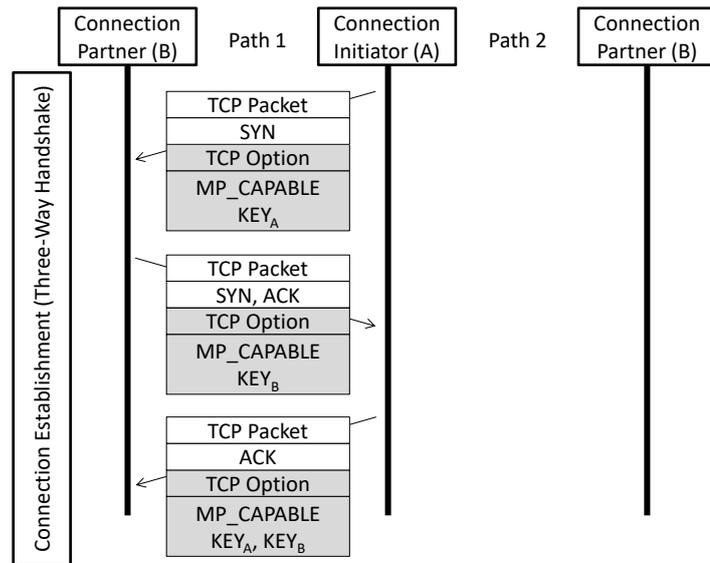


Figure 2.2: Illustration of the Multipath TCP connection establishment with the MP\_CAPABLE and KEY options.

Join subflows

After the connection is established, both communication partners can trigger the establishment of additional subflows using the MP\_JOIN option and a token identifier in a new TCP three-way handshake (Figure 2.3). Multipath TCP additionally provides options to tear down subflows and advertise addresses to cope with middleboxes such as NATs [RFC 6824, 148].

Scheduling dependencies

The previous presentation shows that Multipath TCP requires several round-trip times to establish a connection and add additional subflows. This is an important limitation for short flows that only require a few round-trip times.

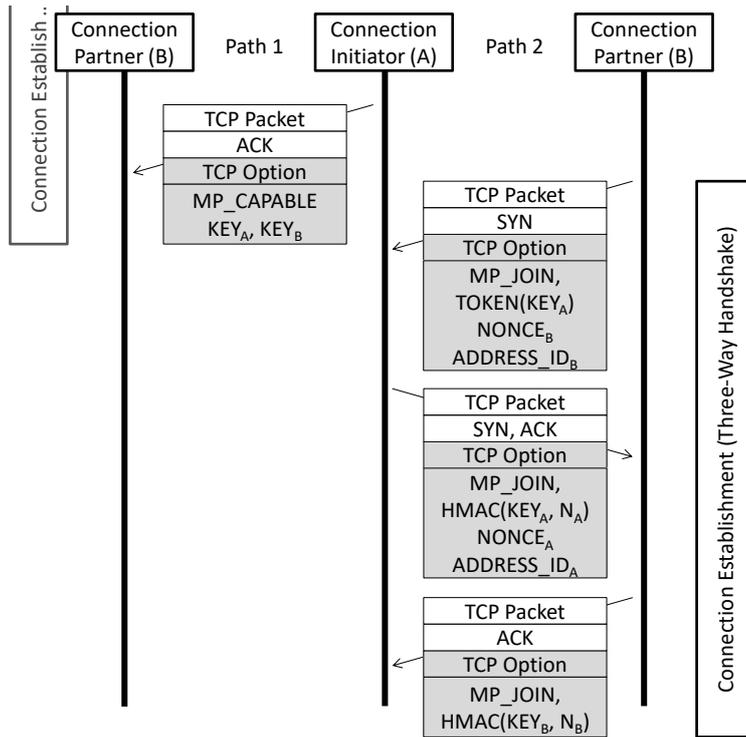


Figure 2.3: Illustration of the MPTCP subflow addition with the MP\_JOIN for a previously established MPTCP connection.

Furthermore, the packet scheduler has to consider that only established subflows can transmit packets.

Multipath TCP relies on TCP options to transfer these control messages. To ensure reliable in-order data delivery across subflows, MPTCP relies on a data sequence number mapping to the subflows sequence number offset and an optional checksum (Figure 2.4). The data sequence number mapping consists of the Data Sequence Number, the Subflow Sequence Number, and the Data Level Length. The complexity of the transferred control information is required to ensure deployability in networks with middleboxes [RFC 6824, 148] and ensure security properties [RFC 6824, RFC 7430].

*TCP options*

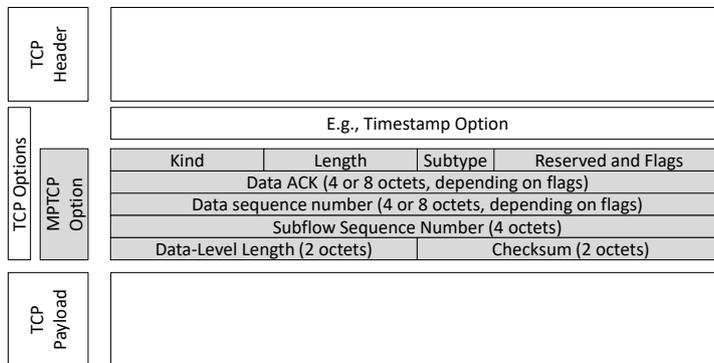


Figure 2.4: Illustration of the MPTCP wire format.

### 2.1.2 Path Management

Trigger subflow  
establishment

Full mesh

Backup mode

Single path mode

Proactive handover

The path management controls the establishment and teardown of subflows using the previously presented MPTCP options and handshakes. A simple yet powerful strategy is to establish a *full mesh* of subflows between all interfaces. With regard to mobile devices, a full mesh might be undesirable as cellular interfaces require energy and are often metered. Paasch et al. [132] propose two alternatives to the *full mesh* path management for mobile devices, namely the *backup mode*, which establishes all subflows, but relies on WiFi as long as possible, and the *single path mode*, which establishes cellular subflows *after* the WiFi connection failed. We have further proposes a *proactive handover* [F21], which triggers the establishment of the cellular subflow *before* the WiFi connection fails to enable a smooth handover while avoiding unnecessary cellular subflows (Figure 2.5). Similar concepts were later proposed by [172]. Alternative path management strategies include *i*) the establishment of subflows if required to sustain throughput goals of the application [F20] and the control of subflow creation and teardown based on path characteristics experienced, e. g., to favor disjoint paths [68].

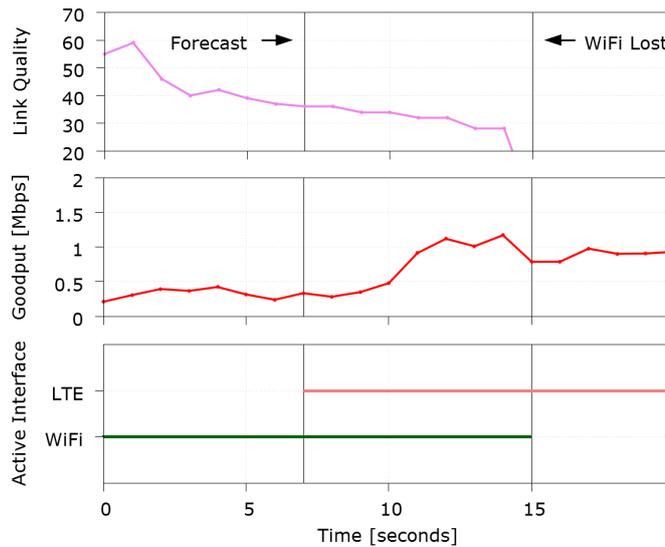


Figure 2.5: Real world measurement of an MPTCP handover using a WiFi connection loss forecast based on the link quality to establish the LTE subflow *before* the WiFi connection is lost [F21].

Extended path  
management API

Non-trivial details

From an implementation perspective, recent papers extended the socket API to enable path management for the Linux kernel MPTCP implementation in the userspace [67, 68]. As path management decisions usually happen at a granularity of milliseconds, the introduced overhead for userspace communication is negligible. We note that even though describing path management strategies seems straightforward, the details for deployable, stable implementations are important and non-trivial. What should happen if an in-

interface fails for a short time period? When should the subflow be recreated?<sup>3</sup> In Section 2.2, we show that MPTCP scheduling strategies exhibit the same mismatch between their *description* and deployable, stable implementations.

While the separation of path management and packet scheduling reduces the overall complexity, we note that both interact with each other in various ways. The scheduler, for example, is limited to the available subflows (controlled by the path management), but can ignore established subflows unilaterally. De Conick et al. [33] showed that an integrated path management and scheduling approach improves mobile handover. Finally, Apple’s MPTCP implementation for iOS uses the `BACKUP_FLAG` to control packet scheduling at the communication partners’ side for the `Interactive` mode.<sup>4</sup>

*Scheduling dependencies*

### 2.1.3 Implementations

As of today, the implementation in the Linux kernel [131] is the publicly available de facto standard Multipath TCP implementation. Apple is known to rely on MPTCP for their voice-based cloud assistant service *Siri*. During the work on this dissertation, Apple started to support MPTCP for a broader range of applications on mobile devices with the advent of iOS11.<sup>5</sup> Even though their implementation is publicly available<sup>6</sup>, public research on MPTCP is mainly restricted to the Linux kernel implementation as of today.

*Linux kernel*

*Apple iOS*

Coudron et al. [30] provide a Multipath TCP implementation for the discrete event simulator ns-3. While this implementation enables systematic, controlled Multipath TCP experiments, the findings have to be analyzed carefully for real world applicability.

*Implementations for simulators*

Tazaki et al. [186, 187] present a *direct code execution* (DCE) approach to run nearly unmodified applications and Linux kernel network stacks in the discrete event simulator ns-3. The authors show that their approach enables ns-3 experiments with the Linux kernel MPTCP implementation. With regard to research on Multipath TCP, all these approaches provide different benefits and drawbacks.

<sup>3</sup>A discussion on the MPTCP developer mailing list is available at <https://listes-2.sipr.ucl.ac.be/sympa/arc/mptcp-dev/2017-02/msg00006.html>.

<sup>4</sup>An overview of the `NSURLSessionMultipathServiceType` for iOS application developers is available at <https://developer.apple.com/documentation/foundation/nsurlsessionmultipathserVICETYPE>.

<sup>5</sup>See <https://support.apple.com/en-ca/HT201373>.

<sup>6</sup>The iOS source is available at <https://github.com/apple/darwin-xnu/tree/xnu-4570.1.46/bsd/netinet>.

## 2.2 MULTIPATH SCHEDULING

In comparison with traditional TCP, Multipath TCP introduces a fundamentally new complexity, as packets of one stream need to be scheduled on multiple subflows. Each side of a Multipath TCP connection runs its own scheduler<sup>7</sup> for outgoing packets. In general, the previously explained data sequence number mapping enables the receiver side to order packets from multiple subflows regardless of the used scheduling.<sup>8</sup> The scheduling decision has a substantial impact on the protocol performance [6]. Bad scheduling decisions may render the advantage of additional paths useless or even reduce the overall performance, e. g., by introducing head-of-line blocking [162].

In this section, we present background and related work on Multipath TCP scheduling and general multipath scheduling. This overview results in the classification of Table 2.1 and an overall discussion.

### 2.2.1 RFCs on MPTCP Scheduling

[RFC 6824] envisions multiple approaches for scheduling that go beyond throughput optimization in paragraph 3.3.8. *Subflow Policy*:

“Within a local MPTCP implementation, a host may use any local policy it wishes to decide how to share the traffic to be sent over the available paths.

In the typical use case, where the goal is to maximize throughput, all available paths will be used simultaneously for data transfer, using coupled congestion control as described in [5] ([RFC 6356] - editor’s note). It is expected, however, that other use cases will appear.

For instance, a possibility is an ‘all-or-nothing’ approach, i. e., have a second path ready for use in the event of failure of the first path, but alternatives could include entirely saturating one path before using an additional path (the ‘overflow’ case). Such choices would be most likely based on the monetary cost of links, but may also be based on properties such as the delay or jitter of links, where stability (of delay or bandwidth) is more important than throughput. Application requirements such as these are discussed in detail in [6] ([RFC 6897] - editor’s note).”

*Backup subflow semantics*

The RFC further specifies *backup subflows* as follows:

“Therefore, the MP\_JOIN option (see Section 3.2) contains the ‘B’ bit, which allows a host to indicate to its peer that this path should be treated as a backup path to use only in the event of failure of

<sup>7</sup>In this dissertation, we use the term *scheduler* to refer to the Multipath TCP scheduler. Schedulers are known in many other disciplines, such as *process* scheduling.

<sup>8</sup>Middleboxes might obstruct this by interfering with the sequence number mapping.

other working subflows (i. e., a subflow where the receiver has indicated B=1 SHOULD NOT be used to send data unless there are no usable subflows where B=0).”

In summary, the RFC is vague with regard to the concrete scheduling, but provides a strict notion of the *backup subflow* semantics.

### 2.2.2 Deployable Multipath TCP Schedulers

In this section, we present an overview of *deployable* Multipath TCP schedulers. Here, we define deployable as implemented in a usable execution environment (typically, the Linux kernel) based on the MPTCP wire protocol.

AVAILABLE IN THE LINUX KERNEL The current Linux kernel implementation is based on a pluggable scheduler [133] and comprises the *round robin*, the *default*, and the *redundant* scheduler. The *round robin* scheduler is known to perform poorly for heterogeneous paths, as slow subflows constrain the overall performance [133], but enables useful test setups. The *default* scheduler considers for each subflow the round-trip time (RTT) and the congestion window. The scheduler assigns packets to the subflow with the lowest RTT that has not exhausted its congestion window yet [148]. Remarkably, related work rarely mentions that the default scheduler only considers subflows that fulfill the TCP short queue condition (i. e., are not throttled by *TSQ*) and are not in a loss state.<sup>9</sup> The *redundant* scheduler in the Linux kernel is a combination of the works in [F7, 113] and aims to reduce latency for applications with moderate bandwidth requirements.<sup>10</sup>

*The three schedulers  
in the Linux kernel*

Although assessing code complexity is difficult, we note that the *naive* round robin scheduler already requires 302 lines of code (LOC). Further commits (two critical and six minor fixes) highlight the error susceptibility of today’s development methodology (Appendix A.6). We revisit these schedulers in Section 5.2 to discuss semantic details and illustrate the strength of specifying schedulers using our programming model.

Inspired by TCPs pluggable Linux congestion control, Paasch et al. [133] introduce a pluggable scheduler for the Linux kernel. While this approach increases flexibility with respect to the scheduler *selection*, it does not contribute abstractions for the *design* and *implementation* of novel schedulers.

*Flexible MPTCP  
Schedulers*

COMPENSATE LOSS IN SHORT DATA-CENTER FLOWS The works in [22, 75] propose Multipath TCP scheduling optimizations for packet loss compensation in data-center environments to improve the tail flow completion time. Hwang et al. [75] propose a *Fast Coupled Retransmission* for unacknowledged packets on non-congested paths when a duplicate acknowledge-

<sup>9</sup>In Section 5.10.2, we discuss this observation in more detail.

<sup>10</sup>While the redundant scheduler is contributed by the author of this dissertation, we present it as background on Multipath TCP scheduling as it is part of today’s open source MPTCP Linux kernel implementation. We present a detailed motivation, analysis, and evaluation of the redundant scheduler in Section 5.4.

*Retransmit when  
loss is suspected*

ment is received. Chen et al. [22] recover packets on alternative subflows by retransmitting the oldest, unacknowledged packet of the subflow with the highest loss rate when loss is *suspected*. Both independent works use slight variations of the same idea. Following their pseudo code, both works implicitly detect the end of flow by checking if the sending queue is empty. An analysis and evaluation of design decisions, such as the choice of the retransmitted packet, is not provided.

*Preference-aware  
video streaming*

VIDEO STREAMING Recent work in [62] presents a preference-aware *Dynamic Adaptive Streaming over HTTP* (DASH) framework, denoted as *MP-DASH*, and shows the potential of differentiating between MPTCP paths for DASH. The authors introduce an additional control loop on top of the scheduler, which controls the *visibility* of subflows for the *default* scheduler. While this concept appears comparable to the established *backup* subflow semantics, the authors do not discuss or justify their design decision to introduce an additional control loop. *MP-DASH* provides an important step towards preference-aware scheduling, but remains confined to the chunk-based DASH video streaming application.

*Earliest  
completion first*

Lim et al. [109, 110] propose an *earliest completion first* (ECF) scheduler. This scheduler considers the amount of data that is queued in the send buffer with the goal of minimizing completion time. While this scheduler is designed as general purpose scheduler, the authors show that it is particularly efficient for video streaming over heterogeneous paths.

*Out-of-order  
transmission*

*Blocking Estimation  
scheduler*

PACKET ORDER AND RECEIVE WINDOW CONSIDERATIONS *Receive buffer blocking* and *out of order* received packets attracted much attention such as in [41, 130, 160, 208] due to their performance impact. Concerning out-of-order-received packets, Yang et al. [208] show the potential of out-of-order transmission to achieve in-order arrival. The authors implemented the proposed scheduler in the Linux kernel and evaluated the performance with large file transfers in a network with seldom buffering. Ferlin et al. [41] propose a *B*locking *E*STimation scheduler (BLEST) to reduce head-of-line blocking due to receive buffer restrictions in heterogeneous environments by estimating the blocking time. The authors further reimplement DAPS [98] and the out-of-order transmission scheduler [208] for a detailed comparison in the Linux kernel. Ou et al. [130] build on these works and propose an MPTCP out-of-order transmission enabled joint congestion and scheduling control. Oh et al. [127] use a constraint-based approach to enable a receiver buffer and network delay based scheduling. Jin et al. [77] focus on throughput and predict the future throughput of paths to avoid under-performing paths. Kim et al. [87] sketch the idea of a scheduler that focuses on flow completion time based on the remaining untransmitted packets.

### 2.2.3 Multipath TCP Schedulers in Simulators

Additional scheduler optimizations were proposed and evaluated based on simulator experiments. Kuhn et al. [98] introduce an analytical model to reduce the receiver buffer blocking time for MPTCP and CMT-SCTP with *delay-aware packet scheduling* (DAPS) and provide an evaluation in the network simulator ns-2. The DAPS scheduler assumes that there is a large difference in delays between the different paths and it assumes that the congestion windows are stable. The evaluation in ns-2 compares the performance with a rather naive *blind round robin* scheduling.

*Delay-aware packet scheduling*

Shreedhar et al. [171] propose a *QueueAware* scheduler and use the MPTCP implementation for the discrete event simulator ns-3 of Coudron et al. [30] to compare the performance with the ns-3 pendant of the default scheduler.

Wu et al. [205] provide a sophisticated analytical model for video streaming with *forward error correction* over heterogeneous wireless networks. The authors implemented their approach together with an own Multipath TCP implementation for the *Exata*<sup>11</sup> network emulator.

### 2.2.4 Multipath TCP Related Schedulers

In this section, we provide an overview of Multipath TCP related schedulers, i. e., schedulers for multipath protocols or with Multipath TCP in mind, but without a Multipath TCP integration and evaluation.

Nikraves et al. [125] conduct a large Multipath TCP user study and identify the scheduling algorithm as source of suboptimal performance. Based on their findings, they present a sophisticated *MPFlex* proxy architecture as *replacement* for Multipath TCP. *MPFlex* uses a single connection to multiplex *all* traffic and runs the scheduling decision and a scheduling policy matching for different applications in the userspace. The *MPFlex* scheduler calling model, the induced overhead at the sending server, and the impact on the scheduling timeliness are not discussed. Nikraves et al. [125] propose two scheduler tunings for the *default* scheduler: *i)* a smarter consideration of the sending buffer, as Kim et al. [88] also noticed that small buffers on lowest RTT might lead to problems, and *ii)* the packet reinjection based on timeouts and receive window sizes.

*MPFlex*

Guo et al. [56, 57] build on the work of Nikraves et al. [125] and propose to accelerate multipath transport through balanced subflow completion. Essentially, the authors use out-of-order transmission on top of the *MPFlex* architecture to improve HTTP-based web traffic.<sup>12</sup>

*Balanced subflow completion*

**REDUNDANCY** Liu et al. [111] and Xu et al. [206] propose redundant transmission on multiple paths for mice flows to reduce flow completion time in data-centers. Both works provide evaluations based on own multipath implementations on top of TCP.

<sup>11</sup>See <http://www.scalable-networks.com/exata>.

<sup>12</sup>This work was developed independently and concurrently to our publication [F5].

	In Linux Kernel	Signaled Appl. Info.	Preferences	Reference	Publication Date	Pseudo Code Lines
<b>Part of the open source MPTCP Linux kernel implementation</b>						
<i>Default</i> , non-exhausted subflow with lowest RTT first	Yes	No	Binary	[148]	2012	-
Bufferbloat mitigation				[133]	2014	-
Opportunistic retransmissions				[133]	2014	-
<i>Round robin</i> , mainly for testing purposes	Yes	No	Binary	[133]	2014	-
<i>Redundant</i> transmission for latency sensitive flows	Yes	No	Binary	[F7, 113]	2016	-
<b>Evaluated in the MPTCP Linux kernel implementation</b>						
Packet loss in data-centers						
Retransmit at the flow end in case of duplicate Acks	Yes, N/A	Implicit	No	[75]	2016	18
Retransmit when loss suspected	Yes	Implicit	No	[22]	2016	8
Preference-aware DASH video streaming	Yes, N/A	No	Yes	[62]	2016	-
<i>Earliest completion first</i> based on subflow buffers	Yes, N/A	No	No	[109, 110]	2017	20
<i>BLEST</i> receive window blocking estimation	Yes <sup>13</sup>	No	No	[41]	2016	-
Out-of-order packet transmission for in-order arrival	Yes <sup>13</sup> , N/A	No	No	[208]	2014	20

<b>Evaluated in MPTCP Simulators</b>							
<i>DAPS</i> , generate delay-aware packet schedule	ns-2 <sup>13</sup>	No	No	[98]	2014	18	
Queue-aware analytical optimization	ns-3, N/A	No	No	[171]	2017	-	
Video streaming FEC in wireless	<i>Exata</i>	Yes	No	[205]	2015	-	
<b>Multipath TCP Related</b>							
<i>MPFlex</i> architecture Novel protocol to replace MPTCP	Over TCP/UDP	Policies per App.	Yes	[125]	2016	-	
Buffer-aware scheduling, Smart reinjection				[125]	2016	-	
Balanced subflow completion for <i>HTTP</i>	<i>MPFlex</i>	Yes	-	[56, 57]	2017	23	
<i>Redundant</i> in data-centers							
<i>RepFlow</i> for mice flows	Over TCP	No	-	[206]	2014	-	
<i>RepNet</i> for mice flows	Over TCP	No	-	[111]	2018	-	
Energy optimized video streaming							
Optimization problem	Over TCP	Video meta data	Yes	[51]	2015	-	
Integer Linear Program	-	Video meta data	-	[29]	2016	-	

Table 2.1: Overview of related work on Multipath TCP scheduling and Multipath TCP related scheduling. A *binary* preference only relies on backup subflows if all other subflows fail. - symbolises that the publication does not state this information.

VIDEO STREAMING Corbillon et al. [29] propose the concept of *cross-layer* schedulers to leverage information from the application and the transport layer. The authors model the video transmission over multiple paths as Integer Linear Program and use multipath traces to evaluate their implementation as an application layer scheduler on top of Multipath TCP.

Go et al. [51] propose an energy-efficient adaptive HTTP-based video streaming over heterogeneous wireless networks. The authors aim to benefit from Multipath TCP, yet they use traditional TCP for their evaluation.

ADDITIONAL SCHEDULERS AS OPTIMIZATION PROBLEM Rueckelt [154] introduces an additional multipath transport protocol and models the scheduling decision as optimization problem. Similarly, Khuda Bukhsh et al. [F11, 86] provide additional optimization problems and solutions of optimal packet distribution on multiple paths.

### 2.2.5 Multipathing for SCTP

In this section, we provide an overview of the *Stream Control Transmission Protocol* (SCTP), which is a notable example of a multipath enabled transport protocol. SCTP is a reliable transport protocol on top of a connectionless packet network such as IP [RFC 4960]. SCTP is motivated by the limitations of TCP. Even though it provides multihoming, the original proposals assume data transfer only on the primary path, as specified in [RFC 4960]:

“By default, an endpoint SHOULD always transmit to the primary path, unless the SCTP user explicitly specifies the destination transport address (and possibly source transport address) to use.”

Retransmitted data might use alternative addresses (and therefore paths), as specified in [RFC 5944]:

“In its current form, SCTP does not do load sharing, that is, multihoming is used for redundancy purposes only. A single address is chosen as the "primary" address and is used as the destination for all DATA chunks for normal transmission. Retransmitted DATA chunks use the alternate address(es) to improve the probability of reaching the remote endpoint, while continued failure to send to the primary address ultimately results in the decision to transmit all DATA chunks to the alternate until heartbeats can reestablish the reachability of the primary.”

*Concurrent  
Multipath Transfer  
for SCTP*

Iyengar et al. [76] propose a *Concurrent Multipath Transfer* (CMT) extension for SCTP, which utilises multiple paths simultaneously. Based on this work, Sarwar et al. [160] propose to delay packets to favor in order delivery

<sup>13</sup>Ferlin et al. [41] (re-) implemented *BLEST* [41], *DAPS* [98], and the out-of-order transmission [208] for the Linux kernel and made it publicly available at [https://bitbucket.org/blest\\_mptcp/mptcp-paper-ifip2016](https://bitbucket.org/blest_mptcp/mptcp-paper-ifip2016).

for SCTP which is transferred to MPTCP with their DAPS scheduler [98]. With regard to multipath APIs, Dreibholz et al. [37] propose an extended sender queue information socket API for SCTP.

### 2.2.6 Singlepath Scheduling

Packet scheduling is a basic operation in communication networks. Network switches, for example, schedule packets of multiple flows to egress-ports *in the network*. TCP implementations on the end-host exhibit scheduling, e. g., *i*) the well-known Nagle algorithm (specified in [RFC 896, RFC 1122]) reduces packet header overhead by delaying packet transmission, *ii*) TCP keep alive messages are scheduled based on idle times, and *iii*) different types of packet retransmissions, such as traditional *retransmission timeouts* as well as *fast* and *early* retransmits [43, 149] are scheduled for loss recovery. Besides these widely deployed TCP scheduling optimizations, recent proposals such as Halfback [108] retransmit unacknowledged packets in a reverse order.

From a Multipath TCP scheduling perspective, the dependencies between singlepath packet scheduling and multipath scheduling are important. What are, for example, the semantics of the Nagle algorithm in case packets are split on multiple subflows?

### 2.2.7 MPTCP Scheduling Dependencies

Even though scheduling is conceptually separated from other concerns, we note that all building blocks of MPTCP are closely tied and depend on each other. Today's schedulers, for example, require subflows to be initiated by the path-manager and consider the congestion window maintained by the congestion control. Optimizing MPTCP for high throughput effectively lets the congestion control *schedule* the traffic [204], as the scheduler is blocked by the congestion control. This was, for example, shown and exploited by Popovici et al. [145]. For thinner application limited flows, such as for *request-response* patterns, congestion windows are not permanently exhausted, reducing the impact of the congestion control on the scheduling decision. As most congestion control algorithms increase the congestion window based on transmitted data, the congestion window only builds up where packets are scheduled.

### 2.2.8 Discussion

The anticipated scheduler optimizations confirm our observation that there is a wide range of potential Multipath TCP schedulers and no *one size fits all* scheduler. We note that the proposed scheduler optimizations include many design decisions, which are rarely evaluated. We assume that this lack of fine grained design decision analyses is caused by the required implementation effort and the evaluation overhead. This assumption supports the motivation and significance of our research questions.

*Nagle, ...*

*keep alive, ...*

*and fast retransmit  
are packet  
scheduling strategies*

*Path-manager and  
scheduler*

*Congestion control  
and scheduler*

*No one size fits all  
scheduler*

*The need for efficient  
implementations and  
evaluations*

Besides this lack of fine grained design decision analyses, we further note that evaluation based comparisons of proposed schedulers from *different* publications are seldom.<sup>14</sup> We assume that this is caused by the implementation overhead, as many implementations are not publicly available. Even with the best of intentions, the reimplementing of already proposed schedulers is hindered by the complexity and imprecision of the scheduler description in the publications. Performance evaluations with reimplemented schedulers are error prone due to the significant impact of tuning parameters.

*The need for  
precise scheduler  
specifications*

*The need to  
simplify MPTCP  
scheduler  
implementation*

The classification in Table 2.1 shows that recent works propose scheduling optimizations for multipath protocols, but leave an evaluation within the MPTCP Linux kernel open. We argue that an implementation *within* Multipath TCP is superior for various reasons. First, a real MPTCP implementation is indispensable to assess the real-world performance, as subtle implementation differences may have a large impact. Second, the proposed schedulers benefit from the proved concepts of MPTCP, such as transparent end-to-end connection establishment in the advent of Middleboxes and multipath-enabled congestion controls. Finally, an integration and usage of the wireformat of Multipath TCP is indispensable for interoperability. A typical MPTCP scheduler optimization does not require receiver side changes, whereas an implementation on top of TCP introduces a new application layer protocol.

*The need for  
abstractions*

Finally, we note that a systematic design of application- and preference-aware schedulers is missing. A few examples from the *Multipath TCP related* schedulers show the potential of incorporating application knowledge and preferences. As of today, the development of application- and preference-aware schedulers requires detailed application *and* Linux kernel networking stack knowledge.

---

<sup>14</sup>Ferlin et al. [41] provide a notable example with their implementation of various schedulers from the related work for a detailed evaluation of their proposed *BLEST* scheduler.

In the previous chapters, we identified that Multipath TCP misses fundamental abstractions to foster deployable scheduling innovations and substantiated the significance of our first research question:

*RQ1: How can we enable deployable Multipath TCP scheduling innovations?*

*First Research Questions*

In this chapter, we tackle this research question and propose *ProgMP*, a *programming model* as abstraction for Multipath TCP scheduling. Based on a detailed requirement analysis and an overview of programming models in communication systems, we introduce the *ProgMP* programming model. *ProgMP* consists of the model of the scheduling environment, a bespoke scheduler specification language, the scheduler calling model, and a powerful yet simple API. We further present the implementation of a runtime environment for the programming model in the Linux kernel. For an evaluation of the expressiveness and a detailed discussion of the programming model experiences, we refer to our presentation of novel schedulers in Chapter 5. Parts of this chapter are published in [F5, F15].

*ProgMP* is publicly available together with detailed examples and tutorials at <https://progmp.net> to enable researchers and application developers to benefit from our experiences and to specify executable general purpose, application-, and preference-aware MPTCP schedulers.

*Publicly available*

### 3.1 REQUIREMENT DERIVATION

In this section, we present and discuss requirements for a Multipath TCP scheduler programming model and conclude with a discussion of the target audience and user group.

**EXCHANGE OF IDEAS** The specification language should enable the exchange of scheduling ideas, i. e., the communication between and discussions of domain experts. The specification language has to abstract over implementation details and should provide a compact, crisp, and comprehensive expression of the scheduling logic and eliminate ambiguity. A domain expert should be able to reason about a scheduler and compare schedulers based on an *explicit* specification. Today's lack of a specification language leads to short informal descriptions of scheduling strategies, which tend to be underspecified, e. g., with regard to important corner cases. To illustrate this, imagine a *round robin* scheduler. The name *round robin scheduler* provides a basic impression of the scheduling strategy, but does not specify if a currently unavailable subflow should be skipped or the scheduler should block in this case. About

*A language for communication...*

... to replace  
pseudo code...

half of the presented scheduler publications in Section 2.2 try to mitigate the imprecision by providing *pseudo code*. A scheduler specification language should provide the expressiveness and crispness to replace these *pseudo code* descriptions with a well-defined specification.

**EVALUABLE AND EXECUTABLE** Section 2.2 showed that researchers proposed a variety of multipath scheduling optimizations, but left the implementation of these optimizations in a real Multipath TCP environment, i. e., the Multipath TCP Linux kernel network stack, for future work. Instead, evaluations of these works rely on user-level implementations that mimic MPTCP and established schedulers. We suspect that the lacking MPTCP implementations are caused by the complexity and effort of the development in the Linux kernel networking stack.

Additionally, we note that even the implemented schedulers are in general solely compared with the available schedulers in the Multipath TCP Linux kernel. We suspect that this is caused by the intricateness of integrating existing research results in the Multipath TCP Linux kernel for own publications. This is amplified by the marginal integration of scheduling research optimizations in the Multipath TCP Linux kernel.<sup>1</sup>

... that should be  
executable...

We argue that schedulers have to be *executed* for realistic evaluations and comparisons. This claim is supported by Section 5.10, where we show that emulations and simplifications are error prone and misleading. Thus, a programming model for schedulers should enable rapid evaluations and comparisons of *executable* schedulers in real MPTCP environments and fosters the transition from research to production. In contrast, a non-executable specification language would require a time consuming and error prone reimplementation for evaluations and therefore not encourage real MPTCP evaluations.

... efficiently.

**EFFICIENT EXECUTION** The programming model and the specification language need to enable an efficient execution and timely scheduling decisions.<sup>2</sup> To saturate a 1 Gbit connection with a packet size of 1500 bytes per packet, about 715,827 packets have to be scheduled per second.<sup>3</sup> Thus, the per packet scheduling decision should require at most a few hundred nano seconds. Following this requirement, the language design and implementation of the corresponding runtime environment have to be considered holistically. Thus, the scheduling language should carefully balance language features, expressiveness, and efficient execution, e. g., with regard to usually time con-

<sup>1</sup>The author of this dissertation integrated the redundant scheduler code of ReMP [F7] into the MPTCP Linux kernel ([https://github.com/multipath-tcp/mptcp/blob/mptcp\\_v0.93/net/mptcp/mptcp\\_redundant.c](https://github.com/multipath-tcp/mptcp/blob/mptcp_v0.93/net/mptcp/mptcp_redundant.c)).

<sup>2</sup>Here, we have to emphasize the difference between the *programming language* and the *implementation*. In general, a programming language does not have properties such as *efficient execution*. In this particular case, however, we *design* the language with assumptions about the implementation in mind. Different implementations and underlying hardware might change these assumption. We refer to [36] for an early, general discussion on this differentiation.

<sup>3</sup>This is a simplified example for illustration. In high-throughput environments, TCP segmentation offloading (TSO) enables larger packets in the hosts networking stack depending on the workload and traffic pattern.

suming dynamic memory allocations. Furthermore, the scheduler execution is preferably automatically optimizable by the runtime environment.

**EXPRESSIVENESS** The expressiveness of the programming model and its specification language should preferably enable the specification of *all* schedulers in a convenient way. However, the specification of *all* schedulers conflicts with the reduced complexity of a domain-specific language and the requirement of an efficient execution. Solving optimization problems per packet, e. g., is not feasible for most scenarios. Thus, our goal is the design of an executable specification language that can express all reasonable and efficiently executable schedulers.

*Expressiveness for a wide range of schedulers...*

For complex scheduler logic that cannot be evaluated *online* per packet, the programming model of the specification language should enable the integration of *offline* logic. Thus, a controller might solve optimization problems or incorporate additional knowledge offline and outside of the scheduling loop, and control the online scheduler logic. This concept is similar to the *Open-Flow/SDN* separation of the *Controller* and the *Flow Tables*.

*... and open for external extensibility.*

**GRACEFUL FAILURE HANDLING** The execution semantics of the programming model should allow a graceful failure handling. Typical pitfalls in the selection of both packets and subflows relate to the establishment of new subflows and the sudden failure of established subflows. In a pure C implementation in the Linux kernel, the disappearance of a subflow easily leads to stale references and crashes of the operating system. Ideally, accessing stale subflow references should not only be prohibited or fail gracefully, but should be impossible by design. Further, handling references to packets, for example, for the assignment to multiple subflows, is a substantial source for errors. In particular, packets must not be lost, e. g., when the target subflow ceases after the packet is already removed from the sending queue. Finally, scheduling decisions have to terminate to ensure *liveness* of the communication system.

**TARGET AUDIENCE** Based on the discussed programming model requirements, the target audience includes domain experts and researchers. Beside these, the specification language should enable network administrators of a managed network environment to specify schedulers that incorporate network specifics and optimize in accordance to additional management functions, such as explicit path controls [72]. With regard to application developers, we anticipate that only developers of applications that crucially depend on network performance characteristics will specify an own application-aware scheduler. However, extended application libraries may ensure that entire application classes benefit from improved application-aware schedulers.

## 3.2 PROGRAMMING MODELS IN COMMUNICATION SYSTEMS

In this section, we present an overview of programming models and languages as abstractions in the domain of communication systems.

*The Click modular router...*

The *Click* modular router [94] abstracts over implementation details of *packet processing modules* and enables the composition of complex router logic as *graphs* of packet processing modules. A *Click* configuration is specified with a *declarative* language based on *declarations* and *connections* to abstract over *procedural* packet processing. The abstractions and the model of *Click* were applied on additional use cases, such as network function virtualization [101, 117]. The underlying execution environment was tuned for high speed I/O [7] and to leverage available hardware [89, 185]. While the concepts and the idea of *Click* appear simple, the abstraction proved to be powerful and applicable for a wide range of applications.

*... uses a declarative language.*

*OpenFlow...*

The *OpenFlow* protocol is an additional example for increasing programmability [118]. While *OpenFlow* [118] is sold as *protocol* between switches and controllers, it essentially represents a programming model for the switches based on the manipulation of *flow table entries*. The enabled programmability of *OpenFlow* took the community by storm and expanded to the general concept of *software-defined networking* (SDN) [91], network-wide programmability [47], and more fine-granular programmability of the underlying packet processing pipeline [13].

*... and SDN*

*Programmable packet processing*

Here, *P4* represents an example of a high-level language for programming protocol-independent packet processors [13]. Sivaraman et al. [173, 174] raised the abstraction layer for *scheduling in switches* and propose programmable packet scheduling, e. g., by executing a block of *imperative code* the moment the packet comes *in*.

*Transport protocol specification language*

On end-hosts, configurable traffic shapers already provide abstractions for scheduling decisions *between independent flows*, e. g., based on QoS flags. The *Readable TCP in the Prolac Protocol Language* [93] was suggested for the development and implementation of *comprehensive transport protocols*, i. e., packet header handling and semantics. Basu et al. [9] propose a domain-specific language for the construction and validation of communication system protocols. Arashloo et al. [5] propose a domain-specific congestion specification language and offload it into the network card. Narayan et al. [122] propose programming abstractions and an API to specify congestion controls. Finally, Hong et al. [70] present a programming model to simplify the development for a large number of heterogeneous, distributed devices in the *Internet of Things*.

*Congestion control specification language*

*Domain-specific languages*

We note that the aforementioned programming models and languages use concepts of the software engineering domain to different extent. While there is no single, agreed definition for *domain-specific languages* in the domain of software engineering, Deursen et al. [191] propose the following definition:

“A domain-specific language (DSL) is a programming language or executable specification language that offers, through appro-

priate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.”

Following this definition, the presented examples represent *domain-specific languages*, as they offer notations and abstractions for a particular problem domain and are usually introduced with a corresponding execution environment. The presented overview shows that programming models and languages as abstractions for communication systems gained traction in recent years. Furthermore, these examples confirm that programming models provide powerful abstraction and are a suitable or at least promising approach for communication system research and development.

*Programming models proved to be powerful abstractions for communication systems*

## 3.3 PROGRAMMING MODEL DESIGN

In the following, we present our proposed programming model design, including the scheduler specification language, the calling model, and APIs. The programming model and the specification language are designed to meet the requirements as presented in Section 3.1. A detailed language overview, scheduler examples, and API tutorials are provided at <https://progmp.net>. Appendix A.5 provides an EBNF-based syntax specification.

## 3.3.1 Model of the Scheduling Environment

In this section, we analyze and model the scheduling environment based on the implementation as presented by [8]. Fundamentally, the Multipath TCP scheduler decides *when* to send *which* packet. The scheduler decouples two building blocks: *i*) the sending queues and *ii*) the subflows (Figure 3.1).

*Decouple sending queue and subflows*

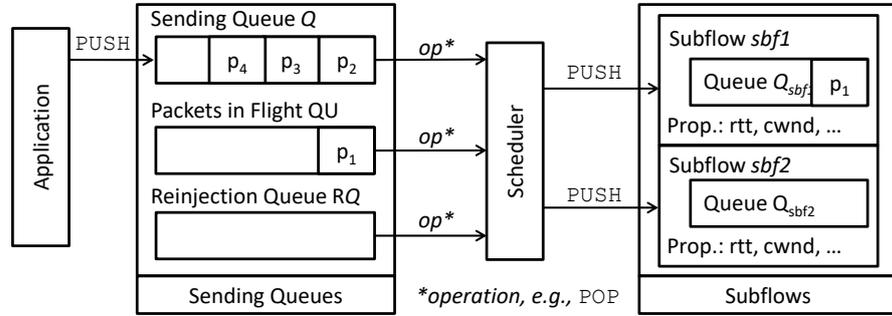


Figure 3.1: Model of the scheduling environment.

The application uses the stream-based TCP socket interface to send data. The network stack fragments this data stream into packets  $p_i$ , which are pushed to the sending queue  $Q = [p_1, \dots, p_m]$ . The scheduler takes these packets from  $Q$  and pushes them into subflow queues. For each packet, the scheduler chooses at least one subflow out of the set of available subflows  $S = \{s_1, \dots, s_n\}$  based on the subflow's properties. Packets might be mapped to multiple subflows, e. g., for redundant transmission or loss recovery.

*Sending queue Q*

TCP has to keep sent packets until they are acknowledged. Accordingly, there is a queue  $QU$  for *unacknowledged* packets in flight. Packets that are removed from the sending queue  $Q$  are stored in  $QU$ . In case a subflow suspects a packet drop, e. g., due to three duplicate acknowledgements or a retransmission timeout, this packet is automatically added to the *reinjection queue RQ*. Packets are not reinjected into the sending queue  $Q$ . This does not reduce the expressiveness with regard to sending duplicates, as the packets are still in  $QU$ . Acknowledged packets are automatically removed from *all* queues.

*Additional queues*

Each communication partner runs its own scheduler. Sender and receiver side are decoupled using the data sequence number mapping of MPTCP, as explained in Section 2.1.1. The sender ensures correct sequence numbers regardless of the used scheduling. The receiver ensures in-order delivery to the receiver application based on these sequence numbers.

### 3.3.2 Language Design

**ENTITIES AND PROPERTIES** We model the elements of the previously presented scheduling environment as entities in the specification language. Thus, the language provides entities for the sending queue *Q*, the packet in flight queue *QU*, and the reinjection queue *RQ*. Queues have properties, e. g., they might be *EMPTY*, and provide operations for selecting and removing packets (*GET*, *TOP*, and *POP*). Packets possess properties, such as their *LENGTH*.

Furthermore, the specification language contains the *SUBFLOWS* entity with the set of all subflows. Subflows provide a *PUSH* operation to schedule a packet and have properties, such as the round-trip time *RTT*, the congestion window *CWND* as maintained by the congestion control algorithm, and the number of packets in flight *PACKETS\_IN\_FLIGHT*. The language further provides function-like properties to retrieve properties of subflow and packet combinations, e. g., to check whether a subflows' receive window can accommodate a packet using *HAS\_WINDOW\_FOR*(*<packet>*).

**DECLARATIVE SELECTION** For both the subflow and the packet selection, we rely on a declarative specification with the operations

*Declarative subflow  
and packet selection*

- *FILTER*(*element => boolean\_predicate(element)* ),
- *MIN*(*element => integer\_predicate(element)* ), and
- *MAX*(*element => integer\_predicate(element)* ).

The declarative specification avoids complex and error prone control flows and directly represents the intention of the developer. Listing 3.1 provides an example excerpt of a scheduler specification. Here, the specified scheduler pushes a packet on the subflow with the minimum round-trip time that has a congestion window larger than the packets in flight.

```
1 SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.PACKETS_IN_FLIGHT).
2 MIN(sbf => sbf.RTT).PUSH(Q.POP());
```

Listing 3.1: Excerpt of a scheduler specification that pushes packets on the subflow with minimum round-trip time (RTT) that has not exhausted its congestion window.

**VARIABLES, TYPE SYSTEM, AND STATE** Dynamic memory allocation is often time consuming and tends to introduce unpredictable delays. Therefore, the specification language is designed to avoid dynamic memory allocation at runtime and to be executable with static memory allocation at *scheduler initialization time*.

*Avoid time  
consuming memory  
allocations*

An early language prototype did not provide variables to keep the language minimal. We found, however, that the support of variables reduces the complexity of our specified schedulers. Based on this experience, the specification language supports variables to store intermediate results *during a single*

*Single-assignment  
variables*

*scheduler execution* in a single-assignment form. Listing 3.2 shows an illustrating example with and without variable support. The runtime environment can allocate the required memory at the initialization of the scheduler and reuse the allocated memory per scheduler execution.

```

1 /* Chosen approach with variables */
2
3 VAR sbfCandis = SUBFLOWS.FILTER(sbf => sbf.CWND > 10);
4
5 IF(!sbfCandis.IS_EMPTY) {
6     sbfCandis.FILTER(...)
7
8 /* Compared with the worse alternative without variables */
9
10 IF(!SUBFLOWS.FILTER(sbf => sbf.CWND > 10).IS_EMPTY) {
11     SUBFLOWS.FILTER(sbf => sbf.CWND > 10) ...

```

Listing 3.2: Illustration of the advantages of a scheduler specification that supports variables. Here, the variable `sbfCandis` stores intermediate results.

*Static type system*

The language provides an implicit and static type system and thereby avoids dynamic type errors, as each variable has the implicit type of its initial assignment. We found that the built-in types are sufficient to express schedulers; user-defined elementary and composite types are neither required nor supported. In contrast to the established implementation of schedulers in C, (type) cast errors, memory leaks, and dangling or wild pointers are impossible without the need for time consuming garbage collection.

*Registers keep state*

Variables do not keep state *between* scheduler executions to prevent stale references to subflows and packets. Instead, each scheduler keeps a limited state of integer values, denoted as `Registers`, between the scheduler executions. In retrospective, a *limited* number of variables that keep state in combination with a well-defined handling of stale references might simplify certain scheduler designs and should be considered for future work. In Section 3.3.4, we present how these registers are additionally used for the application API.

*Limited side effects...*

**SIDE EFFECTS** For simple reasoning on schedulers, only `PUSH` and `SET` operations have side effects.<sup>4</sup> Furthermore, these operations with side effects must not be used in conditions or the predicates of `FILTER`, `MIN`, and `MAX` statements. This avoids common pitfalls, e. g., packets are not accidentally removed due to a statement such as `IF (Q.POP().SENT_ON(sbf))`. Additionally, subflow and packet properties as well as variables are *immutable* during a single scheduler execution. The combination of these restrictions enables sophisticated optimizations, e. g., many operation results may be cached and many operations may be evaluated lazy or in an arbitrary order. These restrictions do not negatively affect the expressiveness, but help the runtime environment and the developer to reason on schedulers.

*... and immutable values...*

*... simplify reasoning.*

<sup>4</sup>Note that while the `SET` statement is not possible in lambda expressions according to the grammar in Appendix A.5, the prohibition of `POP` in a condition or `FILTER` is context sensitive and not part of our provided grammar. A more elaborated grammar might express this constraint at the cost of simplicity and clarity.

**CONTROL FLOW** In addition to the declarative operations, we found that the specification language requires simple control flow primitives, such as `IF . . . ELSE . . .`, to simplify the development of schedulers and to provide the necessary expressiveness for various schedulers. Furthermore, the language provides a `RETURN` statement to avoid deep nesting of other control flow operations. Listing 3.3 provides an example for illustration.

```

1 IF (SUBFLOWS.EMPTY) { /* Nothing to do */
2   RETURN;
3 }
4
5 IF (!RQ.EMPTY) {
6   /* Handle reinjection queue */
7 } ELSE IF (!Q.EMPTY) {
8   /* Handle sending queue */
9 } ELSE {
10  /* We might send redundant data */
11 }

```

Listing 3.3: Illustration of the `IF . . . ELSE . . .` and `RETURN` primitives.

An early prototype of the language did not support loops to keep the language minimal. We found, however, that a `FOREACH`-loop, as shown in Listing 3.4, significantly simplifies the specification of many schedulers. The language does not support `FOR`-loops with an index, as we cannot verify that these loops terminate and they are commonly used to reimplement already provided declarative selections. Focusing on a small language, we found that neither *functions* nor *recursion* are necessary to express schedulers.

*Limited loops*

*Neither functions  
nor recursion*

```

1 FOREACH(VAR sbf IN SUBFLOWS) {
2   sbf.PUSH(Q.TOP);
3 }

```

Listing 3.4: Illustration of the `FOREACH` control flow primitive.

**NULL HANDLING** An early language prototype did not provide `Null` values. We found, however, that the support of `Null` values in combination with graceful but well-defined error handling simplifies the specification of schedulers and enables efficiency improvements during the execution. Listing 3.5 shows an illustrating comparison.

*Graceful but  
well-defined  
Null-handling*

```

1 /* Implemented version: */
2
3 /* Here, MIN might return NULL without a scheduler failure */
4 SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.PACKETS_IN_FLIGHT).
5   MIN(sbf => sbf.RTT).PUSH(Q.POP());
6
7 /* Not implemented version (worse alternative): */
8
9 VAR sbfs = SUBFLOWS.FILTER(sbf=> sbf.CWND > sbf.PACKETS_IN_FLIGHT);
10
11 /* Without this condition, the scheduler might fail at runtime */
12 IF(!sbfs.EMPTY) { sbfs.MIN(sbf => sbf.RTT).PUSH(Q.POP()); }

```

Listing 3.5: `Null` handling example.

In *ProgMP*, the execution of a PUSH operation on NULL does not evaluate the predicate and is therefore ignored. Thus, the POP operation in the example in Listing 3.5 is not executed and the packet is not removed from  $Q$  in case there is no subflow available.

### 3.3.3 Scheduler Triggering and Execution

The *per packet* scheduling decision should be based on the freshest available information and should be taken timely before the packet is actually pushed on the wire. Thus, *scheduling packets in userspace at the moment the data is pushed by the application is not sufficient*. For example, the scheduler would fail to take a reasonable scheduling decision in case all subflows are congestion limited at the time the application pushes data. In this work, we extend the implicit model of Barre et al. [8] and trigger the scheduler execution by a number of events that include the arrival of new packets in  $Q$  or receiving acknowledgements (Figure 3.2).

To simplify the scheduler development, the scheduler is not required to handle all packets in  $Q$  upon one execution. Hence, a scheduler execution rather focuses on a single or a few packets. Whereas a naive scheduler approach might schedule exactly one packet per execution, a single scheduler execution in our programming model can schedule no, one, or even multiple packets per scheduler execution by varying the number of PUSH invocations. We found that this concept fundamentally simplifies many schedulers and discuss the performance impacts of this design decision in Section 3.4.

*No, one, or multiple  
packets per  
execution*

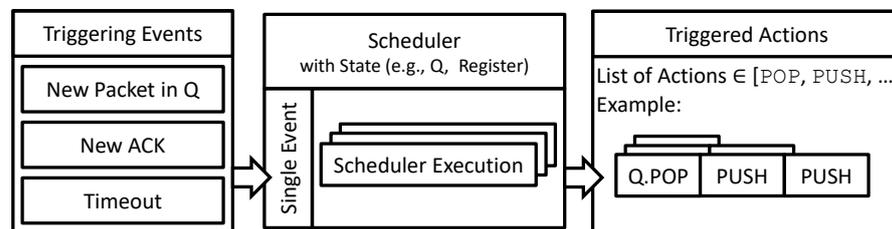


Figure 3.2: The scheduler execution is triggered by various events. A single scheduler execution might trigger multiple actions, e.g., schedule multiple packets.

### 3.3.4 API for Application-aware Scheduling

In the following, we discuss requirements for an extended *scheduling API* and present a corresponding API as part of our programming model, which significantly expands the design space of MPTCP schedulers.

*Enable  
application-aware  
scheduling by ...*

**API REQUIREMENTS** The scheduling API has to allow the application to interact with the scheduler and to provide all important information. Consider, for example, a database where *small requests* that usually consist of a few packets would significantly benefit from redundancy while introducing

a limited overhead. In contrast, *heavy* database *responses* can be transmitted throughput-optimized on the same connection. This example motivates the need for a per-connection scheduler choice and means for the application to inform the scheduler about changing intents. A scheduling intent is an information provided by the application to the scheduler to request a specific scheduling behavior. Note that for a differentiated behavior on packet granularity, e. g., pushing latency sensitive packets, we require the application to be able to provide per-packet scheduling intents. Based on this, we propose an extended API for the interaction of the application with the scheduler.

**CHOOSING A SCHEDULER** The extended scheduler API supports a per-MPTCP-connection scheduler choice permitting a concurrent use of different schedulers for different connections. The application is allowed to *load its own, application-defined* schedulers or reuse loaded schedulers to reduce compilation overhead. Switching schedulers at runtime is discouraged, as we experienced that this provokes inconsistent states, e. g., due to different assumptions and strategies of schedulers. Instead, we encourage to rely on setting registers, as presented next and revisited in Section 5.12.

*... choosing a scheduler per connection...*

**SETTING REGISTERS** An application can *set registers* in the scheduler. This enables, for example, different scheduling modes. Such modes comprise, e. g., an aggressive retransmission mode for latency sensitive content, or a *data flushing* mode that shortens transmission time when the application has no more data to send.

*... setting registers in the scheduler...*

**PACKET PROPERTIES** An application can set packet properties for differentiated handling of packets. High priority packets may, for example, be transferred redundantly or additionally sent on backup subflows. Further, packets can be forced on certain subflows depending on their properties, e. g., to ensure privacy.

*... and setting packet properties.*

### 3.3.5 Programming Model Discussion and Future Work

Table 3.1 provides a summary of the main programming model design decisions and concepts, as presented in this section. The expressiveness and power of the scheduler programming model are confirmed by the wide range of scheduler innovations presented in Chapter 5. In the following, we discuss potential or discarded extensions and pitfalls we found during the programming model design. Additional examples are provided in Appendix A.2.

**POTENTIAL EXTENSIONS** During the development and evaluation of schedulers, we recurrently wanted to check certain properties of our schedulers, e. g., if the scheduler is work conserving, skips packets in the sending queue, or pushes packets of the reinjection queue if a subflow is available. We implemented checks for these properties in the runtime environment but outside of the language. We envision schedulers to *specify strict modes* and

*Strict modes and asserts*

Table 3.1: Programming model design decisions and concepts.

	Concept
Subflow properties	e.g., RTT, RTT_VAR, CWND, QUEUED, PACKETS_IN_FLIGHT, IS_BACKUP, ...
Packet properties	e.g., LENGTH, SENT_ON(sbf)
Subflow selection	Declarative based on FILTER, MIN, and MAX with packet and subflow properties
Packet selection	Declarative based on FILTER, MIN, and MAX with packet and subflow properties
Variables	Single assignment, implicit typing
Types	Static, non-extensible type system (int, bool, packet, subflow, subflow list, packet queue)
Concurrency	Immutable properties and variables
Side Effects	Restricted to PUSH and SET operations
Exceptions	No exceptions by design
Null	Graceful and well-defined Null-handling
Scheduler execution	None, one, or multiple packets per execution
API	Choose scheduler, set register, set packet property

asserts to express these properties and control the runtime environment, as shown in Listing 3.6. These strict modes and assertions might be combined with sophisticated static verification approaches, as envisioned in Section 9.1.

```

1 WORK CONSERVING SCHEDULER;
2 /* ... some code ... */
3 ASSERT(RQ.EMPTY);

```

Listing 3.6: Strict modes for work conserving schedulers and assertions.

We found the presented Null value handling useful for the specification of novel schedulers in Chapter 5. We considered that PUSH operations additionally return if the operation was executed, i. e., if both a packet and a subflow where available. While this extension might simplify recurring scheduler specification patterns (Listing 3.7), it might increase the complexity to read and understand schedulers for inexperienced users.

```

1 /* The current pattern... */
2
3 VAR minSbf = sbfs.MIN(sbf => sbf.RTT);
4 IF (minSbf != NULL) {
5     minSbf.PUSH(...);
6     RETURN;
7 }
8
9 /* ...might be simplified like this */

```

*Feedback for  
PUSH operations*

```

10
11 IF (sbfs.MIN(sbf => sbf.RTT).PUSH(...)) {
12     RETURN;
13 }

```

Listing 3.7: Example how a the *return* value of the PUSH operation might simplify the scheduler specification.

EXPERIENCED PITFALLS Listing 3.8 shows a recurrently experienced pitfall. Here, the scheduler might DROP an unsent packet from the sending queue in case the filter operation returns no subflows. We identify an evolution of the language design to avoid these pitfalls as future work.

```

1 VAR skb = Q.TOP;
2 SUBFLOWS.FILTER(...).MIN(...).PUSH(skb);
3
4 /* Filter might return an empty result, thus,
5  * packet might be dropped but not sent */
6
7 DROP(Q.POP());

```

Listing 3.8: Unintentional drop of an unsent packet.

DISCARDED EXTENSIONS Due to the single assignment variable approach, we considered to support hoisting to simplify scheduler specifications as shown in Listing 3.9. We discarded this, as we found that this feature reduces scheduler comprehension. We found that the *single* assignment approach and *multiple* declarations with assignments provide an inconsistent language experience.

*Discarded hoisting*

```

1 IF(R1 == 0) {
2     VAR sbfCandis = ...;
3 } ELSE {
4     VAR sbfCandis = ...;
5 }
6
7 sbfCandis.FILTER(...)

```

Listing 3.9: Example how hoisting might simplify the scheduler specification.

### 3.4 EXECUTION ENVIRONMENT IMPLEMENTATION

In this section, we provide implementation details of the scheduler runtime environment and the packet handling at the receiver. Further, we discuss the computational overhead of our runtime environment and show that our runtime environment retains scheduling throughput performance.

#### 3.4.1 Scheduler Location and Calling Model

While designing the implementation of the calling model, we considered two alternatives: *i*) userspace up-call to a userspace scheduler and *ii*) in-kernel processing.<sup>5</sup> While the userspace up-call simplifies the development effort for the runtime environment, it introduces latency and computational overhead. We used a *netlink*-based prototype on the same infrastructure used in Section 3.4.5, where we observed that a single up-call requires about 2.4  $\mu$ s. A single scheduler execution in the Linux kernel implementation requires, however, about 0.2  $\mu$ s on the same infrastructure. Although optimizations for kernel-userspace communication, e. g., batch processing as proposed for the OpenVSwitch [142], reduce this overhead, we decided to implement the runtime environment in the kernel between the sending packet queues and the subflows to provide maximum performance (Figure 3.3).

*A runtime in  
the kernel ...*

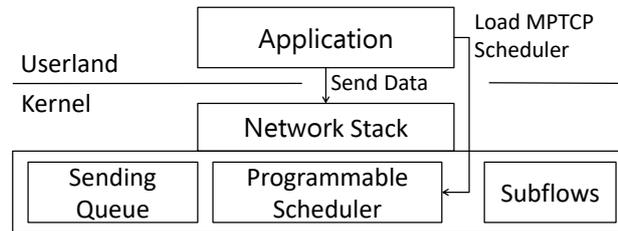


Figure 3.3: The programmable scheduler is located in the network stack between the sending queue and the subflows to enable timely scheduling decisions.

As we anticipate flexible per application and per connection schedulers, schedulers have to be executed in isolation. Failures in one scheduler should not harm other applications or schedulers. To enable applications in a multi tenancy cloud environment to provide and use own schedulers, relying on kernel modules is not possible, as kernel modules do not provide the necessary isolation between applications and tenants. Even a non-malicious implementation error in a C kernel module provided by the user might lead to a *kernel panic* and thereby terminate all running applications or lead to an information disclosure, e. g., by disclosing private memory content of other applications in the network.

*... that ensures  
isolation.*

<sup>5</sup>Note that the programming model abstracts over implementation details, which allows the scheduler developer to be agnostic with respect to both alternatives.

### 3.4.2 Runtime Environment

In this section, we provide an overview of the runtime environment implementation<sup>6</sup>, and discuss notable implementation details, i. e., how to augment PUSH operations and queue handling in the runtime environment API.

#### 3.4.2.1 Overview

We implemented the runtime environment for the programming model with nearly 20,000 LOC in the Linux kernel and integrated it in the existing Multipath TCP Linux kernel implementation version 0.90 (Linux kernel version 4.1.20). Figure 3.4 illustrates the overall implementation. The userland application writes the scheduler specification into the kernel using the *proc* filesystem. The *proc* filesystem is a pseudo-filesystem, which provides an interface to kernel data structures.<sup>7</sup> The runtime environment contains a compiler frontend, i. e., a lexer and a parser, to generate a control flow graph of the input program. We considered to leverage established parser generators, such as *yacc* and *ANTLR*, but found that these tools generate code that is not directly executable in the Linux kernel due to library dependencies.

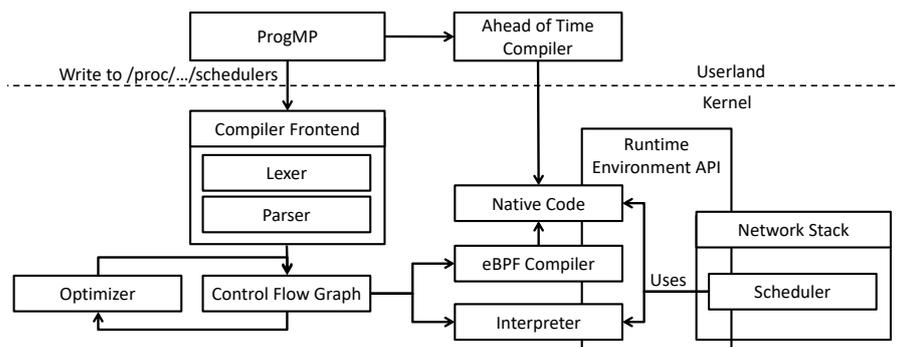


Figure 3.4: Detailed overview of the runtime environment implementation.

The runtime environment implementation further provides an *interpreter* that directly executes the control flow graph representation. This interpreter is a baseline implementation, which does not require changing executable machine instructions at runtime, and works without additional support of the operating system. We further implemented an *ahead-of-time compiler* and an *eBPF-based just-in-time compiler*. The ahead-of-time compiler generates and compiles C functions to be called at runtime. Thus, the programming model is executable without the need of a parser or interpreter in the kernel. Finally, the eBPF compiler translates the intermediate representation into the machine independent eBPF assembly language and uses the eBPF kernel infrastructure [28] to compile to native code. Both the interpreter and native

*Interpreter,...*

*... ahead-of-time,  
and just-in-time  
compiler*

<sup>6</sup>We would like to thank Tobias Erbschäuffer for his implementation support for the *lexer*, *parser*, and *intermediate representations* as part of his activity as student assistant. Parts of the optimization and eBPF compilation were developed by Tobias Erbschäuffer for his Master thesis [S12], which was motivated and supervised by the author of this dissertation.

<sup>7</sup>See <http://man7.org/linux/man-pages/man5/proc.5.html>.

code rely on the *Runtime Environment API* to execute PUSH operations and abstract over implementation details of the underlying queues, as presented in the following.

### 3.4.2.2 Implementation of the PUSH Operation

The programming model supports to schedule none, one, or multiple packets per scheduler execution, as illustrated in Listing 3.10.

```
1 SUBFLOWS.MIN(sbf => sbf.RTT).PUSH(Q.POP());
2 SUBFLOWS.MIN(sbf => sbf.RTT).PUSH(Q.POP());
```

Listing 3.10: Scheduler excerpt that schedules up to two packets per execution.

An analysis of the established scheduler interface implementation [133] shows that the interface is limited to schedule none or one packet per scheduler invocation. To remain compatible with this interface, we considered two alternative implementation approaches: *i)* coroutines or *ii)* decoupling the PUSH operation with a queue.

*Coroutines?*

Coroutines, in contrast to subroutines, keep state between consecutive calls and continue execution at the last state [27]. This makes coroutines ideal to implement iterators. Coroutines are usually implemented using stacks or continuations. In the previous example, a coroutine implementation would return from the scheduler call when reaching the first PUSH and continue at this position when called again. A first analysis in the implementation showed that the execution of the PUSH operation is time consuming. Thus, if we rely on coroutines, we have to ensure that variables do not change between consecutive calls. Listing 3.11 illustrates this, as a round-trip time value change during the execution might lead to an inconsistent state. Furthermore, a coroutine would require to keep the execution state, i. e., the current execution stack.

```
1 VAR sbf = ...;
2
3 IF (sbf.RTT_MS < 100) { PRINT("rtt is less than 100"); }
4
5 /* Time consuming push execution */
6 sbf.PUSH(Q.POP());
7
8 IF (sbf.RTT_MS > 100) { PRINT("rtt is greater than 100"); }
```

Listing 3.11: In a naive scheduler execution environment, subflow properties might change during a time consuming execution of the PUSH operation.

*Deferred execution  
with a queue!*

As we anticipate only a limited number of PUSH operations per scheduler execution, we decided against coroutines and to defer the PUSH operation. We decouple the PUSH operation in the language from the actual execution in the network stack using a small, pre-allocated queue (Figure 3.5). This speeds up the execution due to a higher cache locality and reduces the risk that properties change during the scheduler execution.

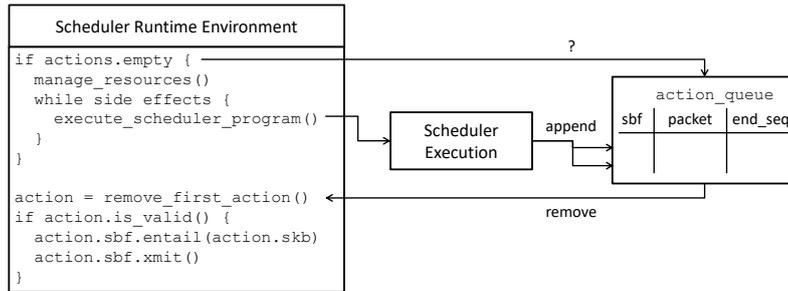


Figure 3.5: Implementation of the deferred PUSH operation with a queue of actions.

The runtime environment ensures that expectable side effects of the PUSH operation are maintained, e. g., as illustrated in Listing 3.12, and that subflows and packets in the action queue are not stale when taken from the queue.

```

1 VAR sbf = ...;
2 VAR skb = ...;
3
4 VAR queued = sbf.QUEUED;
5
6 sbf.PUSH(skb);
7
8 IF (skb.SENT_ON(sbf) AND queued + 1 == sbf.QUEUED) {
9   /* Returns true, as visible side effects are maintained */
10 }

```

Listing 3.12: Example of the maintained visible side effects required for the deferred PUSH execution.

### 3.4.2.3 Implementation of the Queue Abstraction

The proposed programming model enables flexible packet handling in the sending queues. In particular, packets can be removed from the middle of the sending queue or sent without removing them from the sending queue, as illustrated in Listing 3.13.

```

1 VAR topPacket = Q.TOP;
2 VAR sbf = SUBFLOWS.MIN(sbf => sbf.RTT)
3
4 sbf.PUSH(Q.TOP);
5
6 IF (topPacket.SEQ == Q.TOP.SEQ) {
7   /* Returns true, as the top of the queue did not change */
8 }
9
10 /* Remove a packet from the middle of the sending queue */
11 sbf.PUSH(Q.FILTER(skb => skb.USER == 1).POP());

```

Listing 3.13: Example of packet queue operations that go beyond the traditional Linux kernel packet queue operations.

To support these operations, the runtime manages an own `queue_position` pointer in the `sk_write_queue` in addition to the established `sk_send_head`

pointer (Figure 3.6). In conjunction with an additional `in_queue` flag in the `sk_buff` packet, the `queue_position` pointer provides an augmented queue on top of the existing queues. This enables POP operations in the middle of the queue and PUSH operations with TOP packets without removing them from the sending queue `Q`. These changes avoid more complex changes of the existing networking stack. This is important, as, for example, the queue position implicitly maintains the sequence numbers and operations, such as packet fragmentation, operate on these queues.

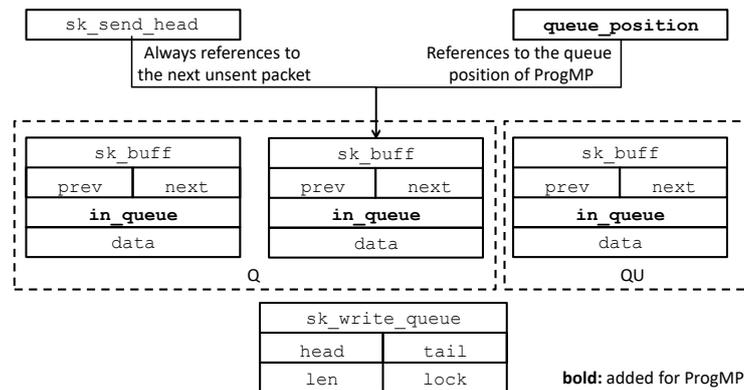


Figure 3.6: Implementation of the augmented queues.

### 3.4.3 Runtime Optimizations and Compilation

We implemented a wide range of optimizations based on the declarative language elements. FILTERS, for example, are evaluated using late materialization. Two additional optimizations are particularly interesting: *i*) constant subflows and *ii*) compressed executions. As the number of Multipath TCP subflows changes rarely, the JIT-compiler optimizes for a constant number of subflows and returns to the original version otherwise. The second optimization increases the number of actions per scheduler execution to compress the number of executions. For example, to push data from `Q` on the subflow that has the lowest RTT *and* that has not exhausted its congestion window, the runtime environment automatically transforms the scheduler to schedule more packets per scheduler execution (Listing 3.14). This reduces the scheduler overhead and increases cache hit probabilities. Note that all optimizations are enabled by the abstractions of the programming model.

```

1 SUBFLOWS.FILTER(sbf => sbf.SKBS_IN_FLIGHT < sbf.CWND).
2   MIN(sbf => sbf.RTT).PUSH(Q.POP())
3
4 /* Is transformed to */
5
6 VAR bestSbf = SUBFLOWS.FILTER(sbf => sbf.SKBS_IN_FLIGHT < sbf.CWND).
7   MIN(sbf => sbf.RTT);
8
9 IF (remainingSpace == 3) {
  
```

```

10 bestSbf.PUSH(Q.POP());
11 bestSbf.PUSH(Q.POP());
12 bestSbf.PUSH(Q.POP());
13 } ELSE IF (remainingSpace == 2) {
14 bestSbf.PUSH(Q.POP());
15 bestSbf.PUSH(Q.POP());
16 } ELSE {
17 bestSbf.PUSH(Q.POP());
18 }

```

Listing 3.14: Automatic transformation to reduce scheduling overhead by scheduling more packets per execution.

To further speed up the scheduler execution, we decided to use *eBPF*, a special-purpose virtual machine in the Linux kernel [28]. This is comparable to recent efforts to speed up P4 with eBPF in the Open vSwitch [141]. The eBPF compilation replaces most interpreter calls with native code and combines scheduler primitives such as FILTER, reducing the number of loops and function calls. The compilation is executed concurrently in a separate thread, therefore not harming network performance. We found the existing eBPF infrastructure in the userspace insufficient because we required to compile to eBPF *in the kernel*. To this end, we implemented an extended version of the linear scan register allocation, specifically, the *Second-Chance Binpacking* algorithm [189], which is computationally superior to iterative computations arising in graph-coloring register allocation.

*eBPF Compilation*

### 3.4.4 API for Application-aware Scheduling

Figure 3.7 shows the implementation of the API for application-aware scheduling. This API consists of the traditional socket, additional *sockopts* to choose the scheduler and set registers and packet properties, as well as an extensive proc-based interface to load schedulers and retrieve performance statistics.

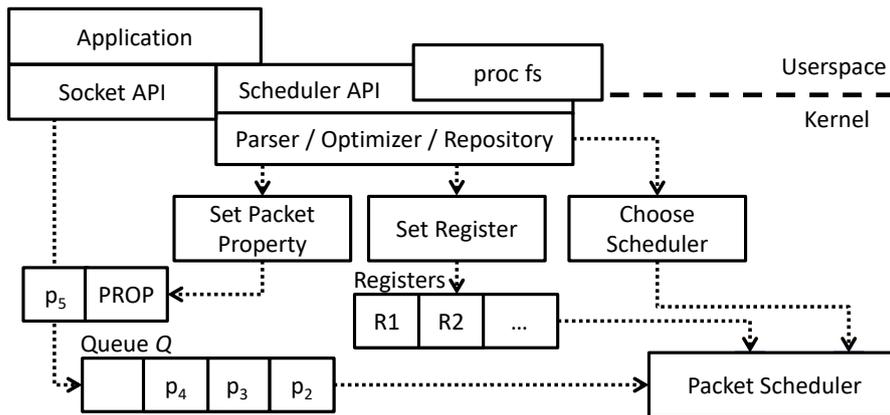
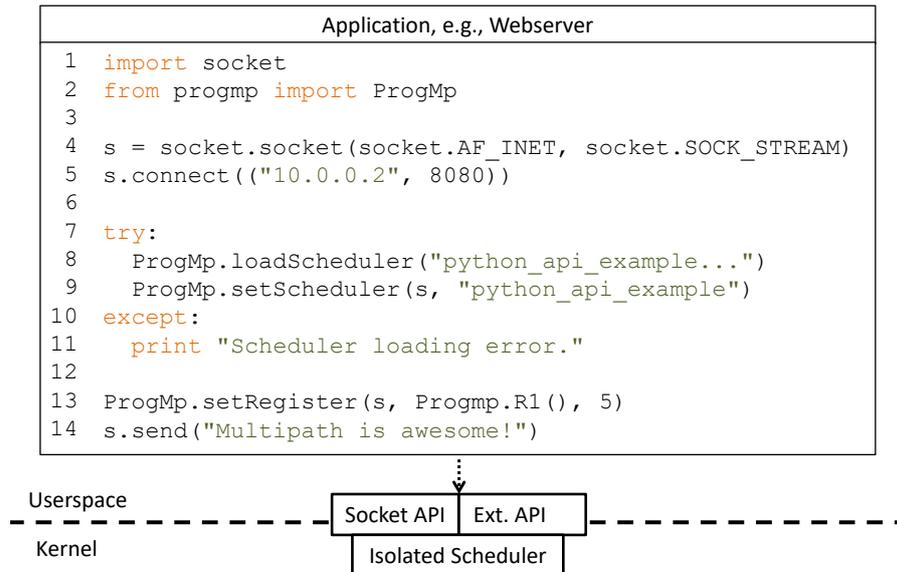


Figure 3.7: Extended scheduler API implementation.

Based on this low-level API, we further provide APIs for Python and C that abstract over the underlying communication details. Figure 3.8 shows an example Python application that controls the *ProgMP* scheduler.

Figure 3.8: Usage example of the Python API for *ProgMP*.

### 3.4.5 Testing and Evaluation

We tested and evaluated the implementation of our runtime environment with regard to the computational overhead, the scalability with the number of schedulers, and the correctness of the implementation.

#### 3.4.5.1 Computational Overhead

We highly optimized our runtime environment implementation, as overhead is a main consideration regarding abstractions. We empirically evaluated our implementation using two bare-metal servers with 64 cores and 128 GB of RAM connected on a 10Gb/s network. We compare the execution times of the C-based default scheduler implementation with a semantically equivalent scheduler specified in our programming model. An isolated measurement of the schedulers' execution times in the Linux kernel is challenging, as collecting and storing measurements introduces significant overhead. Figure 3.9 (top) shows the relative execution times with respect to the default scheduler, where the eBPF-based optimization reduces the computational overhead of the interpreter from  $\sim 44\%$  to  $\sim 25\%$ . Observe that the impact of the number of subflows is marginal. In additional measurements, we found that the execution times and the effectiveness of the implemented optimizations depend on a multitude of factors, such as the network characteristics and the used scheduler operations. Figure 3.9 (bottom) shows the maximum throughput for a CPU-limited connection. It is important to note that the total throughput is not affected by the slower scheduling shown in the top figure.

Our programming model does not target a maximum throughput beyond 10Gb/s for a single connection, but enables schedulers *beyond* the existing throughput optimizations. Our evaluation shows that running our eBPF-based

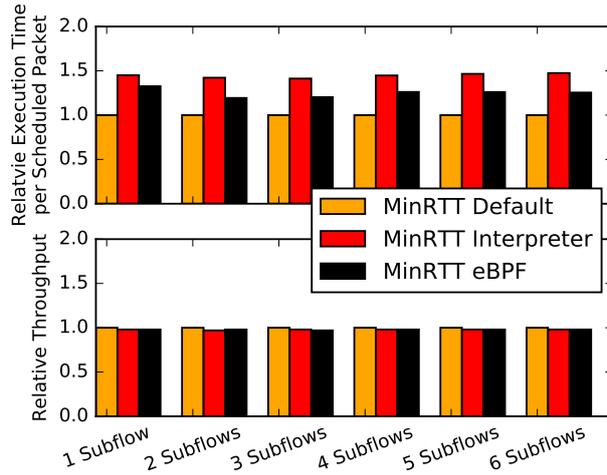


Figure 3.9: Overhead evaluation of our implemented runtime environment for the programming model.

implementation, the overhead of our programming model abstraction is negligible for the targeted application scenarios.

#### 3.4.5.2 Number of Schedulers

As previously loaded schedulers can be reused, we anticipate a limited number of different, concurrently loaded schedulers. The required memory per scheduler depends on the concrete scheduler, e. g., the round robin scheduler requires 3048 byte. Each instantiation of the scheduler requires 328 byte in addition. Compared to the overall network stack, the memory overhead of our runtime environment does not restrict the adoption of our programming model.

#### 3.4.5.3 Testing

The presented runtime environment is a complex implementation with many dependencies, and therefore requires systematic testing. During the development, we relied on the following three testing approaches. First, we used the reproducible executions in the *direct code execution* (DCE) environment for the network simulator ns-3 (see Section 2.1.3), as this environment enables to debug the network stack in a deterministic environment. Second, we relied on large experiment studies with *MACI*, as presented in Chapter 4 and Chapter 5. Third, we used *Packetdrill* [18] for systematic tests with crafted input packet traces. Listing 3.15 shows excerpts of an example *Packetdrill* script to tests if the sending queue size changes as expected (see Appendix A.3 for a full code listing). Therefore, the script configures and loads a scheduler, which uses the PRINT statement to print the queue size (lines 3–6), establishes a Multipath TCP connection with two subflows (lines 8–14), sends 2000 byte which should result in two packets due to the maximum segment size of 1500 byte (lines 16–17), and finally checks the PRINT result of the loaded scheduler.

*Debugging in DCE*

*Large experiments*

*Systematic crafted input traces*

```

1 // Test if the sending queue size reaches 2
2
3 // Configure ProgMP
4 +0 'sysctl -w net.mptcp.mptcp_scheduler=rbs'
5 +0 'echo "SCHEDULER two_in_q; PRINT(\"Q.COUNT %u\", Q.COUNT); IF(Q.
    COUNT > 1) { SUBFLOWS.GET(0).PUSH(Q.POP()); SUBFLOWS.GET(0).
    PUSH(Q.POP()); }" > /proc/net/mptcp_net/rbs/schedulers '
6 +0 'echo "two_in_q" > /proc/net/mptcp_net/rbs/default '
7
8 // Establish sockets and two subflows
9 +0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
10 ...
11 +0 < S 0:0(0) win 32792 <mss 1460,sack0K,nop,nop,nop,wscale 7,
    mp_capable key_a> sock(3)
12 +0 > S. 0:0(0) ack 1 win 28800 <mss 1460,nop,nop,sack0K,nop,wscale
    7,mp_capable key_b> sock(3)
13 ...
14 +0 mp_join_accept(10) = 11
15
16 // Write 2000 byte to the socket stream
17 +0 write(4, ..., 2000) = 2000
18
19 // Give some time to trigger retransmissions
20 +0 'sleep 5'
21
22 // Check if the queue size was 2
23 +0 'dmesg | grep "Q.COUNT 2"'

```

Listing 3.15: Excerpt of a *Packetdrill* script to test if the sending queue size changes as expected.

### 3.4.6 Receiver-Side Packet Handling

*Handle as much as possible*

At the receiver side, all received in-order packets should be pushed to the application as soon as possible. The sequence number mapping of MPTCP, as specified in [RFC 6824], provides the necessary information to map subflow sequence numbers to meta sequence numbers given that each packet carries a full mapping that contains the packet itself. We observe, however, that the current implementation in the Linux kernel *does not deliver* all packets as soon as possible. For certain packet losses and out-of-order patterns between subflows, in-order data is not pushed to the application. We found that only *in-subflow-order* packets without gaps are pushed from the subflow to the meta socket, even though out-of-order packets on the subflow might fit in-order in the meta receive queue.

This is caused by the multilayer queue architecture with both a *receive* and an *out-of-order* queue per subflow and meta socket (Figure 3.10). In this architecture, the subflow queues are ordered by the subflow sequence number, whereas the meta queue is ordered by the data sequence number. This implementation is reasonable, as it significantly reduces the computational effort while the phenomenon appears very seldom with today's default scheduler. We found, however, that the enabled wide range of schedulers increases

the chance that this phenomena appears. In the following, we provide a systematic analysis of the out-of-order packet handling at the receiver side and provide an implementation to overcome these limitations.<sup>8</sup>

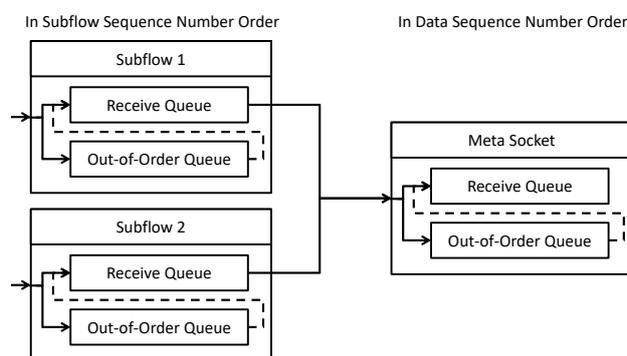


Figure 3.10: Receive queue architecture: Each subflow and the meta socket have an own receive and out-of-order queue. Only in-order packets without gaps are forwarded from the subflow queues to the meta socket.

#### 3.4.6.1 Receiver-Side Packet Handling Analysis

As previously noted, only packets that are in-order on the subflow are pushed to the meta socket. In the following, we provide an analysis of possible queue states that cause unnecessary processing delays. We appreciated the use of the *Packetdrill* [18] tool. *Packetdrill* enables to test the Linux kernel network stack with crafted input packet traces. We used this to extensively test the receiver side packet handling for incoming packet combinations.

**SIMPLEST EXAMPLE** Figure 3.11 illustrates the limitation with the most simple example. Here, a dropped or delayed packet on subflow 2 leads to a gap with regard to the subflow sequence number space. Even though the packet with the data sequence number (dsn) 1001 fits in the global receive space, the implementation handles packets on subflows *after* packets are in subflow sequence number order and without gaps.

**DIFFERING SUBFLOW AND DATA ORDER** Figure 3.12 shows that packets that are ordered by the subflow sequence number in the subflow receive queue do not have to be ordered by the data sequence number. Accordingly, the receiver side packet handling has to consider that the first packet of the out-of-order queue is not always the first packet with regard to the data sequence number.

<sup>8</sup>The *Packetdrill* scripts to replay the analyzed input queue states and the source for our implementation are available at [https://github.com/AlexanderFroemmgen/mptcp\\_ofo\\_queue\\_handling](https://github.com/AlexanderFroemmgen/mptcp_ofo_queue_handling).

<sup>9</sup>The *Packetdrill* script to replay this input queue state is available at [https://github.com/AlexanderFroemmgen/mptcp\\_ofo\\_queue\\_handling/packetdrill\\_scripts/1\\_simple\\_example.pkt](https://github.com/AlexanderFroemmgen/mptcp_ofo_queue_handling/packetdrill_scripts/1_simple_example.pkt)

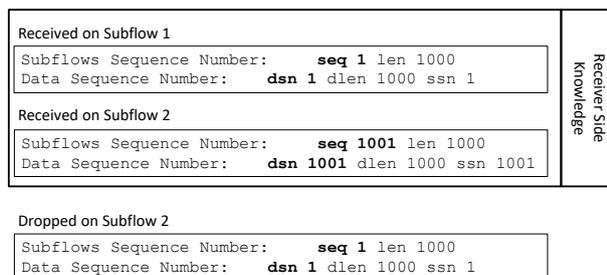


Figure 3.11: Example where the current receiver side does not forward a packet with the data sequence number 1001, even though the packet fits in the data sequence number space.<sup>9</sup>

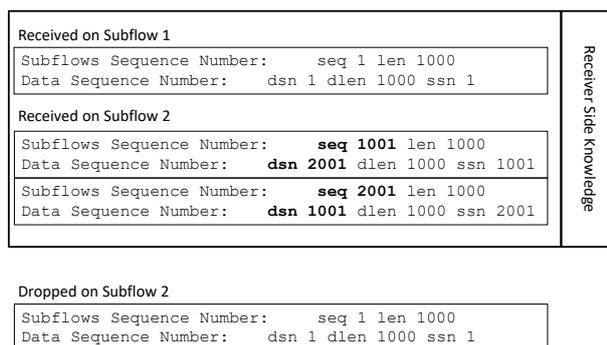


Figure 3.12: The out-of-order queue of a subflow is ordered by the subflow sequence number. Thus, the first packet of the out-of-order queue is not always the first packet with regard the data sequence number.<sup>10</sup>

**PARTIALLY OVERLAPPING PACKETS** Figure 3.13 shows an example of a partially overlapping packet, where the receiver side has to consider that a packet might contribute only partly to the new available data.

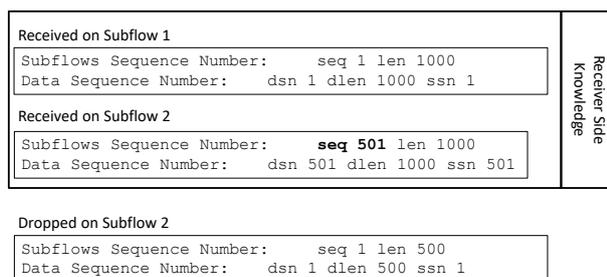


Figure 3.13: Partially overlapping packets might contain additional data.<sup>11</sup>

**MORE DEPENDENCIES BETWEEN SUBFLOWS** Finally, Figure 3.14 shows that a new packet in one subflow might lead to new available packets in

<sup>10</sup>The *Packetdrill* script to replay this input queue state is available at [https://github.com/AlexanderFroemmgen/mptcp\\_ofo\\_queue\\_handling/packetdrill\\_scripts/2\\_differing\\_orders.pkt](https://github.com/AlexanderFroemmgen/mptcp_ofo_queue_handling/packetdrill_scripts/2_differing_orders.pkt).

other subflows. Thus, the receiver side has to recheck if incoming data leads to additional available data in other subflows.

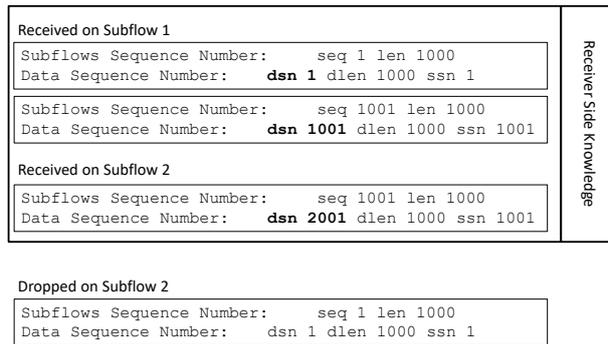


Figure 3.14: In this example, an optimal receiver has to forward the second packet on subflow 1 after handled the out-of-order packet on the second subflow.<sup>12</sup>

**PACKETS WITHOUT MAPPING** Besides these cases, we additionally have to consider packets without a data sequence number mapping. We leave these additional cases as future work.

### 3.4.6.2 Receiver-Side Packet Handling Implementation

We implemented all changes at the receiver-side to avoid unnecessary packet forwarding delays between the subflow and the meta socket. This is computational expensive, as the packets in the subflow queues are sorted by subflow sequence number, but have to be *read* in order of the data sequence number (as shown in Figure 3.12). Figure 3.15 shows the required changes at the receiver side. Note that `ofo_push` only checks all subflows `out_of_order` queues and not the subflow receive queues, as packets in the subflow receive queues are either in the meta receive or the out-of-order queue.

### 3.4.7 Implementation Discussion and Future Work

In this section, we presented our runtime environment implementation for *ProgMP*. This implementation is designed to provide highest flexibility and performance for an isolated scheduler execution. We showed that the implementation fulfills these requirements. In retrospect, we find that the implementation effort was very large and the complexity of the developed source code is very high. For more confined application scenarios and correspondingly adapted requirements, different implementation alternatives might be easier with regard to the implementation effort, the code complexity, and ultimately

<sup>11</sup>The *Packetdrill* script to replay this input queue state is available at [https://github.com/AlexanderFroemmgen/mptcp\\_ofo\\_queue\\_handling/packetdrill\\_scripts/3\\_overlapping\\_packets.pkt](https://github.com/AlexanderFroemmgen/mptcp_ofo_queue_handling/packetdrill_scripts/3_overlapping_packets.pkt).

<sup>12</sup>The *Packetdrill* script to replay this input queue state is available at [https://github.com/AlexanderFroemmgen/mptcp\\_ofo\\_queue\\_handling/packetdrill\\_scripts/4\\_check\\_all\\_ofo\\_queues\\_after\\_new\\_packet.pkt](https://github.com/AlexanderFroemmgen/mptcp_ofo_queue_handling/packetdrill_scripts/4_check_all_ofo_queues_after_new_packet.pkt).

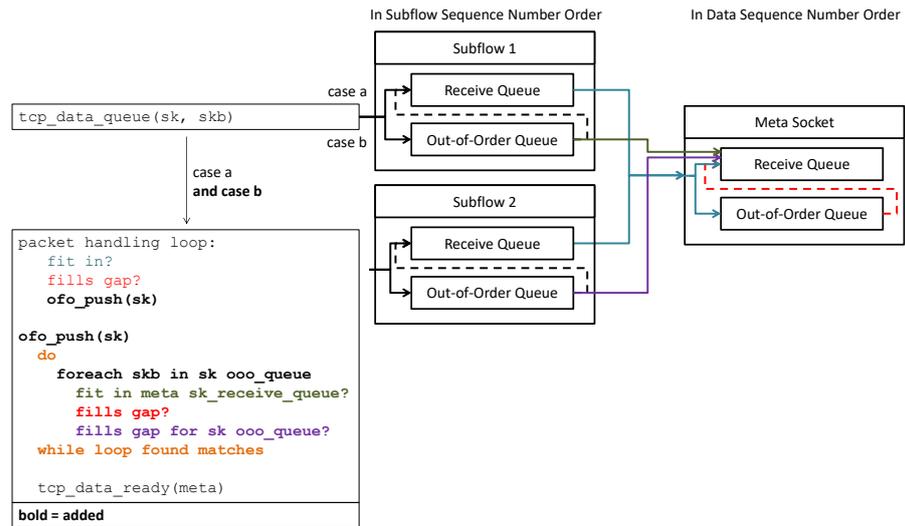


Figure 3.15: Illustration of the added functionality and the achieved packet flow.

the code maintainability. For example, while the approach with the lexer and parser in the Linux kernel ensures isolation between schedulers, as no non-scheduler code can be loaded, a compile infrastructure in the user-land might leverage the same programming model with less effort. Furthermore, our development was driven by avoiding changes in the remaining network stack. We find that a fundamental refracting of the scheduler interface in the Multipath TCP implementation based on our experiences might simplify code complexity.

This is only a limitation with regard to the *implementation effort* that we spent on the reusable execution environment. This is neither a limitation of the programming model nor the usability and efficiency of the implemented execution environment.

In the previous chapters, we identified the need for systematic network evaluations with extensive network experiments. With *ProgMP*, we enable the specification of various MPTCP schedulers. These novel schedulers, however, require careful experimental validations with a variety of traffic patterns in a large number of heterogeneous network environments. This substantiates the significance of our second research question:

RQ II *How can we extensively evaluate and compare the various configurations of communication systems and network protocols in heterogeneous network environments?*

*II Research Questions*

In the following, we provide a detailed motivation and requirement analysis for extensive network experiments that goes beyond the application domain of MPTCP scheduling. We propose a general model, which captures the *recurring* requirements of an experiment driven research process. We present the design and implementation of *MACI*<sup>1</sup>, the first framework for the management, the scalable execution, and the interactive analysis of a large number of network experiments. Finally, we discuss the benefits of *MACI* based on our experiences with various research projects and refer to evaluations with *MACI* in this dissertation, e. g., for systematic evaluations of specified *ProgMP* schedulers or a detailed analysis of DASH video streaming players. Parts of this chapter are published in [F2, F6].

*A general...*

*...framework for extensive experiment.*

*MACI* is open source and publicly available together with detailed examples at <https://maci-research.net> to enable other researchers to increase their efficiency and benefit from our experiences.

*Publicly available*

#### 4.1 MOTIVATION

Communication system research relies on experiments. Accordingly, methods and tools, such as network simulators and their incorporated network models, emerged in the community to enable controlled experiments. There is a plethora of simulators and emulators available, which are tailored for different application scenarios, underlying abstractions, and used network models [2, 21, 63, 106, 123, 129, 153, 183, 192, 212]. Controlled experiments with these execution environments are usually incorporated in the design and development of communication systems to provide recurring feedback.

We note that support for *seamless, systematic, and extensive network experiments and analysis* is missing. This implies *i)* the specification, management and documentation of experiments and their dependent and independent

<sup>1</sup>An initial implementation of *MACI* was implemented by Andreas Bauer as part of his bachelor thesis [S6], which was motivated and supervised by the author of this dissertation.

control parameters, *ii*) the scalable experiment execution, i. e., the parallel execution of a large set of experiments, *iii*) and the interactive analysis of the experiment results based on the previous specified control parameters. We argue that an *integrated* solution is indispensable to increase the efficiency of communication systems research.

## 4.2 OBSERVATIONS AND REQUIREMENT ANALYSIS

To make the case for developing *MACI*, we start by analyzing *recent* observations and *recurring* requirements for conducting network experiments.

*Focus on research*

**IMPROVE RESEARCH EFFICIENCY** The driving requirement for an integrated network experiment framework is to improve research *efficiency*. This allows the researcher to focus on *reasoning*, *questioning* and *improving* the observed behavior. An increased efficiency improves the quality and rigor of evaluations. The specification and management of evaluations has to be intuitive and convenient to ensure wide adoption of reusable components. This includes the initial setup of the framework. Finally, an experiment framework should coordinate experiments and support sharing of results to foster collaboration between researchers.

*Foster collaboration*

*Don't repeat  
yourself*

**OWN RESEARCH EXPERIENCES** Foremost, our motivation for the development of *MACI* was grounded in our own *requirements* and *experiences*. We observed that we recurrently implemented support infrastructure and tools to automate experiments and analyze results during research on various communication systems [F9, F12, F13, F22]. The development of these tools typically started from scratch for every new research project. While we usually started with *just a few scripts*, the tooling evolved with the research project, and finally required a notable fraction of the overall research effort. Even though the overall development of these infrastructure tools is usually straightforward, the development distracts from the actual research and delays the project. Early design decisions for the implementation recurrently turned out to be wrong, causing major refactoring overhead. Further, we note that researchers and developers tend to *write just an own, small* analysis script, which hinders collaboration. Based on these observations, we argue for a *reusable* communication system experiment framework.

*Complex  
dependencies...*

**INCREASING COMPLEXITY** While today's modular, layered communication systems enable optimizations and reduce complexity per layer and concern, research on communication systems has to consider complex cross-layer dependencies. The development and tuning of a transport protocols, for example, has to consider various network environments, application workloads, traffic patterns, and network stack configurations. The performance of a DASH adaptation algorithm, for example, might change significantly when replacing the underlying cubic TCP congestion control with BBR or replacing TCP with alternative transport protocols such as MPTCP and QUIC. The

systematic analysis of these cross-layer dependencies, even if only a single layer should be optimized, requires extensive network experiments.

... require extensive experiments.

**INCREASING INNOVATION SPEED** We observe an increasing speed of network innovations. The recently proposed QUIC transport protocol [60], for example, is designed with the explicit goal to enable frequent iterative improvements [100]. Hence, these *iterative* improvements have to be *repetitively* analyzed with respect to their impact on the application performance, e.g., in the previous *DASH* video streaming example.

Increasing innovation speed ...

Recent advances in network programmability further increase the innovation speed. Our proposed MPTCP scheduler specification language ProgMP, for example, enables rapid scheduler specification. The design decisions of these schedulers, however, have to be evaluated carefully in systematic experiments. Similarly, emerging domain-specific languages for congestion controls simplify writing congestion controls [5]. The specified congestion controls, however, require rigorous evaluations. Since these languages enable rapid specifications of novel communication system algorithms, we argue that we need support for rapid evaluations with systematic experiments.

... and increasing network programmability ...

... require extensive experiments.

**EXTENSIVE EXPERIMENTS** We observe an increasing number of extensive experiment studies in various communication system domains. These studies consist of a large number of individual emulations or simulations.

Recurring extensive experiments for ...

Kakhki et al. [79] identify the rapid evolution of network protocols, i. e., of the QUIC transport protocol. The authors present a rigorous evaluation of various QUIC protocol versions and provide their QUIC specific evaluation infrastructure online<sup>2</sup>. Paasch et al. [134] used an experimental design approach for Multipath TCP. The authors evaluate the dependencies of various parameters, including MPTCP configuration parameters such as buffer sizes, and environment conditions, such as the available capacity and the propagation delay. Hock et al. [69] conducted a detailed testbed-based evaluation of the recently proposed BBR congestion control [19]. In [F9], we contributed to a *DASH* specific evaluation framework and an exhaustive network emulation-based study of *DASH* video streaming. Independently of this work, Zabrovskiy et al. [210] conducted an extensive emulation-based study of *DASH* video streaming. All these examples confirm the need for extensive experiments and contribute frameworks for their confined research domain. A *general reusable* experiment framework for communication systems research remains an open issue.

QUIC,

MPTCP,

BBR,

... and DASH ...

... without a general, reusable framework.

**SCALABLE, PARALLEL EXPERIMENTS** Experiments with many repetitions and variations are time and resource consuming but usually provide better insights and higher confidence. For a long time, the parallel experiment execution was limited by the available infrastructure. Recent infrastructure management advances pave the way for scalable experiment execution.

Increased resource availability ...

<sup>2</sup>The authors made their QUIC specific evaluation framework online available at <https://arashmolavi.github.io/quic/>.

Tools like *OpenStack* and *Proxmox* enable private cloud environments to easily allocate and share computing resources. Public cloud providers apparently provide infinite computing resources.

... and parallel  
nature of network  
experiments.

We note that the workload of evaluations with a large number of experiments is highly parallel, often denoted as embarrassingly parallel [46], as there are no dependencies or communication between the experiments. Thus, parallel experiment execution enables an enormous speed up. We illustrate this for a congestion control experiment: a comparison study of 4 congestion controls with 10 workloads in 10 network environments, where each configuration should be repeated 100 times and each experiment requires about 10 seconds requires over 4 days on a single execution environment. On a cloud platform, the same experiment requires a few minutes and costs less than 10\$.<sup>3</sup> This is in particular important, as researchers require timely evaluations *once* novel concepts are implemented, but usually do not permanently fully utilise their evaluation infrastructure.

Pluggable execution  
environments

**MODULAR FRAMEWORK** The framework has to be modular to support and exchange major components and customize the system. This includes simple APIs for the connection of additional components, e. g., to automatically trigger new evaluations based on previous results. As there are already various optimized execution environments, such as simulators, emulators, hardware testbeds, or real-world infrastructure, the framework should support their *integration* and not their replacement.

**INTERACTIVE ANALYSIS** The framework has to foster a systematic analysis of the conducted experiments. This includes the management, collection, and aggregation of experiment results. Following the experiences of data analytics, business intelligence, and data science, data should be visually inspectable. The researcher should *interact* with the system, e. g., to filter for certain configurations, and trigger the evaluation of additional configurations.

**REPRODUCIBILITY** The conducted experiments have to be reproducible. This is in particular important as research prototype implementations evolve quickly. The framework has to support rerunning previous experiments, which requires access to the used implementations and configurations.

### 4.3 EXPERIMENT-DRIVEN RESEARCH PROCESS

Based on the previous observations and analyzed requirements, *MACI* is designed for an experiment-driven research process. This process relies on recurring evaluations with implementations of systems, protocols, and algorithms. *MACI* supports the entire lifecycle of an iterative research process, including the initial execution and analysis of prototypes with a few varying parameters, the refinement of the underlying algorithms, protocols, and implementations, and the extensive evaluation of matured implementations. Therefore, *MACI*

<sup>3</sup>For a price of 0.50\$ per hour for a medium AWS EC2 instance (January 5th, 2018).

enables the management of experiments, their scalable execution<sup>4</sup>, and the interactive analysis of the experiment results integrated in a seamless fashion. Figure 4.1 illustrates the experiment-driven research process. In the following, we present the design of *MACI* for seamless experiment execution and interactive analysis.

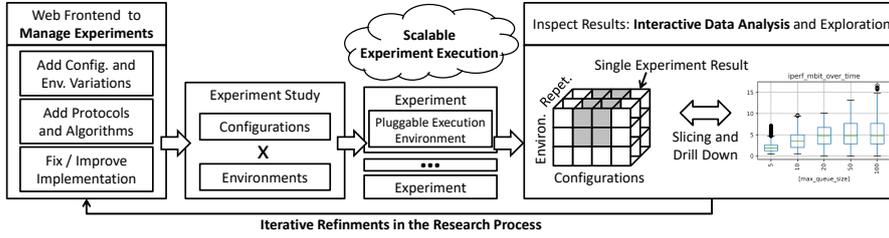


Figure 4.1: Overview of the experiment-driven research process supported by *MACI*. *MACI* enables the management of experiments, their scalable execution, as well as an interactive data analysis to explore the experiment results.

### 4.3.1 A Single Executable Experiment Instance

Based on the previous requirement discussion, *MACI* uses an *Experiment* as smallest, executable instance (Figure 4.2). An experiment consists of the configuration of the analyzed application, the environment conditions of the experiment, and the implicit retrieved version information, such as git commit identifier. The experiment further consists of an experiment script, which specifies the control flow, e. g., by controlling tools such as ns-3, Mininet, or custom simulators. The execution of an experiment results in various measurements, such as target metrics and logging information. In *MACI*, experiments are the smallest, atomic execution units. *MACI* controls the generation and parallel execution of *experiments* in a scalable worker infrastructure.

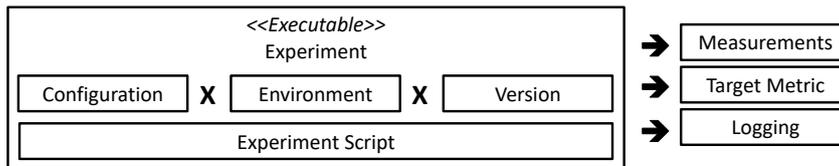


Figure 4.2: A single experiment is defined by the experiment script, a concrete environment specification, a concrete configuration, and the used version of the target application. The execution of a single experiment results in various measurements, including target metrics and logging information.

*MACI* explicitly differentiates between configuration and environment parameters. This should encourage the research to question the experiment setup

*Configuration vs. environment*

<sup>4</sup>It is important to distinguish between scaling a *single* experiment, e. g., a single Mininet emulation with regard to the number of emulated hosts, and scaling the parallel execution of *thousands* of experiments. The former is supported by approaches such as Maxinet [200], whereas the latter is enabled by *MACI*.

and enables *MACI* to automatically prepare for meaningful analysis, e. g., to determine the best configuration per environment as shown in Section 4.3.3.

#### 4.3.2 Structuring Experiments

We structure experiments by decoupling *experiment study templates*, *experiment study configurations*, *experiment studies*, and *experiments* (Figure 4.3). An experiment template is a reusable template that exposes variables, e. g., with regard to the application configuration and the environment conditions. Examples of such templates are test setups for the page load time of web traffic or the average video streaming quality. Experiment configurations are concrete assignments of application configurations and environment conditions for a specific experiment template.

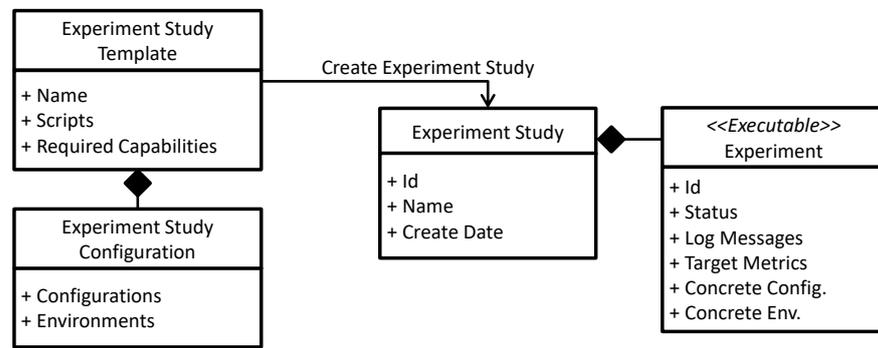


Figure 4.3: Experiment studies are concrete instantiations of experiment study templates and bundle a collection of experiments.

The instantiation of an experiment study template and an experiment study configuration results in an experiment study. Experiment studies consist of a potentially large number of experiments, i. e., combinations of the provided application configuration and environment condition parameters. The presented experiment structure enables researchers to efficiently manage experiments, e. g., by reusing specified templates.

#### 4.3.3 Interactive Data Analysis for Extensive Experiments

In this section, we discuss recurring analysis requirements and their implementation in *MACI*. The data analysis process is inspired by established concepts for the analysis of multidimensional data, i. e., the established OLAP (hyper) cube [26, 53]. The OLAP cube, as illustrated in Figure 4.4, is a multidimensional data cube. In *MACI*, each dimension represents a configuration or environment parameter. The user interface of *MACI* allows the selection of target metrics, as well as the specification of filters and aggregations based on configuration and environment parameters. The result of these operations is represented visually, e. g., as box plots. The interactive analysis and visualization of the data distributions enables researchers to inspect sources of variances by changing filters and aggregations.

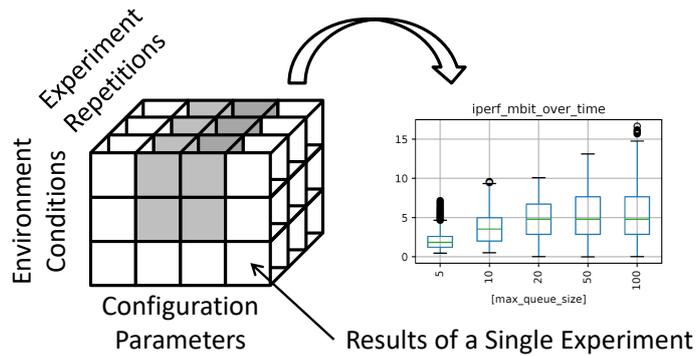


Figure 4.4: The OLAP cube is an established data analysis model in the domain of business intelligence and data warehousing.

*MACI* provides various views to interactively analyze experiment results. These interactive views are *seamlessly* available based on collected experiment results and the provided meta-data from the experiment template. In particular, the data model, e. g., the available configuration parameters, is automatically derived from the specified data in the management frontend. *MACI* automatically supports the following interactive data analysis features:

- Inspect target metric aggregates and distributions

*MACI* enables the comparison of target metrics for different configuration and environment parameters. The analysis is visualized as box plot and can be interactively filtered and aggregated based on the provided configuration and environment parameters to analyze dependencies and inspect the underlying target metric distributions.

- Inspect single experiments

*MACI* further supports a *drill down* operation to select and analyze a single experiment execution, e. g., to plot target metrics over time.

- Inspect trade-offs

We note that applications usually exhibit conflicting optimization metrics. *MACI* automatically generates *Pareto frontiers* to balance conflicting target metrics, such as throughput and latency for congestion controls, loss resilience and overhead for redundant MPTCP schedulers, and video resolution and stalling probability in video streaming.

- Retrieve the best configuration per environment

Inspired by our observation that a single configuration is not always superior in all environments, *MACI* further automatically provides a visual representation of the optimal configuration per environment.

#### 4.4 IMPLEMENTATION

In this section, we present the architecture and implementation of the aforementioned model to facilitate extensive evaluation studies. For the implemen-

tation of *MACI*, we attached importance to *i*) a wide applicability<sup>5</sup>, *ii*) a modular architecture, *iii*) the seamless integration of established tools and frameworks, and *iv*) a comfortable user experience.

Figure 4.5 provides an overview of the *MACI* architecture, which consists of *i*) a central *backend*, *ii*) a web frontend, *iii*) a multitude of worker instances, and *iv*) a *Jupyter notebook server* instance for data analysis. The contribution of *MACI* goes beyond these modules, but stems from their seamless integration to foster the experiment-driven research process. In the following, we present the different modules and their integration.

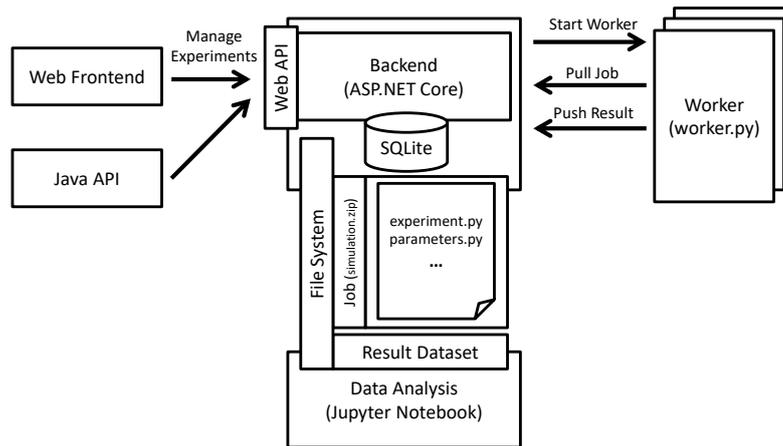


Figure 4.5: The implementation of *MACI* consists of a management web frontend, an *ASP.NET Core* backend, a *Jupyter notebook* server instance for data analysis, and a multitude of worker instances.

#### 4.4.1 Frontend

The web frontend includes an editor and management features for all steps of the experiment lifecycle, e. g., the specification of the experiment and its configuration and environment parameters as well as the monitoring of running experiments. Figure 4.6 shows the management view for experiment study templates. This view includes an editor for the experiment script (left), and the configuration and environment parameters (right). The frontend provides direct feedback, e. g., of the total experiment duration (right), and automates reoccurring manual steps. It is implemented with HTML and JavaScript based on the *Angular.js* and *Bootstrap* framework.

Figure 4.7 shows the management view for a running experiment study. The view provides an overview of the current execution (top right), of all included experiments (bottom right), the executed experiment script (bottom left), and management features, e. g., to add configuration and environment parameters (top left).

<sup>5</sup>We considered a closer integration with cloud computing products such as AWS RDS and AWS S3 for *MACI*'s data storage. However, we preferred to avoid possible lock-in effects to ensure wide applicability.

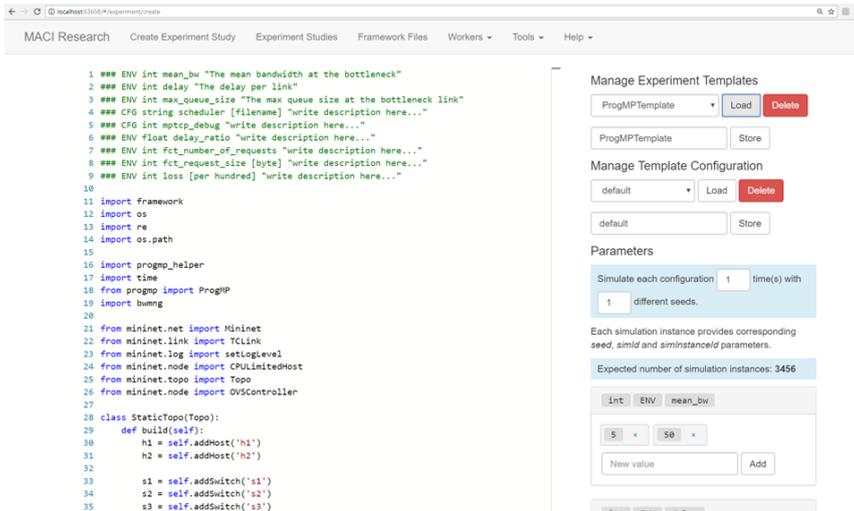


Figure 4.6: Experiment study template view of *MACI*, which enables the management of templates and the creation of experiment studies.

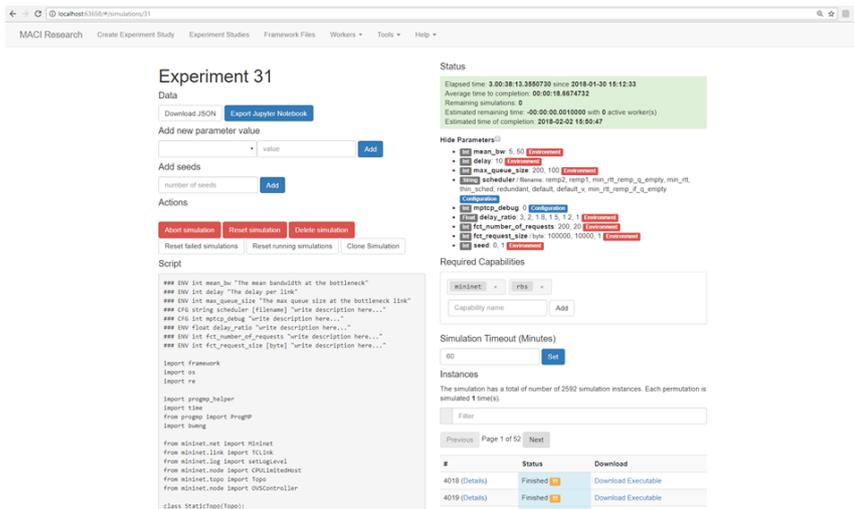


Figure 4.7: Experiment study view of *MACI*.

#### 4.4.2 Backend

The backend is implemented as *ASP.NET Core* server application. *ASP.NET Core* applications are executable on all operating systems that we anticipate for *MACI*, i. e., Windows, Linux and macOS. The server provides a *REST* API for the web frontend and additional management applications, e. g., using a ready to use Java interface. The backend relies on a *SQLite* database as data store, as this allows easy setups and reasonable performance for most workloads. The underlying data store can be easily exchanged for application workloads that go beyond the capabilities of *SQLite*.

#### 4.4.3 Execution Environment Integration

To integrate and control established network simulators and emulators, *MACI* relies on Python scripts. The developer has to specify a single Python script using declared *MACI Variables* for configuration and environment parameters. *MACI* uses dependency injection to control these *MACI Variables* and generate all combinations of configurations and environments. Listing 4.1 shows a simple example.

```

1 # required import of dependencies
2 import maci
3
4 # inform MACI that the simulation starts now
5 maci.start()
6
7 # configure environment and mechanisms
8 framework.param('myParam', default=5)
9 # alternative
10 print "myParam is {{myParam}}"
11
12 # record values
13 value = 42
14 framework.record('key', value)
15 framework.warn('key', value)
16
17 # inform MACI about the end of the experiment
18 framework.stop()

```

Listing 4.1: API usage example for the integration of established experiments.

#### 4.4.4 Scaling Out Experiment Execution

Experiment instances are executed in parallel to speed up the evaluation. *MACI* supports the manual management of workers (servers) as well as the integration with manageable infrastructures, i. e., AWS EC2 and Proxmox.<sup>6</sup>

As many experiments require own operating system modules (e.g., MPTCP experiments in Mininet) and do not support multiple concurrent experiments per host, we follow an *Infrastructure as a Service* cloud model. For experiments with less infrastructure dependencies, we envision more resource efficient serverless computations, such as AWS Lambda.

#### 4.4.5 Analysing a Large Number of Experiments

For the data analysis, we rely on the established *SciPy* [165] data science tool-chain of *Jupyter*, *numpy*, and *pandas*. We discarded commercial alternatives in favor of a publicly available framework. *MACI* provides analysis template scripts that instantly provide interactive analysis features to explore

<sup>6</sup>This work was supported by an AWS research grant. The author of this dissertation would like to thank Amazon.

and drill down experiments intuitively. Figure 4.8 provides an illustrating example with the selection of the filter, grouping, and target metrics (top), the generated scripts for reproducible plot generation (middle), and the generated box plot (bottom). The templates are at the sweet spot of automation and flexibility, as they are easily extendable by researchers with the vast Python software module ecosystem.

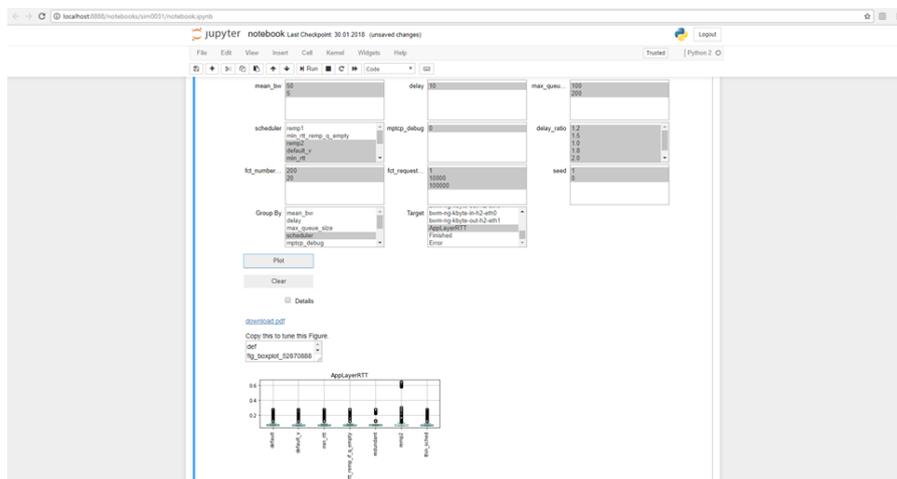


Figure 4.8: Screenshot of the generated analysis interface.

#### 4.5 RELATED WORK

In the following, we provide an overview of related work on experiment frameworks. *MACI* is designed to integrate and benefit from established execution environments [2, 21, 63, 106, 123, 129, 153, 183, 192, 212] and provides *complementary* and orthogonal management and analysis functionality for experiment-driven research projects. In contrast to parameter tuning and performance analysis frameworks [16, 38], *MACI* supports communication system researchers for seamless network experiments throughout the research process. The works in [20, 138] present network monitoring frameworks to *collect* network performance metrics in ns-3. In contrast, *MACI* covers the entire experiment process and includes monitoring *interfaces*. The presented monitoring frameworks for ns-3 might be integrated in *MACI*.

Perrone et al. [139] automate wireless network simulations by guiding the experimenter to avoid methodology flaws in wireless network simulations. Similar frameworks for large-scale simulations and the analysis of results of the simulator ns-2 (ns-3) are presented by Androzzzi et al. [3] (respectively Hallagan et al. [59, 140]). These tools emerged from the same motivation and implement similar concepts as *MACI*. In particular, the authors show that their tools reduce script errors and the required time for network experiments. *MACI* goes beyond these frameworks, i. e., *MACI* covers a wider range of the research process and is more general. The previously proposed frameworks, for example, are deeply integrated with a particular network simulator. The network emulator *Mininet*, for example, is not supported. This makes

these frameworks useless for real network application evaluations, such as our extensive DASH video streaming study in Chapter 8, and evaluations with real operating system network protocol implementations, such as our MPTCP evaluations with *ProgMP* in Chapter 5.<sup>7</sup>

#### 4.6 MACI EXPERIENCES

We greatly benefited from *MACI* during the development and evaluation of Multipath TCP schedulers [F1, F4, F5, F15], the analysis and comparison of DASH video streaming players [F6], and the analysis of a distributed topology graph pattern matching algorithm [F3]. We further provided *MACI* to students and found that *MACI* *i*) increased their speed and systematics by guiding them through the experiment lifecycle and *ii*) helped us to monitor their progress in the management frontend. Table 4.1 provides an overview of the *MACI* applications as of writing this dissertation. Overall, the table shows that *MACI* is a *reusable* tool for a *wide range* of communication system research, as it was successfully used for experiment studies in various domains of communication systems on different layers with different execution environments. For a discussion of our *MACI* experiences with regard to *ProgMP*, *DASH* video streaming, and topology graph pattern matching algorithms, we refer to the corresponding sections of this dissertation, as shown in Table 4.1. In the following, we discuss our *MACI* experiences based on the conducted student theses.

**MPTCP EXPERIMENTAL DESIGN STUDY** In [S6], a bachelor student reproduced the results of a Multipath TCP experimental design study [134]. Besides the necessary evaluation setup for the execution of a single experiment instance, only six lines of code were required to benefit from all *MACI* features. This includes the parallel experiment execution and an analysis with plots that are comparable to the plots of the original publication. Figure A.2 shows a plot, which is automatically generated by the interactive analysis features of *MACI*.

**QUIC MULTIPATH EXTENSION DEVELOPMENT** In [S20], a master student designed and implemented a multipath extension for QUIC, which eventually resulted in [F4]. The student used *MACI* for the iterative development and research process and reports that he found *MACI* helpful. Figure A.3 shows an analysis plot of the final student thesis, which illustrates the application of *MACI* during the thesis. The figure is automatically generated by the interactive analysis features of *MACI* and only slightly modified for presentation purposes. The figure shows a download time comparison of different transport protocols for different file sizes.

---

<sup>7</sup>For the sake of completeness, the network simulator ns-3 provides a *direct code execution* (DCE) extension that enables the usage of Linux kernel transport protocol implementations in ns-3 simulations (Section 2.1.3). In general, however, our claim that the related work is limited to a single simulator, holds true.

Table 4.1: Overview of the applications of *MACI*.

	Execution Environment		Protocol	Student Thesis Publication	
	Layer				
<b>Used by Bachelor and Master Students</b>					
Reproduce MPTCP experimental design study [134]	Mininet	Transport	MPTCP	[S6]	
QUIC multipath extension development	Mininet	Transport	(MP-) QUIC, (MP-) TCP	[S20]	[F4]
Learn and evaluate QUIC congestion controls	Mininet	Transport	QUIC	[S10]	
<b>Used by PhD Students</b>					
Development of novel MPTCP schedulers → Detailed discussion in Section 5.11	Mininet	Transport	MPTCP		[F1, F5, F15]
Analysis of distributed pattern matching → Detailed discussion in Section 7.4.2	Java	Diverse	DPM		[F3]
DASH player analysis and comparison → Detailed discussion in Section 8.4	Mininet	Application	DASH		[F6]

**QUIC CONGESTION CONTROL COMPARISON** In [S10], a bachelor student designed congestion controls for QUIC. The student used *MACI* for the detailed comparison of different congestion controls and reports that he found *MACI* helpful. Figure A.4 shows an analysis of the final student thesis, which illustrates the application of *MACI* during the thesis. The figure is automatically generated by the interactive analysis features of *MACI* and only slightly modified for presentation purposes.

#### 4.7 DISCUSSION AND FUTURE WORK

In this chapter, we presented *MACI*, the first bespoke framework for the *seamless* management, scalable execution, and interactive analysis of a large number of experiments. *MACI* emerged as the result of our experiences, requirements, and learned best practices during various research projects and evolved into a smart combination and integration of established tools to foster rigorous evaluations throughout the research process. *MACI* adopts, for example, the concepts of interactive data analysis from the domains of business intelligence and data science on network experiments. *MACI* follows the zeitgeist of agile development and continuous integration by removing obstacles to fast iterations, which hinder research progress.

While *MACI* provides support for the recurring requirements of an experiment driven research process, we find that many helpful features are still missing. With regard to the expressiveness and model of the experiment configurations and environments, for example, we envision the use of feature models to specify dependencies of configurations. With regard to the scalable execution, spot instances might provide monetary savings for non-urgent experiments. Even though big data analysis frameworks were not required for our use cases so far, we envision that their integration in the seamless research process might be beneficial for certain application scenarios.

Finally, the *MACI* concept of a seamless evaluation integration in the research process might enable the establishment of more advanced experiment concepts in the daily research.

## DESIGN AND ANALYSIS OF NOVEL MPTCP SCHEDULERS

---

In the previous chapters, we presented *ProgMP* for the specification of executable Multipath TCP schedulers and *MACI* for the extensive evaluation of network protocols and applications. In this chapter, we build on these works to tackle the third research question:

III *What is the design space and what are the opportunities of i) general purpose, ii) preference-aware, and iii) application-aware Multipath TCP scheduling?*

III Research Questions

In this chapter, we use *ProgMP* to revisit and extend established schedulers, and systematically design general purpose, application-, and preference-aware schedulers that go beyond the established schedulers. This chapter contributes *i)* an evaluation of the expressiveness, the significance, and the applicability of our programming model *ProgMP*, *ii)* an analysis of the scheduler design space, *iii)* the derivation and specification of novel schedulers, *iv)* the measurement of the achievable improvements, and *v)* an evaluation of *MACI*. Parts of this chapter are published in [F1, F5, F7, F15, F16, F19].

Contributions:  
*ProgMP* and *MACI*  
evaluation and novel  
schedulers

Table 5.1 provides an overview of the discussed schedulers and the outline of this chapter. In Section 5.2, we revisit established schedulers to confirm the expressiveness of the programming model. We improve these schedulers, e. g., by recurrent probing of unused subflows to obtain fresh round-trip time estimates (Section 5.3). We discuss the design decisions for redundant transmission with regard to the choice of fresh and old packets (Section 5.4). We present various preference- and application-aware schedulers to improve the flow completion time in heterogeneous environments (Section 5.5) and to achieve an acceptable round-trip time (Section 5.6) as well as acceptable throughputs (Section 5.7). We introduce a one-way delay-aware scheduler for thin streams (Section 5.8). We further show the benefits of adaptive scheduling during a single connection in a case study of HTTP/2-aware scheduling that demonstrates how all building blocks of the programming model interact to improve page load times and achieve a reduced usage of costly subflows (Section 5.9). Finally, we discuss emulation pitfalls we experienced (Section 5.10), the contribution of *MACI* and *ProgMP* on the design and evaluation of the schedulers (Section 5.11 and Section 5.12), and the potential future work on schedulers (Section 5.13).

### 5.1 EVALUATION SETUP AND SCENARIOS

For our scheduler evaluations and comparisons, we use various applications, traffic patterns, target metrics, and network environments. Most evaluation

	Preferences	Signaled Application Info	Discussed Flavors	Lines of Code with Prefix	
<b>Revisit Established</b>					
<i>Default</i>			1	15	§5.2.2
with backup-semantic	Binary		1	<i>Default</i> + 7 = 22	§5.2.2
<i>Round robin</i>	Binary		2	21 and 35	§5.2.3
<i>Redundant</i> <sup>1</sup>	Binary		1	21	§5.2.4
<b>Novel Schedulers and Features</b>					
Active probing for timely RTT estimates for thin streams			3	31, 34, and 38	§5.3
Balance latency and induced overhead for redundant scheduling			3	17, 21, and 24	§5.4
Minimize flow completion time in heterogeneous environments		End of flow	1	28	§5.5
Preference-awareness to achieve tolerable RTT	Yes	Tolerable RTT	2	23 and 26	§5.6
Preference-awareness to achieve acceptable throughput	Yes	Acceptable throughput	2	22 and 35	§5.7
One-way delay-aware scheduling			1	15	§5.8
Adaptive scheduling for HTTP/2	Yes	HTTP-semantic	1	26	§5.9

Table 5.1: Analyzed Multipath TCP scheduler design space. Most schedulers share a common prefix of 11 lines. This reduces the effectively novel lines of code to 4 till 27 per novel scheduler. See Table 5.2 for a comparison of the lines of code with the existing C implementations.

setups and implementations are available online at <https://progmp.net/evaluations.html>. In the following, we present our recurring evaluation setups and scenarios.

For throughput measurements, we rely on the established *iperf*<sup>2</sup> tool. For measurements of the flow completion time, we implemented small sample applications in Python and C. These applications are configurable with regard to the flow size and flow repetition. For measurements of the application-layer round-trip time, we rely on flow completion time measurements with small flow sizes. We further implemented a constant bitrate stream application to measure application-limited flows. Finally, we use real world applications, such as the *Nghhttpd* HTTP server<sup>3</sup> and the *Postgres* SQL database, to show the applicability for real world applications and traffic patterns.

For the systematic evaluation of different network environments, we rely on Mininet [63] emulations with varying topologies. As Mininet enables the usage of the Linux kernel network stack implementation, it is suitable to run our *ProgMP* runtime environment, as presented in Section 3.4.2. Internally, Mininet relies on network namespaces to provide isolation between emulated hosts and uses TC and netem for traffic shaping.

We carefully designed our emulation studies to avoid measurement errors and discuss experienced pitfalls in Section 5.10. As Mininet emulates multiple hosts on a single machine and is known to be sensitive with regard to the available computational resources, we ensured that the emulation instances were provisioned with enough computation resources.

The mobile real world measurements are conducted with a Laptop running Ubuntu and our *ProgMP* runtime environment as well as schedulers implemented as C modules, i. e., the default and the redundant scheduler. We connected the Laptop with multiple Nexus 5 mobile devices using USB tethering as well as residential WiFi networks. For the cellular connections, we used both the *Deutsche Telekom* and the *O<sub>2</sub>* network in Germany.

*Multiple applications and traffic patterns...*

*... emulated in different network environments ...*

*... and real world environments.*

---

<sup>1</sup>This scheduler was contributed by us but is classified as *established* scheduler, as it was originally developed without *ProgMP*. In fact, the experiences during the design and evaluation of the redundant scheduler motivated the development of *ProgMP*.

<sup>2</sup>See <https://iperf.fr/>.

<sup>3</sup>See <https://nghhttp2.org/documentation/nghhttpd.1.html>.

## 5.2 REVISITING EXISTING SCHEDULERS

In this section, we show the expressiveness of our programming model by revisiting and specifying the MPTCP schedulers that are included in today's MPTCP Linux kernel implementation with the version number 0.94, i. e., the default scheduler, the round robin scheduler, and the redundant scheduler. In the following, we start with a specification of the general reinjection queue handling before presenting the individual schedulers.

### 5.2.1 Preamble and Reinjection Queue Handling

We note that most schedulers use the same, recurring preamble for handling packet reinjections from the reinjection queue (Listing 5.1). This preamble *i*) prioritizes packet reinjection over fresh packets (note that the code is at the beginning of the scheduler and therefore executed first), *ii*) pushes packets on the unsaturated subflow with the lowest round-trip time that did not send this packet so far, and *iii*) removes the packet from the reinjection queue, i. e., does not retransmit the packet on multiple subflows.

```

1  VAR sbfCandidates = SUBFLOWS.FILTER(sbf => !sbf.IS_THROTTLED AND
2    sbf.CWND > sbf.PACKETS_IN_FLIGHT + sbf.QUEUED AND !sbf.IS_LOSSY);
3
4  IF(sbfCandidates.EMPTY) { RETURN; }
5
6  IF (!RQ.EMPTY) {
7    VAR sbfCandidate = sbfCandidates.FILTER(sbf =>
8      sbf.HAS_WINDOW_FOR(RQ.TOP) AND
9      !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
10   IF (sbfCandidate != NULL) {
11     sbfCandidate.PUSH(RQ.POP());
12     RETURN;
13   }
14 }

```

Listing 5.1: Recurring scheduler preamble for packet reinjection.

The recurring preamble is no sign of a deficient language design, but the consequence of today's schedulers. We argue that the design space of packet reinjection goes beyond the currently used strategy, e. g., by retransmitting packets on *multiple* subflows redundantly. Furthermore, removing the recurring packet reinjection pattern from the scheduler specification would hide a fundamental scheduling aspect from the developer.

We omit the preamble in the following schedulers and concentrate on excerpts that reflect the *key functionality* of the schedulers and the programming model. Complete listings of are available in Appendix A.4.

### 5.2.2 (Default) Minimum RTT Scheduler

The current default scheduler in the MPTCP Linux kernel [131] pushes packets from the sending queue on the subflow with the lowest round-trip time

(RTT) and unexhausted congestion window [148]. Listing 5.2 shows a corresponding scheduler specification for the default scheduler (omitting the reinjection queue handling as discussed in Section 5.2.1). Our programming model allows to easily explore a wide range of related schedulers, e. g., considering the RTT variance for jitter reduction or the RTT ratio of the available subflows to minimize packet dispersion.

```

1 IF (!Q.EMPTY) {
2   sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(Q.TOP)).
3   MIN(sbf => sbf.RTT).PUSH(Q.POP());
4 }

```

Listing 5.2: Scheduler specification of the default scheduler.<sup>4</sup>

**BACKUP SEMANTICS** In Section 3.3.8. *Subflow Policy* of [RFC 6824], the `MP_PRI0` option is introduced to communicate subflow priorities. This priority information is used by the default scheduler for a *binary* backup subflow decision. As shown in Listing 5.3, the scheduler filters the set of all subflows to check for non-backup subflows. The scheduler only uses backup subflows when there is not a single non-backup subflow. In the following sections, we show how our programming model provides the flexibility to consider additional metrics, such as round-trip times or capacities, for the backup decision.

*Use backup subflows  
if there is no  
alternative*

```

1 VAR considerBackups = SUBFLOWS.FILTER(sbf => !sbf.IS_BACKUP).EMPTY;
2
3 VAR sbfCandidates = SUBFLOWS.FILTER(sbf => !sbf.IS_THROTTLED AND
4   sbf.CWND > sbf.PACKETS_IN_FLIGHT + sbf.QUEUED AND !sbf.IS_LOSSY
5   AND (sbf.IS_BACKUP == considerBackups));
6
7 sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(Q.TOP)).
8   MIN(sbf => sbf.RTT).PUSH(Q.POP());

```

Listing 5.3: Backup subflow handling of the default scheduler.<sup>5</sup>

**OPPORTUNISTIC RETRANSMISSION** Packets on a slow subflow might cause a receive window block. The *opportunistic retransmission* [148] extends the default scheduler by retransmitting packets from a slower subflow (higher RTT) on a faster subflow (lower RTT) when the receive window is blocked, i. e., using `IF(!minRttSbf.HAS_WINDOW_FOR(Q.TOP))`. Our programming model allows to explore the wide range of related schedulers, e. g., to anticipate such a situation and retransmit packets before the receive window blocks or to selectively retransmit these packets on multiple subflows.

**BUFFERBLOAT MITIGATION** Paasch et al. [133] present a *bufferbloat mitigation scheme*. This scheme compares the smallest experienced RTT and the current RTT to detect bufferbloat [24] and reduces the congestion window

<sup>4</sup>See Listing A.4 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_default](http://progmp.net/progmp.html#dissertation_default).

<sup>5</sup>See Listing A.5 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_default\\_backup](http://progmp.net/progmp.html#dissertation_default_backup).

accordingly. Here, the authors utilize a congestion window that is reduced by a conservative factor such as  $\text{cwnd}_{\text{limit}} = \lambda (\text{sRTT}_{\text{min}} / \text{sRTT}) \text{cwnd}$  with  $\lambda = 3$  as shown by [42]. Optimizations like this can be easily deployed and tweaked in our scheduler model.

### 5.2.3 Round Robin Scheduler

Listing 5.4 presents the specification of the round robin scheduler [133]. Each subflow maintains a quota of packets in the USER-defined per-subflow variable. The quotas are reset when all subflows filled their quota. The implementation provides two tuning parameters, the quota and the behavior in case a subflow is `cwnd_limited`.

The comprehensive specification of the round robin scheduler enables us to reason about details. We note, for example, that in case all subflows are saturated, all quotas are reset. Depending on the experienced traffic patterns, this might result in unexpected behavior.

```

1 /* Tuning parameters */
2 VAR quota = 1;
3 VAR cwnd_limited = 1; /* 1: Fills the cwnd on all subflows. */
4
5 VAR sbfCandidates = SUBFLOWS.FILTER(sbf => !sbf.IS_THROTTLED AND
6   sbf.USER < quota AND !sbf.IS_LOSSY AND (cwnd_limited == 0 OR
7   sbf.CWND > sbf.QUEUED + sbf.PACKETS_IN_FLIGHT));
8
9 /* Take subflow that started to use quota */
10 VAR inUse = sbfCandidates.FILTER(sbf => sbf.USER != 0).GET(0);
11 IF (inUse != NULL) {
12   IF (inUse.CWND > inUse.QUEUED + inUse.PACKETS_IN_FLIGHT) {
13     inUse.PUSH(Q.POP());
14   }
15   RETURN;
16 }
17
18 VAR fresh = sbfCandidates.GET(0);
19 IF (fresh != NULL) {
20   IF (fresh.CWND > fresh.QUEUED + fresh.PACKETS_IN_FLIGHT) {
21     fresh.PUSH(Q.POP());
22   }
23   RETURN;
24 }
25
26 /* Reset quota */
27 FOREACH (VAR sbf IN SUBFLOWS) { sbf.SET_USER(0); }

```

Listing 5.4: The round robin scheduler.<sup>6</sup>

Listing 5.4 represents the round robin scheduler in today's MPTCP Linux kernel implementation. Listing 5.5 shows an alternative specification relying on a cyclic subflow index in register R1. This scheduler skips subflows with an exhausted congestion window.

<sup>6</sup>See Listing A.6 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_round\\_robin](http://progmp.net/progmp.html#dissertation_round_robin).

```

1 IF (R1 >= SUBFLOWS.COUNT) { SET(R1, 0); }
2
3 IF (!Q.EMPTY) {
4     VAR sbf = SUBFLOWS.GET(R1);
5     IF (sbf.CWND > sbf.PACKETS_IN_FLIGHT + sbf.QUEUED AND
6         !sbf.IS_THROTTLED AND !sbf.IS_LOSSY) {
7         sbf.PUSH(Q.POP());
8     }
9     SET(R1, R1 + 1);
10 }

```

Listing 5.5: Alternative round robin scheduler with a cyclic subflow index.<sup>7</sup>

#### 5.2.4 Redundant Scheduler

The MPTCP Linux kernel implementation contains a redundant scheduler, which sends packets redundantly to trade bandwidth for latency. This scheduler is a result of two independent research efforts [F7, 113]. Even though both independent research efforts developed *redundant* schedulers, a detailed comparison of both schedulers is difficult.<sup>8</sup> The term *redundant*, for example, does not specify the behavior given dynamic subflow conditions. Should the scheduler retransmit all packets on a new subflow that becomes available? Our programming model eliminates this ambiguity by providing an *explicit and comprehensive specification*, as shown in Listing 5.6. The implementation in the Linux kernel additionally provides the same backup semantics as the default scheduler.

```

1 FOREACH(VAR sbf IN sbfCandidates) {
2     VAR skb = QU.FILTER(s => !s.SENT_ON(sbf)).TOP;
3     /* Are all QU packets sent on this sbf? */
4     IF(skb != NULL) {
5         sbf.PUSH(skb);
6     } ELSE {
7         sbf.PUSH(Q.POP());
8     }
9 }

```

Listing 5.6: Specification of the redundant scheduler.<sup>9</sup>

<sup>7</sup>See Listing A.7 for a full code example. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_round\\_robin2](https://progmp.net/progmp.html#dissertation_round_robin2).

<sup>8</sup>See <https://github.com/multipath-tcp/mptcp/pull/95> for a discussion on the semantic differences between both original implementations. The discussion illustrates the difficulty of a semantic comparison based on the C implementations.

<sup>9</sup>See Listing A.8 for a full code example. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_redundant](https://progmp.net/progmp.html#dissertation_redundant). A scheduler that considers backup semantics is provided at [https://progmp.net/progmp.html#dissertation\\_redundant\\_backup](https://progmp.net/progmp.html#dissertation_redundant_backup).

## 5.2.5 Discussion

In this section, we showed that our programming model enables a compact and comprehensive specification of today’s Multipath TCP schedulers. An *implementation complexity* comparison between *ProgMP* and the established C implementation is non trivial. The number of required lines of code, a weak complexity indicator, shows that *ProgMP* requires only between 3.4% and 12.4% of a comparable C implementation (Table 5.2). A more compact C implementation might, however, reduce the number of lines for the C implementation. In addition to the original feature commits, today’s C schedulers needed at least twelve commits to fix bugs and add functionalities (Appendix A.6). This supports our claim that C implementations are complex and error prone.

Table 5.2: Comparison of the lines of code required in the original C implementation and our presented specified schedulers.

Scheduler	Lines of code including empty lines			Lines of code without empty lines		
	<i>ProgMP</i>	C	Relative	<i>ProgMP</i>	C	Relative
Default	22	634	3.4%	19	530	3.6%
Round robin	35	302	11.6%	30	241	12.4%
Redundant	28	302	9.3%	25	251	10.0%

5.3 ACTIVE PROBING FOR THIN STREAMS

In the following, we motivate, design, and evaluate an active probing scheduler for thin streams.

5.3.1 Motivation and Analysis

Long, data-intensive connections, such as heavy elephant streams, are usually throughput limited and easily exhaust the congestion window of the lowest round-trip time subflow. Thus, packets of heavy streams are spread over all subflows by the default scheduler (Figure 5.1, top). Thin streams, as predominant for interactive applications, such as online games [55], SSH- and control-connections, however, are long-running connections that exhibit only a few packets per round-trip time (Figure 5.1, bottom). These thin streams usually have tight latency requirements and are sensitive for the application-layer round-trip time rather than the throughput. We note that the default scheduler effectively uses only the subflow with the lowest RTT for thin streams, as a few packets easily fit on a single subflow.

*Thin streams ...*

*... effectively use a single subflow ...*

The default Multipath TCP scheduler relies on the round-trip time estimations of the subflows TCP retransmission timeout calculation. Thus, Multipath TCP *implicitly* obtains subflow round-trip time estimates by sending data on these subflows. As thin streams effectively use a single subflow, estimates of the remaining subflows become stale. Thus, the default scheduler does not notice reducing round-trip times of the remaining subflows, e. g., due to fluctuating cross-traffic and queuing [50, 184].

*... which causes stale RTT estimates.*

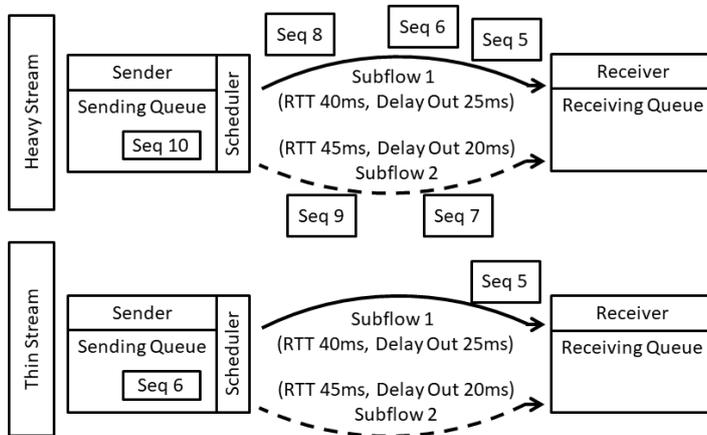


Figure 5.1: Heavy streams are spread over multiple subflows, whereas thin streams usually rely on the minimum round-trip time subflow.

We note that most heavy streams are actually thin in one direction due to a request-response pattern. Many applications and application-layer protocols send small requests and large responses in long-running TCP connections. For example, HTTP/2 connections handle multiple requests and are kept open for a long time to reduce the handshake overhead [120]. Figure 5.2 shows an example where a client C sends a small request to the server S. The

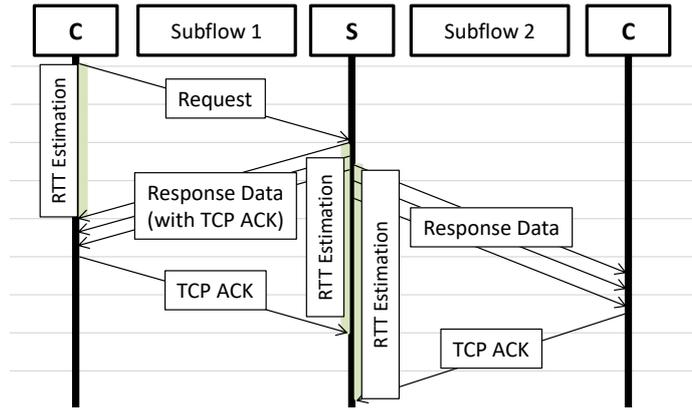


Figure 5.2: Even if one direction of a stream fills all subflows, the other direction might be thin and lack useful RTT estimates.

server replies with a large response. The data packets and the corresponding acknowledgements enable S to estimate the RTT on both subflows. However, C is not able to estimate the RTT on the second subflow with the default scheduler, as this requires sending TCP data and receiving a corresponding TCP acknowledgement. In this section, we propose to *actively probe* unused subflows for thin streams. This probing has to be executed within MPTCP, as alternatives, such as ICMP-based round-trip time estimates, might be prioritized or routed differently by the network.

Active probing ...

### 5.3.2 Scheduler Design

Our previous analysis showed that RTT estimates become stale if subflows are not used regularly. Accordingly, we propose to actively probe unused subflows. In the following, we discuss details of our *redundant probing packets at the end of a burst* design.

...with redundant packets ...

**REDUNDANT PROBING PACKET** We denote the set of subflows that were not used recently as  $\tilde{S}$  out of all subflows  $S$ . A naive probing approach would schedule *the next* packet of the sending queue on a subflow  $s \in \tilde{S}$  as soon as there is at least one stale subflow, i. e.,  $\tilde{S} \neq \emptyset$ . This probing subflow is in general not the subflow with the minimum round-trip. Thus, scheduling a *fresh* packet on this subflow negatively affects the delay for this packet and might increase the flow completion time. We therefore propose to send the probing packet *redundantly* on the best *and* the stale subflows  $\tilde{S}$ .

...at the end of a burst.

**PROBING AT THE END OF THE BURST** Redundancy has to be used carefully to avoid harming performance. Consider a connection is not used for a short time period. Consequently, the scheduler should probe all subflows, as  $\tilde{S} = S$ . If the application sends a burst of packets, a naive *redundant probing packet* scheduler would transmit the first packet redundantly. If the burst saturates the best subflow, however, the redundant packet wastes resources

of the probed subflows. The spare congestion window of the probed subflow should ideally be used for fresh instead of redundant packets. In particular, the last packets of the burst implicitly probe the stale subflows as soon as the best subflow is saturated. We therefore propose to probe stale subflows with redundant packets at the *end of a burst*.

Listing 5.7 shows the specification of the active probing scheduler with *ProgMP*. Lines 10–24 implement active probing in addition to the default scheduler (lines 5–8). The scheduler keeps track of the next probing time per subflow. We use per subflow probing intervals, i. e., a multiple of the subflows RTT, to balance probing aggressiveness for a wide range of environments and avoid synchronized probes on multiple subflows. Line 14 checks if the last packet of a burst is scheduled, i. e., if the sending queue is empty and the TCP PUSH flag is set. Note that relying on the sending queue to determine the end of a burst (Q.EMPTY) is insufficient due to the scheduler timing. When an application pushes data to the socket, the scheduler might be invoked as soon as the first packet is queued. Thus, the sending queue Q contains a single packet when the scheduler is first called even though the application is still pushing additional data. Finally, the scheduler pushes the last packet of the queue (packetToSend) redundantly on stale subflows in case the last packet of a burst is scheduled.

```

1 IF (Q.EMPTY) { RETURN; } /* Nothing to do */
2
3 VAR probingIntervallRttMultiplier = 5; /* Tuning parameter */
4
5 /* Schedule as usual. There is at least one subflow */
6 VAR packetToSend = Q.TOP;
7 VAR bestSbf = sbfCandidates.MIN(sbf => sbf.RTT);
8 bestSbf.PUSH(Q.POP());
9
10 /* Reset subflow probing timeout */
11 bestSbf.SET_USER(CURRENT_TIME_MS + bestSbf.RTT *
12   probingIntervallRttMultiplier);
13
14 IF(Q.EMPTY AND packetToSend.PSH) { /* End of a burst ? */
15   VAR probingSbfs = sbfCandidates.FILTER(sbf =>
16     sbf.USER < CURRENT_TIME_MS);
17
18   FOREACH(VAR sbf IN probingSbfs) {
19     /* Send redundant packet and reset subflow probing timeout */
20     sbf.PUSH(packetToSend);
21     sbf.SET_USER(CURRENT_TIME_MS + sbf.RTT *
22       probingIntervallRttMultiplier);
23   }
24 }

```

Listing 5.7: Specification of an active probing scheduler. Lines 5–8 essentially represent the default scheduler, whereas lines 10–24 implement redundant probing packets at the end of a burst.<sup>10</sup>

<sup>10</sup>See Listing A.9 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_active\\_probing](http://progmp.net/progmp.html#dissertation_active_probing).

### 5.3.3 Evaluation

In the following, we evaluate the impact of active probing on the application-layer RTT, the efficiency, and the maximum achievable throughput.

**PROOF OF CONCEPT** We systematically reduce the round-trip time of the second subflow, starting with a high round-trip time, in a Mininet setup with two disjoint paths. This corresponds to a typical scenario where congestion initially causes high delays on the second subflow [184]. We use a sample application that recurrently sends small requests and waits for a response, thereby tracking the application-layer round-trip time as experienced by this application. Note that the application-layer round-trip time corresponds to the flow completion time for small flows.

Figure 5.3 shows traces of the application-layer RTT for the established schedulers (default, redundant, and round robin), our proposed *redundant probing packet* scheduler, and a slight variation that uses fresh probing packets instead of redundant packets. The default scheduler constantly uses the first subflow and remains unaware of the second subflow’s RTT. The redundant scheduler provides the lowest possible application-layer round-trip time, as the used sample application is not throughput limited. The *redundant probing packet* scheduler uses the minimum RTT subflow at the beginning. When the RTT of the second subflow falls below the RTT of the first subflow, the measurement shows a few smaller application-layer RTT samples caused by the redundant probing packets, before the scheduler eventually prefers the second subflow. The time till the second subflow is preferred depends on the used round-trip time sample smoothing and is an aggressiveness parameters. Finally, the evaluation shows that the naive non-redundant probing introduces high performance degradations, as high RTT subflows increase the flow completion times.

*Default’s RTTs  
become stale*

*Redundant probing  
works!*

*Non-redundant  
probing fails*

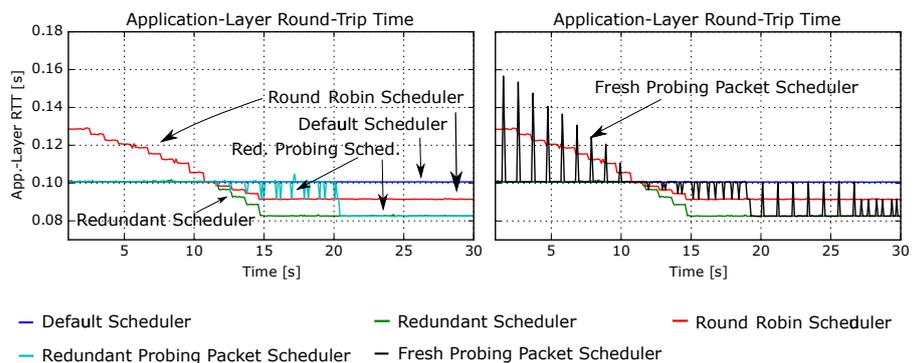


Figure 5.3: Application-layer round-trip time comparison. In the Mininet emulation, the first subflow has a constant 100ms RTT, whereas the second subflow’s RTT decreases from 160ms to 80ms. The measurement shows that the default scheduler is outperformed by the redundant and the active probing with redundancy scheduler.

The round robin scheduler performs bad at the beginning of the trace, as it switches between the low RTT and the high RTT subflow. The detailed analysis shows that the experienced application-layer one-way delay fluctuates with the round robin scheduler (Figure 5.4). Due to a timing dependency of the round robin scheduler of the sender and the receiver, the application-layer round-trip time does not fluctuate.

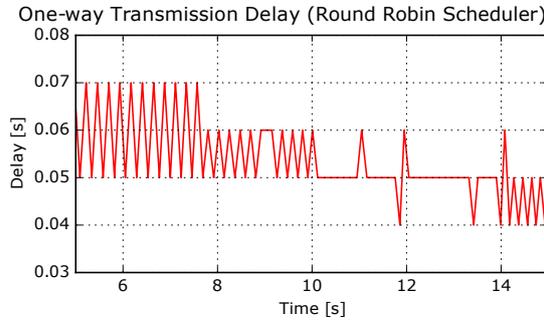


Figure 5.4: Experienced one-way delay with the round robin scheduler.

**END OF BURST PROBING** We further evaluate the design decision to use the *last* packet of a burst for redundant probing. Figure 5.5 compares the flow completion time for an already established MPTCP connection depending on the flow size in a setup with a non-changing 100ms RTT. The evaluation shows that the proposed redundant probing packet at the end of the

*Probing at the end of a burst...*

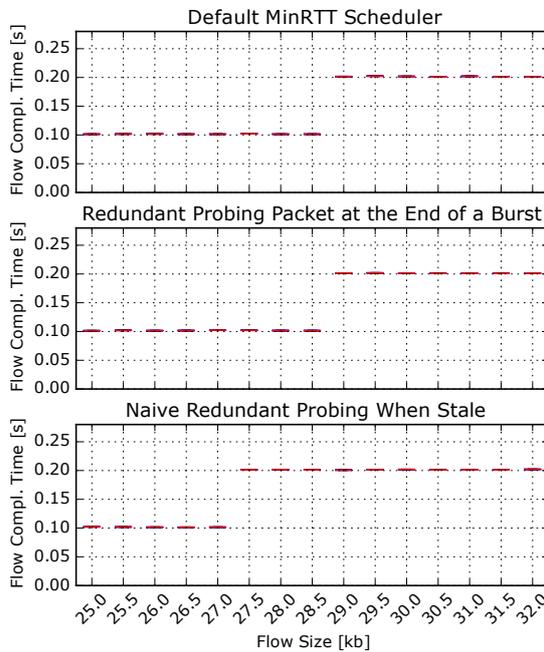


Figure 5.5: The proposed redundant probing packet at the end of a burst does not harm flow completion times, whereas a naive probing as soon as a subflow becomes stale negatively affects the flow completion time depending on the flow size.

... does not harm...

... in contrast to naive approaches.

burst does not harm flow completion times compared to the default scheduler. The naive probing as soon as a subflow becomes stale, however, negatively affects the flow completion time depending on the flow size, i. e., when the redundant packet steals congestion window of fresh packets (for flow sizes between 27.5 and 28.5kb in this example).

**OVERHEAD** For an evaluation of the induced overhead, we compare the totally transferred data on the wire with the actual data for the previous example of Figure 5.3. As expected, the redundant scheduler transfers twice the data in the scenario with two subflows (Figure 5.6). In contrast, the probing scheduler only induces a small overhead. Note that this evaluation shows a single example. In general, the overhead depends on a multitude of aspects, such as the probing interval and the traffic pattern. In particular, the overhead reduces for higher throughput requirements, as shown in the next evaluation.

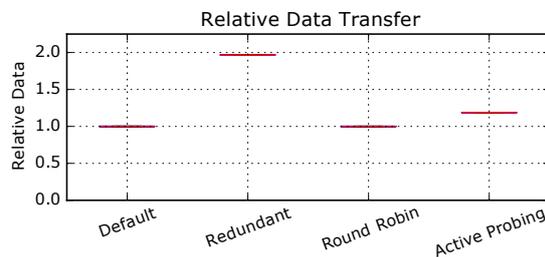


Figure 5.6: Normalized data transfer depending on the scheduler.

**THROUGHPUT** Measurements of the maximum throughput with *iperf* in a scenario with two subflows show only non-significant throughput degradations for the probing scheduler (Figure 5.7). This is reasonable, as the scheduler does not trigger probing packets for the high throughput workload, which saturates all subflows. In contrast, the redundant scheduler provides only half the throughput. As a matter of fairness, we note that the redundant scheduler is designed to sacrifice throughput for lowest possible latency of small flows, as discussed in Section 5.4.

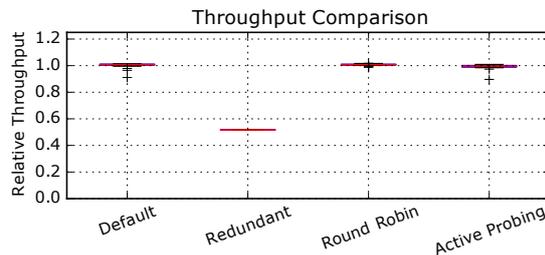


Figure 5.7: Throughput comparison with two subflows.

## 5.4 EXPLORING REDUNDANCY

In the following, we explore the design space of redundant scheduling.

### 5.4.1 Motivation and Analysis

Multipath TCP originates from the need for higher throughput and reliability. Redundant transmission over different subflows – a crude form of forward error correction – was later independently proposed and proved helpful for latency-sensitive applications in lossy environments by the author of this dissertation [F7] and Lopez et al. [113].<sup>11</sup>

The existing redundant scheduler provides full redundancy, as discussed in Section 5.2.4. Thus, the scheduler transmits all packets on all subflows unless the packet is already acknowledged and therefore removed from QU *before* being sent on the slower subflow. We argue that the design space for redundant schedulers is widely unexplored, e. g., with regard to the choice of the next packet, the level of redundancy in environments with multiple subflows, as well as a fine granular control of the redundancy depending on contextual and environmental parameters. Figure 5.8 provides an example for illustration. Here, the packets Seq 3, Seq 4, and Seq 5 as well as an acknowledgement for packet Seq 2 are in flight. The sending queue contains the unsent packets Seq 6 and Seq 7. Which packet should a redundant scheduler transmit next? Should the scheduler prefer a fresh packet from Q, i. e., Seq 6, the already sent packet Seq 5 from QU, or an earlier sent packet, such as Seq 3?

*Large, unexplored design space for redundancy*

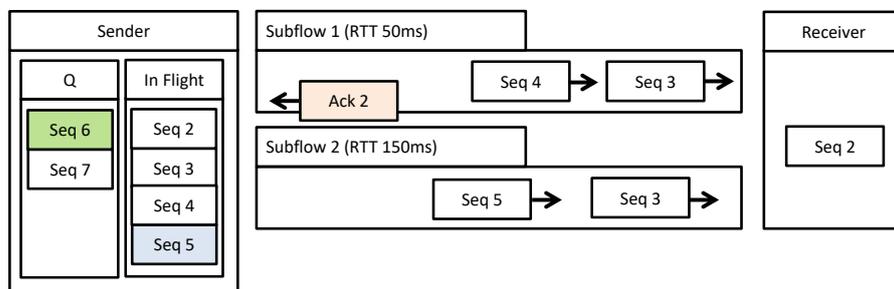


Figure 5.8: Should a redundant scheduler send the fresh packet Seq 6 or the old packet Seq 5 when the acknowledgement arrives at the sender?

### 5.4.2 Scheduler Design

In this section, we propose two novel redundant schedulers. For a revision of the established redundant scheduler, we refer to Section 5.2.4.

<sup>11</sup>Parts of the redundant scheduler as presented in [F7] were implemented by Tobias Erbschäuber as part of his Bachelor thesis [S11], which was motivated and supervised by the author of this dissertation.

*Different  
redundancy flavors*

**OPPORTUNISTIC REDUNDANCY** In general, the redundant scheduler sends all packets redundantly. We note, however, that larger flows fill the congestion windows of all subflows and lead to more queued packets in  $Q$ . Based on this observation, we propose the opportunistic redundant scheduler. This scheduler sends packets on all subflows that have not exhausted their congestion window when a packet is *scheduled for the first time*, as shown in Listing 5.8. For thin flows, this scheduler provides full redundancy. Compared to the redundant scheduler, however, gradually incoming acknowledgements ensure that the scheduler favors *fresh* packets over the transmission of redundant packets in case the sending queue  $Q$  fills.

```

1 IF(!sbfCandidates.EMPTY) {
2   FOREACH(VAR sbf IN sbfCandidates) {
3     sbf.PUSH(Q.TOP);
4   }
5   DROP(Q.POP());
6 }

```

Listing 5.8: Redundancy flavor, denoted as *opportunistic redundancy*, which decides about redundancy the moment the packet is scheduled the first time.<sup>12</sup>

**REDUNDANT IF Q EMPTY** We additionally propose the redundant if  $Q$  empty scheduler. This scheduler always favors fresh packets from  $Q$  and only retransmits packets in case the sending queue  $Q$  is empty (Listing 5.9). Thus, this scheduler behaves like the default scheduler as long as there are unsent packets.

```

1 IF (!Q.EMPTY) {
2   sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(skb)).
3   MIN(sbf => sbf.RTT).PUSH(Q.POP());
4 } ELSE {
5   /* Retransmit on all other subflows...
6    * Start with oldest packet
7    * that was not sent on a sbf that has cwnd */
8
9   VAR skbCandidate = QU.FILTER(skb_ =>
10     !sbfCandidates.FILTER(sbf => ! skb_.SENT_ON(sbf)).EMPTY
11     ).TOP;
12
13   sbfCandidates.FILTER(sbf => !skbCandidate.SENT_ON(sbf)).
14   MIN(sbf => sbf.RTT).PUSH(skbCandidate);
15 }

```

Listing 5.9: Alternative redundancy flavor, denoted as *redundant if  $Q$  empty*, which retransmits if the sending queue is empty.<sup>13</sup>

<sup>12</sup>See Listing A.10 for a full code example. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_opportunistic\\_redundant](https://progmp.net/progmp.html#dissertation_opportunistic_redundant).

<sup>13</sup>See Listing A.11 for a full code example. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_redundant\\_q\\_empty](https://progmp.net/progmp.html#dissertation_redundant_q_empty).

### 5.4.3 Evaluation

**REDUNDANCY IN THE WILD** We evaluate the benefits of redundancy in real world environments with the redundant MPTCP scheduler. Here, we capture the application-layer round-trip time, which corresponds to the flow completion time for small flows. We captured packet traces to determine *what-if* values for singlepath TCP, i. e., pure LTE and WiFi connections.

Figure 5.9 shows a sample measurement between a residential WiFi and LTE network in Heidelberg (Germany) and an AWS EC2 instance in North Virginia in March 2015. Even though the WiFi connection exhibits nearly always lower round-trip times, the rare packet drops and retransmissions lead to a high tail round-trip time for the WiFi connection. As the round-trip times and packet losses of WiFi and LTE show a very low correlation, these WiFi drops are all compensated by the LTE connection. In the measurement, the redundant scheduler reduces the average RTT by 27%, the worst case RTT compared with WiFi by over 78%, and the worst case RTT compared with LTE by over 15%. Regarding the trade-off between minimal average RTT (WiFi) and minimal worst case RTT (LTE), the redundant scheduler provides the best of two worlds and ensures minimal average and minimal worst case RTT at the same time. The cumulated round-trip times show the improvement of the tail of the distribution. Our measurements confirm the results of Chen et al. [23], who reported WiFi packet drop probabilities of 3% (note that our figures show RTT measurements, thus each point represents two packets), LTE packet drop probabilities of 0.1%, and 15ms latency difference between WiFi and LTE.

*Residential WiFi  
and LTE*

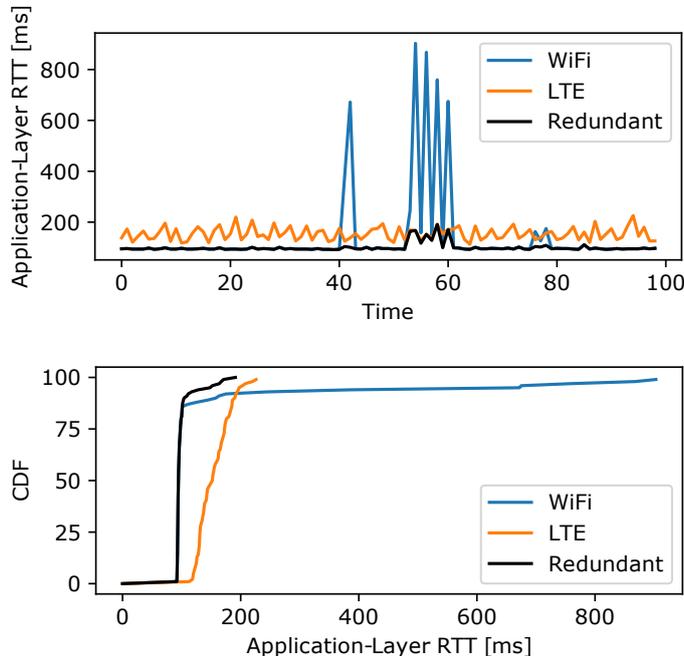


Figure 5.9: Real world measurement of the application-layer round-trip time between Heidelberg and North Virginia. The redundant scheduler compensates WiFi packet drops and reduces the tail round-trip time significantly.

Challenging train environments ...

... benefit significantly from redundancy.

As the redundant scheduler is supposed to outperform singlepath TCP especially in challenging environments, we repeated the application-layer round-trip time measurements to the server in North Virginia in a train moving with up to 160km/h between Heidelberg and Frankfurt. Here, we used two Nexus 5 devices connected with a notebook for two parallel LTE connections. Figure 5.10 shows that the LTE connections experience a lot of packet drops and retransmissions. This leads to extremely high variances in the application-layer round-trip time and a very bad tail latency for singlepath connections. We were surprised that even though both LTE connections used the same carrier, the packet drops show nearly no correlation. This allows the redundant scheduler to more than halve the average application-layer round-trip time and reduce the standard deviation by a factor of 19 for the 100 second trace.

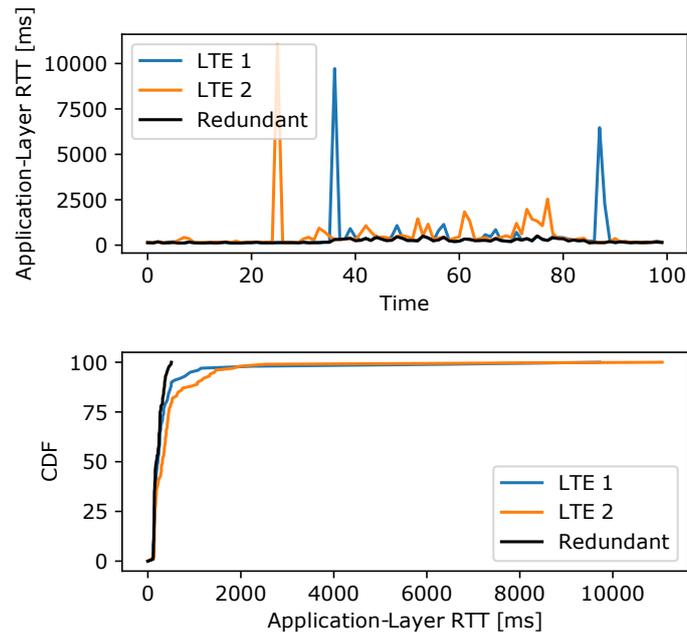


Figure 5.10: Real world measurement of the application-layer round-trip time to North Virginia as experienced in a moving train between Heidelberg and Frankfurt. The redundant scheduler compensates packet drops and large delays effectively.

We repeated the real world measurements multiple times and experienced similar behavior. To further confirm and understand the significant impact of redundancy, we executed additional measurements with the unreliable UDP protocol. The UDP measurements showed that some packets were still successfully transmitted with more than 4 seconds delay. We assume that this artefact might influence the round-trip time estimation and retransmission timeout of the TCP connections.

Systematic redundancy flavor comparison

**CONTROLLED EXPERIMENTS** We used *MACI* for a systematic comparison of the different redundancy flavors. Figure 5.11 shows the average flow completion time depending on the flow size in a Mininet environment with two subflows and 2% loss. Each experiment consists of 200 requests. We

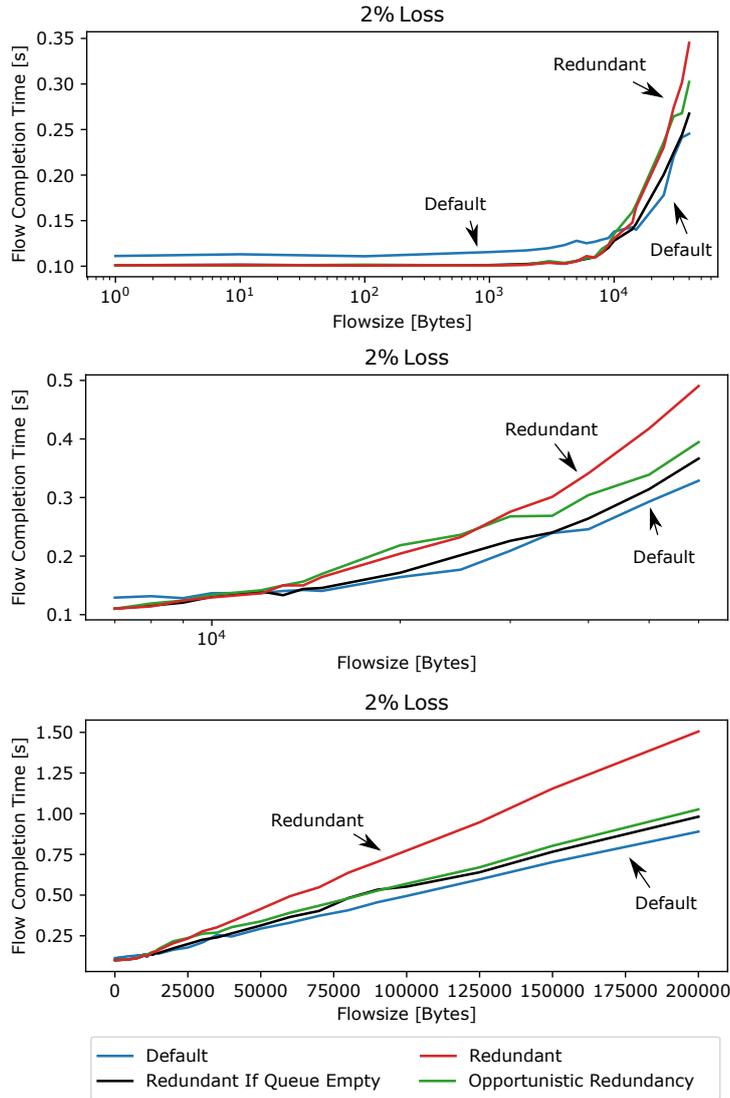


Figure 5.11: Average flow completion time depending on the flow size with two homogeneous subflows and 2% loss in Mininet.

executed these experiments for 32 different flow sizes and repeated each experiment 20 times. The evaluation shows that all redundant schedulers outperform the default scheduler for small flows in a lossy environment. For increasing flow sizes, the opportunistic redundant scheduler and the redundant if Q empty scheduler outperform the redundant scheduler. Here, the default scheduler is slightly better than the best redundancy flavors. A detailed analysis shows that the opportunistic redundant scheduler and the redundant if Q empty scheduler sometimes send redundant packets the moment before new packets are added to the sending queue. Figure 5.12 shows the same experiment with unlimited capacity and without loss. The evaluation confirms that the performance degradation of the redundant schedulers for large flows is caused by the waste of the available capacity.

These experiments compare the flow completion time, i. e., the time till *all* packets arrived. Depending on the traffic pattern and the application, different

target metrics might be more relevant, e. g., the number of *in-order* packets at a certain time. We discuss this for the example of HTTP in Section 5.9.

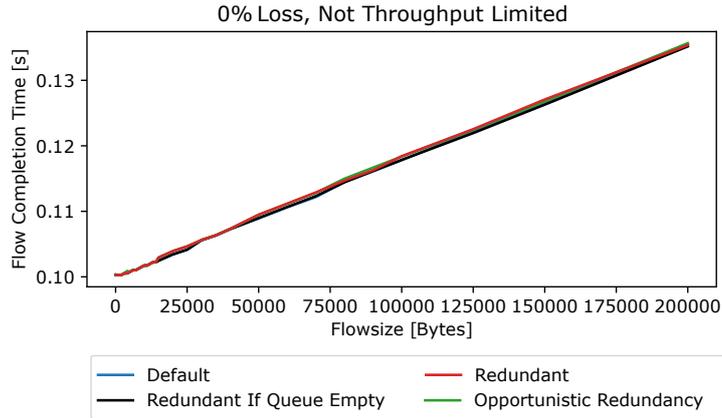


Figure 5.12: Average flow completion time depending on the flow size with two homogeneous subflows and without throughput limitation.

Figure 5.13 shows a comparison of the maximum achievable throughput. We observe that the opportunistic redundant scheduler, the redundant if Q empty scheduler, and the default scheduler provide nearly the maximum achievable throughput for *iperf* measurements with constantly high throughput flows. For bursty flows, however, the throughput depends on fine timing aspects. The opportunistic redundant scheduler and the redundant if Q empty scheduler base their behavior on implicit information, i. e., the queue sizes and incoming acknowledgements. Here, we again note that the schedulers may send redundant packets just before new data arrives in Q. We tackle this problem in the next section with application information.

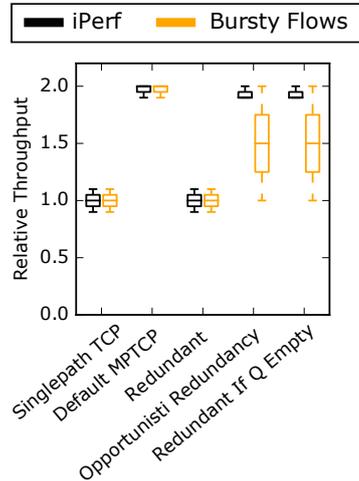


Figure 5.13: Comparison of the maximum throughput normalized to singlepath TCP.

## 5.5 SIGNALING TO BOOST SHORT FLOWS

In the following, we motivate, design, and evaluate the usage of application information and signals to boost short flows.

### 5.5.1 Motivation and Analysis

The significance of MPTCP scheduling decisions increases with the subflow heterogeneity. Specifically, flow completion times of short flows may suffer significantly in environments with heterogeneous round-trip times. Figure 5.14 illustrates a typical situation at the end of a short flow, where the sending queue  $Q$  is empty and packets are still in flight. Here, we do not assume any particular packet scheduling, e. g., packet Seq 35 may have been scheduled on subflow 2 due to an exhausted congestion window at subflow 1. In this simplified example, packet Seq 35 will, in most cases, dominate the flow completion time. However, entirely avoiding the slower subflow is not optimal either as the flow completion time benefits from Seq 32 on the slower subflow.

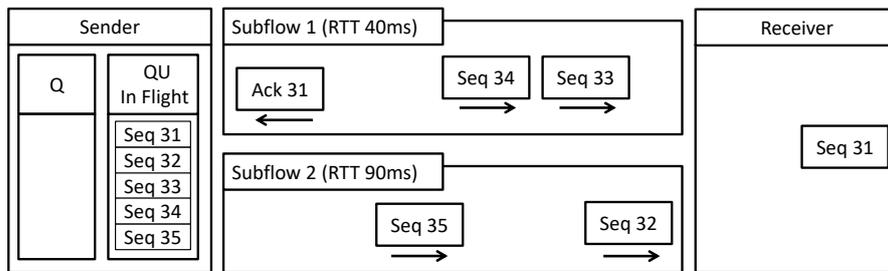


Figure 5.14: The end of a short flow: The impact of heterogeneous subflows on the FCT is most significant when scheduling the last packets of the flow.

In the previous sections, we used redundancy to compensate packet loss and probe subflows. Some of these schedulers used *implicit* application information, i. e., if  $Q$  is empty or the PSH flag of a packet is set, to guess if the end of a flow is reached. In this section, we discuss how *explicit* application signals enable application-awareness and informed redundancy to compensate previous scheduling decisions at the end of a flow.

*Informed  
redundancy*

### 5.5.2 Scheduler Design

The relative ratio of the subflow round-trip times, denoted hereafter RTT ratio, is inherently dynamic in communication networks due to queuing, mobility, and interface-specific properties. This was empirically shown, for example, for mobile networks [32, 35]. Our programming model enables MPTCP schedulers that precisely leverage application information to address subflow heterogeneity. Next, we show how signaling the *end of the flow* by the application leads to improved scheduler performance and optimized FCT for heterogeneous subflows. Listing 5.10 shows a novel selective compensating

*Signal the  
end of the flow*

scheduler that uses this elementary information to compensate for previous scheduling decisions by *selectively* retransmitting packets in flight on subflows where the packets were not sent so far at the signaled end of the flow. To avoid retransmissions of packets that are in flight on a fast subflow, this scheduler *selectively* retransmits packets that are only scheduled on slow subflows. Thus, the selective compensating scheduler retransmits the unacknowledged packets Seq 32 and Seq 35 on subflow 1 but not Seq 33 and Seq 34 on subflow 2 in the example of Figure 5.14.

```

1 /* ... Default scheduler ... */
2
3 VAR minRttRatio = 2;
4
5 /* Use R1 to signal end of flow */
6 IF(R1 == 1 AND Q.EMPTY) {
7   VAR bestSbf = sbfCandidates.MIN(sbf => sbf.RTT);
8   VAR sbfsToCompensate = SUBFLOWS.FILTER(sbf =>
9     sbf.RTT > bestSbf.RTT * minRttRatio);
10
11  /* Packet not on bestSbf but on at least one sbfToCompensate */
12  VAR skbCandidate = QU.FILTER(skb =>
13    !skb.SENT_ON(bestSbf) AND
14    !sbfsToCompensate.FILTER(sbf => skb.SENT_ON(sbf)).EMPTY
15    ).GET(0);
16
17  bestSbf.PUSH(skbCandidate);
18 }

```

Listing 5.10: A scheduler that compensates previous scheduling decisions at the end of a flow.<sup>14</sup>

### 5.5.3 Evaluation

We evaluate the performance of the selective compensating scheduler using a Mininet setup with two heterogeneous subflows while varying the RTT ratio. As a baseline, we consider measurements for the default scheduler, where the flow completion time rapidly increases under high RTT ratios. Being aware of the flow end, the novel scheduler efficiently optimizes the flow completion time under skewed RTT ratios, as shown in Figure 5.15.

Our programming model enables a rapid comparison of different flavors of the selective compensating scheduler. We compare the scheduler with a naive compensating scheduler, which retransmits *all* packets at the end of the flow. The evaluation of the induced overhead (Figure 5.15 right, normalized to the default scheduler) shows that the naive scheduler induces large overheads for small RTT ratios. The selective compensating scheduler, however, is tuned for compensating behavior only for RTT ratios larger than 2 and effectively avoids this overhead for small RTT ratios. A variation of

<sup>14</sup>See Listing A.9 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_compensate](http://progmp.net/progmp.html#dissertation_compensate).

the choice of the retransmitted packet using the TOP packet instead of first (GET(0) in line 15) shows only minor impact on the flow completion time.

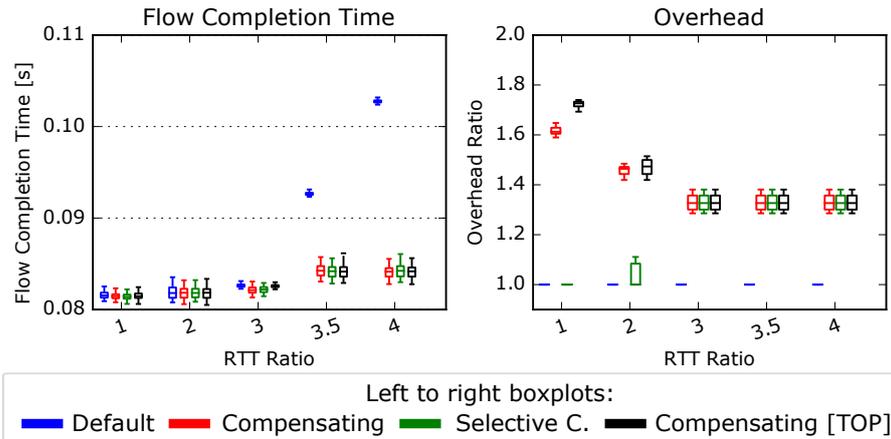


Figure 5.15: Leveraging application information to mitigate the impact of subflow heterogeneity on short flows: The compensating scheduler reduces the flow completion time under increasing RTT ratio by trading flow completion times for transmission overhead.

The selective compensating scheduler shows that the extended scheduling API enables novel schedulers that significantly improve the flow completion time in heterogeneous environments with *informed* redundancy.

## 5.6 BALANCING ROUND-TRIP TIMES AND SUBFLOW PREFERENCES

In the following, we motivate, design, and evaluate schedulers that balance the experienced round-trip time and subflow preferences.

5.6.1 *Motivation and Analysis*

*Latency sensitive applications...*

*... and user preferences.*

Interactive applications, such as voice-based personal assistant systems, usually exhibit request-response communication patterns with a few packets per request. These applications are round-trip time sensitive but usually do not benefit from increased throughput. At the same time, users have preferences with regard to the utilized network resources and corresponding subflows. Paths *in* and *between* data-centers might, for example, be associated with different costs. Users of mobile devices, e. g., often prefer WiFi over metered cellular traffic. A massive, international measurement study [35] of multi-homed wireless MPTCP performance over WiFi and LTE interfaces showed that around 15% of all measurement samples experienced a significantly higher RTT on WiFi compared with LTE. Thus, users do not want to waste their expensive (metered cellular) traffic in case the round-trip time on the preferred subflows is sufficient for the interactive application.

The backup mode of the default scheduler is unable to use the lower round-trip time subflow in these scenarios. As of today, there is no scheduler that schedules packets of interactive applications preference-aware while considering *acceptable* round-trip times.<sup>15</sup>

5.6.2 *Scheduler Design*

We propose schedulers that selectively utilize backup subflows to retain round-trip times below a given acceptable upper round-trip time. Such a latency- and preference-aware scheduler enables substantial performance improvements for interactive applications while preserving subflow preferences.

**INITIAL APPROACH** Listing 5.11 shows a scheduler that only considers backup subflows if no non-backup subflow has a sufficient round-trip time to retain the acceptable upper round-trip time (line 4). The application can control the acceptable upper round-trip time by setting the R1 register with the extended socket API of *ProgMP*.

**ADVANCED APPROACH** The previous example fulfills the basic requirement of a round-trip time and preference-aware scheduler. Imagine, however, a situation where the round-trip time of the non-backup subflow is slightly higher than the round-trip time of the backup subflow. Here, the benefit of the backup subflow is marginal. Listing 5.12 shows a scheduler that only considers backup subflows if all non-backup subflows have a significantly higher

*Only use significantly better backup subflows*

<sup>15</sup>During the finalisation of this dissertation, Apple deployed an *interactive* scheduler mode for iOS, which uses cellular interfaces in case the WiFi round-trip time exceeds 500ms.

round-trip time. Note that this scheduler uses *all* subflows to derive the minimum round-trip time (lines 3–4) instead of the filtered result of available subflows in the variable `sbfCandidates`. The application can control parameters, i. e., the round-trip time differences that are considered to be *significant*, by setting the registers with the extended socket API of *ProgMP*.

```

1 VAR considerBackup = SUBFLOWS.FILTER(sbf => sbf.RTT < R1
2   AND !sbf.IS_BACKUP).EMPTY;
3
4 IF (considerBackup) {
5   /* No acceptable non-backup found, consider all subflows */
6   sbfCandidates.MIN(sbf => sbf.RTT).PUSH(Q.POP());
7 } ELSE {
8   /* Acceptable non-backup found, take best non-backup */
9   sbfCandidates.FILTER(sbf => !sbf.IS_BACKUP).
10  MIN(sbf => sbf.RTT).PUSH(Q.POP());
11 }

```

Listing 5.11: *ProgMP* specification of a scheduler that retains an acceptable upper RTT and subflow preferences.<sup>16</sup>

```

1 VAR bestNonBackup = SUBFLOWS.FILTER(sbf => !sbf.IS_BACKUP).
2   MIN(sbf => sbf.RTT);
3 VAR bestBackup = SUBFLOWS.FILTER(sbf => sbf.IS_BACKUP).
4   MIN(sbf => sbf.RTT);
5
6 VAR considerBackup = bestBackup.RTT_MS < R1 AND
7   bestNonBackup.RTT_MS > R2;
8
9 IF (considerBackup) {
10  /* No acceptable non-backup found, consider all subflows */
11  sbfCandidates.MIN(sbf => sbf.RTT).PUSH(Q.POP());
12 } ELSE {
13  /* Acceptable non-backup found, take best non-backup */
14  sbfCandidates.FILTER(sbf => !sbf.IS_BACKUP).
15  MIN(sbf => sbf.RTT).PUSH(Q.POP());
16 }

```

Listing 5.12: A scheduler that considers RTT differences to retain an acceptable RTT and subflow preferences.<sup>17</sup>

### 5.6.3 Evaluation

**PROOF OF CONCEPT** For a proof of concept, we use a Mininet emulation and systematically increase the RTT on the preferred subflow. Figure 5.16 shows a graphical representation of the experienced round-trip times using the PRINT feature of *ProgMP* during the Mininet emulation.

The default *MinRTT* scheduler uses the non-preferred subflow as soon as it becomes the subflow with the lowest RTT, i. e., after 3 seconds (Figure 5.17,

<sup>16</sup>See Listing A.13 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_rtt\\_preference\\_aware](http://progmp.net/progmp.html#dissertation_rtt_preference_aware).

<sup>17</sup>See Listing A.14 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_rtt\\_preference\\_aware\\_advance](http://progmp.net/progmp.html#dissertation_rtt_preference_aware_advance).

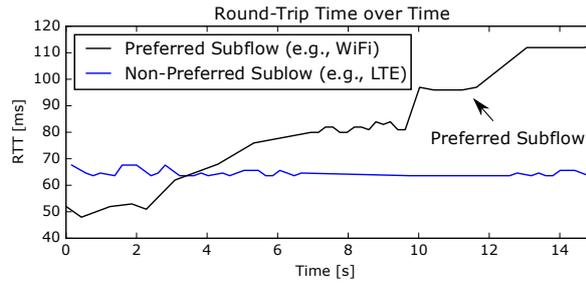


Figure 5.16: Scheduler comparison for a dynamic Mininet scenario. In contrast to the established schedulers, the novel scheduler retains the acceptable round-trip time (90ms) and subflow preferences (prefer WiFi).

top). Setting the non-preferred subflow as backup subflows does not provide remedy, as backup subflows are only used when all non-backup subflows fail. Thus, the backup subflow is not used in the presented round-trip time trace (Figure 5.17, middle). Both examples show that the existing schedulers are not flexible with regard to the backup subflow usage. Finally, Figure 5.17 (bottom) confirms that the scheduler in Listing 5.11 enables RTT- and preference-aware scheduling. Here, the scheduler starts using the backup subflow as soon as the preferred subflow experiences round-trip times above the acceptable RTT, as set to 90ms for the example run.

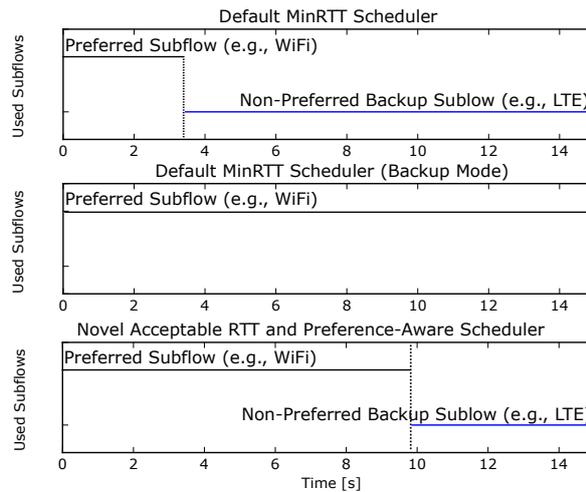


Figure 5.17: Replayed round-trip time trace to compare the behavior of the schedulers during changing round-trip times.

*ProgMP supports a large design space for RTT-aware schedulers*

The design space of possible round-trip time and preference-aware schedulers goes beyond the two presented approaches. *ProgMP* enables the application developer to easily adapt these examples, e. g., to consider additional metrics. The specified schedulers, for example, check if *all preferred subflows* have an unacceptable round-trip time. A slight modification might check if *the available subflows*, i. e., those that have not exhausted their congestion window, have unacceptable round-trip times. Such a scheduler would favor increased throughput over subflow preferences, but might use backup subflows only for a few packets depending on the traffic pattern.

## 5.7 BALANCING THROUGHPUT AND SUBFLOW PREFERENCE

In the following, we motivate, design, and evaluate schedulers that balance the achieved throughput and subflow preferences.

## 5.7.1 Motivation and Analysis

We now consider Multipath TCP schedulers that combine application- and preference-awareness to retain *throughput targets*. Again, subflow preferences might capture a wide range of prioritizations, e. g., to consider asymmetric subflow costs between data-centers or to avoid over-utilizing metered cellular network links.

To illustrate the current inability of applications to leverage preference-aware Multipath TCP scheduling, we conducted real world measurements between a cloud provider and a mobile device on top of off-the-shelf MPTCP with two subflows (WiFi and LTE). Figure 5.18 depicts an example of an interactive streaming session. The first 6 seconds of the stream are encoded with 1MB/s, the remaining part with 4MB/s. Although the 1MB/s stream is sustainable on the 10ms RTT WiFi-subflow, we observe that today’s default scheduler places 30% of the traffic on the higher RTT subflow (LTE, 40ms). This is the result of the throughput and load balancing optimization of the default scheduler in conjunction with specific timings of the congestion control and TCP small queue (TSQ) optimization. We note that setting the LTE subflow to backup mode does not provide remedy as it practically deactivates the subflow. This becomes evident when the stream quality rises to 4MB/s, where the backup mode does not provide enough throughput.

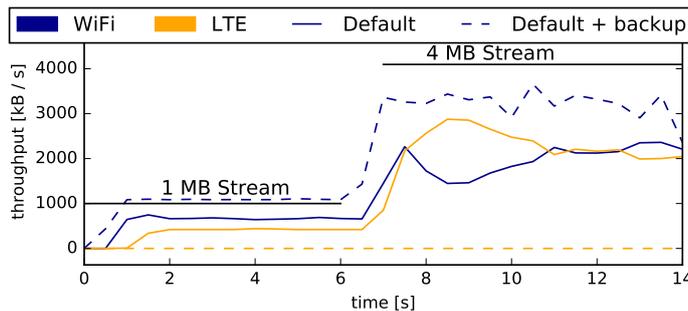


Figure 5.18: Setup and reproducible measurement result of an interactive streaming session over MPTCP using WiFi and LTE with today’s default *MinRTT* scheduler. Neither the default scheduler nor its backup option allow preserving preferences, i. e., exhausting the bandwidth of the faster WiFi subflow before relying on the additional one (LTE) while sustaining the video bitrate of 4MB/s.

### 5.7.2 Scheduler Design

**TARGET DEADLINE** Deadline-driven applications such as *Dynamic Adaptive Streaming over HTTP* (DASH) pose deadlines on the arrival times of data chunks. A preference-aware scheduler might restrict scheduling packets on non-preferred subflows as long as data chunk deadlines are retained. Independently of our work, MP-DASH [62] proposes a similar approach. MP-DASH operates on top of the default scheduler to activate and deactivate the consideration of backup subflows in the default scheduler. A comparable scheduler with *ProgMP* requires only a slight modification of the established default scheduler, as shown in Listing 5.13.

```

1 VAR considerBackups = (R1 == 1) OR /* This line is changed */
2   SUBFLOWS.FILTER(sbf => !sbf.IS_BACKUP).EMPTY;
3
4 VAR sbfCandidates = SUBFLOWS.FILTER(sbf => !sbf.IS_THROTTLED AND
5   sbf.CWND > sbf.PACKETS_IN_FLIGHT + sbf.QUEUED AND !sbf.IS_LOSSY
6   AND (sbf.IS_BACKUP == considerBackups));
7
8 sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(Q.TOP)).
9   MIN(sbf => sbf.RTT).PUSH(Q.POP());

```

Listing 5.13: *ProgMP* specification of a scheduler that can be controlled by the application with regard to the consideration of backup subflows. This scheduler is a slight variation of Listing 5.3.<sup>18</sup>

**TARGET THROUGHPUT** For applications that require a constant bit-rate stream, e. g., interactive video applications, throughput variations are detrimental to the Quality of Experience (QoE). We propose a preference-aware scheduler that only resorts to non-preferred subflows if the acceptable throughput is not achieved. This scheduler has to provide timely scheduling decisions to ensure constant bitrate streaming. Note that such a timely decision is usually not possible in the application layer.

Listing 5.14 shows the specification of the *throughput-* and *preference-aware* TAP scheduler. Here, the application signals the required minimum throughput to the scheduler by setting the `target` throughput in register R1. The scheduler uses the up-to-date subflow properties *per scheduling decision* to calculate the expected throughput. Non-preferred subflows are only used if required and are restricted to transmit the leftover fraction  $\frac{\text{targetBw} - \text{capPrefSbf}}{\text{targetBw}}$ .

<sup>18</sup>See Listing A.15 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_backup\\_controlled](http://progmp.net/progmp.html#dissertation_backup_controlled).

```

1 VAR targetBwKB = R1;
2 VAR prefAhead = R4;
3 VAR factor = 100;
4 VAR mss = 1400;
5 VAR maxAhead = 100 * factor;
6
7 VAR prefS = sbfCandidates.FILTER(s=>!s.IS_BACKUP).MIN(s=>s.RTT);
8 IF (prefS != NULL) {
9   prefS.PUSH(Q.POP());
10  IF (R4 < maxAhead) { SET(R4, R4 + factor); }
11 } ELSE {
12  VAR capKB = prefS.CWND / prefS.RTT_MS * mss;
13  VAR ratio = factor * capKB / (targetBwKB - capKB);
14  IF(prefAhead > ratio AND cap < targetBwKB) {
15    sbfCandidates.MIN(s => s.RTT).PUSH(Q.POP());
16    SET(R4, prefAhead - ratio);
17  }
18 }

```

Listing 5.14: *ProgMP* specification of a novel scheduler, which retains a target throughput and subflow preferences.<sup>19</sup>

### 5.7.3 Evaluation

We evaluated the TAP scheduler in the wild between a client Laptop and an Amazon EC2 server instance both running our runtime environment. On the client side we used a Nexus 5 device with USB tethering as LTE modem to a major European service provider and an IEEE 802.11n WiFi connection to a typical residential broadband service. Figure 5.19 shows that the TAP

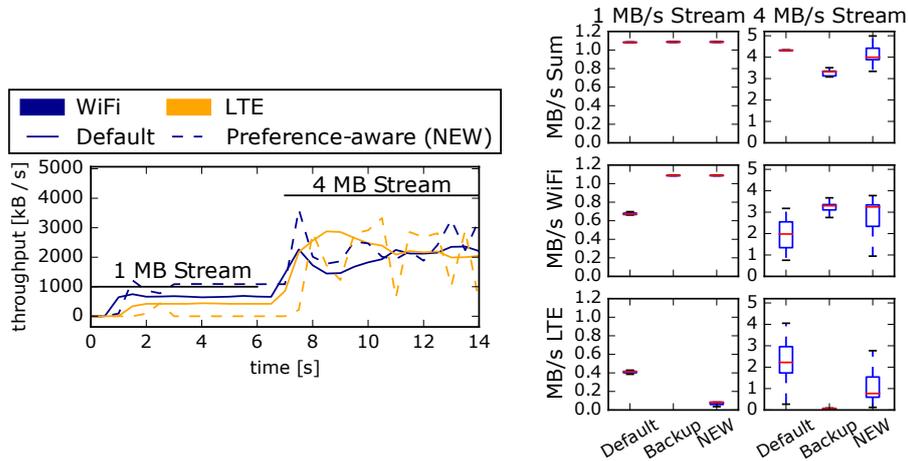


Figure 5.19: Evaluation in the wild: In contrast to the default scheduler, the throughput- and preference-aware (TAP) scheduler uses the signaled target throughput (1MB/s then 4MB/s) and efficiently utilizes subflows according to their preferences.

<sup>19</sup>See Listing A.16 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_throughput\\_preference\\_aware](http://progmp.net/progmp.html#dissertation_throughput_preference_aware).

scheduler significantly reduces the non-preferred LTE usage while sustaining the required overall stream throughput. In particular, the TAP scheduler deals very efficiently with fluctuations in WiFi throughput. *ProgMP* enables a convenient specification and tuning of this throughput- and application-aware scheduler for various throughput sensitive applications, e. g., by implementing different aggressiveness flavors.

## 5.8 ONE-WAY DELAY-AWARE SCHEDULING

In the following, we motivate, design, and evaluate one-way delay-aware Multipath TCP scheduling.

5.8.1 *Motivation and Analysis*

Round-trip times are seldom symmetric. In data-centers, for example, congestion in a single direction causes asymmetric queuing delays. Asymmetric routing in the Internet [136] and properties of access technologies, such as HSDPA, LTE, and WiFi [99], cause asymmetric round-trip times. We argue that thin streams, which require low application-layer round-trip times instead of high throughput, benefit from scheduling packets on the subflow with the lowest one-way delay instead of the lowest round-trip time. In particular, one-way delay-aware scheduling enables application-layer round-trip times below the minimum subflow round-trip time, as illustrated in Figure 5.20.

*Round-trip times are seldom symmetric*

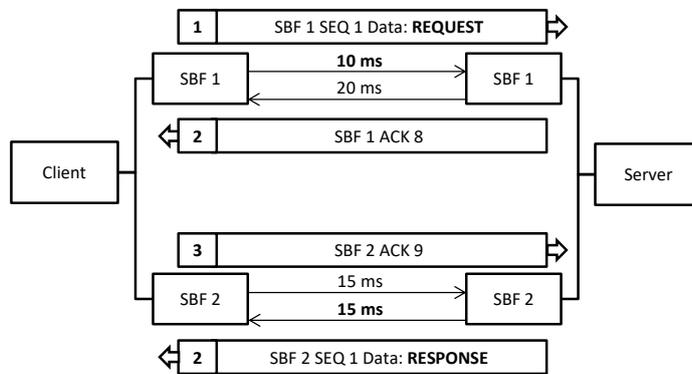


Figure 5.20: With one-way delay-aware scheduling, *request response* patterns on the application layer can be finished in  $10\text{ms} + 15\text{ms} = 25\text{ms}$ , which is less than the minimum subflow round-trip time ( $10\text{ms} + 20\text{ms} = 30\text{ms} = 15\text{ms} + 15\text{ms}$ ).

One-way delay-aware scheduling requires a one-way delay estimation. This is challenging to obtain in a distributed system without synchronized clocks. In the following, we show how multipathing enables sufficient one-way delay estimates for scheduling decisions.

5.8.2 *Calculating One-Way Delay Estimations*

The default Multipath TCP scheduler uses the round-trip time estimations that are used for the retransmission timeout calculation. Originally, the round-trip time samples for computing the TCP retransmission timer ([RFC 6298]) relied on Karn's algorithm [80]. Today, the *TCP Timestamp Option* as specified in [RFC 1323] (*TCP Extensions for High Performance*) supersedes this algorithm and is ubiquitous for round-trip time measurements [97]. Each TCP timestamp option contains two timestamps, the `TSval` with the sender's times-

tamp and the TSecr with the last received timestamps of the communication partner. These values are used to estimate the round-trip time by comparing a replied timestamp (originated from the local clock) with the local time (of the local clock). As only timestamps of the same local clock are compared, clock synchronization is not necessary.

One-way delay-aware scheduling essentially requires to order all subflows by their outgoing one-way delay. In the following, we show how to order subflows accordingly relying on the established TCP timestamp options to estimate outgoing one-way delay *differences* of different paths without synchronized clocks. This design enables deployable one-way delay-aware MPTCP scheduling without modifications at the communication partner. Figure 5.21 provides a TCP timestamp example with two subflows. Both hosts L and R have different initial clocks (100 vs. 1000). The timestamps of received packets (TSVal and TSecr) originate from different, non-synchronized clocks. We denote this using the indices L and R for the local and remote clocks. The processing time at the remote host is denoted as P. We assume a reasonably small clock drift between the sender and the receiver clock.

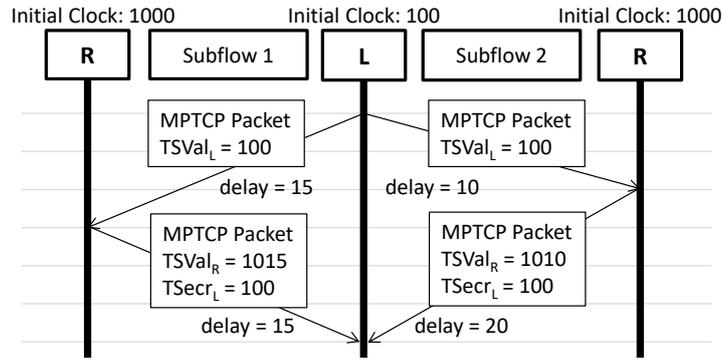


Figure 5.21: TCP timestamps for two hosts (L and R) with non-synchronized clocks and two subflows with asymmetric RTTs.

We calculate two *delay estimators*  $DE_{out}$  and  $DE_{in}$  for each subflow

$$DE_{out} := TSVal_R - P - TSecr_L \quad (5.1)$$

$$DE_{in} := NOW_L - TSVal_R. \quad (5.2)$$

Due to the non-synchronized clocks, the real delays are not derivable. However, the difference of the outgoing delay estimators of two different subflows corresponds to the time difference when the packets were sent at the remote side. Assuming (on average) constant processing times at the remote host, i.e.,  $P_{sbf1} = P_{sbf2}$ , this corresponds to the receive time difference and therefore to the one-way delay difference  $diff$ .

$$\begin{aligned}
DE_{out, sbf1} - DE_{out, sbf2} &= TSVal_{R, sbf1} - P_{sbf1} - TSecr_{L, sbf1} - \\
&\quad (TSVal_{R, sbf2} - P_{sbf2} - TSecr_{L, sbf2}) \tag{5.3} \\
&= TSVal_{R, sbf1} - TSecr_{L, sbf1} - \\
&\quad TSVal_{R, sbf2} + TSecr_{L, sbf2}
\end{aligned}$$

Note that the initial packets neither have to be sent at the same time nor redundantly. If both packets are sent with a time difference  $\delta$ , the initial timestamps differ by  $\delta$ , i.e.,  $TSecr_{L, sbf1} + \delta = TSecr_{L, sbf2}$ . This further implies that the receive time differs by  $\delta + diff$  and  $TSVal_{R, sbf1} + \delta + diff = TSVal_{R, sbf2}$  and therefore  $TSVal_{R, sbf1} - TSVal_{R, sbf2} + \delta = diff$ .

In Figure 5.21, both subflows have the same RTT of 30 but different one-way delays. Host A calculates the *delay estimators* per subflow

$$DE_{out, sbf1} = 1015 - 100 = 915 \tag{5.4}$$

$$DE_{out, sbf2} = 1010 - 100 = 910. \tag{5.5}$$

The comparison of both estimators shows that the outgoing delay of the first subflow is higher than the second subflow. Accordingly, the MPTCP scheduler at host A should prefer the second subflow. Remarkably, if both hosts use the one-way delay-aware scheduler, the application-layer RTT becomes smaller than the smallest subflow RTT, i.e.,  $10 + 15 = 25 < 30$ .

The authors of [31, 177] discuss opportunities of multipathing and one-way delay estimations. In contrast, this work is the first that explicitly analysis MPTCP scheduling for thin flows and considers the complex dependencies of scheduling, traffic patterns, and the available scheduling informations, together with an implementation and detailed evaluation.

### 5.8.3 Scheduler Design

We implemented the presented one-way delay estimator calculation in the `receive_packet_handler` of the MPTCP Linux kernel implementation and integrated it into *ProgMP*. This allows a convenient specification of the delay-aware scheduler relying on the added subflow properties `DELAY_OUT` and `DELAY_IN`, as shown in Listing 5.15.

```

1 IF (!Q.EMPTY) {
2   sbfCandidates.MIN(sbf => sbf.DELAY_OUT).PUSH(Q.POP());
3 }

```

Listing 5.15: Scheduler specification of the one-way delay-aware scheduler.<sup>20</sup>

<sup>20</sup>See Listing A.17 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_onewaydelay](http://progmp.net/progmp.html#dissertation_onewaydelay).

## 5.8.4 Evaluation

For an evaluation of the one-way delay-aware scheduling, we use Mininet to systematically vary the delay asymmetry  $\lambda = \frac{\text{Delay}_{\text{OUT}}}{\text{Delay}_{\text{IN}}}$  while keeping the round-trip time constant. Figure 5.22 compares the application-layer round-trip time of the default scheduler, the round robin scheduler, the redundant scheduler, and the one-way delay-aware scheduler. The figure shows that the application-layer round-trip time with the default scheduler is independent of the delay asymmetry. The redundant scheduler benefits from asymmetric delays due to the evaluation setup with limited throughput requirements. The round robin scheduler provides on average the same application-layer RTT as the default scheduler, but shows increasing variances for higher asymmetries. Finally, the one-way delay-aware scheduler is able to achieve the same, minimum application-layer RTT as the redundant scheduler. The measurement further confirms that the one-way delay-aware scheduler achieves application-layer round-trip times below the minimum subflow RTT.

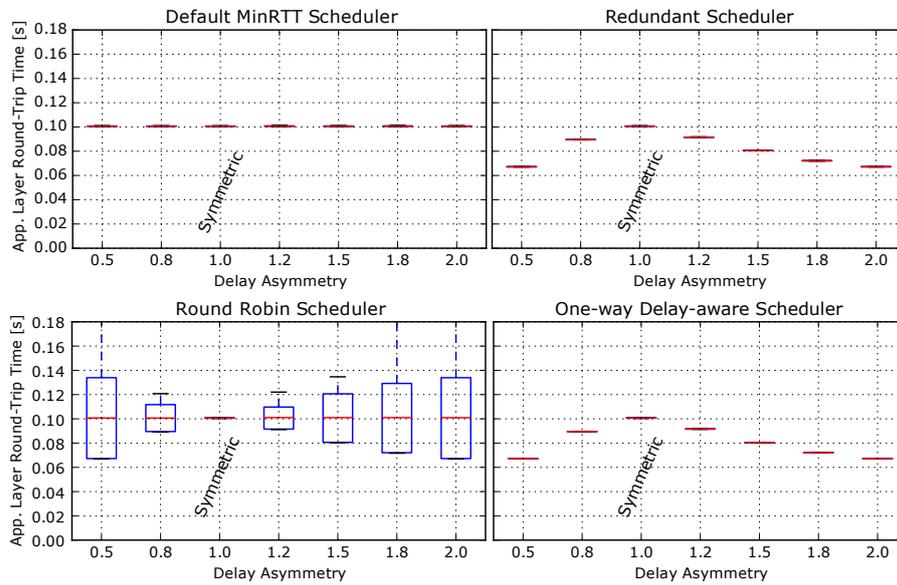


Figure 5.22: Application-layer round-trip time depending on the delay asymmetry. The default scheduler does not benefit from asymmetric delays, whereas the variance increases for the round robin scheduler. The one-way delay-aware scheduler provides the same reduced application-layer RTT than the redundant scheduler without inducing overhead.

The overhead of the redundant scheduler becomes apparent when comparing the flow completion time for larger responses of 500kb (Figure 5.23). These flows require multiple round-trip times to complete. While the one-way delay-aware scheduler and the default scheduler provide nearly the same flow completion times, the performance of the redundant scheduler decreases due to the wasted resources.

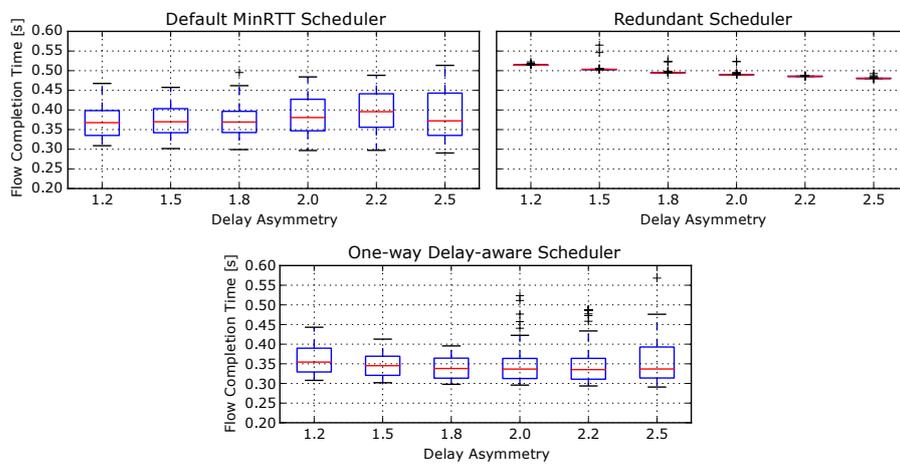


Figure 5.23: The default scheduler and the one-way delay-aware scheduler provide comparable flow completion times for 500kb responses, whereas the performance of redundant scheduler decreases.

## 5.9 TOWARDS HTTP/2-AWARE SCHEDULING

In the following, we motivate, design, and evaluate an HTTP/2-aware Multipath TCP scheduler.

### 5.9.1 *Motivation and Analysis*

HTTP is one of the fundamental protocols of today’s Internet. Thus, boosting the HTTP and mobile web performance is a promising opportunity for MPTCP and a key to *universal* MPTCP deployment. Next, we analyze relevant aspects of today’s web infrastructure and implement a novel HTTP/2-aware scheduler that overcomes existing limitations.<sup>21</sup>

Measurement studies with off-the-shelf MPTCP showed only marginal performance improvements for HTTP [125] and revealed that web protocols incur complex interactions with the multipath-enabled transport layer [61]. Today, browsers, web servers, and the web content are highly tuned and have complex dependencies. The traffic patterns, for example, depend on the used HTTP version, content optimizations, and infrastructure. Multiple short-lived connections, for example, might boost HTTP/1.1 performance but do not benefit from MPTCP due to the slow initial subflow establishment. The widely used optimization to distribute web content on multiple server [54], denoted as *domain sharding*, reduces the amount of data per TCP connection and the potential of MPTCP scheduling optimizations.

These optimizations for HTTP/1.1 are not necessary or even detrimental with the optimizations of HTTP/2 such as framing and multiplexing. With HTTP/2, browsers may, for example, use a single TCP connection efficiently for multiple HTTP requests while signaling priorities to web servers to favor CSS and JavaScript over images to reduce initial loading times. Thus, the transition from HTTP/1.1 to the emerging HTTP/2 protocol particularly improves performance for pages that rely on a single TCP connection and might increase the potential of MPTCP.

### 5.9.2 *Scheduler Design*

Overall, the previous analysis illustrates that HTTP performance has complex dependencies on the content, the infrastructure, and the used protocols. In the following, we highlight and overcome two shortcomings of today’s uninformed MPTCP schedulers. First, packets that refer to external resources may be scheduled on slow subflows, delaying the corresponding 3rd-party content requests and thereby potentially increasing the page load time. One fourth of the Alexa-200 pages have 3rd-party dependencies on their critical path of initial page loading [198]. Second, subflow preferences are not considered. This leads to transferring substantial data amounts, such as images,

---

<sup>21</sup>Parts of the HTTP-aware scheduler were developed by Max Weller as part of his Bachelor thesis [S22], which was motivated and supervised by the author of this dissertation.

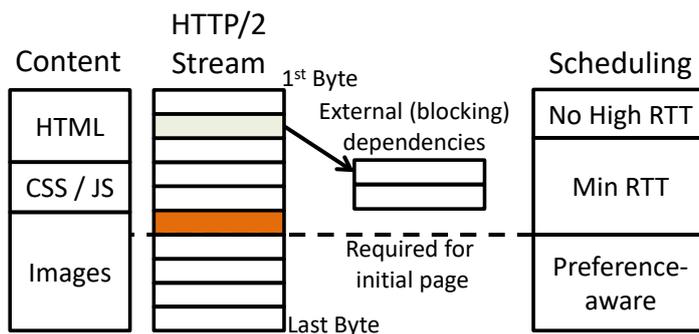


Figure 5.24: Illustration of the different scheduling strategies depending on the content in the HTTP stream.

on non-preferred subflows, such as metered cellular networks, *after* the initial page is loaded, hence, not even increasing the perceived quality by the user.

**HTTP/2-AWARE SCHEDULING** To overcome these shortcomings we implemented a novel HTTP/2-aware scheduler. This scheduler leverages all building blocks of our programming model, using content-dependent scheduling strategies as shown in Figure 5.24. For the initial data, i. e., until the information on external dependencies is transferred, subflows with high round-trip times are avoided. The remaining data that is required to render the initial page is transferred with the default minimum RTT scheduler strategy. For the remaining data that is not required for the initial page and therefore independent of the perceived quality by the user we invoke preference-awareness. Listing 5.16 shows a *ProgMP* scheduler that implements the presented adaptive scheduling.

```

1 IF (Q.EMPTY) { RETURN; }
2
3 VAR modeNoHighRtt = 0;
4 VAR modeMinRtt = 1;
5 VAR modePrefAware = 2;
6
7 IF (Q.TOP.USER == modeNoHighRtt) {
8   VAR minRttSbf = SUBFLOWS.MIN(sbf => sbf.RTT).RTT;
9   sbfCandidates.FILTER(sbf => sbf.RTT < minRttSbf * 3 / 2).
10   MIN(sbf => sbf.RTT).PUSH(Q.POP());
11 } ELSE IF (Q.TOP.USER == modeMinRtt) {
12   sbfCandidates.MIN(sbf => sbf.RTT).PUSH(Q.POP());
13 } ELSE IF (Q.TOP.USER == modePrefAware) {
14   sbfCandidates.FILTER(sbf => !sbf.IS_BACKUP).
15   MIN(sbf => sbf.RTT).PUSH(Q.POP());
16 }

```

Listing 5.16: *ProgMP* specification of a novel scheduler that adapts the scheduling strategy depending on the content of the overlying HTTP connection.<sup>22</sup>

**MPTCP-AWARE WEBSERVER** We extended the Ngthttp2 library [124] to forward HTTP content information through the OpenSSL library to our scheduler API, as shown in Figure 5.25. Thus, each packet is annotated with the content type and the registers contain information about the required bytes for the initial page. This information enables the application-aware MPTCP scheduler of Listing 5.16. Note that content type dependent scheduling can be seen as information disclosure.

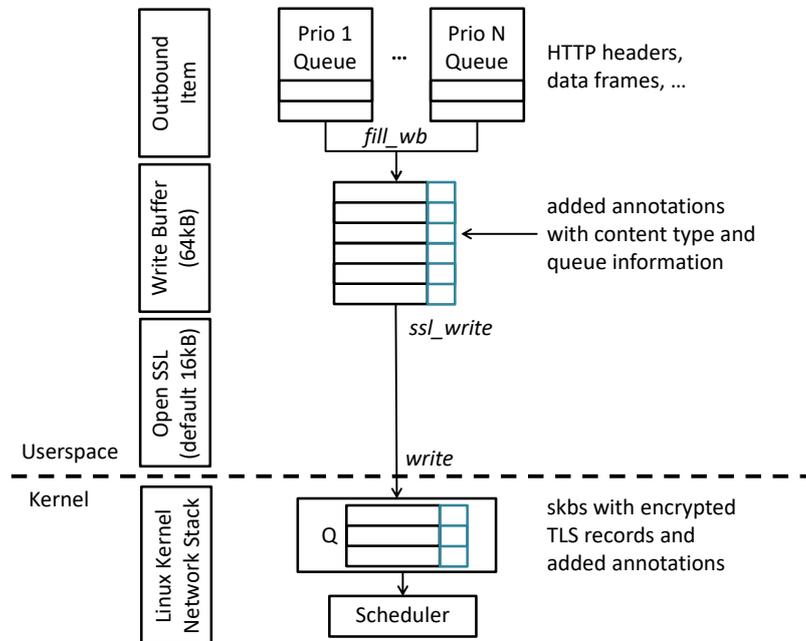


Figure 5.25: Illustration of the buffers and data path of the Ngthttp2 web server. Here, we annotated content type information throughout the data path.

### 5.9.3 Evaluation

We conducted a measurement study in the wild, using our extended the Ngthttp2 library as web server on an EC2 instance and a recent Google Chrome browser (Chrome version 56) on our client with a WiFi and an LTE interface. To evaluate the impact of the RTT-ratio, we systematically increased packet delays on the WiFi interface. Inspired by highly optimized web pages, such as `amazon.com`, we used an exemplary page with optimized HTML, CSS, and JavaScript layout. With this optimized page, more than half of the data, in particular images that are outside of the initial view, are transferred after the initial page. Our measurements show that the HTTP/2-aware scheduler successfully tunes the initial dependency retrieval time for heterogeneous round-trip times (Figure 5.26), and therefore enables *earliest possible dependency resolution*. A deeper analysis of the homogeneous RTT scenario (40ms / 40ms) shows that higher variances on the second subflow cause variance in the load

<sup>22</sup>See Listing A.18 for a full code example. We provide a ready-to-use test environment at [http://progmp.net/progmp.html#dissertation\\_http\\_aware](http://progmp.net/progmp.html#dissertation_http_aware).

time for both MPTCP scheduler. The preference-aware scheduling efficiently reduces the transferred data on the less preferred LTE subflow without affecting the initial page load time.

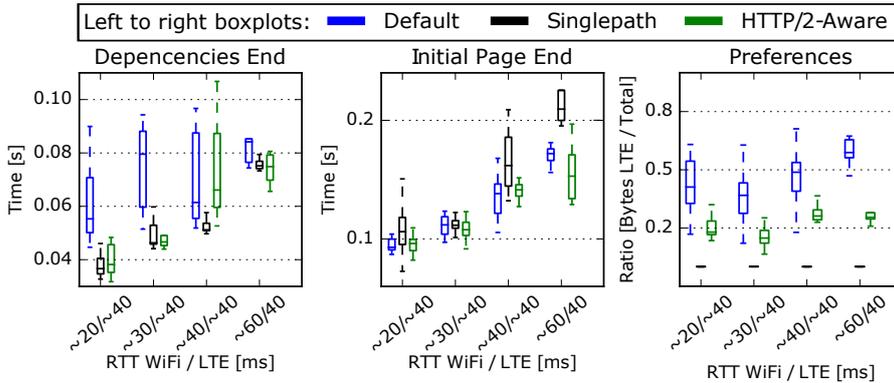


Figure 5.26: Real-world evaluation of a novel HTTP/2-aware MPTCP scheduler. The time to retrieve all dependency information is significantly reduced by avoiding high RTT subflows for the initial packets without affecting the remaining time for non-external content. The preference-aware scheduling of content that is not required for the initial page view significantly reduces the usage of the metered LTE subflow.

The real world measurements of our HTTP-aware scheduler confirm the potential of application- and preference-aware MPTCP scheduling. We envision a systematic improvement of the *overall* HTTP stack in combination with a large, extensive real world evaluation as promising future work.

5.10 PITFALLS IN EMULATION

By carefully analysing the results of our experiments, we found a few recurring pitfalls in network emulations and network experiments in general. In the following, we share notable examples to increase the researchers awareness of these pitfalls.

5.10.1 Using a Fresh Network

We always used a fresh network, i. e., started Mininet from scratch, to avoid measurement artefact and dependencies between experiments. We found, however, that fresh networks introduce measurement artefacts. The initial *ARP* requests ([RFC 826]) increase the experienced round-trip time of the first packet. We assume, however, that most real world MPTCP scenarios are running in networks that are already used and do not require *ARP* requests for a new MPTCP connection setup. Thus, we decided to mitigate this by heating up the network with a few *ICMP* ping messages. Figure 5.27 shows a measurement with and without heating up the network in a network with 100ms round-trip times. Here, we see that the smoothed round-trip time requires mul-

*ARP requests in unused networks...*

*... impact the smoothed RTT estimation.*

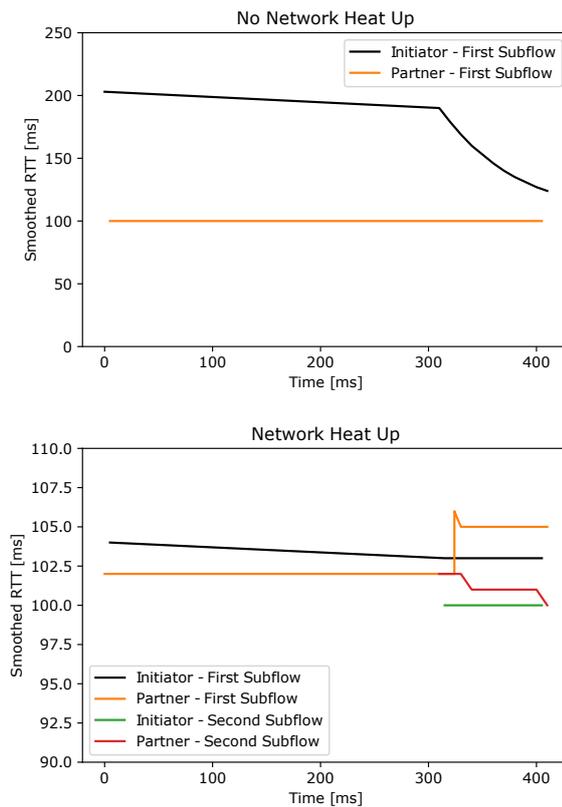


Figure 5.27: Comparison of the smoothed round-trip time as experienced and maintained by the network stack depending on an initial heat up of the network to avoid *ARP* requests.

multiple round-trip times and multiple measurement samples to reach the actual value in the environment without heat up. Additionally, we note that the establishment of the second subflow is significantly delayed due to the experienced higher round-trip times. While these round-trip time artefacts might be less important for many Mininet application scenarios, we note that in particular the MPTCP scheduling decision is very round-trip time sensitive.

Listing 5.17 shows the experienced round-trip times for the initial *ICMP* pings. The values support our argumentation, as the initial round-trip time is twice the actual round-trip time.

```

1 PING 11.0.0.2 (11.0.0.2) 56(84) bytes of data.
2 64 bytes from 11.0.0.2: icmp_seq=1 ttl=64 time=204 ms
3 64 bytes from 11.0.0.2: icmp_seq=2 ttl=64 time=100 ms
4 64 bytes from 11.0.0.2: icmp_seq=3 ttl=64 time=100 ms
5 64 bytes from 11.0.0.2: icmp_seq=4 ttl=64 time=100 ms

```

Listing 5.17: Initial *ICMP* ping measurement output to heat up the network.

Finally, we note that there is not a single *correct* behavior and emulation setup. The emulation should behave like a real world environment. Thus, if a real world environment requires initial *ARP* requests, the emulation should require these as well. In particular, we note the cellular connections experience initial setup delays. These delays depend on the used cellular technology and if the network was already used, e. g., for a DNS request.

*What about cellular connections?*

### 5.10.2 Impact of Netem on the Network Stack

During our work on Multipath TCP scheduling, we initially experienced systematic and reproducible differences between real-world measurements and Mininet experiments.<sup>23</sup> For an application that sends a burst of eight packets in a network with two heterogeneous paths, for example, all eight packets were scheduled on the first subflow (Figure 5.28). In our real world measurement, however, packets were split on both subflows (Figure 5.29).

*Differences between emulations and the real world*

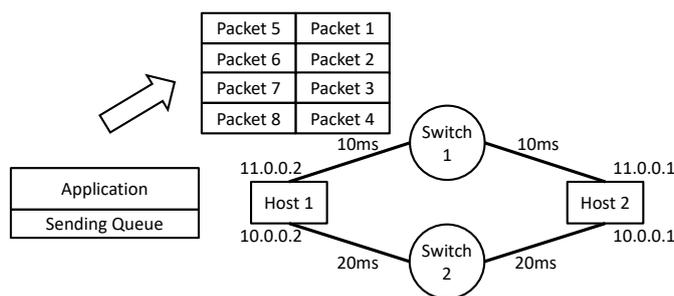


Figure 5.28: All packets are scheduled on the first subflow in our Mininet setup.

Digging into the Multipath TCP implementation, we found that the scheduler stops using the first path due to *TCP small queues* (TSQ).<sup>24</sup> TSQ is an

<sup>23</sup>Parts of these findings were developed in cooperation with Max Weller and previously published in [F16] and his bachelor thesis [S22], supervised by the author of this dissertation.

<sup>24</sup>See <https://lwn.net/Articles/507065>.

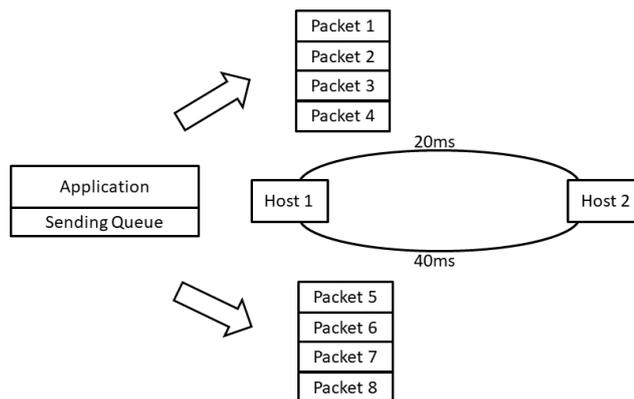


Figure 5.29: Packets are split on both subflow in the real world environment.

effort to reduce queue sizes in the network stack. Large queues in the network stack are known to increase latency due to queuing delays. During the development of Multipath TCP, a commit was added to avoid sending packets on subflows that are considered throttled by TSQ (Listing 5.18).<sup>25</sup>

```

1 /* If TSQ is already throttling us, do not send on this subflow. When
2  * TSQ gets cleared the subflow becomes eligible again.
3  */
4
5 if (test_bit(TSQ_THROTTLED, &tp->tsq_flags))
6     return 0;

```

Listing 5.18: The Multipath TCP default scheduler checks if a subflow is already throttled by TSQ.

Mininet relies on netem<sup>26</sup> for delay emulation. Checking the implementation of netem<sup>27</sup>, we found that packets that are delayed by netem are not considered queued by TSQ (Listing 5.19).

```

1 /* If a delay is expected, orphan the skb. (orphanning usually takes
2  * place at TX completion time, so _before_ the link transit delay)
3  */
4 if (q->latency || q->jitter || q->rate)
5     skb_orphan_partial(skb);

```

Listing 5.19: The netem implementation removes delayed packets from the sending queue.

Thus, MPTCP never experiences a TSQ throttled subflow in Mininet setups with delayed links at the sender. To substantiate our claim, we repeated the previous Mininet emulation with a slightly modified topology (Listing 5.20). In this topology, the first link at the sender does not use netem. We found

<sup>25</sup>This feature was added to MPTCP in July 2014 with commit <https://github.com/multipath-tcp/mptcp/commit/5c278893b37fe48c66ff226793607687b8482ba9>. Based on our findings, it was removed in March 2018 <https://github.com/multipath-tcp/mptcp/commit/d50611004d1f05da5d839aea36c7cd247fee15da>.

<sup>26</sup>See <https://wiki.linuxfoundation.org/networking/netem>.

<sup>27</sup>See [https://elixir.free-electrons.com/linux/v4.14/source/net/sched/sch\\_netem.c#L462](https://elixir.free-electrons.com/linux/v4.14/source/net/sched/sch_netem.c#L462).

that Mininet emulations with this setup behave comparable to our real world experiments and relied on a comparable setups for our experiments.

```

1 class StaticTopo(Topo):
2     def build(self):
3         h1 = self.addHost('h1')
4         h2 = self.addHost('h2')
5
6         /* first path */
7         s1 = self.addSwitch('s1')
8         self.addLink(h1, s1, bw=100)
9         self.addLink(h2, s1, bw=100, delay="20ms")
10
11        /* second path */
12        s2 = self.addSwitch('s2')
13        self.addLink(h1, s2, bw=100)
14        self.addLink(h2, s2, bw=100, delay="40ms")

```

Listing 5.20: Python code snippet for a modified topology that has no netem delay on the sender's link.

### 5.10.3 Changing the Network at Runtime

During our experiments with changing round-trip times and bandwidths, we initially relied on the Mininet API to set link properties.<sup>28</sup> However, we found that Mininet *resets* the underlying traffic shaper by *destroying* and *recreating* them, which causes significant measurement artefacts. Figure 5.30 shows that a direct change of the underlying traffic shaper without the Mininet API leads to plausible, smooth behavior.

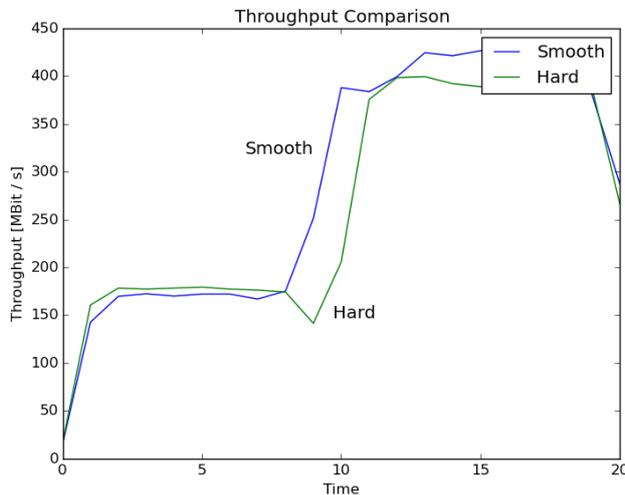


Figure 5.30: Comparison of the used *smooth* change of the traffic shaper and the *hard* reset of Mininet.

<sup>28</sup>Parts of these findings were developed in cooperation with Denny Stohr and previously published in [F19]. A pull request to fix this issue in Mininet was not accepted so far (<https://github.com/mininet/mininet/pull/650>).

## 5.11 MACI PERSPECTIVE

**ITERATIVE RESEARCH PROCESS** We used *MACI* for the development, evaluation, and analysis of the MPTCP schedulers as presented in this chapter.<sup>29</sup> Therefore, we followed an iterative approach that consists of an analysis of the scheduling environment, the derivation of scheduling optimizations, their prototypical implementation with *ProgMP*, the experiment-based scheduler evaluation, and the refinement of the scheduler prototypes. *MACI* supported us during all steps of this iterative research process, as illustrated in Figure 5.31. In particular, we found *MACI* beneficial for the iterative improvements and bug fixes, as it enabled us to quickly evaluate the impact of the improvements on *all* relevant experiments.

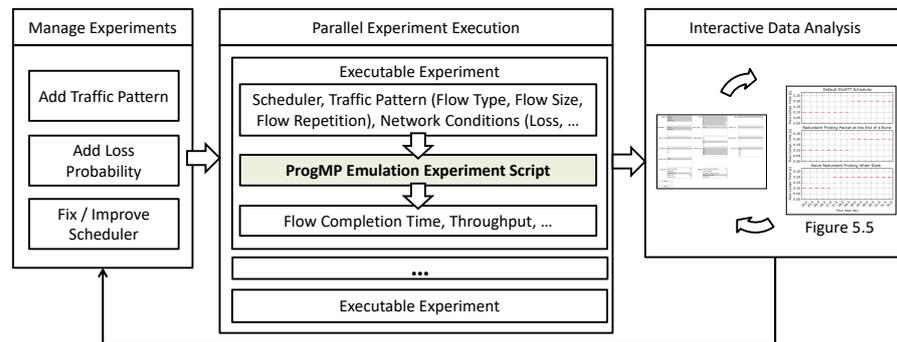


Figure 5.31: Experiment-driven research process for the *ProgMP* evaluations.

**INTERACTIVE ANALYSIS** We benefited from the interactive analysis of experiment results with *MACI*. We used *MACI* to compare the performance of multiple schedulers in various environments for different application workloads using the filter and aggregation selection in the interactive analysis. Most visualizations in this chapter are generated with *MACI* and are only slightly modified for illustration purposes.

**SCALABLE EXECUTION** Finally, the scalable execution significantly increased our evaluation speed. The evaluation of different redundancy flavors in Section 5.4.3, for example, consists of 30,720 experiments, where each experiment requires nearly one minute. With *MACI*, we executed these experiments with up to 20 workers on Amazon AWS in about a single day instead of 21 days on a single machine. These experiments on Amazon AWS *t2.xlarge* instances cost about 95\$. Since we pay \$0.1856 per hour, the parallel execution costs the same as the sequential execution.

During these parallel executions, we noted that our current *MACI* implementation provides only limited scalability. We found that this is caused by a bottleneck in the *Entity Framework* of the *ASP.NET Core* backend and the underlying *SQLite* database. Future work on the implementation might include

<sup>29</sup>More precise, parts of these scheduler were developed and analyzed with early prototypes of *MACI*, as *ProgMP* and *MACI* were developed concurrently.

30,720 experiments  
for 95\$

concepts like message queues to overcome this implementation limitation. Additionally, our current implementation does not support scaling virtual machines in multiple AWS regions and therefore is restricted to the offered scalability of a single AWS region. Future work on the implementation might overcome this limitation and allow to execute the previous mentioned experiments in a few minutes.

As we refined the schedulers during our research and repeated the experiments, we repeatedly benefited from the *MACI* speed up. We found these speed ups important for our research efficiency, as it enabled us to focus on a scheduler problem without interruption and long waiting periods.

## 5.12 PROGMP PERSPECTIVE

*ProgMP enables  
scheduler  
innovations*

First of all, we note that the extensive discussion, analysis, and implementation of novel schedulers in this chapter was enabled by the *ProgMP* programming model and its provided abstractions. This confirms our claim that *ProgMP* enables MPTCP scheduler innovations. Overall, Table 5.3 shows that *ProgMP* fulfils the requirements as presented in Section 3.1. In the following, we support this claim with a discussion of our *ProgMP* experiences during the design and analysis of novel schedulers throughout this chapter.

The wide range of presented and evaluated schedulers confirms the expressiveness of our programming model. In particular, our programming model enabled us to evaluate and compare design decisions, such as the choice and timing of the probing packet in Section 5.3 and different flavors of redundancy in Section 5.4. Thus, we showed that the programming model bridges the gap between profound Linux kernel knowledge, networking details, and the application logic. As the number of lines required to express schedulers is about ten times lower than equivalent implementations in C, we argue that the presented programming model is much more usable than any existing approach. In particular, we argue that the novel *ProgMP* schedulers with up to 38 lines of code (Table 5.1) are enabled by the abstractions of *ProgMP* and would require complex C implementations. Overall, we found that the recurring pattern to handle the reinjection queue, as presented in Section 5.2.1, contributes a significant number of lines and complexity to all presented schedulers.

Table 5.3: *ProgMP* requirement fulfilment.

Requirement	
Exchange of ideas	Used in publications, discussions, and throughout this chapter to exchange ideas
Evaluable and executable	Execution environment implemented and various schedulers executed in this chapter
Efficient execution	Evaluated in Section 3.4.5.1 Executed various schedulers in this chapter
Expressiveness	Expressed various schedulers in this chapter
Graceful failure handling	Yes

We consider a detailed usability study as promising future work. Yet, we conducted a limited user study with computer science students where 9 out of 10 participants, who never modified Linux kernel sources, successfully implemented a new scheduler with predefined functionality using our programming model. Further, we found *ProgMP* useful for education in the *Communication Networks II* and *Communication Networks IV* lectures in the winter semester

2017 at TU Darmstadt. Finally, we note that the *ProgMP* demo page with the web editor is well suited to discuss general MPTCP scheduler concepts.<sup>30</sup>

**ADAPTIVE SCHEDULING AT RUNTIME** A single Multipath TCP connection might face changing network conditions, different traffic patterns, and varying application requirements, as discussed in the motivation of this dissertation (Section 1.1). We found *ProgMP* powerful to handle these changes in various of the presented schedulers. The preference-aware schedulers, for example, change their behavior, i. e., the considered subflows, depending on the environment conditions. The HTTP/2-aware scheduler changes the scheduling strategy depending on the application payload. Listing 5.21 summarizes the recurring patterns to enable adaptive scheduling with *ProgMP*.

```

1 VAR defaultScheduling = 0;
2 VAR redundantScheduling = 1;
3 VAR environmentAwareScheduling = 2;
4
5 /* R1 controlled by the application */
6 IF (R1 == defaultScheduling) {
7   /* Default scheduling */
8 } ELSE IF (R1 == redundantScheduling) {
9   /* Redundant scheduling */
10 } ELSE IF (R1 == environmentAwareScheduling) {
11   /* Environment-aware scheduling */
12   VAR bestNonBackup = SUBFLOWS.FILTER(sbf => !sbf.IS_BACKUP).
13     MIN(sbf => sbf.RTT);
14   VAR bestBackup = SUBFLOWS.FILTER(sbf => sbf.IS_BACKUP).
15     MIN(sbf => sbf.RTT);
16
17   VAR considerBackup = bestBackup.RTT_MS < bestNonBackup.RTT_MS;
18   IF (considerBackup) { /* ... */

```

Listing 5.21: *ProgMP* snippet that illustrates adaptive scheduling strategies depending on the signaled application information and the experienced network conditions.

**COMPOSABILITY** In this chapter, we presented incremental improvements for established schedulers as well as fundamentally novel schedulers. The primitives of our programming model were sufficient to express and compose all these schedulers and features. A combination of the active probing features, for example, with the round-trip time and preference-aware scheduler is easily expressible with *ProgMP*. We envision a direct support and concept for composability to naively combine features as future work.

**LANGUAGE EXTENSIBILITY** So far, we mostly explored the design space of scheduling based on the confined set of metrics that are available in *ProgMP* and that are typically available in the network stack, such as the measured round-trip time and the congestion window. The language extension to incorporate one-way delay estimations was straight forward, but re-

<sup>30</sup>See <https://progmp.net/progmp.net>.

quired changes in the runtime environment. Overall, we note that the presented language enables a wide range of schedulers and allows the extension with additional metrics. A concept for systematic language extensibility was not required but is promising future work.

### 5.13 DISCUSSION AND FUTURE WORK

In this chapter, we explored the design space of MPTCP scheduling and proposed more than eight novel schedulers or scheduling optimizations. We identified that in particular the incorporation of application knowledge and subflow preferences significantly extends the design space and opportunities of MPTCP scheduling. Overall, this chapter contributes to the fundamental understanding of Multipath TCP scheduling.

While we presented and evaluated more than eight novel scheduler, we anticipate that *ProgMP* enables a lot more, optimized schedulers. In particular, we anticipate schedulers that incorporate active capacity probing, more fine-granular redundancy and backup semantics (e. g., use two out of three subflows, only use backup subflow if less than two subflows are available), and more application-aware schedulers (e. g., to consider TLS records as logical chunks of data or to support transitions on upper layer protocols [115, 152]).

*Towards thousands  
of schedulers*

## A PROGRAMMING MODEL FOR ADAPTATION DECISIONS

---

In the previous chapters, we showed that programming models provide well suited abstractions for the design and implementation of communication systems, i. e., for the MPTCP scheduling decision. In this chapter, we go beyond the domain of MPTCP and focus on the domain of adaptive communication systems to tackle the fourth research question:

*RQ IV: What are suitable abstractions for the specification of the adaptation decision for adaptive communication systems?*

We tackle this research question with a programming model for the adaptation decision of adaptive communication systems. We propose to specify the adaptation trigger with *event condition action* (ECA) rules. As alternative to the direct specification of ECA rules, we propose to automatically learn ECA rules with extensive experiments in a reproducible execution environment for a given utility function. We implement the *Fossa* framework, which consists of an ECA engine and a learning framework, and show the potential of our learning approach for an adaptive search overlay scenario. In this dissertation, we focus on the general concepts of *Fossa* and refer to [F13, F14] for a detailed presentation. We further refine the ECA-based programming model for topology adaptations in Chapter 7.

### 6.1 MOTIVATION AND APPROACH

Adaptivity is often used to provide high performance in dynamic environments. However, even the simple adaptation of a single system parameter, such as TCP's congestion window, proved to be challenging, as shown by the continued work on TCP congestion controls [19, 52, 122, 161, 202, 211]. Distributed systems with a multitude of complex adaptations, e. g., exchanges of mechanisms and protocol implementations during runtime, introduce additional interdependencies and non-linearities that are hard to model and predict. Understanding the consequences of a congestion control *exchange* at runtime, for example, requires a detailed model of the involved congestion controls. Even for a single congestion control, characteristics, such as the steady state throughput, depend on network specifics (e. g., the used queueing policy and experienced packet loss distributions) and protocol details (e. g., the used fast retransmission and timeout schema) [135]. Thus, developing realistic analytical performance models is extremely hard for non-trivial systems but required for direct modeling of the adaptation decision.

We note that *the definition of relevant performance metrics* is often feasible even though the derivation of an explicit model of the performance functions

*Fourth Research Questions*

*A programming models as abstractions for the adaptation decision*

*Challenging single parameter adaptation*

*More mechanisms, more complexity*

*Implementations and optimization metrics are available*

for a *given implementation* is challenging. The gap between the specification of a utility-based adaptation decision and the concrete adaptation rules hinders the incorporation of adaptations for many distributed systems.

*Learn adaptation decisions*

In this section, we propose *offline learning* for the adaptation decision logic, i. e., *when* to switch to *which* mechanism or parameter, automatically based on a utility function without the need for explicit performance models. This approach leverages the benefits of a utility-oriented adaptive behavior specification, especially the high abstraction level and the straightforward expression of the optimization goal, without the need for a detailed analytical performance model of the application and the underlying distributed infrastructure.

*Fossa*

We present the *Fossa* framework, which consists of *i*) an ECA engine to trigger and execute adaptations in communication systems based on ECA rules and *ii*) an offline learner that uses a reproducible execution environment (e. g., a network simulator), representative workloads, and expected environment conditions to automatically derive the adaptation decision represented as *event condition action* (ECA) rules. The well-defined syntax of the ECA rules allows an efficient genetic programming exploration strategy. If required, *Fossa* generates a Pareto frontier of solutions for multiple utility functions, obviating the need to balance different metrics in advance and making the factual trade-offs visible.

*Learn ECA rules*

*ECA rules as abstraction for the adaptation decision.*

Overall, this chapter introduces two programming models for adaptive communication systems with different abstraction layers. First, the developer of an adaptive communication system can specify ECA rules and benefit from the underlying concepts of complex event processing [15] and stream processing concepts. In particular, the abstractions to specify complex event conditions and the *decoupling* of the monitoring components (information provider) from the adaptation decision logic (information receiver) in a distributed adaptive system are promising. Second, in case the (performance) dependencies of the adaptive communication system and its mechanisms are too complex, the developer benefits from the higher abstraction of a utility-based specification and the automatic derivation of corresponding ECA rules.

## 6.2 BACKGROUND AND RELATED WORK

In the following, we present an overview of related work on adaptive systems and automated learning in software engineering.

### 6.2.1 Adaptive Systems

Adaptive systems incorporate mechanisms to reason about and trigger adaptations, e. g., relying on utility functions, goal policies, or action policies [84]. We note that concepts for adaptive systems are published with slightly varying motivations in the area of *adaptive systems*, *self-adaptive systems* [11, 25, 34], *autonomic computing* [74], and *organic computing*.

*Utility Functions*

**UTILITY FUNCTIONS** Utility functions are a widely used concept to specify preferred configurations with a high abstraction level [84]. Walsh

et al. [195], for example, use utility functions to optimize the resource allocation in a dynamic, distributed environment. To derive the concrete adaptation decisions at runtime, these solutions require a detailed system model to forecast the utility of different configurations and the costs of the adaptations [11].

Utility functions are often used in communication systems to specify implicit adaptations resulting from the network dynamics. Utility functions for the specification of routing based on the flow bandwidth and experienced latencies have been proposed recurrently, e. g., as vision for the Internet [170] or for emerging software-defined networks [58]. Lehn [104] uses utility functions to specify the trade-off of throughput and information staleness for *InterestCast*, a distributed event dissemination mechanism. Further applications of utility functions are in the area of ad-hoc networks [17, 179].

*Utility functions in communication systems*

**ADAPTATION RULES** There are a variety of established approaches to describe the concrete adaptive behavior with rules, such as condition-action rules [44], adaptation strategies and adaptation operators in the rainbow framework [48], priority rules [45], policies [4], and action policies [84]. Additional approaches model the whole feedback loop [194] or extend use case diagrams to explicitly describe the adaptive behavior [114]. These approaches assume *i*) a developer or domain expert who provides the rules or *ii*) a performance model of the application to retrieve the rules (e. g., Gueyoung et al. [78] use queuing models).

*Adaptation rules*

### 6.2.2 Learning and Search-Based Software Engineering

Search based optimization techniques for software engineering [65] often use meta heuristics such as simulated annealing and genetic programming to mitigate the need for analytical models and to cope with large search spaces [144]. Notable works proposed to search offline for superior configurations and tuning parameters of non-adaptive programs [14, 197] or use genetic improvement at runtime to redeploy improved configurations [66].

**GENETIC PROGRAMMING** Evolutionary algorithms enable computers to solve problems automatically based on a given fitness function. For a comprehensive overview, we refer to Poli et al. [144]. Genetic programming is an evolutionary algorithm that operates on a representation of a computer program. The algorithm transforms populations of programs stochastically with *mutation* and *crossover* operations into new, hopefully better populations of programs. The efficiency of genetic programming benefits from a crisp specification of the solution space. Accordingly, a strong type system allows to generate more likely valid candidates [144].

*Evolutionary algorithms*

*Genetic programming*

*Mutation and crossover*

**LEARNING CONGESTION CONTROLS** In the area of communication systems, end-to-end congestion control algorithms are analyzed in detail and discussed controversially [202]. Winstein et al. recently presented *Remy* [203], a program that generates congestion control algorithms for a given objective

*Computer generated congestion controls*

function. Remy internally represents the congestion control as a set of rules that map input values, such as experienced round-trip time ratios, to changes of the current congestion window size. Remy improves these rules by executing (more or less arbitrary) refined rules in a network simulator to judge their performance. The authors showed that Remy outperforms highly optimized congestion controls that are based on analytical models in ns-3 simulations [175, 203]. Remy is tailored to the congestion control domain and suggests the potential of learning adaptation rules for distributed systems.

### 6.3 SPECIFYING ADAPTATION DECISIONS WITH ECA RULES

We propose to use ECA rules to specify the adaptation decision. ECA rules are triggered by *events* which cause the evaluation of the *condition*. If the condition evaluates to true, the *action* leads to a concrete adaptation.

Listing 6.1 illustrates the overall concept with an example for an adaptive, distributed search overlay application. If the number of received messages (MsgRcvEvent) in the last 30 seconds is greater than 5 and the number of hops a message is forwarded (NumOfHops) is less than 10, the parameter transition MyParaTrans is executed with the parameter 10. The event and condition expressions are inspired by event processing languages such as the ESPER Event Processing Language (EPL)<sup>1</sup> and StreamSQL.

```

1 on first match
2   (count(MsgRcvEvent, 30s) > 5) and (NumOfHops < 10):
3   execute parameter transition MyParaTrans at self set value (10);

```

Listing 6.1: Example of an ECA rule.

The *Fossa* ECA engine is implemented efficiently with an event processing unit, which works as message broker between the incoming events and the corresponding ECA rules. The overall architecture of the *Fossa* ECA Engine ensures that only relevant monitoring values are collected to minimize the monitoring overhead for the adaptivity.

### 6.4 LEARNING ECA RULES

The *Fossa* learner is optimized to derive adaptation decision logic represented as ECA rules based on a given utility function. *Fossa* derives a set of ECA rules for adaptive distributed systems that maximizes the experienced aggregated utility of the system execution, i. e.,

$$R_{max} = \operatorname{argmax}_R \mathcal{A}_w^W u(\mathcal{E}(R, w)) \quad (6.1)$$

where  $W$  represents the set of all considered environments and workloads,  $\mathcal{E}(R, w)$  represents the execution of the system with a given rule set  $R$  in an environment and for a workload  $w$ , and  $u$  represents the scalar utility of a given execution.  $\mathcal{A}$  is the aggregation function used to aggregate the utilities achieved in different environments.

<sup>1</sup>See <http://esper.codehaus.org/>.

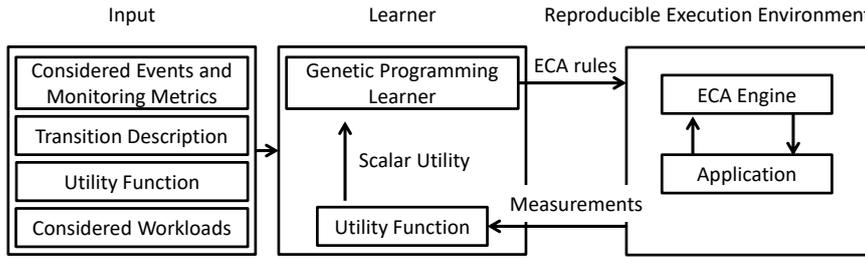


Figure 6.1: Based on the provided input, *Fossa* derives suitable ECA rules relying on multiple test executions in a reproducible execution environment.

Figure 6.1 illustrates the *Fossa* architecture. *Fossa* uses various input parameters for a genetic programming optimization relying on a reproducible execution environment. In particular, *Fossa* relies on:

- The *events* and *monitoring values* that specify the input metric space for the ECA rule events and conditions.
- The *transition description*, which provides a model of the action space of the *event condition action* rules, i. e., of the available adaptations and mechanisms (see Section 6.5).
- The *utility function*, which specifies the fitness of a concrete execution based on the monitored performance metrics.
- A *reproducible execution environment* with an implementation of the system to collect measurements during the execution of a given ECA rule set.
- A set of representative sample workloads and environments.

During the offline learning, *Fossa* triggers the execution of the distributed system with subsets  $R = \{r_1, \dots, r_n\}$  of all possible ECA rules  $\mathcal{R} \supset R$ .<sup>2</sup> The execution  $\mathcal{E}(R)$  of the application with a rule set  $R$  represents a mapping function  $\mathcal{E}(R) \rightarrow \mathcal{M}$ , where  $\mathcal{M} = \{(time, node, type, value), \dots\}$  is the set of all measurements during the execution. The utility function  $u$  maps this set  $\mathcal{M}$  to a scalar utility value  $u(\mathcal{M}) \rightarrow \mathbb{R}$ . Thus, *Fossa* requires a *reproducible execution environment* to retrieve the utility of a given rule set  $R$ .

The reproducible execution environment replaces complex formal and analytical models of the application and the adaptive behavior. Network simulators such as ns-2, ns-3 [126], OMNet++ [128], and PeerfactSim.KOM [183], network emulators such as Mininet [63], and real world testbeds such as Emulab.Net [39] and PlanetLab [143], which are commonly used to develop and evaluate distributed systems, are well suited as underlying infrastructure for a reproducible execution environment. To avoid a mismatch between the specified application behavior and the real implementation, *Fossa* runs the real implementation of the application on top of this infrastructure.

<sup>2</sup>As conditions can contain arbitrary expression combinations, the set of all ECA rules  $\mathcal{R}$  is infinite. For practical reasons the used subset of rules  $R$  contains only a limited rule set.

*Fossa architecture*

*Fossa input*

*Mapping executions to utility*

*Simulators, emulators, and testbeds*

## 6.4.1 Exploration Strategy: Genetic Programming

In the following, we present how *Fossa* uses genetic programming to derive ECA rules for adaptive distributed systems (Figure 6.2). Based on an initial generation of random rule sets or developer provided initial rules, *Fossa* generates and evaluates additional rule sets. Depending on the fitness (specified by the utility function), the evaluated ECA rule set is added to the evolution pool. *Fossa* prefers smaller ECA rule sets in case two different sets lead to the same utility. This should lead to smaller ECA rule sets and allows the developer to interpret the result.

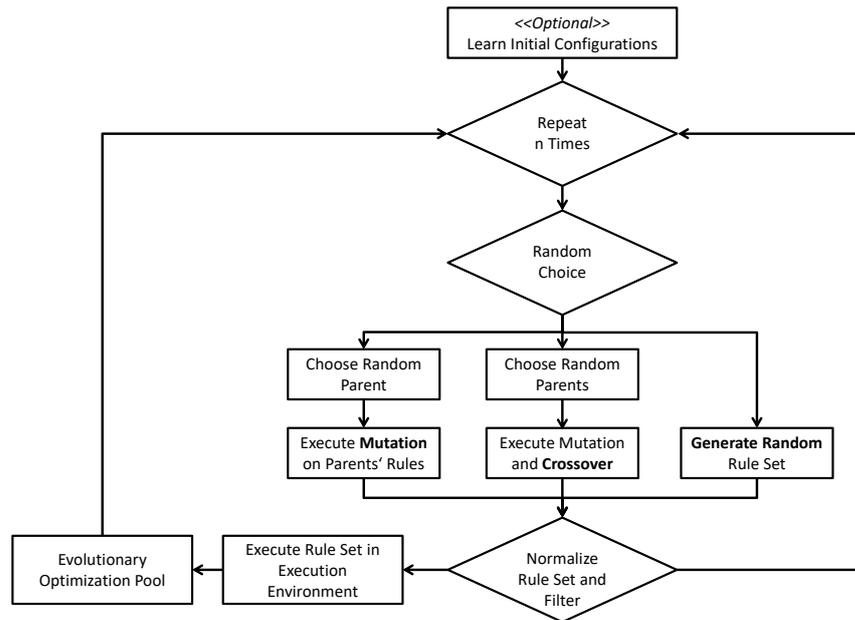


Figure 6.2: Overview of the genetic programming exploration strategy with mutations, crossovers, and random rule generations.

*Fossa*'s genetic programming algorithm operates on the abstract syntax tree of the ECA rules. We illustrate this process for the *Abstract Syntax Tree* (AST) of Listing 6.1, as shown in Figure 6.3.

*Operate on the AST*

The rule has one condition with multiple terms and one action with a parameter transition. The modifications of the genetic programming algorithm change values of the AST nodes and add or remove branches. It is important that these modifications are designed to have a high probability to find useful ECA rule sets. Compared to naive optimizations, the bespoke ECA rule language with its type system and well defined grammar as well as the specification of the monitoring values allows to efficiently avoid invalid ECA rules. In the following, we present *Fossa*'s basic operations to manipulate ECA rules.

*Random rules*

**GENERATE A RANDOM RULE** The condition and the action of a random rule is generated recursively. Therefore, each term chooses random (but lexically and syntactically correct) children. To favor small rules, the probability of a leaf node increases with increasing recursion depth.

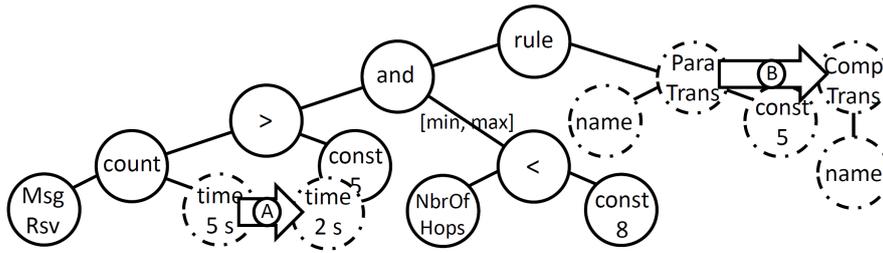


Figure 6.3: Example of mutations on the abstract syntax tree representation of the ECA rule of Listing 6.1.

**MUTATE A RULE** Both the condition and the action of a rule are modified by mutations in different ways:

*Mutations*

- Change the value of a constant (e. g., mutation (A) in Figure 6.3 changes the time interval from 30 seconds to 2 seconds).
- Change an operator, e. g., an *and* to an *or* for two boolean operands.
- Change events or monitoring values.
- Add, remove, or replace subexpressions (branches) based on the node types (e. g., mutation (B) in Figure 6.3 replaces the adaptation action).

Each modification and combinations of modifications are executed with a certain probability. This ensures that *Fossa* tries slight variations as well as huge changes of the existing rules. Additionally, rules and mutations of these rules are used at the same time with a certain probability. This supports the generation of more specialized rules.

**CROSSOVER OF TWO RULES OR RULE SETS** Two rule sets are combined by *i*) generating a new rule set as the union of random subsets of their parents' rule sets and *ii*) combining rules from the different sets, e. g., by replacing branches of one rule with branches of the other rule. The crossover of two rule sets can be very effective. In case, for example, one rule set is optimized for a certain situation and another set for another situation, a combination might perform well in both situations.

*Crossovers*

#### 6.4.2 Efficiency Improvements

As the number of required evaluation executions limits the learning performance, we propose efficiency improvements that rely on the integration of the *ECA Engine* and the *ECA Learner*.

**INCREASE THE MATCHING PROBABILITY** As ECA rules that always evaluate to false have no effect, we propose to increase the probability of a rule match to increase the probability of finding good rules.

*Increased matching probability*

*Fossa* automatically builds a model of the monitoring values during the learning process each time a rule is evaluated. *Fossa* leverages this model to

*Build a model*

generate conditions and expressions that have a high matching probability. If, for example, the monitoring value `Number of Hops` is in the interval  $[20, 50]$ , the expression `Number of Hops < 10` will never evaluate to true (Figure 6.3, leftmost branch). Therefore, *Fossa* uses the monitoring value intervals for a symbolic execution [90] of the expression. The resulting information is used to generate constants that increase the matching probability.

*Favor small  
rule sets*

*Fossa* further monitors the match count of each rule during the execution and removes rules that never match. In case *Fossa* detects that a subexpression always evaluates to the same value during an execution, this subexpression is replaced by the constant term. This reduces the size of the rules without affecting their semantics.

*Decrease required  
evaluations*

**DECREASE THE NUMBER OF REQUIRED EVALUATIONS** During a long running genetic programming optimization, there is a high probability that rule sets are generated twice. We propose to filter semantically equal rule sets to avoid costly repetitive evaluations. As semantically equal rule sets might have different representations, we normalize the rules for duplicate detection. The expression `5 < MonitoringValue`, for example, has the same semantic as `MonitoringValue > 3 + 2`. We normalize rules with constant folding and ordering of commutative operation operands.

*Filter semantically  
equal rule sets*

*Fossa* further schedules experiments to minimize the number of required evaluations by discarding a rule set as soon as the aggregated utility cannot reach the required minimum.

## 6.5 TRANSITION DESCRIPTION AND EXECUTION

For the *description* of the considered and available adaptations during the exploration and for their *execution* at runtime, *Fossa* relies on our transition model [F14]. This model allows the definition and execution of transitions following a well defined lifecycle for component exchange and provides proxy-based abstractions for the application developer. Figure 6.4 illustrates the lifecycle of an elementary *runrun transition*, as introduced in [F14] and further refined by [151, 155]. The state transitions in the lifecycle are annotated with the triggering state transition events and triggered component transitions. All components that implement this lifecycle can be controlled and considered by *Fossa* without modifications. Alternative approaches for compositional adaptation [119], such as [40, 82, 157, 158], might be integrated in future work and might benefit from *Fossa*'s learning approach as well.

*Transition lifecycle*

## 6.6 EVALUATION

In the following, we present previous evaluations of *Fossa* for an adaptive search overlay sample application with five different environment settings during the learning process [F12, F13]. For the performance of the genetic programming approach, there are two baselines: the non-adaptive static application and randomly generated ECA rules. Figure 6.5 shows that *Fossa*

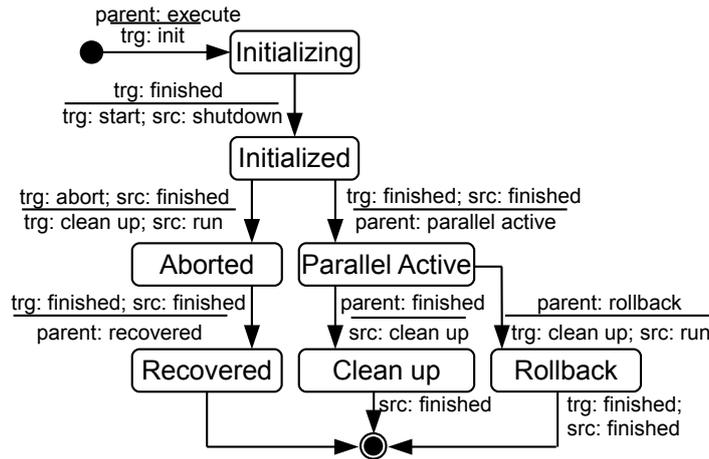


Figure 6.4: Lifecycle of a *runrun* transition as introduced in [F14].

outperforms both baselines after a few hundred simulations. Even though the average utility for all environment settings is strictly increasing with the genetic programming search, the minimum and maximum utilities for certain environments might decrease. Note that we cannot compare our results with an optimal solution, due to the lack of a corresponding performance model and as a full search is not computationally feasible due to the infinite space of possible ECA rules.

*Outperforms static and random solutions*

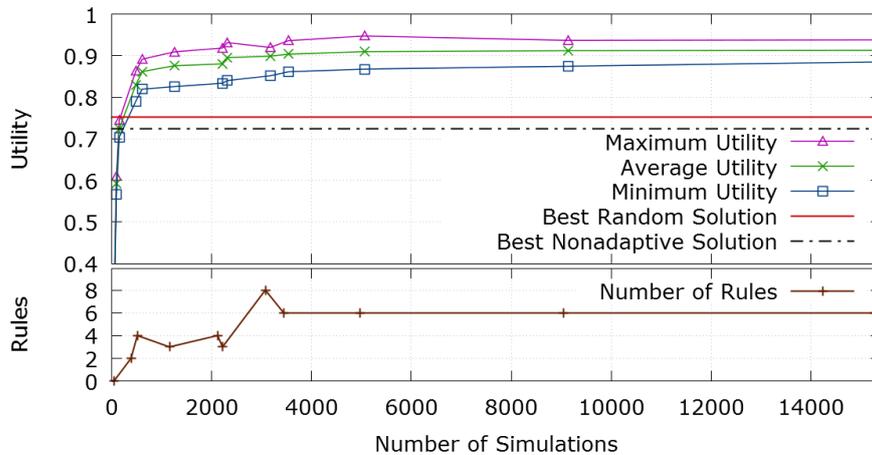


Figure 6.5: Improvement of the utility during the execution of *Fossa* for an adaptive search overlay sample application.

The evaluation in [F12, F13] further shows that the proposed efficiency improvements (Section 6.4.2) reduce the required number of simulations to 30% for this application scenario.

Figure 6.6 shows an example how *Fossa* optimizes for two contradictory optimization goals. Therefore, *Fossa* keeps all ECA rule sets that are on the Pareto frontier of both optimization goals in the evolutionary pool. Here, we balance the resource consumption (X-axis) and the number of failed requests (Y-axis) of the search overlay sample application. The black points repre-

*Pareto frontier generation*

sent all static configurations without ECA rules. The red points represent all learned ECA rule sets that lead to the corresponding performance metrics. As lower values are better, the figure shows that the adaptive solutions with the learned ECA rules dominate all static configurations. *Fossa* and the generated Pareto frontier allows the developer to balance the contradictory optimization goals and the aggregated utility function based on the actually necessary trade-offs.

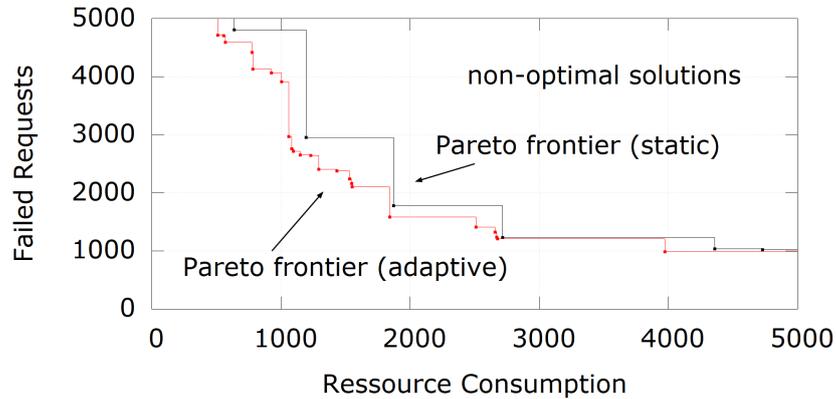


Figure 6.6: Pareto frontier for two contradictory optimization goals (lower is better). The adaptive solutions with the learned ECA rules dominate all static configurations.

#### Further applications

We further found the concept of ECA rules useful for the specification of adaptation decisions for the transition between consensus protocols in Zookeeper [F18], the activation of cellular interfaces in multipath environments [F20, F21], and as starting point for the development of the topology adaptation rule language TARL [F8].

## 6.7 DISCUSSION AND FUTURE WORK

### 6.7.1 ECA Rules for Adaptation Decision Logic

#### Yet another rule language?

In this work, we focus on the offline learning of adaptation decision logic and introduced the concise, well-defined ECA model for adaptation rules. In general, we assume that *Fossa*'s genetic programming optimization is applicable on other executable adaptation descriptions with well-defined semantics. However, we showed that the integration of the ECA model and the learning algorithms enables significant efficiency improvements, e. g., by using the notion of semantically equal ECA rules and controlling required monitoring based on the requirements of the specified rules.

#### Stream processing abstractions for adaptation rules

To the best of our knowledge, we are the first to explicitly propose stream processing concepts for *self-adaptive systems* and *autonomic computing*. This enables to use well known, powerful abstractions for the adaptation decision, e. g., to integrate aggregations in conditions instead of artificially creating new, pre-aggregated metrics.

*Fossa* can be classified in the explicit *MAPE-K* reference model for autonomous computing by Kephart et al. [83]. The *Fossa ECA Engine* runs the monitoring and the execution at runtime to execute ECA rules, whereas the *Fossa ECA Learner* executes the analyze and plan part offline. This reduces the overhead of the adaptivity at runtime, as the evaluation of the ECA rules is usually more efficient than online learning. Our work implicitly addresses research challenges identified by Cheng et al. in their research roadmap for self-adaptive systems [25]. For example, in an adaptive system, it is important to ensure that the monitoring overhead does not nullify the benefit of the adaptivity. In existing adaptive systems, it is complex to model the performance impact of the monitoring and the execution of the adaptation. As our proposed methodology optimizes based on the overall utility, the impact of the used ECA rules and their execution is always considered.

*MAPE loop*

*Adaptive systems  
research roadmap*

### 6.7.2 Learning Approach

*Fossa* provides the benefits of a utility-oriented adaptive behavior, especially the high abstraction level and the straightforward expression of the optimization goal, without the need for a complex and detailed analytical performance model of the application and the environment. As *Fossa* uses utility functions as black boxes, arbitrary utility functions are supported. In contrast, many optimization methods require certain properties and representations, such as a closed-form expression, continuity, or differentiability.

*The best of  
two worlds:  
utility and rules*

*Fossa* automatically learns rules that are tested in a set of representative workloads and environments. There are, however, general reservations with regard to the applicability of learned algorithms. A prediction for the behavior of learned rules with slightly different workloads and environments is impossible. This is a general debate for learned algorithms. Learned rules require a careful test and deployment. As the ECA rules are human readable, the developer should examine them to obtain insights on the adaptive behavior.

*Can we trust  
the rules?*

Even in application scenarios that do not tolerate only partially understood learned solutions, the learned rules provide a benchmark for the performance of alternative solutions. Sivaraman et al. [175], e. g., used their learned congestion control as benchmark to discuss the potential of congestion controls, as the *optimal* achievable performance of a distributed coordination algorithm is unknown.

*Benchmark for the  
potential*

Figure 6.7 illustrates alternative approaches to learn adaptation logic relying on emerging learning frameworks [1] or established classifications and regressions, as proposed by the author of this dissertation in [F22]. Accordingly, these approaches have different representations of the learned model, e. g., a neuronal network, and require additional feature engineering to choose relevant metrics and their aggregations. The aggregation is implicitly learned in *Fossa*, as it is part of the underlying stream processing model of the ECA rules. Furthermore, *Fossa* learns the consequences of an adaptation, including the costs of an adaptation, and includes a mechanism to choose additional required simulations.

*Alternative:  
classification and  
regression*

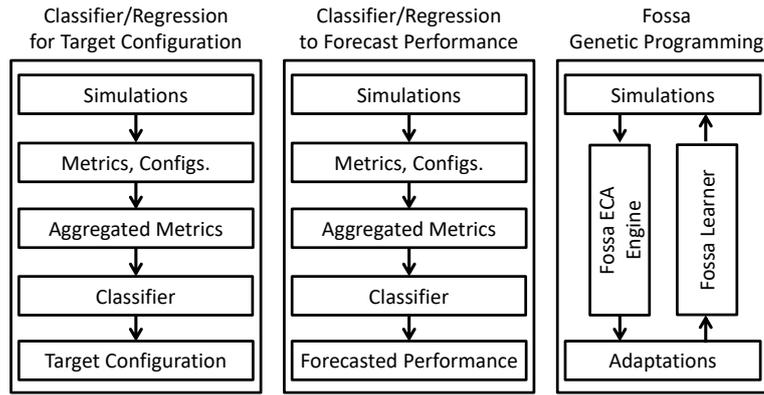


Figure 6.7: Overview of classification- and regression-based alternatives for *Fossa*'s ECA-based genetic programming approach.

*Search space reduction*

*Fossa* showed the feasibility and potential of learning adaptation rules based on utility functions. We envision that *Fossa* would benefit from the integration of rule generation concepts from complex event processing [116] and more sophisticated semantics of the adaptation execution, e. g., to incorporate knowledge about valid configurations and component compositions [10] to reduce the search space and ensure reconfiguration consistency. Like most machine learning approaches, *Fossa* provides a lot of adjustable hyper-parameters to tune the learning process. A detailed analysis and automatic tuning of these parameters, similar to [150], remains promising future work.

*Generic Fossa*

Based on our findings, we envision a *Generic Fossa* for domain-specific languages in communication systems as shown in Figure 6.8. *Generic Fossa* takes a specification of the domain-specific language<sup>3</sup>, e. g., the context-free grammar of the language as Backus–Naur form, an attribute grammar [92], or even the specification of the language type system and formal semantics, as input. The genetic programming algorithm operates on the model of the domain-specific language and further requires a reproducible execution environment with a runtime environment of the domain-specific language to automatically learn programs in the communication system.

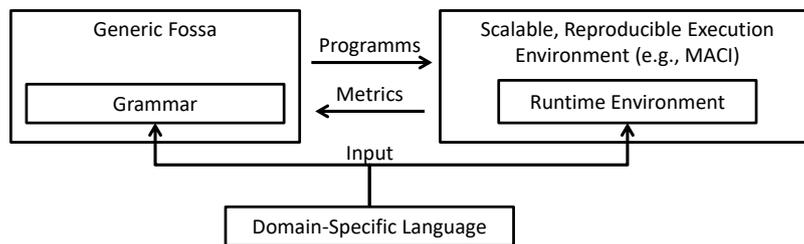


Figure 6.8: Vision of a *Generic Fossa* in *MACI*, which leverages genetic programming for various domain-specific languages given their grammar.

<sup>3</sup>Note that this is not a specification *in* the domain-specific language, but a specification *of* the domain-specific language.

## A PROGRAMMING MODEL FOR TOPOLOGY ADAPTATIONS

---

In the previous chapter, we presented a programming model as abstraction for the specification of adaptation decisions based on ECA rules that are optimized to meet a given utility function. In this chapter, we focus on *topology adaptations* in the domain of adaptive communication systems and refine our solution for the fourth research question with regard to *topology adaptations*.

*RQ IV: What are suitable abstractions for the specification of the adaptation decision for adaptive communication systems?*

*Fourth Research Questions*

In this chapter, we present our extension of the presented ECA rule model to the domain of topology adaptations.<sup>1</sup> For a detailed presentation, we refer to the respective publications in [F3, F8].

### 7.1 MOTIVATION AND APPROACH

Topology adaptations enable networking applications to cope with changing environments and requirements, e. g., by switching between mesh and tree topologies [201]. Video streaming overlays, for example, adapt their video distribution topology to cope with fluctuating peers [137, 156, 169, 193, 196]. Topology adaptations are applied in wireless sensor networks to reduce energy consumption [164], for event dissemination in dynamic virtual realities [104, 105], for load balancing in search overlays [95], and connectivity maintenance in peer-to-peer networks [112]. We note that the related work on topology adaptations in these domains focuses on concrete topology adaptations for their applications and optimization goals. *Abstractions* or models for the specification and execution of topology adaptations are missing.

*Executable topology adaptations ...*

*... are missing abstractions.*

*TARL*

We propose the topology adaptation rule language *TARL*, which provides abstractions to specify and execute topology adaptation. *TARL* reduces the complexity of topology adaptations by decoupling the adaptation logic from the application logic. *TARL* provides a reusable runtime environment, which enables rapid development and evaluation of novel topology adaptations.

---

<sup>1</sup>The content of this section emerged from a cooperation with Michael Stein and Roland Kluge. The author of this dissertation contributed major parts of the *TARL* language syntax and expressiveness evaluation, whereas Michael Stein and Roland Kluge contributed the analysis of topology adaptation characteristics, the conceptual architecture for topology adaptations, and their general background in topology adaptations.

## 7.2 TOPOLOGY ADAPTATION RULE LANGUAGE TARL

In the following, we present the topology adaptation rule language TARL. In contrast to existing adaptation rule languages in Section 6.2.1, TARL is specifically designed for the characteristics and requirements of topology adaptations in networking applications.

*Characteristics of  
local topology  
adaptations*

*Conceptual  
architecture*

Stein et al. [F8, 180, 181] identified the four characteristics of local topology adaptations: *i*) locality, *ii*) simple graph operations, *iii*) pattern based decision making, and *iv*) multiple topologies. The authors further present a conceptual architecture for local topology adaptations, which decouples the local topology adaptation logic, the topology provider, and the application. TARL is designed based on this conceptual architecture and these four characteristics. Accordingly, TARL supports the wide range of applications that execute topology adaptations locally on each device in a decentralized way.

*Rule-based  
adaptation logic*

TARL rules specify the topology adaptation logic in a declarative rule-based manner. A TARL rule consists of three subsequent parts, *i*) a *preamble* with constant and selector declarations, *ii*) the *condition* specification, and *iii*) the *action* part. The execution part is triggered depending on the condition evaluation. In the following, we focus on the condition and execution part.

*Example: low delay  
jump topology  
adaptation*

ILLUSTRATING EXAMPLE Listing 7.1 shows a TARL rule example. The TARL rule specifies the *low delay jump*, as proposed by Wang et al. [196]. The low delay jump minimizes latency in overlay video streaming topologies by reducing the depth of the streaming tree. For this purpose, nodes try to replace their parents with alternative nodes that have a lower tree depth. The corresponding TARL rule contains two match expressions to specify the *currentParent* (line 3) and all *parentCandidates* (line 5). In case the *treeDepth* of the *parentCandidate* with the lowest *treeDepth* (line 4–6) is smaller than the current *treeDepth* (line 1–7), both neighbors are exchanged (line 8). Following the conceptual architecture of TARL, the specified rule relies on a monitoring of the topology and assumes an actor to manipulate the topology.

```

1 filter(
2   join(
3     match(TP, Tree, currentParent - e0 -> self),
4     min(
5       match(TP, Neighborhood, parentCandidate - e1 -> self),
6       parentCandidate.treeDepth)),
7   currentParent.treeDepth > parentCandidate.treeDepth)
8 execute every match:
9   at (self, TP, Tree) move_neighbor (currentParent, parentCandidate)

```

Listing 7.1: TARL rule that implements the *low delay jump* topology adaptation, as proposed by Wang et al. [196].

*Topology graph  
pattern matching*

CONDITIONS Table 7.1 provides an overview of the language primitives to handle topology graph patterns in the condition. Here,  $M$  represents a set of matches. A match is a mapping from nodes of the pattern  $P$  to nodes in

Table 7.1: TARL language primitives for the condition specifications to express topology graph patterns.

---

$M \leftarrow \text{match}(\text{Topology Provider}, \text{Topology Name}, \text{Pattern } P)$
$M \leftarrow \text{filter}(M, f_B)$
$M \leftarrow \text{join}(M, M)$
$M \leftarrow \text{min}(M, f_R) \text{ and } \text{max}(M, f_R)$
$\text{Number} \leftarrow \text{count}(M)$

---

the topology graph and from edges in the pattern  $P$  to edges in the topology graph.<sup>2</sup> Thus, the `match` operation returns the set of all matches, which represents the set of all found occurrences of the pattern  $P$  in the topology graph. Further operations allow to filter matches based on boolean predicates, join matches of potentially different topologies, select subsets of matches that lead to the `min` or `max` values of numeric properties, or count the number of matches.

**ACTIONS** The execution part is triggered for each match in the condition. Table 7.2 provides an overview of the language primitives to modify the topology based on the identified three recurring graph operations.

*Topology pattern modification*

Table 7.2: TARL language primitives for the action specification to express topology graph pattern modifications.

---

<code>add_neighbor(N, Topology Provider, Topology Name)</code>
<code>remove_neighbor(N, Topology Provider, Topology Name)</code>
<code>move_neighbor(N1, N2, Topology Provider, Topology Name)</code>

---

### 7.3 EXPRESSIVENESS EVALUATION

In the following, we evaluate the expressiveness of TARL. Therefore, we revisit established topology adaptation algorithms and discuss an eye tracker supported TARL user study.

**REVISIT TOPOLOGY ADAPTATION ALGORITHMS** To evaluate the expressiveness of TARL, we specified representative topology adaptation algorithms from the overlay video streaming and wireless sensor network domains with TARL (Table 7.3).<sup>3</sup> We followed the description and presentation of the original papers. A proof of the semantic equality of the TARL representation and the presented algorithms in the original papers is beyond the scope of this section. We discuss the general requirement for a notion of semantic equality in Section 7.5.

*Expressiveness*

<sup>2</sup>We provide a graph based specification of the operator semantics in [F8].

<sup>3</sup>The modeled TARL rules of the original publication [F8] are available at <http://www.dvs.tu-darmstadt.de/tarl>.

Table 7.3: Revisited topology adaptation algorithms for an evaluation of the expressiveness of TARL.

Algorithm	Modeled	Lines of Code	Filter	Join	Max/Min	Selector	Topologies	Add	Remove	Move
<b>Domain: Video Streaming Overlay</b>										
<b>Chunkyspread [193]</b>										
Load Optimization	✓	18	✓	✓		✓	≥ 3			✓
Latency Optimization	✓	25	✓	✓		✓	≥ 3			✓
Distributed Market [137]	✓	42	✓	✓	✓	✓	≥ 2			✓
<b>DCO [169]</b>										
Provider Selection	✓	10	✓		✓		2	✓		
Requester Selection	✓	7			✓		2	✓		
<b>mTreebone [196]</b>										
Low Delay Jump	✓	9	✓	✓			2			✓
High Degree Preemption	✓	20	✓	✓			2	✓	✓	✓
<b>TRANSIT TopT [156]</b>										
Low Delay Jump	✓	21	✓	✓	✓	✓	≥ 2			✓
High Degree Preemption	✓	41	✓	✓	✓	✓	≥ 3	✓	✓	✓
<b>Domain: Wireless Sensor Networks</b>										
kTC [164]	✓	17	✓				2			✓
Gabriel [199]	✓	9	✓				2			✓
Relative-Neighborh. [81]	✓	9	✓				2			✓
Yao [209]	✓	13	✓		✓	✓	2	✓		
LMST [107]	no									

*Appropriate level of abstraction*

We successfully specified 13 out of the 14 analyzed algorithms. As all language operators are used repeatedly in both domains, we conclude that TARL provides an appropriate level of abstraction to specify a wide range of topology adaptation algorithms. Although wireless sensor networks exhibit multiple topologies, the corresponding conditions only rely on one of these topologies. Thus, join operations are only used in the video streaming overlays. TARL cannot express LMST [107], as this would require matching paths of arbitrary length. Even though TARL could be extended to support LMST, we discarded an extension solely for a single use case.

*Convenient specification*

Our specified TARL rules require between 7 and 42 lines of code. A comparison with the required lines of code in the original publications is hindered by the accessibility of their source code and the effort to analyze their implementations. However, we claim that a central specification of the adaptation logic in a few lines is convenient and maintainable for the developer.

**USER STUDY** To substantiate the claim that TARL increases the efficiency of the developer and the maintainability of the application, we conducted a limited eye tracker supported user study with 12 students from TU Darmstadt.<sup>4</sup> During the study, participants were asked to *interpret* a given adaptation rule, i. e., to specify the topology change caused by the rule (Figure 7.1). Our study shows a measurable advantage of TARL with regard to accuracy, solution time, and visual effort compared with a Java-based alternative. Figure 7.2, for example, shows that TARL significantly reduces the number of fixations and therefore the visual effort.

*Eye tracker supported user study*

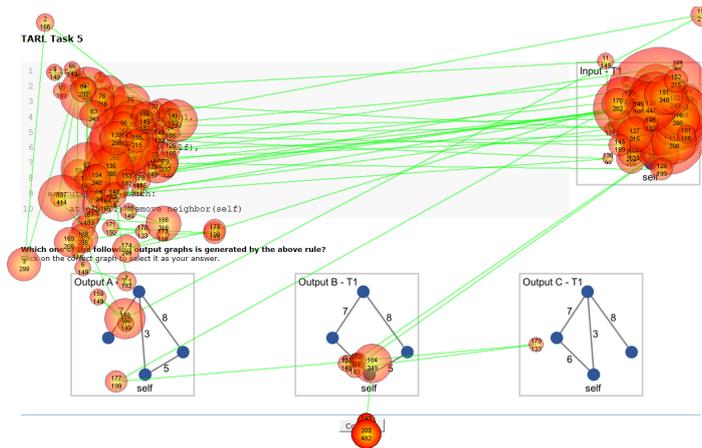


Figure 7.1: User study setup and visual fixations. The participants were asked to *interpret* a given adaptation rule (top left). The orange circles visualize eye fixations of the participants during the task. Figure taken from [S4].

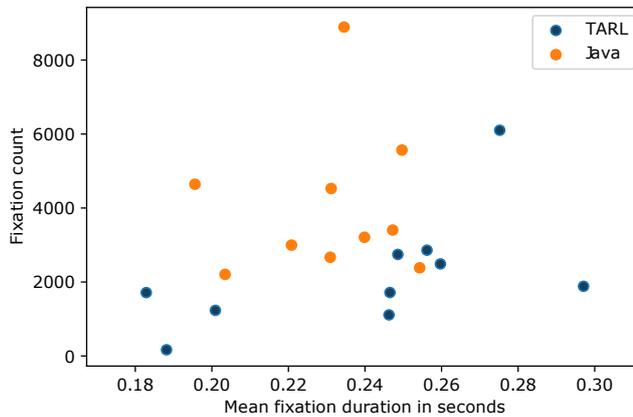


Figure 7.2: While the participants showed comparable fixation durations with TARL and Java, the number of fixations is significantly lower with TARL. Figure taken from [S4].

<sup>4</sup>This user study was conducted by Gregor Albrecht as part of his bachelor thesis [S4], which was motivated and supervised by the author of this dissertation.

## 7.4 DISTRIBUTED TOPOLOGY PATTERN MATCHING

*Programming model  
abstracts over  
implementation  
details...*

*... and enables  
optimizing execution  
environments.*

*Distributed topology  
pattern matching*

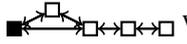
TARL enables the specification of topology adaptation logic and abstracts over implementation details of the execution environment. Accordingly, the TARL execution environment can optimize, e. g., the underlying topology pattern matching. In [F8], we present the implementation of a proof-of-concept TARL runtime environment and discuss potential optimizations. In [F3], we further present the first *distributed pattern matching* (DPM) protocol<sup>5</sup> for topology adaptations. In this protocol, participating network nodes send partial pattern matches based on their local knowledge to their neighbors. The neighbors try to complete the pattern matching process based on their local knowledge. Thus, the DPM protocol enables the matching of patterns that are larger than the local view size of the participating nodes.

*The case for  
extensive  
experiments  
with MACI.*

DPM differs fundamentally from traditional topology pattern matching. Accordingly, we identify the need for a systematic analysis of the distributed pattern matcher, i. e., with regard to major configuration and environment parameters and their impact on performance metrics. We use *MACI* to systematically analyze the impact of *i*) the topology graph with its graph parameters such as its node degree, *ii*) the pattern and its size, and *iii*) the local view size. While details of the DPM protocol and detailed case studies are contributed by Michael Stein and presented in [F3], we focus on a summary of the experimental design and the *MACI* perspective. In the following, we start with a presentation of the evaluation and the major findings before we discuss the contribution of *MACI*.

## 7.4.1 Exploring Distributed Topology Pattern Matching

*A lower pattern  
frequency...*

In this evaluation, we use Erdős-Rényi graphs with varying *i*) number of nodes, *ii*) average node degree, and *iii*) edge weights. We use the topology pattern  with the self-node , as shown in Listing 7.2.<sup>6</sup> To systematically analyze the impact of the *pattern frequency* on the required number of messages per node, we further use a constraint for the edge weight in the pattern to control the pattern frequencies by changing the edge weights in the graph. Here, the *pattern frequency* is the average number of matches per node ( $\frac{\# \text{ matches}}{\# \text{ nodes}}$ ). In general, the pattern frequency is a contextual property based on the interference between the pattern and the topology graph.

```
1 match( self <- e1 -> n1, n1 <- e2 -> n2, n2 <- e3 -> n3,
2   n1 <- e4 -> n4)
3 execute every match: ...
```

Listing 7.2: TARL rule with a topology pattern of a horizon of 3.

<sup>5</sup>We refer to the distributed pattern matching approach as *protocol*, because it essentially defines how messages are exchanged. From a conceptual point of view, the *protocol* implements a distributed pattern matching *algorithm*.

<sup>6</sup>Our evaluation focuses on the *distributed* pattern matching protocol. Therefore, we use patterns that require a view size of more than two and that are usually infeasible with traditional pattern matching (horizon > 2). Additional optimizations, e. g., moving the *self* node in the pattern, would reduce the required horizon of the pattern.

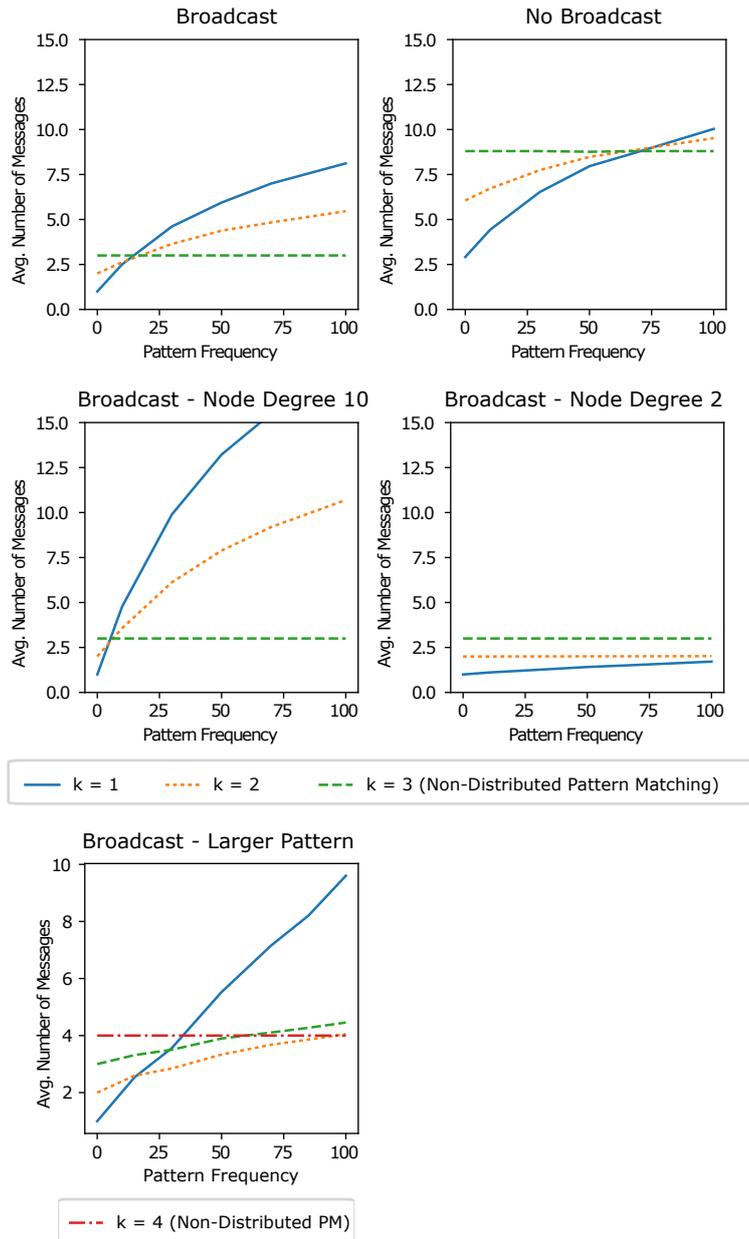


Figure 7.3: While the non-distributed pattern matcher induces a constant overhead regardless of the pattern frequency, the number of messages of the distributed pattern matcher depends on the pattern frequency. This visualization is automatically provided by *MACI* and only slightly modified for the presentation in this dissertation.

Figure 7.3 shows the average number of messages per node depending on the pattern frequency and the local view size  $k$ . The message overhead of the non-distributed pattern matcher depends on the view size but is independent of the pattern frequency. The distributed pattern matcher overhead depends on the number of matching candidates and matches, as the algorithm sends search messages for partial matches. Thus, the number of messages increases with the pattern frequency for distributed matching, whereas it is constant for the non-distributed case (Figure 7.3,  $k = 3$  left,  $k = 4$  right).

... increases the benefit of DPM.

*Missing broadcast functionalities...*

A basic communication primitive in various network environments is broadcast, which denotes the operation of delivering a message sent only once to all 1-hop neighbors of the sender. Broadcast messages are not available in all applications and environments, e.g., wireless networks usually provide broadcasts thanks to a shared medium, whereas overlay networks typically do not. Broadcasts reduce the cost to retrieve a local view because they provide an efficient local communication. Figure 7.3 compares *Broadcast* and *No Broadcast* scenarios. The benefit of the distributed pattern matcher is higher for environments without broadcasts.

*... increase the benefit of DPM.*

*The optimal k...*

We further evaluate the impact of graph parameters in Figure 7.3. In the chosen setting, small node degrees favor small local views: For an average node degree of 2, the configuration with  $k = 1$  is superior for all pattern frequencies. The evaluation with a *Larger Pattern*, which requires  $k = 4$  for the non-distributed matcher, in Figure 7.3 supports these observations. In this example, the optimal  $k$  is 1, 2 or 4 depending on the pattern frequency. This shows that the optimal  $k$  for distributed matching is not always 1 but depends on both the pattern and the topology graph.

*... depends on the pattern and environment.*

#### 7.4.2 MACI Perspective

To conduct the previous experimental design study, we relied on *MACI*, our framework for the seamless execution and analysis of extensive network experiments, as presented in Chapter 4. Overall, *MACI* provided support for all *non-TARL* and *non-DPM specific* tasks and enabled us to focus on the specific aspects for our analysis of distributed topology pattern matching.

**CUSTOM SIMULATOR** The *TARL* runtime environment and the *DPM* protocol are implemented in a Java-based simulator. We extended the existing simulator with a few lines of code to store the target metrics in *MACI*. Therefore, we relied on the Java API of *MACI*, as shown in Listing 7.3.

```

1 public void record(final String key, final int value)
2     throws IOException;
3
4 public void record(final String key, final int value,
5     final long time) throws IOException;
6
7 public void record(final String key, final String value)
8     throws IOException;
9
10 /* ... */
11
12 public void warn(final String key, final String message)
13     throws FileNotFoundException;
```

Listing 7.3: Java API to store target metrics with *MACI*.

*Interactive analysis*

**INTERACTIVE ANALYSIS** We benefited from the interactive analysis features of *MACI*, which generated the visualizations as shown in Figure 7.3

based on our specified target metric, filter, and aggregation selection. The visualizations are only slightly modified for the presentation in this dissertation. Besides these visualizations, additional time-based *MACI* visualizations are used in [F3, 180].

**SCALABLE EXECUTION** The used simulator provides full isolation and therefore does not require an own operating system per execution, as Mininet does. Thus, we benefited from the parallelism of the setup with up to 100 parallel experiment executions.

## 7.5 DISCUSSION AND FUTURE WORK

In this section, we introduced the executable specification language TARK as abstraction for topology adaptations. TARK closes the gap between the domain of general graph pattern matching and graph rewriting [12, 49, 102], and the diverse topology adaptations in various domains of communication systems [95, 104, 105, 112, 137, 156, 164, 169, 193, 196, 201].

**USER STUDY** Established eye tracker supported user studies in the area of programming languages [64, 168, 190] showed that subtle differences in the syntax and variable naming have a significant impact on the understandability of source code. Accordingly, our user study, which indicates that TARK rules require less visual effort and are easier to read than comparable Java implementations, has to be treated carefully. In general, we note that the design of domain-specific languages with regard to the user experience is based on educated guesses and lacks well-established best-practices.

**FUTURE WORK ON SEMANTICS** While TARK provides well defined semantics based on graph pattern matching, we note that semantics and consistency models for the environment monitoring and the execution of topology modifications are missing. The topology monitoring, for example, does not specify at which point in time the reported topology on a local node was monitored in the network. The distributed nature of communication systems, the usually asynchronous communication, and the constant environment changes introduce many imprecisions. Thus, the reported topology might be a combination of different snapshots and might never have been in the reported state. The execution part of TARK rules does neither guarantee that all topology modifications are executed atomically in the distributed environments nor what happens in case an operation fails. We note that the revisited application scenarios do neither provide nor require these semantics due to their nature of *best effort* overlay networks and wireless sensor networks.

However, these semantics are required to reason about topology adaptations. For future work, we anticipate a notion of *semantically equal* topology adaptations based on the envisioned consistency models. This would enable us to discuss the semantic equality of the TARK representation and the presented algorithms in the original papers. Furthermore, a notion of semantic

*Future work on semantics and consistency models...*

*...for topology monitoring...*

*...and topology modifications.*

*Notion of semantic equality*

equality is required to discuss meaningful optimizations for TARL and reason about correctness properties. For example, it seems trivial that both patterns  $\text{self} - e_0 - n_0 - e_1 - n_1$  and  $n_0 - e_0 - \text{self} - e_1 - n_1$  are semantically equal. However, in a highly dynamic distributed environment, the required two hop monitoring data might be outdated whereas the one hop data finds the match. This limitation makes reasoning about *i*) the correctness and semantic equality of local topology and distributed topology pattern matching and *ii*) the concurrent execution of multiple topology adaptations meaningless. Note that such a consistency model does not have to provide strong guarantees but concepts to reason about possible states.

In this chapter, we present a *MACI* case study with a DASH video streaming analysis using extensive network experiments.<sup>1</sup> The result of this analysis is a valuable contribution for the video streaming domain that goes beyond an evaluation of *MACI*, our solution for the second research question. In this chapter, we focus on a presentation of the general concepts and findings. For a detailed presentation, we refer to our publications [F2, F6, F9].

*MACI (RQ II)*  
evaluation...  
... and video  
streaming  
contribution.

## 8.1 MOTIVATION AND BACKGROUND

Multimedia communication systems have to be carefully designed to provide a satisfying user experience [182]. Dynamic Adaptive Streaming over HTTP (DASH) [176] adapts the video quality at runtime to cope with changing network conditions. The HTTP server offers each video segment in various video bitrate representations. At the client side, the *DASH player* uses an *adaptation algorithm* to dynamically choose the representation of the next segment considering the previously observed network conditions. The DASH player optimizes for various metrics that impact the perceived *Quality of Experience* (QoE) of the user [167], such as a high video quality and a low number of playback interrupts (stallings).

*Dynamic Adaptive  
Streaming over  
HTTP (DASH)*

There is a large number of DASH players, such as *DASH.JS*<sup>2</sup> and *Shaka*<sup>3</sup>, which implement their particular default adaptation algorithms as well as additional algorithms. These adaptation algorithms are based on different adaptation principles, such as *throughput-based adaptation* and *buffer-based adaptation*. We refer to [96] for a survey on rate adaptation techniques for DASH. Each player provides an own set of configuration parameters. We notice that empirical evaluations of adaptation algorithms are typically conducted in a single player [73, 159, 178] or academic DASH emulators such as *AStream*<sup>4</sup>.

*Various DASH  
player...  
... with various  
adaptation  
algorithms...*

*... and various  
configuration  
parameters...*

Furthermore, today's DASH players have to ensure a high user satisfaction in a wide range of fluctuating network conditions. A recent publications of Google, for example, show that their customers experience mean round-trip times between 38ms in South Korea and 188ms in India [100].

*... have to operate in  
various network  
environments.*

<sup>1</sup>The presented work in this chapter is the result of a joint effort with Denny Stohr and was previously published in [F6, F9]. The author of this dissertation contributed the general methodology and implementation for large extensive network emulations, whereas Denny Stohr contributed the DASH video streaming specific aspects and implementations, such as the used DASH player abstraction and video metric collections. The analysis and interpretation of the results are joint work.

<sup>2</sup>See <https://github.com/Dash-Industry-Forum/dash.js>.

<sup>3</sup>See <https://github.com/google/shaka-player>.

<sup>4</sup>See <https://github.com/pari685/AStream>.

The case for systematic extensive MACI experiments.

Considering this large evaluation space, we make the case for systematic experimental comparisons and evaluations of today’s DASH player designs and implementations. The required systematic extensive experiments for the application scenario of DASH video streaming substantiates our argument for the *reusable* experiment framework *MACI*.

## 8.2 EXPERIMENTAL DESIGN APPROACH

60 configurations...

... in 20 environments.

In this section, we present our experimental design for a large DASH player and adaptation algorithm analysis. Based on the previous motivation, we chose the player, the adaptation algorithm, the video segment lengths, a range of target playout buffer sizes, and the network environment parameters as control variables (depicted in Table 8.1). For an evaluation of the network parameters, we concentrate on a systematic comparison of the available bandwidth with varying statistical properties. We refer to our previous publication for an analysis of the impact of the loss rate [F9] and a detailed presentation of the DASH player specific abstractions [F6]. We use the established *Tears of Steel*<sup>5</sup> DASH data set prepared by [103], featuring nine H.264-AVC encoded representations in the range of 0.243Mbps to 10Mbps.

Table 8.1: Experiment control variables.

	Variable	Values
Configuration	Player	DASH.JS, Shaka, AStream
	AA	standard, BOLA
	segment length	1, 2, 6, 10, 15 [s]
	target buffer size	default, 5, 20 [s]
Environment (Avail. BW)	$\mu_{\text{BW}}$	0.8, 2, 5, 7.5, 10 [Mbps]
	$\sigma_{\text{BW}}^2$	0, 0.8, 2, 5 [Mbps <sup>2</sup> ]

## 8.3 DASH ANALYSIS OVERVIEW

Increasing bandwidth variance, increasing stalling durations.

15s segments show larger stalling durations than 6s segments.

In this section, we provide an overview of our DASH player and adaptation algorithm analysis results. Figure 8.1 shows the total stalling duration during playback of the analyzed DASH players and their adaptation algorithms under varying network conditions and segment length configuration of 6 seconds. The stalling duration increases with the bandwidth variance for the *DASH.JS* and *Shaka* player, as higher variances increase the bandwidth fluctuations. *AStream*, however, exhibits the opposite trend and shows in general larger stalling durations. To evaluate the impact of the segment length, Figure 8.2 shows the same comparison for 15 second segments. We see that the stalling duration with 15 second segments is significantly larger for the *DASH.JS* and *Shaka* player. We assume that this is a consequence of the used default player

<sup>5</sup>See <https://mango.blender.org/>.

buffer size (between 10 and 12 seconds), which is smaller than the segment length. Again, we note that *AStream* exhibits the opposite trend. A more detailed analysis showed that implementation errors in the emulator code lead to this unexpected and unrepresentative behavior [F6].

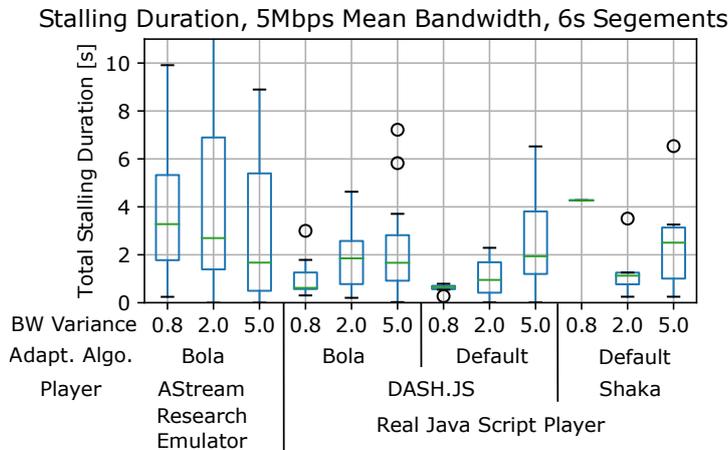


Figure 8.1: Total stalling duration of DASH players with various adaptation algorithms, and available bandwidth volatilities for a segment length configuration of 6s (lower is better, experiment setup as described in Section 8.2).

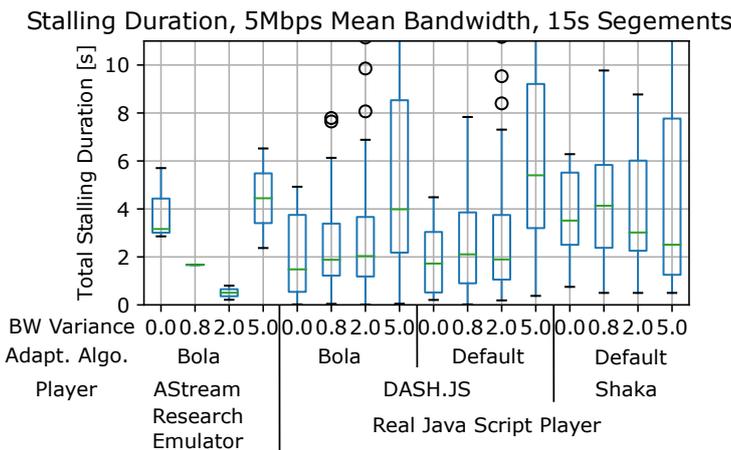


Figure 8.2: Total stalling duration DASH players with various adaptation algorithms and available bandwidth volatilities for a segment length configuration of 15s (lower is better, experiment setup as described in Section 8.2).

Figure 8.3 further investigates the impact on an aggregated stalling *Quality of Experience* (QoE) metric based on the stalling events and their duration [71]. Here, we note that the aggregated stalling QoE metric shows comparable results to the total stalling duration.

Overall, the comparison shows that the choice of the adaptation algorithm is dominated by the choice of the player and its configuration. This illustrative example substantiates the need for a large systematic comparison, as an isolated analysis of single DASH players or parameters is *not sufficient* to evaluate streaming performance.

*The player configuration is more important than the adaptation algorithm.*

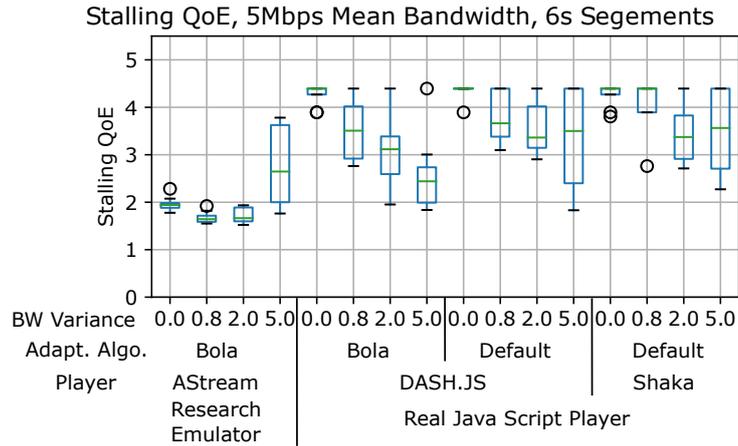


Figure 8.3: Stalling QoE by [71] of DASH players with various adaptation algorithms and available bandwidth volatilities for a segment length configurations of 6s (higher is better, experiment setup as described in Section 8.2).

Performance metric trade-offs...

... visualized as Pareto frontier...

... show that no single configuration dominates both target metrics...

The experienced quality for the user depends on multiple performance metrics, such as the stalling events and the video playback bitrate. Due to the intrinsic difficulty of constructing a single comprehensive quality metric, the choice of the optimal player and its configuration is a multidimensional optimization problem. Figure 8.4 shows a scatter plot to analyze the trade-offs between two exemplary performance metrics, *i*) the playback bitrate and *ii*) the previously introduced stalling QoE.<sup>6</sup> Each entry in the graph denotes the performance of a particular configuration averaged over all considered network conditions. As all player and adaptation algorithm combinations are represented at least once on the Pareto frontier, no single configuration dom-

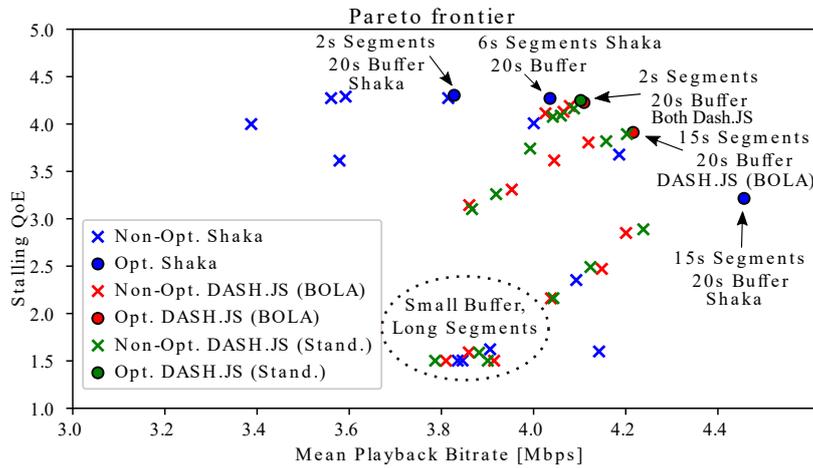


Figure 8.4: Trade-offs between playback bitrate and stalling QoE for different players and adaptation algorithms, aggregated over all environment conditions of Table 8.1. This visualization is automatically provided by *MACI* and only slightly modified for the presentation in this dissertation.

<sup>6</sup>We omit *AStream* due to inconsistent stalling results as discussed in [F6].

inates both target metrics. Thus, the choice of the player and adaptation algorithm only depends on the weighting of an aggregation function for both performance metrics.

A closer analysis shows multiple dependencies and trends: *i)* large buffer sizes dominate the performance for all player configurations for both considered metrics, *ii)* increasing the segment length leads to higher playback bitrates. *iii)* combinations of small buffer sizes and long segments perform badly. Finally, Figure 8.4 shows the minor impact of different adaptation algorithms within *DASH.JS*, i. e., the green and red marks collate without a consistently superior adaptation algorithm.

...and that large buffer sizes dominate the performance for all player.

#### 8.4 MACI PERSPECTIVE

To conduct the previous experimental design study, we relied on *MACI*, our framework for the seamless execution and analysis of extensive network experiments, as presented in Chapter 4. Thus, the DASH evaluation represents a case study and evaluation for *MACI*.

Figure 8.5 illustrates the relationship between *MACI* and the DASH specific aspects. *MACI* provides the reusable infrastructure to manage the experiments, execute them in parallel on a distributed infrastructure, and analyze the collected metrics. The DASH specific aspects are embedded in the emulation script for the execution of the DASH experiments. Based on the configuration of the emulation script provided by *MACI*, the experiment results in various performance metrics which are again collected by *MACI*. Figure 8.6 shows the automatically generated user interface for the interactive data analysis, which directly leads to visual representations as shown in Figure 8.1, 8.2, 8.3, 8.4. The visualizations are only slightly modified for the presentation in this dissertation, Figure A.1 shows an unmodified example for illustration.

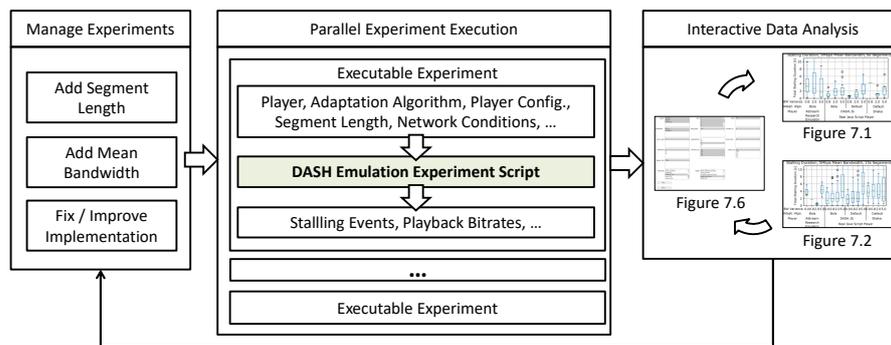


Figure 8.5: Experiment-driven research process for the DASH evaluation study.

**ITERATIVE RESEARCH PROCESS** We developed, tested, and improved the DASH specific measurement features iteratively. The interactive analysis of the experiment results enabled us *i)* to quickly detect errors and inconsistencies in our measurements and implementations and *ii)* to identify regions of interest and to add additional measurement metrics and configurations to

*Iterative Research Process*

further investigate and question our findings within the process. We profited from *MACI* for interactive analysis group sessions to discuss and question hypotheses. The simple repetition of experiment studies with improved and extended implementations was crucial for our efficiency.

### Scalable Execution

**SCALABLE EXECUTION** As a single execution of all configurations in all environments requires more than 40 hours (120s video playback per experiment), the parallel experiment execution significantly increased our iteration speed and enabled us to retrieve reliable results with dozens of repetitions.

Figure 8.6: Automatically generated graphical user-interface for the DASH analysis.

The experience report in this section oversimplifies the internal aspects of the DASH emulation script and setup, which are presented in [F6]. *MACI* provided support for all *non-DASH specific* tasks and enabled us to focus on the DASH specific aspects for the DASH player comparison.

## 8.5 DISCUSSION AND FUTURE WORK

In this chapter, we showed that the large design and evaluation space of DASH video streaming requires systematic extensive experiments. Our extensive evaluation study showed that the impact of the player and its configuration dominates the choice of the adaptation algorithm with regard to various target metrics. In particular, the buffer size dominates all other control parameters with regard to the experienced video playback bitrate and stalling behavior. We found *MACI* indispensable in all phases of the research and development process for this DASH study.

For future work, we envision the consideration and integration of additional DASH players and bitrate adaptation algorithms as well as the consideration of lower network layers [163]. We envision that this might be supported by additional *MACI* features, e. g., to automatically generate and evaluate promising configurations.

## DISCUSSION AND FUTURE WORK

This dissertation contributes *i)* three programming models for the domains of Multipath TCP (*ProgMP*), adaptive communication systems (*Fossa*), and topology adaptations in communication systems (*TARL*), *ii)* more than eight novel, deployable general purpose, preference-, and application-aware Multipath TCP schedulers, and *iii)* the reusable *MACI* framework for the seamless execution and analysis of extensive network experiments (Figure 9.1). Major parts of our contributions are publicly available.<sup>1</sup>

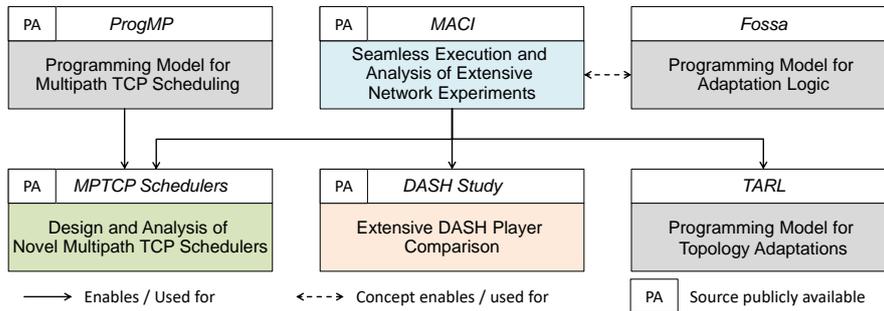


Figure 9.1: Overview of the contributions of this dissertation and their dependencies.

This dissertation overcomes the obstacles for the analysis, implementation, and evaluation of communication systems, i. e., *i)* the missing abstractions and the resulting implementation complexity, and *ii)* the required extensive evaluations for today’s large configuration spaces and heterogeneous network environments, as identified in Chapter 1. *ProgMP* and *TARL* are abstractions for the specification of concrete MPTCP schedulers and topology adaptation strategies, whereas *Fossa* provides abstractions for the specification of the adaptation decision. In addition to these abstractions for the analysis and implementation of communication systems, *MACI* increases the efficiency for the evaluation with extensive experiments (Figure 9.2). *MACI* proved its usefulness for the analysis and evaluation of our proposed *ProgMP* schedulers, the analysis of a distributed topology graph pattern matching protocol for *TARL*, and a systematic comparison of DASH video streaming implementations. These experiments go beyond an evaluation of *MACI* and significantly contribute to the understanding of their domains.

Overall, this dissertation provides various contributions in different domains. We find, however, that the contribution to the *understanding of Multipath TCP scheduling*, i. e., the *ProgMP* programming model, the wide range of novel schedulers, and the detailed experimental evaluation of these schedulers, is the most fundamental contribution of this dissertation.

<sup>1</sup>See <https://progmp.net> and <https://maci-research.net>.

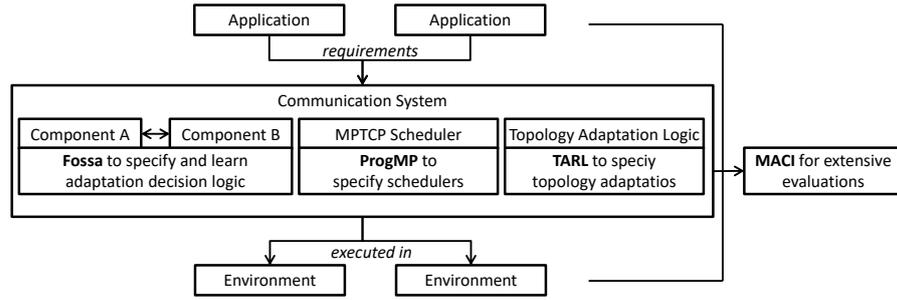


Figure 9.2: Illustration of the contributions in the context of today’s decoupled communication system components.

In the following, we compare our presented programming models, discuss the extensive network experiment approach, and conclude with an overview of identified future work.

### 9.1 PROGRAMMING MODELS FOR COMMUNICATION SYSTEMS

In this dissertation, we identified missing abstractions in communication system research and development. We presented three programming models for different target domains in the area of communication systems (Figure 9.3). First, the Multipath TCP scheduler programming model *ProgMP* enables to specify *when* to send *which* packet on *which* subflow. Second, the *Fossa* ECA- and utility function-based programming model for adaptation decisions enables to specify *when* to switch to *which* mechanism. Third, the topology adaptation rule language *TARL* enables to specify *when* and *how* to modify the surrounding topology.

<i>ProgMP</i>	What: <b>When</b> to send <b>which</b> packet on <b>which</b> subflow?
Programming Model for Multipath TCP Scheduling	Scope: Confined for MPTCP scheduling, small set of decision variables
	Declarative packet and subflow selection
	Runtime Environment: Linux kernel network stack
<i>Fossa</i>	What: <b>When to switch</b> to <b>which</b> mechanism?
Programming Model for Adaptation Decisions	Scope: Broad application domain, open and integrative model
	Declarative aggregations and conditions
	Runtime Environment: Java simulator
<i>TARL</i>	What: <b>When and how</b> to modify the topology?
Programming Model for Topology Adaptations	Scope: Confined for topology operations, open wrt. graph attributes
	Declarative pattern specification
	Runtime Environment: Java simulator

Figure 9.3: Comparison of the presented programming models.

*Programming models as abstractions*

The presented programming models and their languages provide bespoke abstractions for their domains. Thus, the concepts and primitives of the programming models differ due to their target domains. *Fossa*, for example, targets a broad application domain and is therefore designed to be open and in-

tegrative for different metrics, events, and possible mechanisms. In contrast, the Multipath TCP scheduling decision is a rather confined domain with a small set of possible decision variables.

We expressed a large number of notable programs and algorithms of their respective domains to confirm the expressiveness, efficiency, and understandability of the proposed programming models. All three programming models rely on declarative specifications, e. g., for the subflow selection in *ProgMP*, for aggregations and condition in *Fossa*, and topology patterns in *TARL*.

The domain-specific languages of all three presented programming models are executable. Here, we note that the significance of the *ProgMP* runtime environment goes beyond the two other environments, as it supports and is compatible with the de facto multipathing transport protocol MPTCP. Thus, *ProgMP* is directly usable by all applications that rely on TCP and run on top of our *ProgMP* Linux kernel runtime environment.

*Expressive ...*

*... and declarative specifications.*

*Executable in runtime environment*

## 9.2 EXTENSIVE NETWORK EXPERIMENTS

In this dissertation, we identified the need for the seamless execution and analysis of extensive network experiments. We presented *MACI* as a *generic, reusable* framework for large, extensive network experiments. We benefited from *MACI* during the analysis and evaluation of *ProgMP*-based MPTCP schedulers, the evaluation of the distributed topology pattern matching algorithm for *TARL*, and during an extensive *DASH* video streaming comparison.

*MACI* is inspired by our first experiences with the execution of a large number of evaluations during the development of the genetic programming learner of *Fossa*. The concept of *MACI*, the support for extensive experiments, enables the genetic programming learner of *Fossa*.

## 9.3 FUTURE WORK

While we discussed future work throughout this dissertation, we present an overview of the most promising future work in the following. Figure 9.4 illustrates the different areas of future work and their relationship to the contributions of this dissertation.

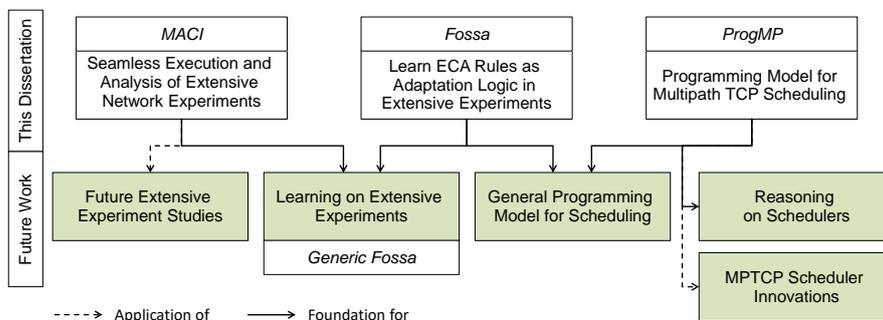


Figure 9.4: Illustration of the different areas of future work based on the contributions of this dissertation.

*Future extensive  
experiment studies*

**MACI** We made *MACI* publicly available to enable other researchers to benefit from *MACI* for their extensive experiment studies. While *MACI* provides a comprehensive support for an iterative research process, we envision helpful additional features, e. g., to increase the level of automation by automatically generating additional experiments for promising configurations.

*Generic Fossa*

**GENERIC FOSSA** Besides the use of the ECA- and the learning-based approach of *Fossa* for more applications, we envision a *Generic Fossa* for the emerging programming models in communication systems. The *Generic Fossa* might take the specification of a domain-specific language as input to generate optimized programs for the corresponding domain with *genetic programming* and extensive network experiments. The *Generic Fossa* would, for example, enable to learn optimized *TARL* rules based on the *TARL* grammar and the presented evaluation setups in *MACI*.

*A general  
programming model  
for packet scheduler*

**A GENERAL PROGRAMMING MODEL FOR SCHEDULING** *ProgMP* proved to be a powerful enabler for deployable Multipath TCP scheduler innovations. We showed that *ProgMP* enables a compact specification of a wide range of Multipath TCP schedulers. We envision an extension of *ProgMP* towards a general packet scheduler programming model, e. g., as illustrated in Listing 9.1. The inclusion of event-based expressions, e. g., as used in the ECA language of *Fossa*, might enable the specification of monitoring metrics beyond the confined Multipath TCP scheduling model and enable more fine-granular timing primitives. In combination with the support for the specification of packet queues, this would enable the specification of a broader range of packet schedulers but might increase the complexity for the specification of Multipath TCP schedulers.

```

1 /* Support for events, e.g., to manage metrics */
2 DECLARE smooth_rtt
3   ON receive_packet packet EVENT
4   WITH CONDITION packet.ACK = TRUE
5   ACTION SET smooth_rtt = smooth_rtt * 0.4 + packet.RTT * 0.6;
6
7 /* Support to define persistent queues */
8 DEFINE QUEUE reinjectionQueue;
9
10 reinjectionQueue.PUSH(QU.FILTER(skb =>
11   skb.SENT_TIME > CURRENT_TIME_MS + 2 * smooth_rtt));

```

Listing 9.1: Envisioned *ProgMP* extensions towards a general packet scheduler programming model.

*Reasoning  
on ProgMP  
schedulers...  
...with Generic  
Fossa...  
...and symbolic  
execution.*

**REASONING ON SCHEDULERS** We further envision advanced reasoning on *ProgMP* schedulers. The envisioned *Generic Fossa*, for example, might automatically derive and refine schedulers in extensive experiments with genetic programming. Further, we envision that the confined *ProgMP* programming model enables powerful verifications. Symbolic execution, for example, might efficiently verify correctness and liveness properties of schedulers. The

verification and analysis might help the scheduler developer and increase trust in learned, automatically derived schedulers.

**MPTCP SCHEDULER INNOVATIONS** The contributions of this work pave the way for future Multipath TCP scheduler innovations. This includes improved general purpose, application-, and preference-aware schedulers. We envision, for example, DASH-aware Multipath TCP schedulers that are optimized to provide a high quality of experience while preserving user preferences. Finally, we envision improved Multipath TCP schedulers based on large real-world evaluations.

*Future scheduler innovations*



## BIBLIOGRAPHY

---

- [F1] Alexander Frömmgen, Jens Heuschkel, and Boris Koldehofe. “Multipath TCP Scheduling for Thin Streams: Active Probing and One-way Delay-awareness”. In: *Proceedings of the IEEE International Conference on Communications (ICC)*. 2018.
- [F2] Alexander Frömmgen, Denny Stohr, Amr Rizk, and Boris Koldehofe. *Don’t Repeat Yourself: Seamless Execution and Analysis of Extensive Network Experiments*. Tech. rep. 2018. URL: <https://maci-research.net>.
- [F3] Michael Stein, Alexander Frömmgen, Roland Kluge, Wang Lin, Augustin Wilberg, Boris Koldehofe, and Max Mühlhäuser. “Scaling Topology Pattern Matching: A Distributed Approach”. In: *Proceedings of the ACM/SIGAPP Symposium on Applied Computing (SAC)*. 2018.
- [F4] Tobias Viernickel, Alexander Frömmgen, Amr Rizk, Boris Koldehofe, and Ralf Steinmetz. “Multipath QUIC: A Deployable Multipath Transport Protocol”. In: *Proceedings of the IEEE International Conference on Communications (ICC)*. 2018.
- [F5] Alexander Frömmgen, Amr Rizk, Tobias Erbschäuber, Max Weller, Boris Koldehofe, Alejandro Buchmann, and Ralf Steinmetz. “A Programming Model for Application-defined Multipath TCP Scheduling”. In: *Proceedings of the ACM/FIP/USENIX Middleware Conference, Best Paper Award*. ACM, 2017, pp. 134–146. URL: <https://progmp.net>.
- [F6] Denny Stohr, Alexander Frömmgen<sup>2</sup>, Amr Rizk, Michael Zink, Ralf Steinmetz, and Wolfgang Effelsberg. “Where are the Sweet Spots?: A Systematic Approach to Reproducible DASH Player Comparisons”. In: *Proceedings of the ACM Conference on Multimedia (MM)*. 2017, pp. 1113–1121. URL: <https://maci-research.net/dash>.
- [F7] Alexander Frömmgen, Tobias Erbschäuber, Torsten Zimmermann, Klaus Wehrle, and Alejandro Buchmann. “ReMP TCP: Low Latency Multipath TCP”. In: *Proceedings of the IEEE International Conference on Communications (ICC)*. Idea proposed in CoNEXT’15 Student Workshop. 2016.

---

<sup>2</sup>The two first authors contributed equally to this work.

- [F8] Michael Stein, Alexander Frömmgen, Roland Kluge, Frank Löffler, Andy Schürr, Alejandro Buchmann, and Max Mühlhäuser. “TARL: Modeling Topology Adaptations for Networking Applications”. In: *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM. 2016, pp. 57–63.
- [F9] Denny Stohr, Alexander Frömmgen, Jan Fornoff, Michael Zink, Alejandro Buchmann, and Wolfgang Effelsberg. “QoE Analysis of DASH Cross-Layer Dependencies by Extensive Network Emulation”. In: *Proceedings of the SIGCOMM Workshop on QoE-based Analysis and Management of Data Communication Networks (Internet-QoE)*. ACM, 2016, pp. 25–30.
- [F10] Alexander Frömmgen, Jens Heuschkel, Patrick Jahnke, Fabio Cuzzo, Immanuel Schweizer, Patrick Eugster, Max Mühlhäuser, and Alejandro Buchmann. “Crowdsourcing Measurements of Mobile Network Performance and Mobility During a Large Scale Event”. In: *Proceedings of the International Conference on Passive and Active Network Measurement (PAM)*. Springer International Publishing. 2016, pp. 70–82.
- [F11] Wasiur R. KhudaBukhsh, Amr Rizk, Alexander Frömmgen, and Heinz Koepl. “Optimizing Stochastic Scheduling in Fork-Join Queuing Models: Bounds and Applications”. In: *Proceedings of the IEEE INFOCOM*. 2017.
- [F12] Alexander Frömmgen, Robert Rehner, Max Lehn, and Alejandro Buchmann. “Fossa: Using Genetic Programming to Learn ECA Rules for Adaptive Networking Applications”. In: *Proceedings of the Local Computer Networks (LCN)*. IEEE. 2015, pp. 197–200.
- [F13] Alexander Frömmgen, Robert Rehner, Max Lehn, and Alejandro Buchmann. “Fossa: Learning ECA Rules for Adaptive Distributed Systems”. In: *Proceedings of the International Conference on Autonomic Computing (ICAC)*. IEEE. 2015, pp. 207–210.
- [F14] Alexander Frömmgen, Björn Richerzhagen, Julius Rückert, David Hausheer, Ralf Steinmetz, and Alejandro Buchmann. “Towards the Description and Execution of Transitions in Networked Systems”. In: *Proceedings of the IFIP International Conference on Autonomous Infrastructure, Management and Security (AIMS)*. Springer International Publishing. 2015, pp. 17–29.
- [F15] Alexander Frömmgen and Boris Koldehofe. “Demo: Programming Application-defined Multipath TCP Schedulers”. In: *Proceedings of the ACM/IFIP/USENIX Middleware Conference: Posters and Demos*. ACM, 2017, pp. 13–14.

- [F16] Alexander Frömmgen. *Mininet/Netem Emulation Pitfalls: A Multipath TCP Scheduling Experience*. Tech. rep. 2017. URL: <https://progmp.net/MininetPitfalls.pdf>.
- [F18] Alexander Frömmgen, Stefan Haas, Martin Pfannemüller, and Boris Koldehofe. “Switching ZooKeeper’s Consensus Protocol at Runtime”. In: *Proceedings of the International Conference on Autonomic Computing (ICAC) Poster Track*. 2017.
- [F19] Alexander Frömmgen, Denny Stohr, Jan Fornoff, Wolfgang Efelsberg, and Alejandro Buchmann. “Demo: Capture and Replay: Reproducible Network Experiments in Mininet”. In: *Proceedings of the Conference of the Special Interest Group on Data Communication (SIGCOMM)*. ACM. 2016, pp. 621–622.
- [F20] Jens Heuschkel, Alexander Frömmgen, Jon Crowcroft, and Max Mühlhäuser. “VirtualStack: Adaptive Multipath Support through Protocol Stack Virtualization”. In: *Proceedings of the International Network Conference (INC)*. Lulu.com. 2016, p. 73.
- [F21] Alexander Frömmgen, Sreeram Sadasivam, Sabrina Müller, Anja Klein, and Alejandro Buchmann. “Poster: Use Your Senses: A Smooth Multipath TCP WiFi/Mobile Handover”. In: *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM. 2015, pp. 248–250.
- [F22] Alexander Frömmgen, Patrick Wagner, and Alejandro Buchmann. “Simulation-based Retrieval of Adaptation Knowledge”. In: *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT) Student Workshop*. ACM. 2015.
- [F24] Alexander Frömmgen, Max Lehn, and Alejandro Buchmann. “A Property Description Framework for Composable Software”. In: *Proceedings of the European Conference on Software Architecture (ECSA)*. Springer International Publishing. 2014, pp. 267–282.
- [S4] Gregor Albrecht. *Eye-Tracker Supported Evaluation of a Domain-Specific Topology Adaptation Language*. Bachelor Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [S6] Andreas Bauer. *Eine Plattform zur Ausführung und Evaluation von Netzwerksimulationen mit vielen Konfigurationen*. Bachelor Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [S10] Nikolas Eller. *Maschinelles Lernen der Staukontrolle im QUIC Transportprotokoll*. Bachelor Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2018.

- [S11] Tobias Erbschäuer. *Optimierung der Latenz in Multipath-TCP Netzwerken durch Vervielfältigung und quantitative Verteilung der Datenpakete*. Bachelor Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2015.
- [S12] Tobias Erbschäuer. *Optimization of Custom Schedulers in Multipath TCP*. Master Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [S20] Tobias Viernickel. *Verbesserte Web-Performance mit Multipath Scheduling für HTTP/2 und QUIC*. Master Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [S22] Max Weller. *Optimierte Zusammenarbeit von HTTP/2 und Multipath-TCP-Schedulern*. Bachelor Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [RFC 826] David Plummer. *Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48. bit Ethernet address for transmission on Ethernet hardware*. RFC 826. Internet Engineering Task Force, 1982. URL: <https://tools.ietf.org/html/rfc826>.
- [RFC 896] John Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. Internet Engineering Task Force, 1984. URL: <http://www.ietf.org/rfc/rfc896.txt>.
- [RFC 1122] Robert Braden. *Requirements for Internet Hosts – Communication Layers*. RFC 1122. Internet Engineering Task Force, 1989. URL: <http://www.ietf.org/rfc/rfc1122.txt>.
- [RFC 1323] Van Jacobson, Bob Braden, and Dave Borman. *TCP Extensions for High Performance*. RFC 1323. Internet Engineering Task Force, 1992. URL: <https://tools.ietf.org/html/rfc1323>.
- [RFC 2616] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force, 1999. URL: <https://tools.ietf.org/html/rfc2616>.
- [RFC 4960] Randall Stewart. *Stream Control Transmission Protocol*. RFC 4960. Internet Engineering Task Force, 2007. URL: <http://www.ietf.org/rfc/rfc4960.txt>.
- [RFC 5944] Charles E. Perkins. *IP Mobility Support for IPv4, Revised*. RFC 5944. Internet Engineering Task Force, Nov. 2010. URL: <http://www.ietf.org/rfc/rfc5944.txt>.
- [RFC 6298] Vern Paxson, Mark Allman, Jerry Chu, and Matt Sargent. *Computing TCP’s Retransmission Timer*. RFC 6298. Internet Engineering Task Force, 2011. URL: <https://tools.ietf.org/html/rfc6298>.

- [RFC 6356] Costin Raiciu, Mark Handley, and Damon Wischik. *Coupled Congestion Control for Multipath Transport Protocols*. RFC 6356. Internet Engineering Task Force, 2011. URL: <http://www.ietf.org/rfc/rfc6356.txt>.
- [RFC 6824] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824. Internet Engineering Task Force, 2013. URL: <http://www.ietf.org/rfc/rfc6824.txt>.
- [RFC 6897] Michael Scharf and Alan Ford. *Multipath TCP (MPTCP) Application Interface Considerations*. RFC 6897. Internet Engineering Task Force, 2013. URL: <http://www.ietf.org/rfc/rfc6897.txt>.
- [RFC 7430] Marcelo Bagnulo, Christoph Paasch, Olivier Bonaventure, and Costin Raiciu. *Analysis of Residual Threats and Fernando Gont and Possible Fixes for Multipath TCP (MPTCP)*. RFC 7430. Internet Engineering Task Force, 2015. URL: <http://www.ietf.org/rfc/rfc7430.txt>.
- [1] Martin Abadi et al. “TensorFlow: A System for Large-scale Machine Learning”. In: *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 265–283.
- [2] Alexander Afanasyev, Ilya Moiseenko, Lixia Zhang, et al. “ndnSIM: NDN simulator for NS-3”. In: *University of California, Los Angeles, Tech. Rep* (2012).
- [3] Matteo Maria Andreozzi, Giovanni Stea, and Carlo Vallati. “A Framework for Large-scale Simulations and Output Result Analysis with NS-2”. In: *Proceedings of the International Conference on Simulation Tools and Techniques (Simutools)*. 2009, pp. 1–7.
- [4] Richard John Anthony. “A Policy-Definition Language and Prototype Implementation Library for Policy-based Autonomic Systems”. In: *Proceedings of the International Conference on Autonomic Computing (ICAC)*. IEEE. 2006, pp. 265–276.
- [5] Mina Tahmasbi Arashloo, Monia Ghobadi, Jennifer Rexford, and David Walker. “HotCocoa: Hardware Congestion Control Abstractions”. In: *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. 2017.
- [6] Behnaz Arzani, Alexander Gurney, Shuotian Cheng, Roch Guerin, and Boon Thau Loo. “Impact of Path Characteristics and Scheduling Policies on MPTCP Performance”. In: *IEEE International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 2014.

- [7] Tom Barbette, Cyril Soldani, and Laurent Mathy. “Fast Userspace Packet Processing”. In: *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. 2015, pp. 5–16.
- [8] Sebastien Barre, Christoph Paasch, and Olivier Bonaventure. “MultiPath TCP: From Theory to Practice”. In: *Proceedings of the IFIP International Conference on Networking Research (Networking)*. 2011.
- [9] Anindya Basu, Mark Hayden, Greg Morrisett, and Thorsten von Eicken. *A Language-Based Approach to Protocol Construction*. Cornell University.
- [10] Thais Batista, Ackbar Joolia, and Geoff Coulson. “Managing Dynamic Reconfiguration in Component-based Systems”. In: *Software Architecture*. Springer, 2005.
- [11] Matthias Becker, Markus Luckey, and Steffen Becker. “Model-driven Performance Engineering of Self-adaptive Systems: A Survey”. In: *ACM SIGSOFT Quality of Software Architectures (QoSA)*. 2012, pp. 117–122.
- [12] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. “Viatra 3: A Reactive Model Transformation Platform”. In: *Theory and Practice of Model Transformations*. Vol. 9152. Lecture Notes in Computer Science (LNCS). 2015, pp. 101–110.
- [13] Pat Bosshart et al. “P4: Programming Protocol-independent Packet Processors”. In: *SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.
- [14] Nevon Brake, James R. Cordy, Elizabeth Dancy, Marin Litoiu, and Valentina Popescu. “Automating Discovery of Software Tuning Parameters”. In: *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2008, pp. 65–72.
- [15] Alejandro Buchmann and Boris Koldehofe. “Complex Event Processing”. In: *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik* (2009), pp. 241–242.
- [16] Anton Burtsev, Nikhil Mishrikoti, Eric Eide, and Robert Ricci. “Weir: A Streaming Language for Performance Analysis”. In: *Proceedings of the Workshop on Programming Languages and Operating Systems (PLOS)*. 2013.
- [17] John Byers and Gabriel Nasser. “Utility-based decision-making in wireless sensor networks”. In: *Annual Workshop on Mobile and Ad Hoc Networking and Computing (MobiHOC)*. 2000, pp. 143–144.

- [18] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkupati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. “Packetdrill: Scriptable Network Stack Testing, from Sockets to Packets”. In: *Presented as part of the USENIX Annual Technical Conference (USENIX ATC)*. 2013.
- [19] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. “BBR: Congestion-Based Congestion Control”. In: *ACM Queue* 14. 2016.
- [20] Gustavo Carneiro, Pedro Fortuna, and Manuel Ricardo. “Flow-Monitor: A Network Monitoring Framework for the Network Simulator 3 (NS-3)”. In: *Proceedings of the International ICST Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*. 2009.
- [21] Min-Cheng Chan, Chien Chen, Jun-Xian Huang, Ted Kuo, Li-Hsing Yen, and Chien-Chao Tseng. “OpenNet: A Simulator for Software-Defined Wireless Local Area Network”. In: *Wireless Communications and Networking Conference (WCNC)*. IEEE. 2014, pp. 3332–3336.
- [22] Guo Chen, Yuanwei Lu, Yuan Meng, Bojie Li, Kun Tan, Dan Pei, Peng Cheng, Layong Larry Luo, Yongqiang Xiong, Xiaoliang Wang, et al. “Fast and Cautious: Leveraging Multi-path Diversity for Transport Loss Recovery in Data Centers”. In: *Presented as part of the USENIX Annual Technical Conference (USENIX ATC)*. 2016.
- [23] Yung-Chih Chen, Yeon-sup Lim, Richard J Gibbens, Erich M Nahum, Ramin Khalili, and Don Towsley. “A Measurement-Based Study of Multipath TCP Performance over Wireless Networks”. In: *Proceedings of the International Conference on Internet Measurements (IMC)*. 2013.
- [24] Yung-Chih Chen and Don Towsley. “On bufferbloat and delay analysis of multipath TCP in wireless networks”. In: *Proceedings of the IFIP International Conference on Networking Research (Networking)*. 2014.
- [25] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, et al. “Software Engineering for Self-Adaptive Systems: A Research Roadmap”. In: *Software Engineering for Self-Adaptive Systems*. Vol. 5525. Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg, 2009, pp. 1–26.
- [26] Edgar Codd, Sharon Codd, and Clynch Salley. *Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate*. 1993.

- [27] Melvin E. Conway. “Design of a Separable Transition-diagram Compiler”. In: *Communications of the ACM* 6.7 (1963), pp. 396–408.
- [28] Jonathan Corbet. *Extending extended BPF*. <https://lwn.net/Articles/603983/>. 2014.
- [29] Xavier Corbillon, Ramon Aparicio-Pardo, Nicolas Kuhn, Geraldine Texier, and Gwendal Simon. “Cross-layer Scheduler for Video Streaming over MPTCP”. In: *Proceedings of the International Conference on Multimedia Systems (MMSys)*. 2016.
- [30] Matthieu Coudron and Stefano Secci. “An Implementation of Multipath TCP in ns3”. In: *The International Journal of Computer and Telecommunications Networking* (2017), pp. 1–11.
- [31] Matthieu Coudron, Stefano Secci, and Guy Pujolle. “Differentiated Pacing on Multiple Paths to Improve One-way Delay Estimations”. In: *Proceedings of the IFIP International Conference on Networking Research (Networking)*. 2015.
- [32] Quentin De Coninck, Matthieu Baerts, Benjamin Hesmans, and Olivier Bonaventure. “A First Analysis of Multipath TCP on Smartphones”. In: *Proceedings of the International Conference on Passive and Active Network Measurement (PAM)*. 2016.
- [33] Quentin De Coninck and Olivier Bonaventure. *Every Millisecond Counts: Tuning Multipath TCP for Interactive Applications on Smartphones*. Tech. rep. 2017.
- [34] Rogerio De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. “Software Engineering for Self-Adaptive Systems: A Second Research Roadmap”. In: *Software Engineering for Self-Adaptive Systems II*. Lecture Notes in Computer Science (LNCS). Springer, 2013, pp. 1–32.
- [35] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. “WiFi, LTE, or both?: Measuring Multi-homed Wireless Internet Performance”. In: *Proceedings of the International Conference on Internet Measurements (IMC)*. 2014.
- [36] Edsger W. Dijkstra. “On the Role of Scientific Thought”. In: *Selected Writings on Computing: A Personal Perspective*. Springer, 1982, pp. 60–66.
- [37] Thomas Dreibholz, Robin Seggelmann, and Martin Becke. *Sender Queue Info Option for the SCTP Socket API*. Internet-Draft. Internet Engineering Task Force, 2013.

- [38] Dmitry Duplyakin, Jed Brown, and Robert Ricci. “Active Learning in Performance Analysis”. In: *Proceedings of the IEEE Cluster Conference*. 2016.
- [39] *Emulab*. <http://www.emulab.net/>.
- [40] Pascal Felber, Benoit Garbinato, and Rachid Guerraoui. *Towards Reliable CORBA: Integration vs. Service Approach*. Tech. rep. 1997.
- [41] Simone Ferlin, Oezgue Alay, Olivier Mehani, and Rokšana Boreli. “BLEST: Blocking Estimation-based MPTCP Scheduler for Heterogeneous Networks”. In: *Proceedings of the IFIP International Conference on Networking Research (Networking)*. 2016.
- [42] Simone Ferlin-Oliveira, Thomas Dreiholz, and Ozgu Alay. “Tackling the Challenge of Bufferbloat in Multi-Path Transport over Heterogeneous Wireless Networks”. In: *IEEE International Symposium of Quality of Service (IWQoS)*. 2014, pp. 123–128.
- [43] Tobias Flach, Nandita Dukkupati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. “Reducing Web Latency: The Virtue of Gentle Aggression”. In: *SIGCOMM Computer Communication Review* 43.4 (2013), pp. 159–170.
- [44] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. “Modeling and Validating Dynamic Adaptation”. In: *Models in Software Engineering*. Ed. by Michel R.V. Chaudron. Vol. 5421. Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg, 2009, pp. 97–108.
- [45] Franck Fleurey and Arnor Solberg. “A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems”. In: *Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 606–621.
- [46] Foster, Ian. *Designing and Building Parallel Programs*. Addison–Wesley, 1995. ISBN: 9780201575941.
- [47] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. “Frenetic: A Network Programming Language”. In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2011.

- [48] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. “Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure”. In: *Proceedings of the International Conference on Autonomic Computing (ICAC)*. 2004.
- [49] Rubino Geiß and Moritz Kroll. “GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool”. In: *Applications of Graph Transformations with Industrial Relevance*. Vol. 5088. Lecture Notes in Computer Science (LNCS). 2008, pp. 568–569.
- [50] Jim Gettys and Kathleen Nichols. “Bufferbloat: Dark Buffers in the Internet”. In: *Queue* (2011).
- [51] Yunmin Go, Oh Chan Kwon, and Hwangjun Song. “An Energy-Efficient HTTP Adaptive Video Streaming With Networking Cost Constraint Over Heterogeneous Wireless Networks”. In: *IEEE Transactions on Multimedia* 17.9 (2015), pp. 1646–1657.
- [52] Prateesh Goyal, Mohammad Alizadeh, and Hari Balakrishnan. “Rethinking Congestion Control for Cellular Networks”. In: *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. 2017, pp. 115–121.
- [53] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals”. In: *Data Mining and Knowledge Discovery* (1997), pp. 29–53.
- [54] Ilya Grigorik. “Making the Web Faster with HTTP 2.0”. In: *Queue* 11.10 (2013), p. 40.
- [55] Carsten Griwodz and Pål Halvorsen. “The Fun of using TCP for an MMORPG”. In: *Workshop on Network and Operating Systems support for Digital Audio and Video*. ACM. 2006.
- [56] Yihua Ethan Guo, Ashkan Nikraves, Z. Morley Mao, Feng Qian, and Subhabrata Sen. “Accelerating Multipath Transport Through Balanced Subflow Completion”. In: *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM. 2017, pp. 141–153.
- [57] Yihua Ethan Guo, Ashkan Nikraves, Z. Morley Mao, Feng Qian, and Subhabrata Sen. “Demo: DEMS: DEcoupled Multipath Scheduler for Accelerating Multipath Transport”. In: *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2017, pp. 477–479.

- [58] Nikola Gvozdiev, Brad Karp, and Mark Handley. “FUBAR: Flow Utility Based Routing”. In: *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. 2014.
- [59] Andrew Hallagan, Bryan Ward, and L. Felipe Perrone. “An Experiment Automation Framework for NS-3”. In: *Proceedings of the International ICST Conference on Simulation Tools and Techniques (Simutools)*. 2010, 38:1–38:2.
- [60] Ryan Hamilton, Janardhan Iyengar, Ian Swett, and Alyssa Wilk. *QUIC: A UDP-based secure and reliable transport for HTTP/2*. Internet-Draft. Internet Engineering Task Force, 2016.
- [61] Bo Han, Feng Qian, Shuai Hao, Lusheng Ji, and NJ Bedminster. “An Anatomy of Mobile Web Performance over Multipath TCP”. In: *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM. 2015.
- [62] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. “MP-DASH: Adaptive Video Streaming Over Preference-Aware Multipath”. In: *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM. 2016.
- [63] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. “Reproducible Network Experiments Using Container-based Emulation”. In: *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM. 2012, pp. 253–264.
- [64] Michael Hansen, Robert L Goldstone, and Andrew Lumsdaine. “What makes code hard to understand?” In: *arXiv preprint arXiv:1304.5257* (2013).
- [65] Mark Harman. “The Current State and Future of Search Based Software Engineering”. In: *Proceedings of the Conference on the Future of Software Engineering (FOSE)*. IEEE. 2007, pp. 342–357.
- [66] Mark Harman, Yue Jia, William B. Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. “Genetic Improvement for Adaptive Software Engineering (Keynote)”. In: *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2014, pp. 1–4.
- [67] Benjamin Hesmans and Olivier Bonaventure. “An Enhanced Socket API for Multipath TCP”. In: *IRTF Applied Networking Research Workshop*. 2016.

- [68] Benjamin Hesmans, Gregory Detal, Raphael Bauduin, Olivier Bonaventure, et al. “SMAPP: Towards Smart Multipath TCP-enabled APplications”. In: *Proceedings of the International Conference on emerging Networking Experiments and Technologies (CoNEXT)*. ACM. 2015.
- [69] Mario Hock, Roland Bless, and Martina Zitterbart. “Experimental Evaluation of BBR Congestion Control”. In: *Proceedings of the International Conference on Network Protocols (ICNP)*. IEEE. 2017.
- [70] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwalder, and Boris Koldehofe. “Mobile Fog: A Programming Model for Large-Scale Applications on the Internet of Things”. In: *Proceedings of the ACM SIGCOMM Workshop on Mobile Cloud Computing*. ACM. 2013, pp. 15–20.
- [71] Tobias Hoffeld, Raimund Schatz, Ernst Biersack, and Louis Plissonneau. “Internet Video Delivery in Youtube: From Traffic Measurements to Quality of Experience”. In: *Lecture Notes in Computer Science (LNCS) 7754 (2013)*, pp. 264–301.
- [72] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. “Explicit Path Control in Commodity Data Centers: Design and Applications”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015.
- [73] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. “A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service”. In: *Proceedings of the ACM SIGCOMM*. 2014, pp. 187–198.
- [74] Markus C. Huebscher and Julie A. McCann. “A Survey of Autonomic Computing-degrees, Models, and Applications”. In: *ACM Computing Surveys (CSUR) 40.3 (2008)*, p. 7.
- [75] J. Hwang, A. Walid, and J. Yoo. “Fast Coupled Retransmission for Multipath TCP in Data Center Networks”. In: *IEEE Systems Journal (2016)*.
- [76] Janardhan R. Iyengar, Paul D. Amer, and Randall Stewart. “Concurrent Multipath Transfer using SCTP Multihoming over Independent End-to-End Paths”. In: *IEEE/ACM Transactions on Networking 14.5 (2006)*, pp. 951–964.
- [77] Xiaomin Jin, Yuanan Liu, Wenhao Fan, Fan Wu, and Hongguang Zhang. “A Throughput Improved Path Selection Method Based on Throughput Prediction Model and Available Bandwidth for MPTCP”. In: *International Journal of Future Generation Communication and Networking 8.2 (2015)*, pp. 105–114.

- [78] Gueyoung Jung, K.R. Joshi, M.A. Hiltunen, R.D. Schlichting, and C. Pu. “Generating Adaptation Policies for Multi-tier Applications in Consolidated Server Environments”. In: *Proceedings of the International Conference on Autonomic Computing (ICAC)*. 2008, pp. 23–32.
- [79] Arash Kakhki, Samuel Jero, David Choffnes, Christina Nita-Rotaru, and Alan Mislove. “Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols”. In: *Proceedings of the International Conference on Internet Measurements (IMC)*. 2017.
- [80] Phil Karn and Craig Partridge. “Improving Round-trip Time Estimates in Reliable Transport Protocols”. In: *ACM SIGCOMM Computer Communication Review* (1987), pp. 2–7.
- [81] Brad Karp and H. T. Kung. “GPSR: Greedy Perimeter Stateless Routing for Wireless Networks”. In: *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. 2000, pp. 243–254.
- [82] Eric P. Kasten, Philip K. McKinley, Seyed Masoud Sadjadi, and R. E. Kurt Stirewalt. “Separating Introspection and Intercession to Support Metamorphic Distributed Systems”. In: *Distributed Computing Systems Workshops*. 2002.
- [83] Jeffrey O. Kephart and D. M. Chess. “The Vision of Autonomic Computing”. In: *Computer* (2003), pp. 41–50.
- [84] Jeffrey O. Kephart and Rajarshi Das. “Achieving Self-Management via Utility Functions”. In: *IEEE Internet Computing* 11.1 (2007), pp. 40–48.
- [85] Ramin Khalili, Nicolas Gast, Miroslav Popovic, Utkarsh Upadhyay, and Jean-Yves Le Boudec. “MPTCP is not Pareto-optimal: Performance Issues and a Possible Solution”. In: *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM. 2012.
- [86] Wasiur R. KhudaBukhsh, Bastian Alt, Sounak Kar, Amr Rizk, and Heinz Koepl. “Collaborative Uploading in Heterogeneous Networks: Optimal and Adaptive Strategies”. In: *Proceedings of the IEEE INFOCOM*. 2018.
- [87] Hyungjik Kim and Sunwoong Choi. *Data Path Selection for Multipath TCP Considering RTT*. Tech. rep. 2016.
- [88] Hyungjik Kim and Sunwoong Choi. “The effect of routing path buffer size on throughput of multipath TCP”. In: *Proceedings of the International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2016.

- [89] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. “NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors”. In: *Proceedings of the European Conference on Computer Systems*. ACM. 2015, p. 22.
- [90] James C. King. “Symbolic Execution and Program Testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.
- [91] Keith Kirkpatrick. “Software-defined Networking”. In: *Communications of the ACM* 56.9 (2013), pp. 16–19.
- [92] Donald E. Knuth. “Semantics of Context-free Languages: Correction”. In: *Theory of Computing Systems* 5.2 (1971), pp. 95–96.
- [93] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. “A Readable TCP in the Prolac Protocol Language”. In: *Proceedings of the ACM SIGCOMM*. 1999.
- [94] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. “The Click Modular Router”. In: *ACM Transactions on Computer Systems* 18.3 (2000), pp. 263–297.
- [95] Lachezar Krumov, Immanuel Schweizer, Dirk Bradler, and Thorsten Strufe. “Leveraging Network Motifs for the Adaptation of Structured Peer-to-Peer-Networks”. In: *Proceedings of the IEEE GLOBECOM*. 2010, pp. 1–5.
- [96] Jonathan Kua, Grenville Armitage, and Philip Branch. “A Survey of Rate Adaptation Techniques for Dynamic Adaptive Streaming Over HTTP”. In: *IEEE Communications Surveys and Tutorials* 19.3 (2017), pp. 1842–1866.
- [97] M. Kühlewind, S. Neuner, and B Trammell. “On the State of ECN and TCP Options on the Internet.” In: *Proceedings of the International Conference on Passive and Active Network Measurement (PAM)*. 2013, pp. 135–144.
- [98] Nicolas Kuhn, Emmanuel Lochin, Ahlem Mifdaoui, Golam Sarwar, Olivier Mehani, and Roksana Boreli. “DAPS: Intelligent Delay-aware Packet scheduling for Multipath Transport”. In: *Proceedings of the IEEE International Conference on Communications (ICC)*. 2014.
- [99] Markus Laner, Philipp Svoboda, Peter Romirer-Maierhofer, Navid Nikaein, Fabio Ricciato, and Markus Rupp. “A Comparison between One-way Delays in Operating HSPA and LTE Networks”. In: *International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt)*. 2012, pp. 286–292.

- [100] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. “The QUIC Transport Protocol: Design and Internet-Scale Deployment”. In: *Proceedings of the ACM SIGCOMM*. ACM. 2017, pp. 183–196.
- [101] Rafael Laufer, Massimo Gallo, Diego Perino, and Anandathirtha Nandugudi. “Climb: Enabling Network Function Composition with Click Middleboxes”. In: *ACM SIGCOMM Computer Communication Review* 46.4 (2016), pp. 17–22.
- [102] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. “Developing eMoflon with eMoflon”. In: *Theory and Practice of Model Transformations*. Vol. 8568. Lecture Notes in Computer Science (LNCS). 2014, pp. 138–145.
- [103] Stefan Lederer, Christopher Müller, and Christian Timmerer. “Dynamic Adaptive Streaming over HTTP Dataset”. In: *Proceedings of the International Conference on Multimedia Systems (MMSys)*. ACM, 2012, p. 89.
- [104] Max Lehn. “InterestCast: Adaptive Event Dissemination for Interactive Real-Time Applications”. PhD thesis. Technische Universität Darmstadt, 2016.
- [105] Max Lehn, Robert Rehner, and Alejandro Buchmann. “Distributed Optimization of Event Dissemination Exploiting Interest Clustering”. In: *Proceedings of the IEEE Conference on Local Computer Networks (LCN)*. 2013, pp. 328–331.
- [106] Christof Leng. “BubbleStorm: Replication, Updates, and Consistency in Rendezvous Information Systems”. PhD thesis. Technische Universität Darmstadt, 2012.
- [107] Ning Li, Jennifer C. Hou, and Lui Sha. “Design and Analysis of an MST-Based Topology Control Algorithm”. In: *IEEE Transactions on Wireless Communications* 4.3 (2005), pp. 1195–1206.
- [108] Qingxi Li, Mo Dong, and Brighten Godfrey. “Halfback: Running Short Flows Quickly and Safely”. In: *Proceedings of the International Conference on emerging Networking Experiments and Technologies (CoNEXT)*. ACM. 2015.
- [109] Yeon-sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. “ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths”. In: *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 2017, pp. 33–34.

- [110] Yeon-sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. “ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths”. In: *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM. 2017.
- [111] Shuhao Liu, Hong Xu, Libin Liu, Wei Bai, Kai Chen, and Zhiping Cai. “RepNet: Cutting Latency with Flow Replication in Data Center Networks”. In: *IEEE Transactions on Services Computing* (2018).
- [112] Xiaomei Liu, Li Xiao, Andrew Kreling, and Yunhao Liu. “Optimizing Overlay Topology by Reducing Cut Vertices”. In: *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSS-DAV)*. ACM. 2006.
- [113] Igor Lopez, Marina Aguado, Christian Pinedo, and Eduardo Jacob. “SCADA Systems in the Railway Domain: Enhancing Reliability through Redundant MultipathTCP”. In: *Proceedings of the International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2015.
- [114] Markus Luckey, Benjamin Nagel, Christian Gerth, and Gregor Engels. “Adapt Cases: Extending Use Cases for Adaptive Systems”. In: *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2011, pp. 30–39.
- [115] Manisha Luthra, Boris Koldehofe, Pascal Weisenburger, Guido Salvaneschi, and Raheel Arif. “TCEP: Adapting to Dynamic User Environments by Enabling Transitions between Operator Placement Mechanisms”. In: *Proceedings of the ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2018.
- [116] Alessandro Margara, Gianpaolo Cugola, and Giordano Tamburrelli. “Learning from the Past: Automated Rule Generation for Complex Event Processing”. In: *Proceedings of the ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2014, pp. 47–58.
- [117] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. “ClickOS and the Art of Network Function Virtualization”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014, pp. 459–473.
- [118] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. “OpenFlow: Enabling Innovation in Campus

- Networks”. In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.
- [119] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. “Composing Adaptive Software”. In: *Computer* 37.7 (2004), pp. 56–64.
- [120] Patrick McManus. *HTTP/2 is Live in Firefox*. <https://bitsup.blogspot.de/2015/02/http2-is-live-in-firefox.html>. 2015.
- [121] MozillaZine. *Network.http.max-persistent-connections-per-server*. URL: <http://kb.mozillazine.org/Network.http.max-persistent-connections-per-server>.
- [122] Akshay Narayan, Frank J. Cangialosi, Prateesh Goyal, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. “The Case for Moving Congestion Control Out of the Datapath”. In: *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. 2017.
- [123] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. “Mahimahi: Accurate Record-and-Replay for HTTP”. In: *Presented as part of the USENIX Annual Technical Conference (USENIX ATC)*. 2015, pp. 417–429.
- [124] *Nghhttp2: HTTP/2 C Library*. <https://nghhttp2.org/>. 2015.
- [125] Ashkan Nikraves, Yihua Guo, Feng Qian, Z. Morley Mao, and Subhabrata Sen. “An In-depth Understanding of Multipath TCP on Mobile Devices: Measurement and System Design”. In: *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2016.
- [126] *ns-3*. <http://www.nsnam.org/>.
- [127] Bong-Hwan Oh and Jaiyong Lee. “Constraint-based Proactive Scheduling for MPTCP in Wireless Networks”. In: *Computer Networks* 91 (2015), pp. 548–563.
- [128] *OMNet++*. <http://www.omnetpp.org/>.
- [129] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. “Cross-level Sensor Network Simulation with Cooja”. In: *Proceedings of the IEEE Conference on Local Computer Networks (LCN)*. 2006, pp. 641–648.
- [130] Shih-Hao Ou, Chih-Wei Huang, Tzu-Kuan Lee, and Chih-Yang Huang. “Out-of-order Transmission enabled Congestion and Scheduling Control for Multipath TCP”. In: *Proceedings of the International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, 2016.

- [131] Christoph Paasch and Sebastien Barre. *Multipath TCP in the Linux Kernel*. Available from <http://www.multipath-tcp.org>.
- [132] Christoph Paasch, Gregory Detal, Fabien Duchene, Costin Raiciu, and Olivier Bonaventure. “Exploring Mobile/WiFi Handover with Multipath TCP”. In: *Proceedings of the ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design*. 2012.
- [133] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. “Experimental Evaluation of Multipath TCP Schedulers”. In: *Proceedings of the ACM SIGCOMM Workshop on Capacity Sharing*. 2014.
- [134] Christoph Paasch, Ramin Khalili, and Olivier Bonaventure. “On the Benefits of Applying Experimental Design to Improve Multipath TCP”. In: *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM. 2013, pp. 393–398.
- [135] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. “Modeling TCP Throughput: A Simple Model and its Empirical Validation”. In: *ACM SIGCOMM Computer Communication Review* 28.4 (1998), pp. 303–314.
- [136] Abhinav Pathak, Himabindu Pucha, Ying Zhang, Y. Charlie Hu, and Z. Morley Mao. “A Measurement Study of Internet Delay Asymmetry”. In: *Proceedings of the International Conference on Passive and Active Network Measurement (PAM)*. Springer-Verlag, 2008, pp. 182–191.
- [137] Amir H. Payberah, Jim Dowling, Fatemeh Rahimain, and Seif Haridi. “Distributed Optimization of P2P Live Streaming Overlays”. In: *Computing* 94.8-10 (2012), pp. 621–647.
- [138] L. Felipe Perrone, Thomas R. Henderson, Mitchell J. Watrous, and Vinicius Daly Felizardo. “The Design of an Output Data Collection Framework for NS-3”. In: *Proceedings of the IEEE Winter Simulation Conference (WSC)*. 2013, pp. 2984–2995.
- [139] L. Felipe Perrone, Christopher J. Kenna, and Bryan C. Ward. “Enhancing the Credibility of Wireless Network Simulations with Experiment Automation”. In: *Proceedings of the IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*. 2008, pp. 631–637.
- [140] L. Felipe Perrone, Christopher S. Main, and Bryan C. Ward. “Safe: Simulation Automation Framework for Experiments”. In: *Proceedings of the Winter Simulation Conference (WSC)*. 2012.
- [141] Ben Pfaff. *P4 and Open vSwitch*. <http://p4.org/p4/p4-and-open-vswitch>.

- [142] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. “The Design and Implementation of Open vSwitch”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015.
- [143] PlanetLab. <https://www.planet-lab.org/>.
- [144] Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza. *A Field Guide to Genetic Programming*. Lulu.com, 2008.
- [145] Matei Popovici and Costin Raiciu. “Exploiting Multipath Congestion Control for Fun and Profit”. In: *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. 2016.
- [146] Junaid Qadir, Anwaar Ali, Kok-Lim Alvin Yau, Arjna Sathiseelan, and Jon Crowcroft. “Exploiting the Power of Multiplicity: A Holistic Survey of Network-Layer Multipath”. In: *IEEE Communications Surveys and Tutorials* 17.4 (2015), pp. 2176–2213.
- [147] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. “Improving datacenter performance and robustness with Multipath TCP”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. 2011, pp. 266–277.
- [148] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. “How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012.
- [149] Mohammad Rajiullah, Per Hurtig, Anna Brunstrom, Andreas Petlund, and Michael Welzl. “An Evaluation of Tail Loss Recovery Mechanisms for TCP”. In: *ACM SIGCOMM Computer Communication Review* 45.1 (2015), pp. 5–11.
- [150] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. “HyperDrive: Exploring Hyperparameters with POP Scheduling”. In: *Proceedings of the ACM/IFIP/USENIX Middleware Conference*. 2017.
- [151] Björn Richerzhagen. “Mechanism Transitions in Publish/Subscribe Systems-Adaptive Event Brokering for Location-based Mobile Social Applications”. PhD thesis. Technische Universität Darmstadt, 2017.

- [152] Björn Richerzhagen, Stefan Wilk, Julius Rückert, Denny Stohr, and Wolfgang Effelsberg. “Transitions in Live Video Streaming Services”. In: *Proceedings of the Workshop on Design, Quality and Deployment of Adaptive Video Streaming (VideoNext)*. 2014, pp. 37–38.
- [153] George F. Riley and Thomas R. Henderson. “The ns-3 Network Simulator”. In: *Modeling and Tools for Network Simulation* (2010), pp. 15–34.
- [154] Tobias Rückelt. “Connecting Vehicles to the Internet-Strategic Data Transmission for Mobile Nodes using Heterogeneous Wireless Networks”. PhD thesis. Technische Universität Darmstadt, 2017.
- [155] Julius Rückert. “Large-scale Live Video Streaming Over the Internet-Efficient and Flexible Content Delivery Using Network and Application-Layer Mechanisms”. PhD thesis. 2016.
- [156] Julius Rückert, Björn Richerzhagen, Eduardo Lidanski, Ralf Steinmetz, and David Hausheer. “TopT: Supporting Flash Crowd Events in Hybrid Overlay-based Live Streaming”. In: *Proceedings of the IFIP International Conference on Networking Research (Networking)*. 2015.
- [157] Seyed Masoud Sadjadi and Philip K. McKinley. “ACT: An Adaptive CORBA Template to Support Unanticipated Adaptation”. In: *Distributed Computing Systems*. IEEE. 2004.
- [158] Seyed Masoud Sadjadi, Philip. K. McKinley, E. P. Kasten, and Z. Zhou. “MetaSockets: Design and Operation of Runtime Reconfigurable Communication Services: Experiences with Auto-adaptive and Reconfigurable Systems”. In: *Software Practice and Experience* 36.11-12 (2006).
- [159] Jacques Samain, Giovanna Carofiglio, Luca Muscariello, Michele Papalini, Mauro Sardara, Michele Tortelli, and Dario Rossi. “Dynamic Adaptive Video Streaming: Towards a Systematic Comparison of ICN and TCP/IP”. In: *IEEE Transactions on Multimedia* 19.10 (2017), pp. 2166–2181.
- [160] Golam Sarwar, Roksana Boreli, Emmanuel Lochin, Ahlem Mifdaoui, and Graeme Smith. “Mitigating Receiver’s Buffer Blocking by Delay Aware Packet Scheduling in Multipath Data Transfer”. In: *IEEE International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 2013.
- [161] Michael Schapira and Keith Winstein. “Congestion-Control Throwdown”. In: *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. 2017.

- [162] Michael Scharf and Sebastian Kiesel. “Head-of-line Blocking in TCP and SCTP: Analysis and Measurements”. In: *IEEE GLOBECOM*.
- [163] Matthias Schulz, Denny Stohr, Stefan Wilk, Benedikt Rudolph, Wolfgang Effelsberg, and Matthias Hollick. “APP and PHY in Harmony: A framework enabling flexible physical layer processing to address application requirements”. In: *Proceedings of the International Conference and Workshops on Networked Systems (NetSys)*. 2015, pp. 1–8.
- [164] Immanuel Schweizer, Michael Wagner, Dirk Bradler, Max Mühlhäuser, and Thorsten Strufe. “kTC - Robust and Adaptive Wireless Ad-hoc Topology Control”. In: *Proceedings of the International Conference on Computer Communications and Networks (ICCCN)*. 2012.
- [165] *Scripy 0.9.3: Python tools for manage system commands as replacement to bash script*. Python Software Foundation <https://pypi.python.org/pypi/Scripy>.
- [166] SungHoon Seo. *KT's GiGA LTE*. <https://www.ietf.org/proceedings/93/slides/slides-93-mptcp-3.pdf>. 2015.
- [167] Michael Seufert, Sebastian Egger, Martin Slanina, Thomas Zinner, Tobias Hoßfeld, and Phuoc Tran-Gia. “A Survey on Quality of Experience of HTTP Adaptive Streaming”. In: *IEEE Communications Surveys and Tutorials* (2015), pp. 469–492.
- [168] Bonita Sharif and Jonathan I. Maletic. “An Eye Tracking Study on camelCase and Under\_score Identifier Styles”. In: *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE. 2010, pp. 196–205.
- [169] Haiying Shen, Ze Li, and Jin Li. “A DHT-Aided Chunk-Driven Overlay for Scalable and Efficient Peer-to-Peer Live Streaming”. In: *IEEE Transactions on Parallel Distributed Systems* 24.11 (2013), pp. 2125–2137.
- [170] Scott Shenker. “Fundamental Design Issues for the Future Internet”. In: *IEEE Journal on Selected Areas in Communications* 13.7 (1995), pp. 1176–1188.
- [171] Tanya Shreedhar, Nitinder Mohan, Sanjit K Kaul, and Jussi Kangasharju. “More Than The Sum Of Its Parts: Exploiting Cross-Layer and Joint-Flow Information in MPTCP”. In: *arXiv preprint arXiv:1711.07565* (2017).
- [172] Hassan Sinky, Bechir Hamdaoui, and Mohsen Guizani. “Proactive Multi-Path TCP for Seamless Handoff in Heterogeneous Wireless Access Networks”. In: *IEEE Transactions on Wireless Communications* (2016).

- [173] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. “Packet Transactions: High-Level Programming for Line-Rate Switches”. In: *Proceedings of the ACM SIGCOMM*. 2016.
- [174] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. “Programmable Packet Scheduling at Line Rate”. In: *Proceedings of the ACM SIGCOMM*. 2016.
- [175] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. “An Experimental Study of the Learnability of Congestion Control”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. 2014, pp. 479–490.
- [176] Iraj Sodagar. In: *IEEE MultiMedia* 18.4 (2011), pp. 62–67.
- [177] Fei Song, Hongke Zhang, Sidong Zhang, Fernando Ramos, and Jon Crowcroft. *An Estimator of Forward and Backward Delay for Multipath Transport*. Tech. rep. 2009.
- [178] Kevin Spiteri, Rahul Uргаonkar, and Ramesh K. Sitaraman. “BOLA: Near-optimal bitrate adaptation for online videos”. In: *Proceedings of the IEEE INFOCOM*. 2016.
- [179] Vikram Srinivasan, Carla F. Chiasserini, Pavan Nuggehalli, and Ramesh R. Rao. “Optimal Rate Allocation and Traffic Splits for Energy Efficient Routing in Ad Hoc Networks”. In: *Proceedings of the IEEE INFOCOM*. 2002.
- [180] Michael Stein. “Local Algorithms for Distributed Topology Adaptation”. PhD thesis. Technische Universität Darmstadt, 2018.
- [181] Michael Stein, Mathias Fischer, Immanuel Schweizer, and Max Mühlhäuser. “A Classification of Locality in Network Research”. In: *ACM Computing Surveys* 50.4 (2017), 53:1–53:37.
- [182] Ralf Steinmetz. *Multimedia: Computing Communications & Applications*. Pearson Education India, 2012.
- [183] Dominik Stingl, Christian Gross, Julius Rückert, Leonhard Nobach, Aleksandra Kovacevic, and Ralf Steinmetz. “PeerfactSim.KOM: A Simulation Framework for Peer-to-Peer Systems”. In: *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*. 2011, pp. 577–584.
- [184] Stephen D. Strowes. “Passively Measuring TCP Round-trip Times”. In: *ACM Queue* (2013).

- [185] Weibin Sun and Robert Ricci. “Fast and Flexible: Parallel Packet Processing with GPUs and Click”. In: *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2013, pp. 25–35.
- [186] Hajime Tazaki, Emilio Mancini, Daniel Camara, Thierry Turletti, and Walid Dabbous. “Direct Code Execution: Realistic Protocol Simulation with Running Code.” In: *Demonstration of DCE at MSWIM*. 2013.
- [187] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. “Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments”. In: *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. ACM. 2013, pp. 217–228.
- [188] Viet-Hoang Tran, Quentin De Coninck, Benjamin Hesmans, Ramin Sadre, and Olivier Bonaventure. “Observing Real Multipath TCP Traffic”. In: *Computer Communications* (2016).
- [189] Omri Traub, Glenn Holloway, and Michael D. Smith. “Quality and Speed in Linear-scan Register Allocation”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1998.
- [190] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. “An Eye-tracking Study Assessing the Comprehension of C++ and Python Source Code”. In: *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM. 2014, pp. 231–234.
- [191] Arie Van Deursen, Paul Klint, and Joost Visser. “Domain-Specific Languages: An Annotated Bibliography”. In: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36.
- [192] Andras Varga and Rudolf Hornig. “An Overview of the OM-NeT++ Simulation Environment”. In: *Proceedings of the International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. 2008.
- [193] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. “Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast”. In: *Proceedings of the International Conference on Network Protocols (ICNP)*. IEEE. 2006, pp. 2–11.
- [194] Thomas Vogel and Holger Giese. “A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels”. In: *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM. 2012, pp. 129–138.

- [195] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. “Utility Functions in Autonomic Systems”. In: *Proceedings of the International Conference on Autonomic Computing*. IEEE, 2004, pp. 70–77.
- [196] Feng Wang, Yongqiang Xiong, and Jiangchuan Liu. “mTreebone: A Hybrid Tree/Mesh Overlay for Application-Layer Live Video Multicast”. In: *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2007.
- [197] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. “Searching for Better Configurations: A Rigorous Approach to Clone Evaluation”. In: *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 2013, pp. 455–465.
- [198] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. “Demystifying Page Load Performance with WProf”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2013.
- [199] Yu Wang. “Topology Control for Wireless Sensor Networks”. In: *Wireless Sensor Networks and Applications*. Signals and Communication Technology. Springer, 2008, pp. 113–147.
- [200] Philip Wette, Martin Draxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. “Maxinet: Distributed Emulation of Software-defined Networks”. In: *Proceedings of the IFIP International Conference on Networking Research (Networking)*. 2014.
- [201] Matthias Wichtlhuber, Björn Richerzhagen, Julius Rückert, and David Hausheer. “TRANSIT: Supporting Transitions in Peer-to-Peer Live Video Streaming”. In: *Proceedings of the IFIP International Conference on Networking Research (Networking)*. 2014, pp. 1–9.
- [202] Jörg Widmer, Robert Denda, and Martin Mauve. “A Survey on TCP-friendly Congestion Control”. In: *IEEE Network* 15.3 (2001), pp. 28–37.
- [203] Keith Winstein and Hari Balakrishnan. “TCP Ex Machina: Computer-Generated Congestion Control”. In: *SIGCOMM Computer Communication Review*. Vol. 43. 4. 2013, pp. 123–134.
- [204] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. “Design, Implementation and Evaluation of Congestion Control for Multipath TCP”. In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2011.

- [205] Jiyan Wu, Chau Yuen, Bo Cheng, Ming Wang, and Junliang Chen. “Streaming High-Quality Mobile Video with Multipath TCP in Heterogeneous Wireless Networks”. In: *IEEE Transactions on Mobile Computing* 15.9 (2015).
- [206] Hong Xu and Baochun Li. “RepFlow: Minimizing Flow Completion Times with Replicated Flows in Data Centers”. In: *Proceedings of the IEEE INFOCOM*. 2014.
- [207] Mingwei Xu. *Delay-based Congestion Control for MPTCP*. Internet Engineering Task Force, 2014.
- [208] Fan Yang, Qi Wang, and Paul D. Amer. “Out-of-order Transmission for In-order Arrival Scheduling for Multipath TCP”. In: *IEEE International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 2014.
- [209] Andrew Chi-Chih Yao. “On Constructing Minimum Spanning Trees in k-Dimensional Spaces and Related Problems”. In: *SIAM Journal on Computing* 11.4 (1982), pp. 721–736.
- [210] Anatoliy Zabrovskiy, Evgeny Kuzmin, Evgeny Petrov, Christian Timmerer, and Christopher Mueller. “AdViSE: Adaptive Video Streaming Evaluation Framework for the Automated Testing of Media Players”. In: *Proceedings of the International Conference on Multimedia Systems (MMSys)*. ACM, 2017, pp. 217–220.
- [211] Doron Zarchy, Radhika Mittal, and Scott Schapira Michael and; Shenker. “An Axiomatic Approach to Congestion Control”. In: *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. 2017.
- [212] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. “GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks”. In: *Proceedings of the IEEE Workshop on Parallel and Distributed Simulation*. 1998, pp. 154–161.



## APPENDICES

## A.1 ILLUSTRATING MACI VISUALIZATIONS

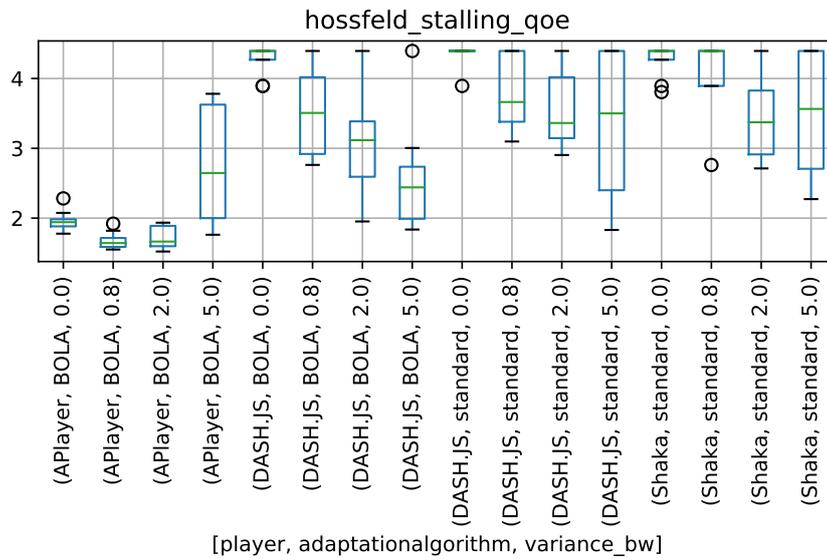


Figure A.1: Unmodified, automatically generated *MACI* visualization for the DASH evaluation (Figure 8.3). Stalling QoE by [71] of DASH players with various adaptation algorithms and available bandwidth volatilities for a segment length configurations of 6s (higher is better, experiment setup as described in Section 8.2).

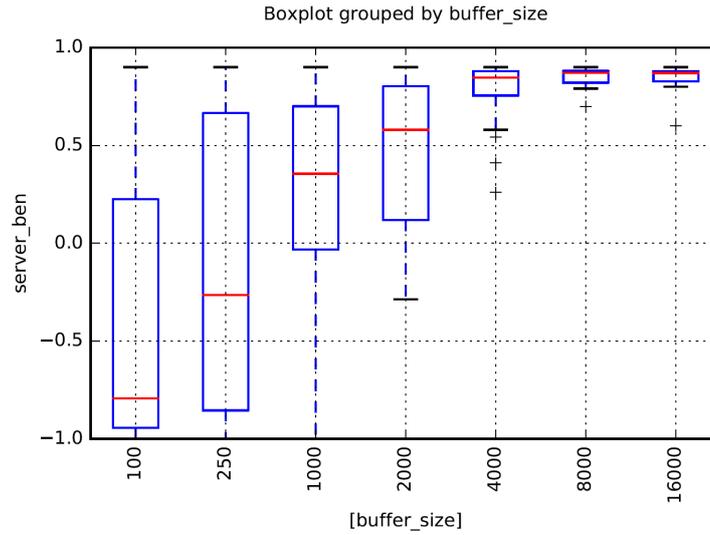


Figure A.2: Reproduction of the aggregation benefit evaluation of [134] by [S6] with *MACI*. Taken from [S6].

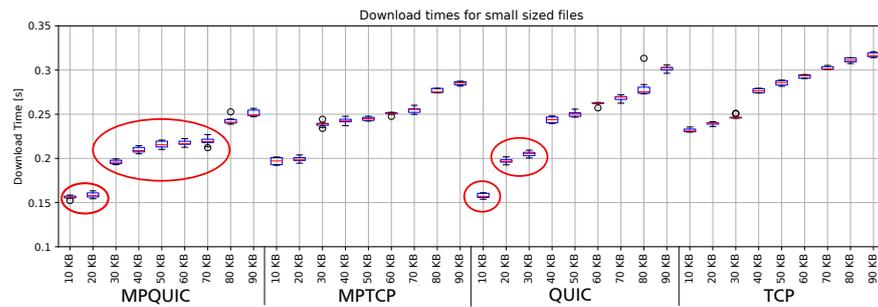


Figure A.3: Download time comparison of different transport protocols as illustration for the application of *MACI*. Taken from [S20].

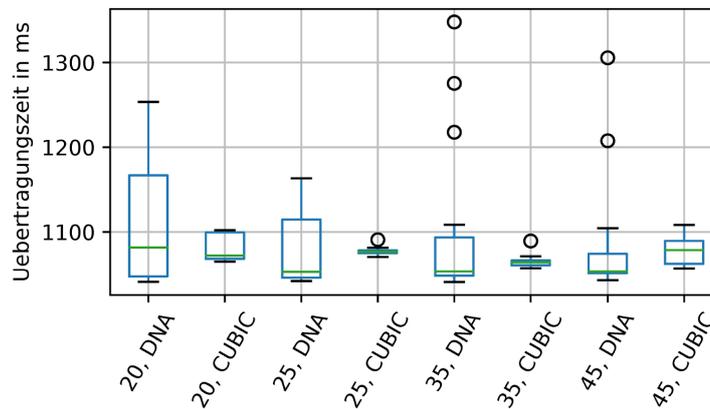


Figure A.4: Download time comparison of different transport protocols as illustration for the application of *MACI*. Taken from [S10].

## A.2 ADDITIONAL PROGMP EXPERIENCES

In this section, we discuss complementary experiences with the *ProgMP* scheduler specification language design.

A.2.1 *Was the Packet Sent on all Subflows?*

An early language draft provided the `SENT_ON_ALL_SUBFLOWS` packet operation. This operation appeared convenient, as the information is recurrently required during scheduling. When using the operation during our experiments, however, we found that this operation is misleading. As illustrating example, assume we want to remove a packet from the reinjection queue if it was sent on *all* subflows. During our experiments, we found that this scheduler stops working in case a subflow becomes lossy, as we only push packet on non-lossy subflows. We considered to change the operation to only check if it was sent on all currently available subflows. However, we found that in many situations, we are actually interested if the packet was sent on all available subflows that have congestion window left. Eventually, we decided to remove the packet operation `SENT_ON_ALL_SUBFLOWS`, as we found that there is no reason for a special operation as this operation does not summarize a recurring pattern. All flavours of this pattern can be easily expressed in the language, as shown in Listing A.1.

```

1 VAR skb = Q.TOP; /* Just an example packet */
2
3 VAR alternative1 = skb.SENT_ON_ALL_SUBFLOWS;
4
5 VAR alternative2 = SUBFLOWS.FILTER(skb => !skb.SENT_ON(skb)).EMPTY;
6
7 VAR alternative3 = SUBFLOWS.FILTER(skb => skb.LOSSY AND !skb.
    SENT_ON(skb)).EMPTY;
8
9 VAR alternative4 = SUBFLOWS.FILTER(skb => skb.LOSSY AND skb.CWND >
    skb.SKBS_IN_FLIGHT + skb.QUEUED AND !skb.SENT_ON(skb)).EMPTY;

```

Listing A.1: Patterns to check if a packet was sent on all subflows.

A.2.2 *Print Statements: Analysing the Scheduling Environment and Decisions*

*ProgMP* enables a convenient analysis of the dynamic scheduling environment and scheduling decisions. Interactive debugging of scheduling is infeasible due to the timing constraints and complexity of debugging Linux kernel code.<sup>1</sup> Thus, the developer usually has to rely on output for interference. *ProgMP* provides a `PRINT` statement for these outputs. In contrast to traditional Linux kernel print statements, *ProgMP* enables a fast update of schedulers and print statements.

<sup>1</sup>The complexity of debugging Linux kernel code might be intended considering <http://lwn.net/2000/0914/a/lt-debugger.php3>.

During our scheduler analysis in this dissertation, we recurrently show timeplots of different variables. These are captured with PRINT statements, such as Listing A.2 for round-trip time statistics. We found that PRINT statements have a huge performance impact and should be used carefully. To balance the induced overhead the induced overhead, we restrict PRINT operations to occur only once in a time interval, e. g., every 200ms (line 1–3).

```
1 VAR interPrintInterval_ms = 200;
2
3 IF (R6 < CURRENT_TIME_MS) {
4   SET(R6, CURRENT_TIME_MS + interPrintInterval_ms);
5   PRINT("Number of subflows is %d", SUBFLOWS.COUNT);
6
7   FOREACH(VAR sbf IN SUBFLOWS) {
8     PRINT("Subflow with id = %d ...", sbf.ID);
9     PRINT("... has RTT = %d ...", sbf.RTT);
10    PRINT("... and is backup %d.", sbf.IS_BACKUP);
11  }
12 }
```

Listing A.2: *ProgMP* snippet that provides analysis information.<sup>2</sup>

---

<sup>2</sup>A full code example and a ready test environment is provided at <https://progmp.net/progmp.html#debug>.

## A.3 PACKETDRILL TESTSCRIPT FOR THE PROGMP IMPLEMENTATION

```

1 // Test if the sending queue size reaches 2
2
3 // Clean output
4 0 'dmesg -c > /dev/null'
5
6 // Configure ProgMP
7 +0 'sysctl -w net.mptcp.mptcp_scheduler=rbs'
8 +0 'echo "SCHEDULER two_in_q; PRINT(\"Q.COUNT %u\", Q.COUNT); IF(Q.
    COUNT > 1) { SUBFLOWS.GET(0).PUSH(Q.POP()); SUBFLOWS.GET(0).
    PUSH(Q.POP()); }" > /proc/net/mptcp_net/rbs/schedulers '
9 +0 'echo "two_in_q" > /proc/net/mptcp_net/rbs/default '
10
11 // Establish sockets
12 +0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
13 +0 setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
14 +0 bind(3, {sa_family = AF_INET, sin_port = htons(13000), sin_addr
    = inet_addr("192.168.0.1")}, ...) = 0
15 +0 listen(3, 1) = 0
16
17 +0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 5
18 +0 setsockopt(5, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
19 +0 bind(5, {sa_family = AF_INET, sin_port = htons(13001), sin_addr
    = inet_addr("192.168.0.1")}, ...) = 0
20 +0 listen(5,1) = 0
21
22 +0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 10
23 +0 setsockopt(10, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
24 +0 bind(10, {sa_family = AF_INET, sin_port = htons(13002),
    sin_addr = inet_addr("192.168.0.1")}, ...) = 0
25 +0 listen(10, 1) = 0
26
27 // Open first subflow with MPTCP
28 +0 < S 0:0(0) win 32792 <mss 1460,sackOK,nop,nop,nop,wscale 7,
    mp_capable key_a> sock(3)
29 +0 > S. 0:0(0) ack 1 win 28800 <mss 1460,nop,nop,sackOK,nop,wscale
    7,mp_capable key_b> sock(3)
30 +0 < . 1:1(0) ack 1 win 257 <mp_capable key_a key_b> sock(3)
31 +0 accept(3, ..., ...) = 4
32
33 // Open second subflow with MPTCP
34 +0 < S 0:0(0) win 32792 <mss 1460,sackOK,nop,nop,nop,wscale 7,
    mp_join_syn address_id=1 token=sha1_32(key_b)> sock(10)
35 +0 > S. 0:0(0) ack 1 win 28800 <mss 1460,nop,nop,sackOK,nop,wscale
    7,mp_join_syn_ack address_id=1 sender_hmac=trunc_l64_hmac(
    key_b key_a)> sock(10)
36 +0 < . 1:1(0) ack 1 win 32792 <mp_join_ack sender_hmac=
    full_160_hmac(key_a key_b)> sock(10)
37 +0 mp_join_accept(10) = 11
38
39 // Write 2000 byte to the socket stream
40 +0 write(4, ..., 2000) = 2000
41
42 // Give some time to trigger retransmissions

```

```
43 +0 'sleep 5'  
44  
45 // Check if the queue size was 2  
46 +0 'dmesg | grep "Q.COUNT 2"'
```

Listing A.3: *Packetdrill* script to test if the sending queue size changes as expected (see Section [3.4.5.3](#) for a discussion).

## A.4 PRESENTED PROGMP SCHEDULERS

```

1  VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.THROTTLED AND !sbf.LOSSY);
2
3  IF(sbfCandidates.EMPTY) { RETURN; }
4
5  IF (!RQ.EMPTY) {
6    VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR
    (RQ.TOP) AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
7    IF (sbfCandidate != NULL) {
8      sbfCandidate.PUSH(RQ.POP());
9      RETURN;
10   }
11 }
12
13 IF (!Q.EMPTY) {
14   sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(Q.TOP)).MIN(sbf =>
    sbf.RTT).PUSH(Q.POP());
15 }

```

Listing A.4: Full specification of the default scheduler as discussed in Section 5.2. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_default](https://progmp.net/progmp.html#dissertation_default).

```

1  VAR considerBackups = SUBFLOWS.FILTER(sbf => !sbf.IS_BACKUP).EMPTY;
2  VAR sbfCandidates = SUBFLOWS.FILTER(sbf => !sbf.THROTTLED AND sbf.
    CWND > sbf.SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.LOSSY AND ((sbf.
    IS_BACKUP AND considerBackups) OR (!sbf.IS_BACKUP AND !
    considerBackups)));
3
4  IF (!RQ.EMPTY) {
5    VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR
    (RQ.TOP) AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
6    IF (sbfCandidate != NULL) {
7      sbfCandidate.PUSH(RQ.POP());
8      RETURN;
9    } ELSE IF (!considerBackups) {
10     VAR sentOnAllNonBackupSubflows = SUBFLOWS.FILTER(sbf => !RQ.TOP.
    SENT_ON(sbf)).EMPTY;
11
12     VAR backupSbfCandidate = SUBFLOWS.FILTER(sbf => !sbf.THROTTLED
    AND sbf.CWND > sbf.SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.LOSSY
    AND sbf.IS_BACKUP).MIN(sbf => sbf.RTT);
13     IF (backupSbfCandidate != NULL) {
14       backupSbfCandidate.PUSH(RQ.POP());
15       RETURN;
16     }
17   }
18 }
19
20 IF (!Q.EMPTY) {
21   sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(Q.TOP)).MIN(sbf =>
    sbf.RTT).PUSH(Q.POP());
22 }

```

Listing A.5: Full specification of the default scheduler with backup semantics as discussed in Section 5.2. Note that this scheduler considers backup subflows for packets from the reinjection queue in case the packet was sent on all non-backup subflows, even though non-backup subflows are available. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_default\\_backup](https://progmp.net/progmp.html#dissertation_default_backup).

```

1 IF (!RQ.EMPTY) {
2   VAR sbfCandidate = SUBFLOWS.FILTER(sbf => !sbf.THROTTLED AND sbf.
      CWND > sbf.SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.LOSSY AND sbf.
      HAS_WINDOW_FOR(RQ.TOP) AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.
      RTT);
3   IF (sbfCandidate != NULL) {
4     sbfCandidate.PUSH(RQ.POP());
5     RETURN;
6   }
7 }
8
9 /* tuning parameters */
10 VAR quota = 1;
11 VAR cwnd_limited = 1; /* 1: tries to fill the cwnd on all subflows.
      */
12
13 VAR sbfCandidates = SUBFLOWS.FILTER(sbf => !sbf.THROTTLED AND sbf.
      USER < quota AND !sbf.LOSSY AND (cwnd_limited == 0 OR sbf.CWND
      > sbf.QUEUED + sbf.SKBS_IN_FLIGHT));
14
15 /* Take subflow which started to use quota */
16 VAR inUse = sbfCandidates.FILTER(sbf => sbf.USER != 0).GET(0);
17 IF (inUse != NULL) {
18   IF(inUse.CWND > inUse.QUEUED + inUse.SKBS_IN_FLIGHT) {
19     inUse.PUSH(Q.POP());
20   }
21   RETURN;
22 }
23
24 VAR fresh = sbfCandidates.GET(0);
25 IF (fresh != NULL) {
26   IF(fresh.CWND > fresh.QUEUED + fresh.SKBS_IN_FLIGHT) {
27     fresh.PUSH(Q.POP());
28   }
29   RETURN;
30 }
31
32 /* reset quota */
33 FOREACH(VAR sbf IN SUBFLOWS) {
34   sbf.SET_USER(0);
35 }

```

Listing A.6: Full specification of the round robin scheduler as discussed in Section 5.2. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_round\\_robin](https://progmp.net/progmp.html#dissertation_round_robin).

```

1 VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.THROTTLED AND !sbf.LOSSY);
2
3 IF(sbfCandidates.EMPTY) { RETURN; }
4
5 IF (!RQ.EMPTY) {
6   VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR
    (RQ.TOP) AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
7   IF (sbfCandidate != NULL) {
8     sbfCandidate.PUSH(RQ.POP());
9     RETURN;
10  }
11 }
12
13 IF (R1 >= SUBFLOWS.COUNT) { SET(R1, 0); }
14
15 IF (!Q.EMPTY) {
16   VAR sbf = SUBFLOWS.GET(R1);
17   IF (sbf.CWND > sbf.PACKETS_IN_FLIGHT + sbf.QUEUED AND
18     !sbf.IS_THROTTLED AND !sbf.IS_LOSSY) {
19     sbf.PUSH(Q.POP());
20   }
21   SET(R1, R1 + 1);
22 }

```

Listing A.7: Full specification of the alternative round robin scheduler as discussed in Section 5.2. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_round\\_robin2](https://progmp.net/progmp.html#dissertation_round_robin2).

```

1 VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.THROTTLED AND !sbf.LOSSY);
2
3 IF(sbfCandidates.EMPTY) { RETURN; }
4
5 IF (!RQ.EMPTY) {
6   VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR
    (RQ.TOP) AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
7   IF (sbfCandidate != NULL) {
8     sbfCandidate.PUSH(RQ.POP());
9     RETURN;
10  }
11 }
12
13 FOREACH(VAR sbf IN sbfCandidates) {
14   VAR skb = QU.FILTER(s => !s.SENT_ON(sbf)).TOP;
15   /* are all QU packets sent on this sbf? */
16   IF(skb != NULL) {
17     sbf.PUSH(skb);
18   } ELSE {
19     sbf.PUSH(Q.POP());
20   }
21 }

```

Listing A.8: Full specification of the redundant scheduler as discussed in Section 5.2. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_redundant](https://progmp.net/progmp.html#dissertation_redundant).

```

1  VAR probingIntervallRttMultiplier = 5; /* Tuning parameter */
2
3  VAR sbfCandidates = SUBFLOWS.FILTER(sbf => !sbf.THROTTLED AND sbf.
    CWND > sbf.SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.LOSSY);
4
5  IF (sbfCandidates.EMPTY) { RETURN; }
6
7  IF (!RQ.EMPTY) {
8    VAR sbfCandidate = sbfCandidates.FILTER(sbf =>
9      sbf.HAS_WINDOW_FOR(RQ.TOP) AND
10     !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
11    IF (sbfCandidate != NULL) {
12      sbfCandidate.PUSH(RQ.POP());
13      RETURN;
14    }
15  }
16
17  IF (Q.EMPTY) { RETURN; }
18
19  /* Schedule as usual. There is at least one subflow */
20  VAR packetToSend = Q.TOP;
21  VAR bestSbf = sbfCandidates.MIN(sbf => sbf.RTT);
22  bestSbf.PUSH(Q.POP());
23
24  /* Reset subflow probing timeout */
25  bestSbf.SET_USER(CURRENT_TIME_MS + (bestSbf.RTT *
    probingIntervallRttMultiplier));
26
27  IF (Q.EMPTY AND packetToSend.PSH) {
28    /* End of a burst ? */
29    FOREACH (VAR sbf IN sbfCandidates.FILTER(sbf => sbf.USER <
    CURRENT_TIME_MS)) {
30      /* Send redundant packet and reset subflow probing timeout */
31      sbf.PUSH(packetToSend);
32      sbf.SET_USER(CURRENT_TIME_MS + (sbf.RTT *
    probingIntervallRttMultiplier));
33    }
34  }

```

Listing A.9: Full specification of the active probing scheduler as discussed in Section 5.3. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_active\\_probing](https://progmp.net/progmp.html#dissertation_active_probing).

```

1  VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.THROTTLED AND !sbf.LOSSY);
2
3  IF(sbfCandidates.EMPTY) { RETURN; }
4
5  IF (!RQ.EMPTY) {
6    VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR
    (RQ.TOP) AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
7    IF (sbfCandidate != NULL) {
8      sbfCandidate.PUSH(RQ.POP());
9      RETURN;
10   }
11  }

```

```

12
13 /* we are sure that there is at least one subflow */
14 FOREACH(VAR sbf IN sbfCandidates) {
15   sbf.PUSH(Q.TOP);
16 }
17 DROP(Q.POP());

```

Listing A.10: Full specification of the opportunistic redundant scheduler as discussed in Section 5.4. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_opportunistic\\_redundant](https://progmp.net/progmp.html#dissertation_opportunistic_redundant).

```

1  VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.THROTTLED AND !sbf.LOSSY);
2
3  IF(sbfCandidates.EMPTY) { RETURN; }
4
5  IF (!RQ.EMPTY) {
6    VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR
    (RQ.TOP) AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
7    IF (sbfCandidate != NULL) {
8      sbfCandidate.PUSH(RQ.POP());
9      RETURN;
10   }
11 }
12
13 IF (!Q.EMPTY) {
14   sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(Q.TOP)).MIN(sbf =>
    sbf.RTT).PUSH(Q.POP());
15 } ELSE {
16   /* retransmit on all other subflows... start with oldest skb
    which was not sent on a sbf which has cwnd */
17
18   VAR skbCandidate = QU.
19     FILTER(skb_ =>
20       !sbfCandidates.FILTER(sbf => ! skb_.SENT_ON(sbf)).EMPTY
21       ).TOP;
22
23   sbfCandidates.FILTER(sbf => !skbCandidate.SENT_ON(sbf)).MIN(sbf
    => sbf.RTT).PUSH(skbCandidate);
24 }

```

Listing A.11: Full specification of a scheduler which retransmits packets if the sending queue is empty as discussed in Section 5.4. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_redundant\\_q\\_empty](https://progmp.net/progmp.html#dissertation_redundant_q_empty).

```

1  VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED
2    AND !sbf.THROTTLED AND !sbf.LOSSY);
3  IF(sbfCandidates.EMPTY) { RETURN; }
4
5  IF (!RQ.EMPTY) {
6    VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.
    HAS_WINDOW_FOR(RQ.TOP)
7      AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
8    IF (sbfCandidate != NULL) {

```

```

9      sbfCandidate.PUSH(RQ.POP());
10     RETURN;
11   }
12 }
13
14 IF (!Q.EMPTY) {
15   sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(Q.TOP)).MIN(sbf
=> sbf.RTT).PUSH(Q.POP());
16 }
17
18 VAR minRttRatio = 2;
19
20 IF(R1 == 1 AND Q.EMPTY) {
21   VAR bestSbf = sbfCandidates.MIN(sbf => sbf.RTT);
22   VAR sbfsToCompensate = SUBFLOWS.FILTER(sbf => sbf.RTT > bestSbf.
RTT * minRttRatio);
23
24   /* packet not on bestSbf but sent on at least one sbfToCompensate
*/
25   VAR skbCandidate = QU.FILTER(skb => !skb.SENT_ON(bestSbf) AND !
sbfToCompensate.FILTER(sbf => skb.SENT_ON(sbf)).EMPTY).GET(0);
26
27   bestSbf.PUSH(skbCandidate);
28 }

```

Listing A.12: Full specification of a scheduler that compensates previous scheduling decisions as discussed in Section 5.5. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_compensate](https://progmp.net/progmp.html#dissertation_compensate).

```

1  VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
SKBS_IN_FLIGHT + sbf.QUEUED
2     AND !sbf.THROTTLED AND !sbf.LOSSY);
3
4  IF(sbfCandidates.EMPTY) { RETURN; }
5
6  IF (!RQ.EMPTY) {
7     VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.
HAS_WINDOW_FOR(RQ.TOP)
8         AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
9     IF (sbfCandidate != NULL) {
10        sbfCandidate.PUSH(RQ.POP());
11        RETURN;
12    }
13 }
14
15 IF (Q.EMPTY) { RETURN; }
16
17 VAR considerBackup = SUBFLOWS.FILTER(sbf => sbf.RTT < R1 AND !sbf.
IS_BACKUP).EMPTY;
18
19 IF (considerBackup) {
20   sbfCandidates.MIN(sbf => sbf.RTT).PUSH(Q.POP());
21 } ELSE {
22   sbfCandidates.FILTER(sbf => !sbf.IS_BACKUP).MIN(sbf => sbf.RTT).
PUSH(Q.POP());
23 }

```

Listing A.13: Full specification of a round-trip time- and preference-aware scheduler as discussed in Section 5.6. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_rtt\\_preference\\_aware](https://progmp.net/progmp.html#dissertation_rtt_preference_aware).

```

1 VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED
2   AND !sbf.THROTTLED AND !sbf.LOSSY);
3
4 IF(sbfCandidates.EMPTY) { RETURN; }
5
6 IF (!RQ.EMPTY) {
7   VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.
    HAS_WINDOW_FOR(RQ.TOP)
8   AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
9   IF (sbfCandidate != NULL) {
10    sbfCandidate.PUSH(RQ.POP());
11    RETURN;
12   }
13 }
14
15 IF (!Q.EMPTY) {
16   VAR bestNonBackup = SUBFLOWS.FILTER(sbf => !sbf.IS_BACKUP).MIN(
    sbf => sbf.RTT);
17   VAR bestBackup = SUBFLOWS.FILTER(sbf => sbf.IS_BACKUP).MIN(sbf
    => sbf.RTT);
18
19   VAR considerBackup = bestBackup.RTT_MS < R1 AND bestNonBackup.
    RTT_MS > R2;
20
21   IF (considerBackup) {
22     sbfCandidates.MIN(sbf => sbf.RTT).PUSH(Q.POP());
23   } ELSE {
24     sbfCandidates.FILTER(sbf => !sbf.IS_BACKUP).MIN(sbf => sbf.
    RTT).PUSH(Q.POP());
25   }
26 }

```

Listing A.14: Full specification of a round-trip time- and preference-aware scheduler as discussed in Section 5.6. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_rtt\\_preference\\_aware\\_advance](https://progmp.net/progmp.html#dissertation_rtt_preference_aware_advance).

```

1 VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.THROTTLED AND !sbf.LOSSY);
2
3 IF(sbfCandidates.EMPTY) { RETURN; }
4
5 IF (!RQ.EMPTY) {
6   VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR
    (RQ.TOP) AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
7   IF (sbfCandidate != NULL) {
8     sbfCandidate.PUSH(RQ.POP());
9     RETURN;
10  }

```

```

11 }
12
13 FOREACH(VAR sbf IN sbfCandidates) {
14   VAR skb = QU.FILTER(s => !s.SENT_ON(sbf)).TOP;
15   /* are all QU packets sent on this sbf? */
16   IF(skb != NULL) {
17     sbf.PUSH(skb);
18   } ELSE {
19     sbf.PUSH(Q.POP());
20   }
21 }

```

Listing A.15: Full specification of a scheduler with externally controlled backup subflow semantics as discussed in Section 5.7. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_redundant](https://progmp.net/progmp.html#dissertation_redundant).

```

1  VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED
2    AND !sbf.THROTTLED AND !sbf.LOSSY);
3  IF(sbfCandidates.EMPTY) { RETURN; }
4
5  IF (!RQ.EMPTY) {
6    VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.
    HAS_WINDOW_FOR(RQ.TOP)
7      AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
8    IF (sbfCandidate != NULL) {
9      sbfCandidate.PUSH(RQ.POP());
10     RETURN;
11   }
12 }
13
14 VAR targetBwKB = R1;
15 VAR prefAhead = R4;
16 VAR factor = 100;
17 VAR maxAhead = 100 * factor;
18
19 VAR prefS = sbfCandidates.FILTER(s=>!s.IS_BACKUP).MIN(s=>s.RTT);
20 IF (prefS != NULL) {
21   prefS.PUSH(Q.POP());
22   IF (R4 < maxAhead) {
23     SET(R4, R4 + factor);
24   }
25 } ELSE {
26   VAR mss = 1400;
27   VAR capKB = prefS.CWND / prefS.RTT_MS * mss;
28   VAR ratio = factor * capKB / (targetBwKB - capKB);
29   IF (prefAhead > ratio AND capKB < targetBwKB) {
30     sbfCandidates.MIN(s => s.RTT).PUSH(Q.POP());
31     SET(R4, prefAhead - ratio);
32   }
33 }

```

Listing A.16: Full specification of the throughput- and preference-aware scheduler as discussed in Section 5.7. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_throughput\\_preference\\_aware](https://progmp.net/progmp.html#dissertation_throughput_preference_aware).

```

1 VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.THROTTLED AND !sbf.LOSSY);
2
3 IF(sbfCandidates.EMPTY) { RETURN; }
4
5 IF (!RQ.EMPTY) {
6   VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR
    (RQ.TOP) AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
7   IF (sbfCandidate != NULL) {
8     sbfCandidate.PUSH(RQ.POP());
9     RETURN;
10  }
11 }
12
13 IF (!Q.EMPTY) {
14   sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR(Q.TOP)).MIN(sbf =>
    sbf.DELAY_OUT).PUSH(Q.POP());
15 }

```

Listing A.17: Full specification of the one-way delay-aware scheduler as discussed in Section 5.8. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_onewaydelay](https://progmp.net/progmp.html#dissertation_onewaydelay).

```

1 VAR sbfCandidates = SUBFLOWS.FILTER(sbf => sbf.CWND > sbf.
    SKBS_IN_FLIGHT + sbf.QUEUED AND !sbf.THROTTLED AND !sbf.LOSSY);
2
3 IF(sbfCandidates.EMPTY) { RETURN; }
4
5 IF (!RQ.EMPTY) {
6   VAR sbfCandidate = sbfCandidates.FILTER(sbf => sbf.HAS_WINDOW_FOR
    (RQ.TOP) AND !RQ.TOP.SENT_ON(sbf)).MIN(sbf => sbf.RTT);
7   IF (sbfCandidate != NULL) {
8     sbfCandidate.PUSH(RQ.POP());
9     RETURN;
10  }
11 }
12
13 IF (Q.EMPTY) { RETURN; }
14
15 VAR modeNoHighRtt = 0;
16 VAR modeMinRtt = 1;
17 VAR modePrefAware = 2;
18
19 IF (Q.TOP.USER == modeNoHighRtt) {
20   VAR minRttSbf = SUBFLOWS.MIN(sbf => sbf.RTT).RTT;
21   sbfCandidates.FILTER(sbf => sbf.RTT < minRttSbf * 3 / 2).MIN(sbf
    => sbf.RTT).PUSH(Q.POP());
22 } ELSE IF (Q.TOP.USER == modeMinRtt) {
23   sbfCandidates.MIN(sbf => sbf.RTT).PUSH(Q.POP());
24 } ELSE IF (Q.TOP.USER == modePrefAware) {
25   sbfCandidates.FILTER(sbf => !sbf.IS_BACKUP).MIN(sbf => sbf.RTT).
    PUSH(Q.POP());
26 }

```

---

Listing A.18: Full specification of the HTTP-aware scheduler as discussed in Section 5.9. We provide a ready-to-use test environment at [https://progmp.net/progmp.html#dissertation\\_http\\_aware](https://progmp.net/progmp.html#dissertation_http_aware).

## A.5 PROGMP LANGUAGE SYNTAX

Listing A.19 provides an EBNF-based syntax specification of the *ProgMP* language for the parser generator ANTLR4<sup>3</sup>. While we used ANLTR for the implementation of a validator for the editor at <http://progmp.net/demo>, the runtime environment in the Linux kernel relies on a manually implemented parser as presented in Section 3.4.

```

1 grammar ProgMP;
2
3 scheduler: 'SCHEDULER' IDENTIFIER ';' statements;
4
5 /* Statements */
6
7 statements: statement+;
8
9 statement: (dropStatement | foreachStatement | ifStatement |
10            printStatement | setStatement | varStatement | pushStatement |
11            setUserStatement);
12
13 dropStatement: DROP '(' skbValue ')' ';' ;
14
15 foreachStatement: FOREACH '(' foreachDeclaration ')' '{' statements
16                 '}' ;
17
18 foreachDeclaration: 'VAR' IDENTIFIER 'IN' (skbList | sbfList |
19            IDENTIFIER | genericList);
20
21 ifStatement: ifBlock (ELSE ifBlock)* (ELSE '{' scopeArea '}')?;
22
23 ifBlock: IF '(' boolExpression ')' '{' scopeArea'}';
24
25 scopeArea: (statements returnStatement?) | returnStatement;
26
27 printStatement: PRINT '(' stringValue (',' (intExpression |
28            boolExpression))? ')' ';' ;
29
30 returnStatement: RETURN ';' ;
31
32 setStatement: SET '(' register ',' intExpression ')' ';' ;
33
34 varStatement: VAR IDENTIFIER '=' (identifierHelper | boolExpression
35            | intExpression | skbValue | skbList | sbfValue | sbfList |
36            genericList) ';' ;
37
38 identifierHelper: IDENTIFIER;
39
40 pushStatement: sbfValue '.' PUSH '(' skbValue ')' ';' ;
41
42 setUserStatement: sbfValue '.' SET_USER '(' intExpression ')' ';' ;
43
44 /* Generic */
45
46 genericList: IDENTIFIER ( '.' FILTER '(' IDENTIFIER '=>'
47            boolExpression ')' ) * ;

```

<sup>3</sup>See <http://www.antlr.org/>.

```

40
41 genericValue: genericList '.' 'GET(' intExpression ')';
42
43 /* Packets */
44
45 skbValue: (genericValue | skbPop | skbTop | IDENTIFIER);
46
47 skbPop: (skbList | genericList) '.' POP '()';
48
49 skbTop: (skbList | genericList) '.' (TOP | 'GET(' intExpression ')')
      );
50
51 skbList: ((Q | QU | RQ) (skbFilter)*);
52
53 skbFilter: '.' FILTER '(' IDENTIFIER '=>' boolExpression ')';
54
55 /* Subflows */
56
57 sbfValue: (sbfList | genericList) '.' (sbfMinMax | (GET '('
      intExpression ')') ) | IDENTIFIER | genericValue;
58
59 sbfList: ('SUBFLOWS' (sbfFilter)*);
60
61 sbfFilter: '.' FILTER '(' IDENTIFIER '=>' boolExpression ')';
62
63 sbfMinMax: (MIN | MAX | 'SUM') '(' IDENTIFIER '=>' intExpression ')
      ';
64
65 /* Expressions */
66
67 boolExpression: (boolLiteral | boolProperty | boolFromInt |
      boolFromComparison | boolNotExpression | boolBracketExpression
      | IDENTIFIER) ((AND | OR | '==' | '!=') boolExpression)?;
68
69 boolNotExpression: '!' boolExpression;
70
71 boolBracketExpression: '(' boolExpression ')';
72
73 boolLiteral: (TRUE | FALSE);
74
75 boolFromInt: intExpression ('<' | '>' | '==' | '!=' | '<=' | '>=')
      intExpression;
76
77 boolFromComparison: (skbValue ('==' | '!=') skbValue) |
78   (sbfValue ('==' | '!=') sbfValue) |
79   /* The order is important, as NULL comparisons do not have a
      type */
80   (nullComparisonIdentifier ('==' | '!=') NULL) |
81   (skbValue ('==' | '!=') NULL) |
82   (sbfValue ('==' | '!=') NULL);
83
84 nullComparisonIdentifier: IDENTIFIER;
85
86 boolProperty: ((sbfList '.' EMPTY) |
87   (skbList '.' EMPTY) | (genericList '.' EMPTY) | (skbValue '.' PSH)
      | (skbValue '.' SENT_ON '(' sbfValue ')') |

```

```

88 (sbfValue '.' 'HAS_WINDOW_FOR' '(' skbValue ')') | (sbfValue '.' '
    IS_BACKUP') | (sbfValue '.' THROTTLED) | (sbfValue '.' LOSSY));
89
90 intExpression: (INT | intProperty | intBracketExpression |
    IDENTIFIER (('+' | '-' | '*' | '/' | '%') intExpression?);
91
92 intBracketExpression: '(' intExpression ')';
93
94 intProperty: ('CURRENT_TIME_MS' | register | 'RANDOM' |
95 intSbfListProperty |
96 intGenericProperty |
97 intSkbListProperty |
98 intGenericListProperty |
99 intSkbProperty | intSbfProperty
100 );
101
102 intGenericProperty: IDENTIFIER '.' USER;
103
104 intSkbProperty: skbValue '.' ( LENGTH | SEQ | USER);
105
106 intSbfProperty: sbfValue '.' (CWND | ID | 'LOST_SKBS' | RTT | '
    RTT_MS' | 'RTT_VAR' |
107 'SKBS_IN_FLIGHT' | QUEUED | DELAY_IN | DELAY_OUT | USER | '
    DELAY_OUT_ESTIMATOR' | 'SSTHRESH');
108
109 intSkbListProperty: (skbList '.' COUNT);
110
111 intSbfListProperty: (sbfList '.' COUNT);
112
113 intGenericListProperty: (genericList '.' COUNT);
114
115 register: ('R1' | 'R2' | 'R3' | 'R4' | 'R5' | 'R6');
116
117 stringValue: STRING;
118
119 expression: (boolExpression | intExpression);
120
121 /* LEXER Rules */
122
123 STRING: '"' ~('\r' | '\n' | '"')* '"';
124
125 INT: [0-9]+;
126
127 TRUE: 'TRUE';
128
129 FALSE: 'FALSE';
130
131 DROP: 'DROP';
132
133 POP: 'POP';
134
135 TOP: 'TOP';
136
137 FOREACH: 'FOREACH';
138
139 Q: 'Q';
140

```

```
141 RQ: 'RQ';
142
143 QU: 'QU';
144
145 IF: 'IF';
146
147 ELSE: 'ELSE';
148
149 PRINT: 'PRINT';
150
151 RETURN: 'RETURN';
152
153 SET: 'SET';
154
155 VAR: 'VAR';
156
157 FILTER: 'FILTER';
158
159 AND: 'AND';
160
161 OR: 'OR';
162
163 EMPTY: 'EMPTY';
164
165 COUNT: 'COUNT';
166
167 LENGTH: 'LENGTH';
168
169 QUEUED: 'QUEUED';
170
171 RTT: 'RTT';
172
173 ID: 'ID';
174
175 CWND: 'CWND';
176
177 USER: 'USER';
178
179 SEQ: 'SEQ';
180
181 SENT_ON: 'SENT_ON';
182
183 MIN: 'MIN';
184
185 MAX: 'MAX';
186
187 GET: 'GET';
188
189 PSH: 'PSH';
190
191 THROTTLED: 'THROTTLED';
192
193 LOSSY: 'LOSSY';
194
195 NULL: 'NULL';
196
197 PUSH: 'PUSH';
```

```
198
199 SET_USER: 'SET_USER';
200
201 DELAY_OUT: 'DELAY_OUT';
202
203 DELAY_IN: 'DELAY_IN';
204
205 IDENTIFIER: [a-z] [a-zA-Z_]*; /* we enforce a lower case character
    in the parser.
206 We changed it to force it in the lexer to avoid ambiguity and
    improve error messages. */
207
208 WS: [ \t\r\n]+ -> skip;
209
210 SL_COMMENT: '//' .*? '\n' -> skip;
211
212 COMMENT: '/*' .*? '*/' -> skip;
```

Listing A.19: ANLTR grammar for the presented *ProgMP* scheduler specification language.

## A.6 MPTCP SCHEDULER COMMITS

In this section, we list commits of the original MPTCP Linux kernel schedulers implementations, which consist of additions of features and semantic aspects, as well as bug fixes.

1. mptcp: Don't prevent scheduling on subflows with TSQ-flag set <https://github.com/multipath-tcp/mptcp/commit/d50611004d1f05da5d839aea36c7cd247fee15da>
2. mptcp: Avoid over-counting packets\_out with the redundant scheduler <https://github.com/multipath-tcp/mptcp/commit/0a00c9bbe56f9ae87d48d74530c226e3ab26b809>
3. Fix NULL pointer dereference in redundant scheduler with empty subflow list <https://github.com/multipath-tcp/mptcp/commit/8cf663220767588824ee55326502c748ae59885d>
4. Fix skipping packets with the redundant scheduler <https://github.com/multipath-tcp/mptcp/commit/5bb34f4de990003187364b7593bb867b4f6d1ce5>
5. mptcp: sched: Improve active/backup subflow selection <https://github.com/multipath-tcp/mptcp/commit/f9ca33df8007af2e31c887b6cee6a51b493801c7>
6. mptcp: Do not schedule on tsq-throttled subflows <https://github.com/multipath-tcp/mptcp/commit/5c278893b37fe48c66ff226793607687b8482ba9>
7. mptcp: Don't use tcp\_cwnd\_test in mptcp\_is\_available <https://github.com/multipath-tcp/mptcp/commit/4c10c4895551cbf5f4a22f766e4f7f8455339069>
8. mptcp: Check for skb in get\_available\_subflow <https://github.com/multipath-tcp/mptcp/commit/6a5eb7c98f00a2aa02361dec831bf437824107df>
9. mptcp: fix penal in slow start <https://github.com/multipath-tcp/mptcp/commit/a155a39e9b9e89fbcab2be3e402f5039192fb9e9>
10. mptcp: scheduler: Support interfaces that do not support TSO <https://github.com/multipath-tcp/mptcp/commit/b9a46ab805291f506bf3da8c86ee5dbfceb26a4>
11. mptcp: Fix quota checking in roundrobin scheduler <https://github.com/multipath-tcp/mptcp/commit/8b98611bf0d15bc88b11632d34d294b0ccf739c4>

12. mptcp: mptcp\_dss\_len and mptcp\_sched\_rr can be static  
<https://github.com/multipath-tcp/mptcp/commit/2025d6605d155db514c994cbb63c5023f735232e>
13. mptcp: Disable nagle at meta-level considering segment type  
<https://github.com/multipath-tcp/mptcp/commit/58fa7ade4fd94a7a45186f2f2c8969fd3934c079>



## ERKLÄRUNG LAUT §9 DER PROMOTIONSORDNUNG

---

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe.

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

*Darmstadt, 2018*

---

Alexander Frömmgen





## WISSENSCHAFTLICHER WERDEGANG DES VERFASSERS

---

### **Personal Information**

Name	Alexander Frömmgen
Date of Birth	March 29, 1988
Place of Birth	Koblenz
Nationality	German

### **Education**

2016 – 2018	Technische Universität Darmstadt Research Assistant – Multimedia Communications Lab
2013 – 2016	Technische Universität Darmstadt Research Assistant – Databases and Distributed Systems Group
2011 – 2013	Technische Universität Darmstadt Computer Science – Degree: Masters of Science
2008 – 2011	DHBW Mannheim Wirtschaftsinformatik – Degree: Bachelor of Science



## SUPERVISED THESIS

---

The author of this dissertation supervised the following student thesis:

- [S1] Othmane Achoual. *Switching the TCP Congestion Control at Runtime*. Bachelor Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2016.
- [S2] Patrick Adler. *Die Wahl von Datenstrukturen in Java: Kriterien, Auswirkungen und mögliche Automatisierung*. Bachelor Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2015.
- [S3] Ejaz Ahmed. *Consensus Protocol Analysis and Performance Improvements for Zookeeper*. Master Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2016.
- [S4] Gregor Albrecht. *Eye-Tracker Supported Evaluation of a Domain-Specific Topology Adaptation Language*. Bachelor Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [S5] Sohaib Amir. *Self-Adaptive Systems: A Survey on Representations of Adaptation Logic*. Master Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2016.
- [S6] Andreas Bauer. *Eine Plattform zur Ausführung und Evaluation von Netzwerksimulationen mit vielen Konfigurationen*. Bachelor Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [S7] Fabio Cuozzo. *Erfassung und Analyse von Problemen mobiler Netze in Überlastsituationen*. Bachelor Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2015.
- [S8] Saju Daniel. *MPTCP Ex Machina: Integration der Remy Staukontrollen in MPTCP*. Master Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [S9] Anay Deshpande and Soumya Bhowmik. *Design and Implementation of Multipath UDP*. Seminar Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [S10] Nikolas Eller. *Maschinelles Lernen der Staukontrolle im QUIC Transportprotokoll*. Bachelor Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2018.
- [S11] Tobias Erbschäuer. *Optimierung der Latenz in Multipath-TCP Netzwerken durch Vervielfältigung und quantitative Verteilung der Datenpakete*. Bachelor Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2015.

- [S12] Tobias Erbschäuer. *Optimization of Custom Schedulers in Multipath TCP*. Master Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [S13] Jan Fornoff. *Retrieving Adaptation Knowledge by Offline Simulation in MPEG DASH Video Streaming*. Bachelor Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2016.
- [S14] Gómez Guillermo. *ATP: Adaptive Transport Protocol and Socket Builder*. Master Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2016.
- [S15] Stefan Haas. *Transitionsstrategien zum Wechsel zwischen Peer-to-Peer-Suchoverlays*. Bachelor Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2015.
- [S16] Claudius Kleemann. *Derivation of Optimized Protocols for Distributed Systems using Genetic Programming*. Master Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2018.
- [S17] Alexander Köhler. *Eine Gegenüberstellung von selbst-adaptiver Software mit TCP-basierten Überlastkontrollen und ABR-Algorithmen*. Bachelor Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2016.
- [S18] Johannes Rüsche. *Testgetriebene Ansätze zum automatischen Eingliedern von VNFs und Cloud-Anwendungen*. Master Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2018.
- [S19] Sreeram Sadasivam. *Use your Senses: A Smooth MPTCP WiFi/Mobile Handover*. Seminar Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2015.
- [S20] Tobias Viernickel. *Verbesserte Web-Performance mit Multipath Scheduling für HTTP/2 und QUIC*. Master Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.
- [S21] Patrick Wagner. *Simulation Based Retrieval of Adaptation Knowledge*. Master Thesis, DVS, TU Darmstadt, Supervised by Alexander Frömmgen. 2015.
- [S22] Max Weller. *Optimierte Zusammenarbeit von HTTP/2 und Multipath-TCP-Schedulern*. Bachelor Thesis, KOM, TU Darmstadt, Supervised by Alexander Frömmgen. 2017.