FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Platform for Measuring Mobile Broadband Performance

Analysis and Implementation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Leonhard Wimmer, Bakk.rer.soc.oec.
Matrikelnummer 00025267

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Dr. Özgü Alay, Simula Research Laboratory, Norway
Çise Mıdoğlu, M.Sc, Simula Research Laboratory, Norway

Wien, 1. Juni 2019

Leonhard Wimmer	Wolfgang Kastner

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Platform for Measuring Mobile Broadband Performance

Analysis and Implementation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Leonhard Wimmer, Bakk.rer.soc.oec.
Registration Number 00025267

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Dr. Özgü Alay, Simula Research Laboratory, Norway
Çise Mıdoğlu, M.Sc, Simula Research Laboratory, Norway

Vienna, 1st June, 2019

Leonhard Wimmer	Wolfgang Kastner

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Leonhard Wimmer, Bakk.rer.soc.oec.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juni 2019

Leonhard Wimmer

Acknowledgements

Foremost, I would like to thank my supervisors, *Wolfgang Kastner*, *Özgü Alay*, and *Çise Mıdoğlu* for giving me the opportunity to work on this thesis and for their support.

Especially without Çise this thesis would have never happened. I want to thank her for all the support with the internship, the stay in Norway, the measurement campaigns, the MONROE nodes, the analysis of the results, and the encouragement to finish this thesis.

My heartfelt thanks go to *Isabel Sánchez*, especially for her mental and culinary support in the final writing phases, but also for proofreading and valuable input.

Furthermore, I would like to thank *Simula Research Laboratory* for hosting my internship for six months in 2017. Without this internship, the work on this thesis would have not been possible. Also the team behind the MONROE platform deserve recognition for providing the platform I used in the course of this thesis work.

During the internship at Simula, several people gave me invaluable help with various topics. Besides *Isabel Sánchez*, I would also like to thank *Andra Lutu* for helping me in some battles against the MONROE nodes and *Carsten Griwodz* for sharing his deep knowledge about the inner workings of the TCP protocol.

My sincere thanks also go to *Marion Scholz* for her help with the organization of this thesis and to *Gerti Kappel* for her persuading words to finally finish this thesis. I also want to thank my current employer, *ecosio GmbH*, for giving me the flexibility to work on the final part of this thesis.

Finally, I would also like to extend my thanks to my parents and all my close friends for (almost) not giving up the hope that I would someday finish this thesis and finally graduate.

The work for this thesis was partially funded by the EU H2020 research and innovation programme under grant agreement No. 644399 (MONROE), and by the Norwegian Research Council project No. 250679 (MEMBRANE).

Abstract

The Internet has become an important part of our daily life and we rely on communication networks more than ever before. In addition, the enormous prevalence of mobile devices like smartphones, combined with the availability of high-capacity Fourth Generation (4G) mobile networks, has radically changed the way most people access and use the Internet.

Our demands for network performance are increasing, and therefore proper network performance measurements become extremely relevant.

In this thesis, we focus on network performance measurements and analysis. We describe our motivation for focusing our performance measurements on Mobile Broadband (MBB) networks, give an overview of the state of the art in network measurements, and present related work on MBB measurement platforms.

We develop a network measurement client, that is slim enough to run on low-resource measurement nodes, while collecting detailed measurement results and accompanying metadata.

Our solution is based on the open-source measurement platform *RTR-Netztest*, released by the *Austrian Regulatory Authority for Broadcasting and Telecommunications (RTR)*, and deployed by several National Regulatory Authorities (NRAs) in Europe, such as in Austria, Croatia, Czech Republic, Luxembourg, Norway, Serbia, Slovakia, and Slovenia. This original software was initially developed with major contributions by the author of this thesis.

We perform controlled measurements to validate the accuracy and viability of our solution and describe observed protocol overheads in detail. Afterwards, we perform large-scale measurement campaigns over operational MBB networks in Norway and Sweden, in the context of the EU-funded *Measuring Mobile Broadband Networks in Europe (MONROE)* research project. We present and analyze the results of the measurement campaigns and go into detail about selected interesting findings.

Keywords: mobile broadband networks, large-scale measurements, network performance evaluation, throughput, TCP

Kurzfassung

Das Internet ist zu einem wichtigen Teil unseres täglichen Lebens geworden und wir sind mehr denn je auf Kommunikationsnetze angewiesen. Darüber hinaus hat die enorme Verbreitung mobiler Geräte wie Smartphones in Verbindung mit der Verfügbarkeit leistungsfähiger Mobilfunknetze der vierten Generation (4G) die Art und Weise, wie die meisten Menschen auf das Internet zugreifen und es nutzen, grundlegend verändert.

Unsere Anforderungen an Kommunikationsnetze steigen, daher werden Messungen der Netzwerk-Performance immer wichtiger.

In dieser Arbeit konzentrieren wir uns auf Messungen und Analysen der Netzwerkleistung. Wir beschreiben unsere Motivation, unsere Leistungsmessungen auf mobile Breitbandnetze zu konzentrieren, geben einen Überblick über den Stand der Forschung bei Netzmessungen und präsentieren verwandte Arbeiten über Messplattformen für mobile Breitbandnetze.

Wir entwickeln einen Messclient für Netzwerke, der schlank genug ist, um auf schwacher Hardware zu laufen, während wir detaillierte Messergebnisse und zugehörige Metadaten sammeln.

Unsere Lösung basiert auf der Open-Source-Messplattform *RTR-Netztest*, die von der Österreichischen *Rundfunk und Telekom Regulierungs-GmbH (RTR-GmbH)* veröffentlicht und von mehreren nationalen Regulierungsbehörden in Europa, wie z.B. in Österreich, Kroatien, Tschechien, Luxemburg, Norwegen, Serbien, der Slowakei und Slowenien, eingesetzt wird. Diese Software wurde ursprünglich mit wesentlichen Beiträgen des Autors dieser Arbeit entwickelt.

Wir führen kontrollierte Messungen durch, um die Genauigkeit und Durchführbarkeit unserer Lösung zu überprüfen und die beobachteten Protokoll-Overheads im Detail zu betrachten. Anschließend führen wir im Rahmen des EU-finanzierten Forschungsprojektes *Measuring Mobile Broadband Networks in Europe (MONROE)* groß angelegte Messkampagnen über Mobilfunknetze in Norwegen und Schweden durch. Wir präsentieren und analysieren die Ergebnisse der Messkampagnen und gehen detailliert auf ausgewählte, interessante Ergebnisse ein.

Contents

CHAPTER 1

Introduction

1.1 Preface

The work behind this thesis was primarily conducted in 2017 during an internship at *Simula Research Laboratory* in Norway. As a result the following publications with major contributions by the author of this thesis were published. For these publications, parts of this thesis were taken as input and are therefore not to be viewed as plagiarism.

- **L. Wimmer**, C. Midoglu, A. Lutu, Ö. Alay, and C. Griwodz. „Concept and Implementation of a Configurable Nettest Tool for Mobile Broadband“. In: *2018 IEEE Wireless Communications and Networking Conference (WCNC): IEEE WCNC2018 Student Program (IEEE WCNC 2018 Students)*. Barcelona, Spain, 2018-04 [106]
- C. Midoglu, **L. Wimmer**, A. Lutu, Ö. Alay, and C. Griwodz. „MONROE-Nettest: A Configurable Tool for Dissecting Speed Measurements in Mobile Broadband Networks“. In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2018-04, pp. 342–347. DOI: 10.1109/INFCOMW.2018.8406836 [65] **(Best Paper Award[1])**

The source code produced in the course of this thesis work is publicly available under these URLs:

- https://github.com/lwimmer/rmbt-client
- https://gitlab.com/lwimmer/rmbt-client (Alternative)
- https://github.com/MONROE-PROJECT/Experiments/tree/master/experiments/nettest

[1] https://web.archive.org/web/20190203125544/https://infocom2018.ieee-infocom.org/content/workshop-cnert-computer-and-networking-experimental-research-using-testbeds-program

1.2 Motivation

The need for monitoring and measuring broadband performance has grown tremendously over the past years; interested parties include service providers, regulatory bodies, end users, and researchers.

From the end users' perspective, broadband performance can be defined by the metrics (i) download speed, (ii) upload speed, and (iii) latency as they are experienced by the end user [19].

The use of Mobile Broadband (MBB) networks has increased over the last few years due to the vast popularity of powerful mobile devices, combined with the availability of high-capacity Third Generation (3G)/Fourth Generation (4G) mobile networks. Mobile data traffic grew almost 79 % year-on-year in Q3 2018 and is expected to grow 5-fold by 2024, summing up to a total of 30 % of all Internet Protocol (IP) traffic [24]. Moreover, mobile video traffic represented around 60 % of all mobile data traffic in 2016 and is expected to grow to 78 % by 2021 [21]. All this shows the insatiability of mobile data consumers and the rapid increase in demand for faster mobile connectivity.

One approach to measuring MBB performance is to push the monitoring efforts towards the edge of the network where quality metrics are most relevant. Combined with the paradigm of crowdsourcing, this task can take the form of a mobile application for conducting network measurements on end-user devices [64]. Crowdsourcing is a neologism, originally coined by Jeff Howe in 2006 as a portmanteau of *crowd* and *outsourcing*[2].

There are existing solutions claiming to measure *mobile speed* for end users. Notable examples, providing applications enabling crowdsourced measurements to end users, include *Speedtest.net* [81], *MobiPerf* [69, 33], *OpenSignal App* [82], and *Netradar* [74, 101, 100].

Some of these performance monitoring entities provide rankings and reports on global network performance (e.g., *Speedtest Awards*[3], *OpenSignal Reports*[4], *Netradar country reports*[5]), further steering the public opinion in the mobile broadband market. As a response, mobile carriers have been partnering with such entities to control their service performance from the end-user perspective [105].

National Regulatory Authorities (NRAs) are also providing crowdsourced measurement platforms to the end users, to allow end users and authorities to independently assess and compare the performance of national MBB networks. Notable examples include *Federal Communications Commission (FCC) Speed Test* [25] in the US, *Breitbandmessung*[6] in Germany, and tools based on the *RTR Multithreaded Broadband Test (RMBT)* such as

[2] https://web.archive.org/web/20180702175118/http://www.crowdsourcing.com/cs/

[3] https://www.speedtest.net/awards/

[4] https://opensignal.com/reports/

[5] https://www.netradar.com/category/country-report/

[6] https://breitbandmessung.de/

the *RTR-Netztest* in Austria [96] and similar tools in other countries such as Croatia, Czech Republic, Luxembourg, Norway, Serbia, Slovakia, and Slovenia (see section 3.2 for more detail).

All these crowdsourced tools use different measurement parameters, such as the utilized network protocol, the order of the measurement phases, the measurement duration, and the number of parallel Transmission Control Protocol (TCP) flows (if applicable) [30, 31, 27, 108].

In this thesis, we address the problem of measuring MBB performance with the optimal parameters. We design a flexible, open measurement tool that is able to measure download/upload speed and latency as experienced by the end user. We implement a system able to perform automated measurements on a mobile measurement platform (Measuring Mobile Broadband Networks in Europe (MONROE)[7]) and perform large-scale measurement campaigns over operational MBB networks in order to investigate the optimal parameters for efficient and accurate measurement results.

1.3 Problem Statement

Currently, most of the tools measuring speed are proprietary and the measurement methodology is not open (aside from high-level explanations in the tool descriptions). Therefore, it is not obvious whether the methodology has any hidden biases. The lack of peer review can also result in controversy, as in the case of *Ookla* designating *Airtel* as India's fastest 4G network in 2016, which was a move questioned by another operator, *Reliance Jio*, and subsequently caused a complaint made to the Advertising Standards Council of India (ASCI) [2].

Existing tools differ in the metrics they collect and focus on. Although all of them share the target metrics of upload/download speed and latency, they have different approaches for measuring them. They target different measurement servers, utilize different protocols and use different measurement parameters, therefore they potentially achieve different results.

As measurement results of aforementioned tools impact end-user behavior (e.g., customer churn) and inform regulatory policy (e.g., network neutrality investigations, operator benchmarking), it is important to understand how mobile speed currently is and, more importantly, should be measured.

Therefore, we formulate the research question: What is the optimal measurement design for collecting performance metrics on MBB networks?

We address this question by designing a flexible measurement tool, that is able to perform automated, large-scale measurements with varying parameters. We utilize the resulting tool by performing measurement campaigns in operational MBB networks, in which we investigate a number of measurement parameters and their influence on the results.

[7] For more information regarding the MONROE platform see section 3.7

1.4 Aim of the Work

The aim of this work is to design a measurement software for MBB networks and perform large-scale measurements with the resulting software, where the product is an efficient, flexible implementation of the measurement methodology identified to be the most accurate, allowing for large-scale measurements and open for future improvement. The expected outcomes are the following:

- analysis of existing solutions for performing speed measurements
- implementation of an efficient network speed measurement software
  - able to perform automated, large-scale measurements
  - flexible in terms of configuration and widely compatible with different scenarios
  - open and available for the research community
- ensure compatibility of the measurement tool with the MONROE measurement platform
- performing large-scale measurement campaigns
  - on operational MBB networks to investigate the suitability of selected metrics and parameters
  - to showcase the potential of the implemented tool
- analysis of the measurement campaigns results

1.5 Methodological Approach

In the first step, an extensive literature analysis is conducted to obtain the necessary background knowledge on speed measurements.

In the next step, we determine the relevant parameters and metrics that follow from the literature analysis. We select the most relevant ones for our work and focus on these parameters and metrics in the further work.

Furthermore we analyze available measurement methodologies by examining other, similar tools that are already available and in active usage. We use the gathered insights to select the basis we build our approach on and to optimize our measurement solution.

Next, we implement and test the measurement software and setup the measurement system so that it can be used on the targeted MONROE platform. We start our measurement campaigns by running large-scale measurement campaigns in a controlled environment to validate the results, reported by our implemented measurement client.

After reaching confidence in the reported results, we focus our measurement campaigns on operational MBB networks. We take advantage of the MONROE platform to perform

measurement campaigns over six operational MBB networks in Norway and Sweden. Finally, we run large-scale measurement campaigns over operational MBB networks to investigate the effects of these parameters and provide an analysis of selected results.

1.6 Structure of the Thesis

This thesis is structured in the following chapters:

Chapter 2 gives an overview of the state of the art in network measurements. We also explain why we base our measurements on the TCP protocol, briefly describe the importance of MBB networks and present related work in MBB measurement platforms.

In chapter 3, we give an overview of the key concepts, present the used codebase, and describe the utilized measurement protocol and phases in detail. We also show the used measurement algorithm and give a mathematical representation of the throughput calculation. In the last section, we give an introduction into the MONROE platform.

In chapter 4, we describe the software — implemented in the course of this thesis — in detail. We give an overview of the resulting source code, explain the necessary steps for compiling the code, how we installed the servers, and how the measurements were performed. We also describe all configuration parameters, the collected metrics, and the result format in detail.

Chapter 5 gives an overview of the conducted measurement campaigns. We explain the measurement setup, give the reasoning for selecting the measurement parameters, and explain why certain campaigns were conducted. We also describe the characteristics of the measurements and present selected results.

Chapter 6 summarizes the findings of this thesis and lists a few, potentially interesting topics for future work.

In appendix A, we list all the measurement campaigns mentioned in chapter 5, and give detailed information about the measurement parameters and the results.

Appendix B gives listings of the resulting source code.

CHAPTER 2

Background

This chapter gives an overview of the state of the art in network performance measurements. We also explain the reasoning behind using TCP for our measurement platform and motivate the focus on MBB networks and their performance assessment. Finally, we go into details about related work in MBB measurement platforms.

2.1 Key Concepts

In this section, we define important terms relevant to understand the following chapters.

Throughput is the average number of data units per time unit, successfully transmitted from the sender to the receiver. In this thesis, we use bits per second as a base unit (bit/s), with common International System of Units (SI) prefixes (e.g. kbit/s, Mbit/s, Gbit/s). We are focusing on measuring the *maximum achievable throughput* (see also section 2.2). Section 5.3.2 goes into more detail about the overhead of various protocols.

Client is the device running the measurement client software and initiating the measurement to the measurement server. The direction of the throughput measurement (Downlink (DL)/Uplink (UL)) is viewed from the point of view of the client.

Server is the device running the measurement server software listening for connections from the client.

Downlink (DL) is the communication direction from the server to the client.

Uplink (UL) is the communication direction from the client to the server.

2.2 Network Measurements

As communication networks are becoming more prevalent in our daily lives, measuring the performance of these networks is becoming more important than ever before. Knowing the status and performance of the network is not only important for end user satisfaction and marketing, but also to be able to proactively detect and correct network problems and to have an insight into areas that require improvement.

Especially with the release of newer technologies (e.g. Fifth Generation (5G) mobile networks), being able to get a view on the current performance of the network and its improvement is a great aid in developing and rolling out new technologies.

Tanenbaum and Wetherall (2011) [104] define the primary parameters for network flows as: (i) bandwidth, (ii) delay, (iii) jitter, and (iv) loss. Jin and Tierney (2003) [40] go into the important difference of *achievable throughput* vs. *available bandwidth*.

In this thesis, we focus on measuring the *maximum achievable throughput*, as we want to evaluate the network performance from the viewpoint of the *end user*. Because of various overhead, the gross bit rate is never achievable. In sections 5.3.2 and 5.3.4, we go into more detail of network overheads.

For applications running on end user devices, the theoretical available bandwidth is (for a number of reasons) not always achievable [32, 39]. Rather, the actual achievable throughput, is the most relevant metric from the end users' perspective.

Paxson, Almes, Mahdavi, and Mathis (1998) [86] define a framework for IP performance metrics in RFC 2330. Network performance monitoring is carried out in a passive or active manner [85, 28].

Passive measurements have the advantage of being non-intrusive and they don't suffer from the "Heisenberg" effects (Paxson, 1996 [85]), in which they perturb the network and may bias the results. It is also possible to perform passive measurements on a single point in the network, whilst active measurements (in packet switched networks) consist of at least two parties, a sender and a receiver.

On the other hand, one needs access to particular vantage points in the network — such as routers or base stations — to perform passive measurements, which is not always feasible. There are also privacy concerns, if data, sent by others, is collected during the measurement [85].

Active measurements work by producing traffic in the network, and measuring one or more parameters on the sender and/or receiver sides. In contrast to passive measurements, active measurements can normally be easily performed by end users.

In this thesis, we focus on active measurements, as our purpose is to measure the network from the viewpoint of the user. We also aim to provide a tool, which enables the user to perform measurements under their chosen conditions and without the need for any privileged access to critical network equipment.

2.3 Mobile Broadband Networks

The historical and technological evolution of mobile networks is often divided into generations. With early analog cellular networks being commercially available in the 1970s, the first digital mobile networks — commonly referred to as Second Generation (2G) — came in the early 1990s to the market [68]. Commercial wireless Internet access was even available as early as 1986 in Sweden and 1991 in the U.S., with the *MOBITEX* system [84]. Higher speeds became available as a part of 3G networks in the early 2000s and even higher speeds with All-IP, 4G networks around 2010. Recently the first launches of 5G networks have been announced, providing more spectral efficiency and higher speeds [24].

Ericsson (2018) [24] estimates the worldwide mobile traffic per month to be over 20 EB ($20 * 10^{18}$ Bytes) at the end of 2018 and expect it to grow to 136 EB at the end of 2024. The number of mobile subscriptions is estimated to be around $8 * 10^9$ in 2018 and expected to rise to around $9 * 10^9$ in 2024.

Users are more mobile than ever and this trend seems to continue, they want to be connected anywhere, anytime. Also, the type of traffic shifts to types that demand higher bandwidth, such as video streaming.

When it comes to the mobile traffic type, it is expected that the amount of video traffic will rise from around 60 % in 2018 to around 74 % in 2024 [24]. This is also part of the trend of users requiring higher resources from the networks in the future.

Nikravesh, Choffnes, Katz-Bassett, Mao, and Welsh (2014) [75] show that there is a significant variance in key performance metrics of MBB networks, making measuring these networks a challenging task.

Intrusive testing of achievable throughput — by sending as much data as possible in a time frame – also uses a lot of (potentially limited) data quota, therefore it is important to keep the data volume as low as possible. In this thesis, we use intrusive testing for measuring MBB networks, to generate valuable data, which can later be used to fine tune measurement parameters — such as the measurement duration — to keep the used data volume as low as possible, while still providing accurate results.

Throughput and latency are regarded as the two main factors that impact user experience [20]. We will therefore focus on these factors in this thesis work.

2.4 Using TCP for Throughput Measurements

To measure the maximum achievable throughput, as the end user would experience it, we need to use the most prominent network protocol in use.

Li, Jiang, Chung, and Claypool (2018) [48] provide a recent overview of the most prominently used Internet protocols over 4G mobile networks. They show that over

99.98 % of the total traffic is either TCP or User Datagram Protocol (UDP), while around 90 % of the traffic is TCP.

The vast amount of the UDP flows consist of Domain Name System (DNS) queries, which are insignificant in volume. The major part of the UDP traffic consists of the currently rising Quick UDP Internet Connections (QUIC) protocol, mostly in use by YouTube [48]. QUIC works similarly to TCP + Transport Layer Security (TLS) + HTTP/2 and behaves similarly to TCP when it comes to congestion control, although QUIC tries to improve some shortcomings of TCP [62, 73]. Although a relatively high amount of traffic uses QUIC, it is still under standardization by the Internet Engineering Task Force (IETF), which makes it change quickly and very often and many features are in an early experimental stage[1].

UDP has no feedback loop implemented for acknowledging the successful packet delivery [90]. On the transport layer, the sender gets no information if a sent packet has successfully reached the receiver, or it has been dropped on the transmission path (e.g. due to congestion). Therefore, measuring achievable throughput utilizing UDP requires the implementation of a feedback loop, similar to the congestion control algorithms of TCP, or using a protocol on top of UDP, like QUIC. The other, very intrusive, option is to send with a data rate higher than the expected available bandwidth, flooding the network with packets [29], which is in any case undesirable.

When it comes to measuring throughput form the end users' perspective, TCP is therefore the logical protocol to use to replicate the behavior an end user would experience.

TCP limits its sending rate to avoid congestion, by utilizing a *window size*, limiting the maximum amount of data not yet acknowledged as successfully received by the target. Therefore, the maximum throughput (T) of a TCP connection is limited by the current window size (WS) and the Round Trip Time (RTT):

$$T < \frac{WS}{RTT} \tag{2.1}$$

Mathis, Semke, Mahdavi, and Ott (1997) [59] refine this simple model by taking packet loss (p) and a constant (C) into account and simplifying by using the Maximum Segment Size (MSS) instead of the window size:

$$T < \frac{MSS}{RTT}\frac{C}{\sqrt{p}} \tag{2.2}$$

As Mathis et al. show, C depends on the Acknowledgment (ACK) strategy and the loss pattern.

[1] https://web.archive.org/web/20190316211455/https://datatracker.ietf.org/wg/quic/charter/

2.5 Mobile Broadband Measurement Platforms

There is a plethora of related work on tools and platforms for measuring network performance. Traditionally, controlled platforms for measuring broadband performance were the norm [11, 3, 10].

Bauer, Clark, and Lehr (2010) [13] look at broadband speed measurements and provide a comparison of popular methodologies (ComScore, Ookla Speedtest, Akamai, YouTube Speed, Network Diagnostic Tool (NDT)), while focusing on fixed broadband networks and browser-based testing.

Li et al. (2013, 2015) [49, 50] show that browser-based and app-based solutions have limitations regarding their achievable measurement accuracy.

Goel, Wittie, Claffy, and Le (2016) [26] provide a comprehensive overview of crowdsourced tools that are intended for end user measurements of mobile devices.

Baltrūnas (2017) [12] gives a list of crowdsourced platforms and applications for measuring MBB networks.

Such tools generally can be categorized by the publisher of the tool:

Commercial Bodies with prominent tools such as *Speedtest by Ookla* [81] or *OpenSignal App* [82]. They generally provide free service to end-users, while commercially selling the measurement data to interested parties.

Regulatory Authorities provide tools to citizens for a neutral way of evaluating and comparing operator performance. In some countries (e.g. members of the European Union (EU)), there are regulations for NRAs to transparently monitor performance metrics and ensure end users can achieve the performance indicated in the contract by their operators [79].

Notable examples are *FCC Speed Test* [25] and *RTR-Netztest* [96] (and other tools based on the same codebase as RTR-Netztest; for more information see section 3.2).

Academic Institutions also produced several comparable mobile measurement tools such as *Mobilyzer/MobiPerf* [69, 33, 93, 76], *Netalyzr* [45, 44], and *Netradar* [74, 101]. Researchers are definitely interested in measuring the performance of the network from different perspectives, but the lack of a measurement tool that can adapt to their requirements motivated the development of similar tools by several different research teams. The release of an open-source, well documented measurement software, would have saved invaluable time to some of those teams, which could also benefit from the significant amount of open data extracted in the measurements.

Besides these measurement platforms, there are also other simple, but highly versatile, configurable measurement tools like the open-source software *iperf* [36, 23, 109].

We look at the following mobile measurement tools in more detail to investigate the measurement parameters and measured performance metrics:

Speedtest.net is the most prominent and widely used Internet speed test, developed by *Ookla*. They claim that over $23 * 10^9$ speed tests were performed (as of March of 2019)[2] over their network of around 7000 servers worldwide in 190 countries[3]. They are providing a browser-based solution and apps for several mobile platforms and publish the *Speedtest Global Index*[4] and the *Speedtest Awards*[5], comparing operator speeds and granting awards to the "fastest operators".

While providing a free service to end-users to perform measurements, the measurement data is sold to interested parties.[6]

OpenSignal App are mobile apps for Android and iOS by the commercial company *OpenSignal*, which mostly focuses on coverage mapping, while also providing a speed test solution. Despite having "Open" in the name, neither the code, nor the data are open [82, 18].

They are regularly publishing *OpenSignal Reports*,[7] in which they provide in-depth analysis and statistics about network coverage, availability and speed, globally and per-country.

RTR-Netztest is a free and open-source tool [95], initially released by the Austrian Regulatory Authority for Broadcasting and Telecommunications (RTR), focusing on measuring service quality of mobile and fixed line Internet connections. Applications are available for Android, iOS, and web browsers, and they measure download/upload throughput and latency. The mobile applications also include Quality of Service (QoS) and signal strength measurements[8] [96]. The measurements are also released as open data [97]. After the initial release in Austria, it has also been released by several regulatory authorities in multiple European countries (Croatia, Czech Republic, Luxembourg, Norway, Serbia, Slovakia, and Slovenia; for more information see section 3.2)

MobiPerf is a free and open-source [70], academic measurement platform for mobile networks, built on the open-source measurement library *Mobilyzer*. It is a joint work by University of Michigan, Northeastern University, University of Washington,

[2] https://www.ookla.com/about

[3] https://web.archive.org/web/20190308121213/https://www.speedtest.net/global-index/about

[4] https://www.speedtest.net/global-index

[5] https://www.speedtest.net/awards/

[6] https://www.ookla.com/speedtest-intelligence

[7] https://opensignal.com/reports/

[8] https://web.archive.org/web/20190308123811/https://www.rtr.at/en/tk/netztestfaq_allgemein_0400

Tool	DL/UL speed	Number of flows	Measurement server	Reported results
Speedtest	TCP	5	dedicated server network	adjusted
OpenSignal	HTTP	4	CDN	adjusted
RTR-Netztest	TCP	3	local servers	raw
MobiPerf	TCP	1	dedicated server network	adjusted

Table 2.1: Different methodology aspects for the four tools we analyze

and Google/M-Lab[9] [69, 33, 93, 76]. The raw measurement data is released to the public without restriction [54].[10] The development of the mobile measurement client seems to have ceased in 2014[11].

We observe that while OpenSignal is using Hypertext Transfer Protocol (HTTP) for mobile speed measurements, the other tools use TCP flows directly. The number of threads used, however, varies from one tool to another. For example, RTR-Netztest uses up to 3 flows with a nominal duration of 7 seconds while Speedtest uses up to 5 flows with a nominal duration of 10 seconds.

The location of the server each tool utilizes for its measurements also varies significantly. For example, OpenSignal targets Content Delivery Network (CDN) servers hosted by cloud providers (Akamai, Cloudfront, Google Cloud), claiming that this is what users experience considering most mobile data traffic is served by similar solutions. Speedtest has its own network of target servers distributed across multiple Internet providers and geographical regions, for which a list is provided [80].

Tools by NRAs — such as RTR-Netztest — usually use a small set of target servers, since they target a certain country. MobiPerf, on the other hand, uses a dedicated server network by *Measurement Lab* [61].

Finally, we observe that all the tools except RTR-Netztest apply certain post-processing to the speed measurements. In other words, the reported mobile speed is larger than the average mobile speed measured, indicating that these tools partially eliminate the slow start phase of the TCP transmission.

Table 2.1 presents an overview of the analyzed tools and the respective parameters. We focus on the approach for measuring DL/UL speed, the number of flows, the location of the measurement server each tool uses as target and the post-processing of the results.

[9] https://web.archive.org/web/20150308021021/http://www.mobiperf.com/contact

[10] https://www.measurementlab.net/data/

[11] At the time of this writing (2019) the last commit was from 2014: https://github.com/Mobiperf/MobiPerf

We choose to base our implementation on the *RTR-Netztest*, as it is (i) free and open-source software, (ii) actively developed, (iii) in widespread use, and (iv) highly configurable (e.g. the number of flows and measurement duration).

The RTR-Netztest consists of several parts, such as the control server, the database, the statistic server, and the QoS servers. The measurement core of the RTR-Netztest is called *RTR Multithreaded Broadband Test (RMBT)* [103]. Since we are just focusing on the measurement core, we will refer to it as *RMBT* in the remainder of this thesis.

CHAPTER 3

Measurement Tool Design and Specification

This chapter describes the planning and design of the measurement software before discussing the practical implementation in chapter 4.

3.1 Requirements

We define the functional and non-functional requirements for the measurement software that are needed to perform the planned large-scale measurement campaigns in commercial mobile networks.

3.1.1 Functional requirements

- As discussed in chapter 2, the measurement system needs to be able to perform active network measurements to measure the maximum achievable throughput.
- It should utilize multiple, parallel TCP flows to perform the measurements.
- The measurement system should consist of a server and a client part. The client has to be able to initiate the measurement.
- A measurement run needs to measure the latency and the throughput in both directions.
- The system needs to report the raw, collected time series in high resolution and also the aggregated, calculated measurement results.

3.1.2 Non-functional requirements

- The measurement client needs to be able to run on systems with very low resources and still provide accurate measurements up to the gigabit range. A requirement is therefore, that the measurement part is carefully implemented with resource usage in mind.

- The measurements needs to be highly configurable, to be able to investigate the influence of certain parameters on the result, without the need for a change in the server or client software.

- The measurements will be performed in an automated manner. The server therefore needs to be able to run constantly and has to be ready for measurements, initiated by the client, at all times.

- The server needs to be able to handle multiple measurements from multiple clients simultaneously. The limiting factor should be only the network connection.

- The client has to accept configuration data in a machine-generated form (such as JavaScript Object Notation (JSON)), to be able to schedule large-scale measurements.

- Changes in the measurement parameters should not require a server reconfiguration or restart.

- The measurement results need to be in a machine-readable format, to be able to automate the result collection and enable large-scale data analysis after measurement campaigns.

- The system needs to be verified in a controlled setup before measurement campaigns in commercial mobile networks are performed.

- The system needs to be robust, so it can handle failures during automated, large-scale measurement campaigns, without aborting the whole campaigns on a single failure.

- The source code of the measurement platform should be free and open-source software.

- The software and the configuration and result data need to be thoroughly documented.

Country	Regulatory Authority	Tool
Austria	RTR	RTR-Netztest[1]
Croatia	HAKOM	HAKOMetar Plus[2]
Czech Republic	ČTÚ	NetMetr[3]
Luxembourg	ILR	checkmynet[4]
Norway	Nkom	Nettfart Mobil[5]
Serbia	RATEL	RATEL NetTest[6]
Slovakia	RÚ	RU MobilTest[7]
Slovenia	AKOS	AKOS Test Net[8]

Table 3.1: RMBT installations by European regulatory authorities

3.2 RMBT as Codebase

In November 2012, the Austrian Regulatory Authority for Broadcasting and Telecommunications (RTR)[9] announced the RTR-Netztest at the "72. Mobilregulierungsdialog" [94], as a crowdsourced broadband speedtest. The measurement core of the RTR-Netztest is called *RTR Multithreaded Broadband Test (RMBT)* [103] and is released as free and open-source software[10] under the Apache License [95, 107]. The measurements conducted with the RTR-Netztest are released as Open Data [97].

The software was developed jointly by the Austrian software company *alladin-IT GmbH*[11] and RTR. The author of this thesis was a proprietor of *alladin-IT* and was responsible for the initial implementation and enhancements of the RTR-Netztest software from 2012 to 2016[12].

RTR-Netztest (also known as RTR-NetTest) took as a basis the recommendations by the Organisation for Economic Co-operation and Development (OECD) [78] and currently also follows The Body of European Regulators for Electronic Communications (BEREC) measurement recommendations [14]. The tool was therefore a good choice for other European regulatory authorities as well, which have to fulfill the relevant European regulations [79].

[1] https://www.netztest.at/
[2] https://hakometarplus.hakom.hr/
[3] https://www.netmetr.cz/
[4] https://checkmynet.lu/
[5] http://www.nettfart.no/
[6] https://www.nettest.ratel.rs/
[7] https://www.meracinternetu.sk/sk/index
[8] https://www.akostest.net/
[9] https://www.rtr.at/
[10] https://github.com/rtr-nettest
[11] https://alladin.at/
[12] https://web.archive.org/web/20190215204802/https://alladin.at/downloads/RTR-Nettest-and-Expansion-Modules-2012-2014.pdf

At the time of writing, there are eight known European regulatory authorities providing a measurement software based on RMBT to the public. Table 3.1 shows the currently known countries, the responsible regulatory authorities and the name of the product.

Rationale for the Reimplementation of the Client in C

As discussed in section 2.5, we choose RMBT as the codebase for the implementation of the measurement software.

The initial primary target platforms for the RMBT client were Android based smartphones [94], therefore the RMBT client code was implemented in Java, as it was the primary programming language for Android.

One of the requirements for the measurement software in the specification of this thesis is to be able to run measurement clients smoothly on embedded devices, which enables large-scale remote measurements on different platforms.

Running the Java based client on the MONROE platform would be technically possible, but is deemed as to be too resource hungry to deploy it on low resource systems. As the MONROE platform consists of hardware nodes with relatively low resources, it is obligatory to use a measurement client that uses as little resources as possible. Especially the high Random Access Memory (RAM) usage and the long initial startup time of Java applications is deemed to be too much for the low powered nodes. Therefore, the client part was reimplemented from scratch in C as the only available RMBT client at this time, was the Java based client.

In addition, a more low-level language such as C allows for direct access to low-level socket functions provided by the C standard library. Functions such as `setsockopt` allow getting and setting TCP socket options, which are normally inaccessible by programming languages such as Java. Beneficial socket options to set are for example `TCP_CORK` and `TCP_NODELAY` (see section 3.4.1 for details). The socket option `TCP_INFO` can provide usable insights into the state of the TCP connection (see section 4.5.3 for details).

Using C as the programming language also allows for more control over buffering of input and output data and avoids garbage collections runs, which could influence accurate timestamping.

Consequently, the decision was made to reimplement the RMBT client codebase in C instead of Java.

3.3 RMBT Infrastructure Overview

Figure 3.1 gives an overview of the infrastructure and the communication involved to perform an RMBT measurement [103]. The following steps occur during a complete measurement run:

1. The *Client* sends a (HTTPS) *Test Request* to the *Control Server*.

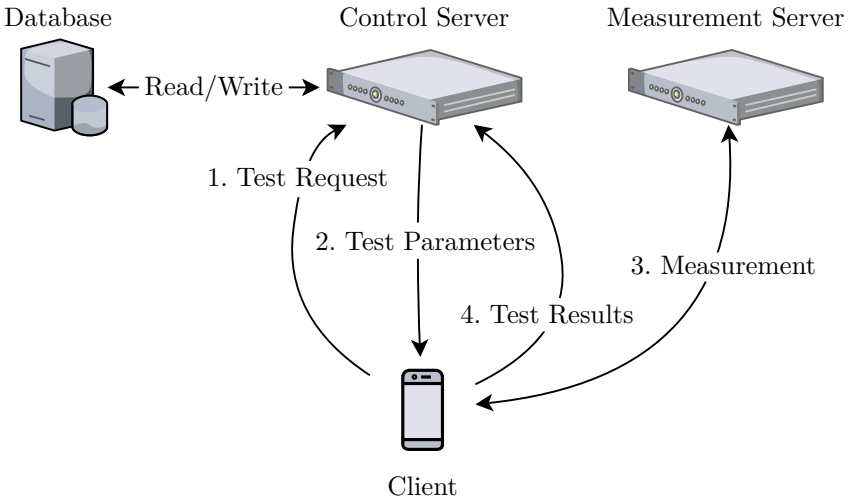


Figure 3.1: Overview of an RMBT run

2. The *Control Server* checks if the *Client* is eligible to perform a measurement and selects a suitable *Measurement Server* which has free capacity and is reasonably close to the client. If the measurement request can be fulfilled, the *Measurement Server* sends back the *Test Parameters* to the *Client*, indicating the measurement parameters, the socket address of the selected *Measurement Server*, the token (see section 3.4.2), and the time slot in which the *Client* is allowed to perform the measurement (i.e. the token validity).

3. The *Client* performs the *Measurement* against the selected *Measurement Server*. The details of this phase are outlined in section 3.4.

4. After the *Measurement* is finished, the *Client* reports the *Test Results* back to the *Control Server*. The *Control Server* stores the results in the *Database* for archiving and further statistical analysis.

In the end, only the *Measurement Server* code of the RMBT source code [95] is used in the scope of this thesis' work. The client code is used as a guide for the reimplementation and the rest of the RMBT code is not required for running the measurements on the MONROE platform, as the needed configuration and the result output format is different from the RMBT project.

3.4 RMBT Protocol

In this section, we describe the RMBT protocol that takes place between the measurement client and server in order to perform a measurement. The design of the RMBT protocol is not part of this thesis, but the protocol was developed with major contribution by the author of this thesis in 2012, including further changes in the following years [103].

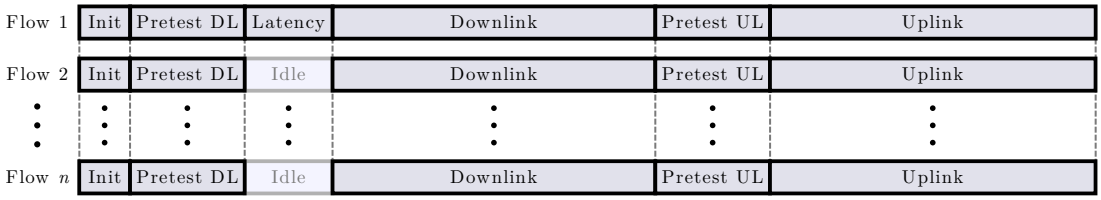


Figure 3.2: Measurement phases of an RMBT measurement

3.4.1 Preliminary Concepts

A measurement consists of the following phases:

1. Initialization Phase
2. Pretest-Downlink Phase
3. Latency Phase
4. Downlink Phase
5. Pretest-Uplink Phase
6. Uplink Phase

Figure 3.2 gives a graphical overview of the order of the measurement phases. It shows n parallel TCP flows, performing the aforementioned measurement phases in sync. Before the start of each of the phases, all threads performing the measurements, wait on a thread synchronization barrier [63], before synchronously starting the next phase. The dashed lines in the figure denote the synchronization barriers.

The *latency phase* (see section 3.4.4) is only performed on one of the TCP flows, to not influence the latency measurement. The other flows are idle during this phase and waiting at the synchronization barrier.

As we show in section 5.3.5, the measurement result is influenced by the number of parallel TCP flows performing the measurement [27, 108]. Section 3.5.1 goes into detail about the calculation of the throughput with multiple TCP flows.

The protocol is visualized with Message Sequence Charts (MSCs) in figures 3.3 to 3.8 for easy readability. In each of the phases, the protocol consists of *messages* and *commands* sent over a TCP or TLS connection. We use *messages* for the direction from server to client and *commands* from client to server. Each *message* or *command* is terminated with an American Standard Code for Information Interchange (ASCII) line feed control character (`LF` which is `0A` in hexadecimal). *Messages* and *commands* start with an uppercase ASCII word, optionally followed by a space `20` in hexadecimal) and one or more parameters (explained in detail in the following sections).

TLS [22] can be used on top of the TCP flows. Since the transmitted data is randomly generated, data security per se is not a concern for using TLS. However, TLS makes possible to perform measurements even within networks protected by firewalls and proxy servers, and to additionally prevent compression. The usage of TLS is a configurable option that can be set at run-time.

The cryptographic handshakes performed during TLS connection establishment are not performed during the actual measurement phase and therefore won't influence the measurement result. To quantify the impact on the measurement result, section 5.3.4 goes into more detail about the overhead of using TLS.

The sent measurement data is generated randomly to prevent any compression. The random data is pre-generated on the server side and the client reuses the received random data for the uplink measurement.

Nagle's Algorithm

In Requests for Comments (RFC) 896, John Nagle described his algorithm for the so called "small-packet problem" [72, 16]. This algorithm is nowadays known as *Nagle's algorithm* and basically works like this:

1. If there is enough data in the buffer to send a packet with the maximum size (i.e. Maximum Segment Size (MSS)), then send it.
2. If there is not enough data to send a packet with the maximum size, send it only when all prior packets have been acknowledged (i.e. ACK has been received).

This algorithm is beneficial for minimizing the packet overhead by trying to avoid sending small packets unnecessarily. However, this algorithm might also introduce a higher latency for sent data, therefore it could interfere with our latency measurements [67]. For this reason, we use the Linux *setsockopt* option `TCP_NODELAY` to turn off Nagle's algorithm during the latency measurement (see section 3.4.4).

Linux has also another interesting option, `TCP_CORK`, which enables a more aggressive version of this algorithm by introducing a delay of up to 200 ms before sending a non-full packet [52]. We activate this option during the throughput measurement phases to minimize the overhead caused by the packet headers (see also section 5.3.2).

Chunks and Termination Byte

The goal during the throughput measurements is to fully saturate the buffer in the kernel so that the kernel can send out the TCP packets with the maximum possible size for the given path. As it is not supported by the kernel to send a continuous stream of data with the socket functions directly, and since it would be very inefficient to send each octet separately to the system buffers, the data is passed on to the operating system kernel in

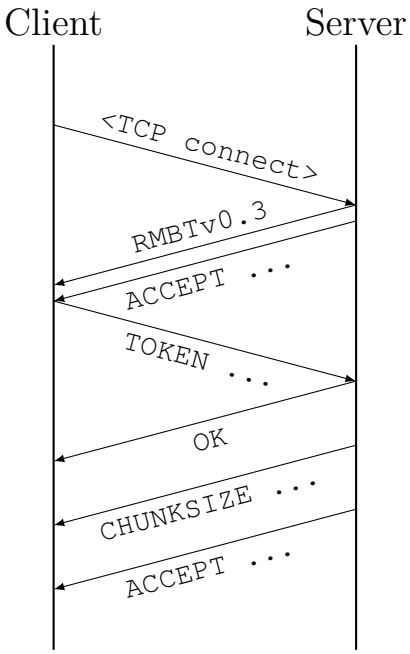


Figure 3.3: Message Sequence Chart of the Initialization Phase

chunks. The chunk size is not related to the packet size of the TCP packets sent over the network. It is only used to pass data from the application to the operating system kernel and read data back from the kernel. RMBT uses a default size of 4096 octets for a chunk, but this size is configurable. For the scope of this thesis, the chunk size is kept at the default value of 4096 octets.

To be able to detect the end of the data transmission in the RMBT protocol, the last octet of each chunk signals if another chunk will follow the current one, or if the current chunk is the last. This last octet is also called the *Termination Byte*. If the Termination Byte is set to all ones (`0xFF`) it indicates that the current chunk is the last, any other octet indicates that another chunk will follow. The sender of the chunks has to make sure to set the Termination Byte to the correct value. The current implementation always sets the octet to all zeros (`0x00`) as a continuation signal and all ones (`0xFF`) as an indication of the end of the data stream.

In the following section, we will describe in detail each of the measurement phases.

3.4.2 Initialization Phase

The measurement starts with the Initialization phase. This phase is marked as *Init* in figure 3.2.

Figure 3.3 shows the initialization phase of the RMBT protocol. This phase starts with the establishment of a TCP connection. If TLS is used, the TLS handshake is done

directly after the establishment of the TCP connection (not part of figure 3.3.) and the rest of the protocol flows over the secured TLS data channel. In case of no TLS usage, the protocol is sent over a plain TCP connection.

After the connection has been established successfully, the server starts by sending the server identification string which includes the protocol version. At the time of writing, the current protocol version is 0.3, therefore the protocol string is `RMBTv0.3`.

Directly following the server identification string, the server sends the `ACCEPT` *message*. Every time the server is ready and listening for a *command* sent by the client it will send an `ACCEPT` *message* indicating the currently supported client *commands*. The `ACCEPT` *message* has the following format:

`ACCEPT <VALUES>`
where `<VAULES>` is a space separated list of currently allowed client *commands*. Possible client *commands* are:

- `TOKEN`
- `GETCHUNKS`
- `GETTIME`
- `PUT`
- `PUTNORESULT`
- `PING`
- `QUIT`

The meaning and usage of these *commands* are described in more detail in the current and the following sections, as part of the different phases of the protocol.

After receiving the `ACCEPT` *message*, the client sends the `TOKEN` *command*. The token *command* has the following format:
`TOKEN <UUID>_<TIMESTAMP>_<HMAC>`
where `<UUID>` is a randomly generated Universally Unique Identifier (UUID) (version 4), `<TIMESTAMP>` is a UNIX timestamp indicating the earliest allowed start time of the measurement and `<HMAC>` is a Base64 encoded HMAC-SHA1 value of `<UUID>_<TIMESTAMP>` [46, 41, 43].

The token is used as a security measure that allows the server to only grant access to clients with a valid token. The token can be handed out by a central control server, and includes a UUID, a timestamp, and a cryptographic signature which only allows for temporary validity, thereby preventing token reuse.

The inclusion of the cryptographic signature allows the RMBT server to check the validity of the token without any need for additional communication with the control server. This allows for an independent and highly distributed nature of the measurement servers.

For the course of this work, the check on the RMBT server was disabled, allowing for direct measurements without the need for a central control server. The server is configured to check the syntax of the token, but neither the validity nor the cryptographic signature (see `CHECK_TOKEN` in section 4.2.).

If the token is correct and valid, the server will respond with an `OK` and `CHUNKSIZE` *message*. In the case of an incorrect token, the server will send an `ERR` *message* and close the connection.

The `CHUNKSIZE <CHUNKSIZE>` *message* indicates the size of a chunk used during the measurement phases in bytes (i.e. octets) to the client (see section 3.4.1).

Finally, the server concludes the Initialization phase with an `ACCEPT` *message*.

3.4.3 Pretest-Downlink Phase

The Pretest-Downlink phase is the second phase and is marked as *Pretest-Downlink* in figure 3.2. It makes sure that the Internet connection is in an "active" state. Some Internet access technologies (mobile and also Digital Subscriber Line (DSL)) put the connection in a power-saving mode which comes with higher latency and potentially lower throughput [1]. Most of these technologies switch to a different mode if there is an active data transfer.

In the Pretest-Downlink phase, the client requests data blocks of the current chunk size, as indicated by the server in the *Initialization Phase*. The first request will start with one chunk and further requests will each double the previous number of chunks. The n-th request will therefore request 2^{n-1} chunks. The total number of requested chunks after m iterations will be $2^m - 1$. After a defined amount of time (`cnf_dl_pretest_duration_s`; see section 4.4) no more chunks will be requested and the Pretest-Downlink phase ends.

Figure 3.4 shows an example of the Pretest-Downlink phase with 3 iterations and therefore a total of 7 chunks $(1 + 2 + 4)$ sent.

Each iteration starts by the client sending the `GETCHUNKS` *command*:
`GETCHUNKS <CHUNKS>`
where `<CHUNKS>` is the number of chunks (each of chunk size) that shall be sent by the server to the client. The server responds by sending data with the length of the current chunk size times the number of chunks. After the transmission is complete, the client sends an `OK` *message*. The server concludes the iteration with a `TIME <TIME>` and an `ACCEPT` *message*. The `<TIME>` *message* indicates the duration of time, measured on the server side, between the reception of the `GETCHUNKS` *command* and the reception of the `OK` *command* in nanoseconds (`d_server`; see also section 4.5.2).

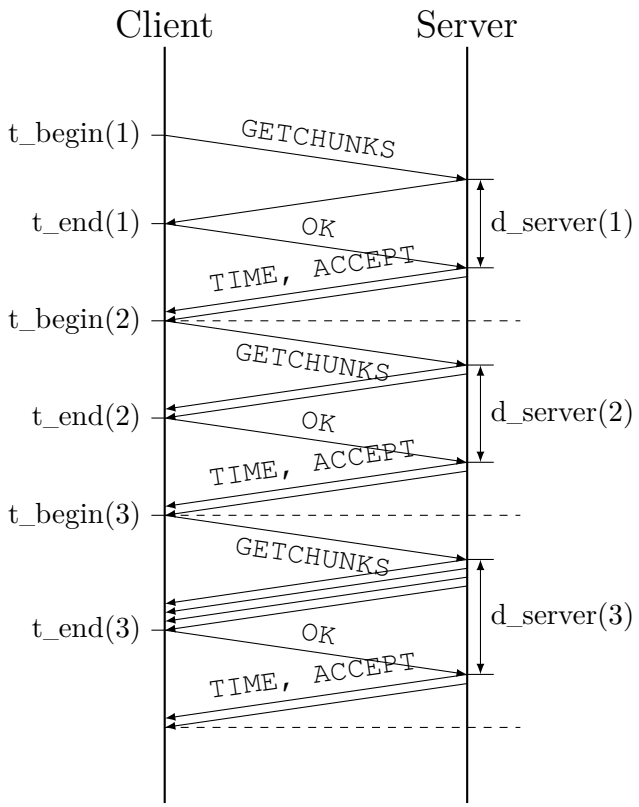


Figure 3.4: Message Sequence Chart of the Pretest-Downlink Phase

Algorithm 3.3 provides an algorithmic description of the Pretest-Downlink phase.

3.4.4 Latency Phase

After the Pretest-Downlink phase, the Latency phase follows (*Latency* in figure 3.2). It only utilizes one TCP connection, the other connections remain idle for the duration of this phase. The goal of this phase is to get an estimate of the Round Trip Time (RTT) of the TCP connection. The measurement is done on client and server side. Figure 3.5 shows the MSC of one RTT measurement. The RMBT client will do `cnf_rtt_tcp_payload_num` (see section 4.4) successive measurements.

The timestamps `time_start_rel_ns` and `time_end_rel_ns` are timestamps in nanoseconds relative to `res_time_start_s`. The durations `rtt_client_ns` and `rtt_server_ns` are the measured RTT in nanoseconds on the client and respectively the server sides. The measured values are reported in the *flows* output file (see section 4.5.2).

The client performs an RTT measurement by sending a `PING` *command* to the server which immediately responds with a `PONG` *message*. The client immediately sends back an `OK` *command* to the server so that a server side RTT measurement is possible. After

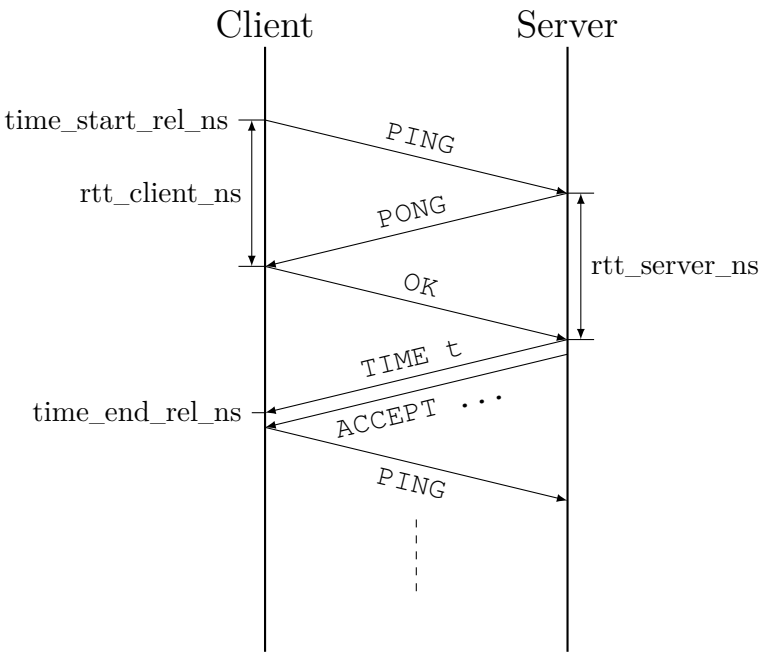


Figure 3.5: Message Sequence Chart of the Latency Phase

receiving the `OK` *command*, the server sends a `TIME <TIME>` *message* to the client in which the measured RTT (`rtt_server_ns`) is reported back to the client so that the client is able to collect all the measurements together. After the RTT measurement is complete, the server sends an `ACCEPT` *message* again to indicate its readiness for further *commands*.

See algorithm 3.2 for an algorithmic description of the Latency phase.

3.4.5 Downlink Phase

In the Downlink phase (*Downlink* in figure 3.2), the main throughput measurement over the DL path takes place. Figure 3.6 shows an MSC of this phase. The client starts this phase by recording the current relative test time `time_start_rel_ns` (see section 4.5.2) and sending a `GETTIME <DURATION>` *command* with `<DURATION>` set to the nominal measurement duration in seconds `cnf_dl_duration_s` (see section 4.4). The server immediately begins sending data continuously until the nominal duration is reached.

After receiving the last data chunk — indicated by the termination byte (see section 3.4.1) — the client sends an `OK` *command* to the server. When the server receives the `OK` *message* it stops the time measurement and sends the measured time in nanoseconds (`duration_server_ns`, see section 4.5.2) with a `TIME <TIME>` *message* back to the client.

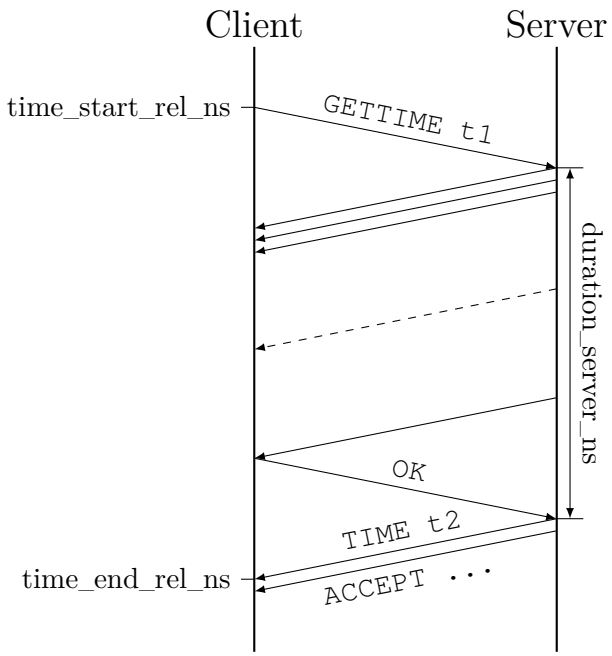


Figure 3.6: Message Sequence Chart of the Downlink Phase

See algorithm 3.4 for an algorithmic description of the Downlink phase.

3.4.6 Pretest-Uplink Phase

The Pretest-Uplink phase is analogous to the *Pretest-Downlink Phase* (cf. section 3.4.3) but with reversed roles for the client and server, in order to enable the throughput mesasurement in that direction. The client starts this phase by sending a `PUTNORESULT` *command*, which allows the client to send data chunks without getting intermediate results (similar to the `PUT` *command*, cf. section 3.4.7). The server responds with an `OK` *message*, indicating its readiness to receive data. The client then starts — analogous to the Pretest-Downlink phase — by sending one data chunk, which is followed by the server sending a `TIME <TIME>` *message*, where `<TIME>` indicates the measured time in nanoseconds (`d_server`). After the `TIME` *message*, the server sends an `ACCEPT` *message* indicating its readiness for further *commands*. The phase continues with the server iterating and sending — analogous to the Pretest-Downlink phase — double the data chunks in each iteration. Figure 3.7 shows one iteration as a MSC.

See algorithm 3.5 for an algorithmic description of the Pretest-Uplink phase.

3.4.7 Uplink Phase

The Uplink phase works analogous to the Downlink phase (cf. section 3.4.5) but with the client sending the chunks of data instead of the server. The client starts this phase

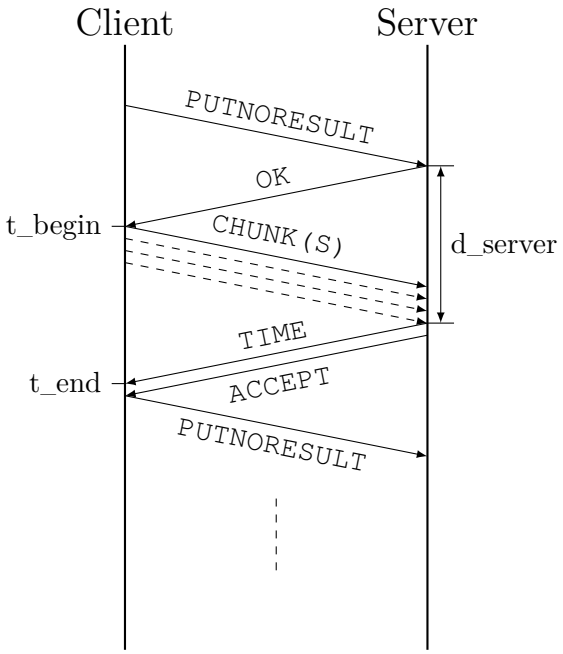


Figure 3.7: Message Sequence Chart of the Pretest-Uplink Phase

by sending a `PUT` *command*, which works similar to the `PUTNORESULT` *command* (see section 3.4.5), but with the server sending intermediate results during the measurement in the form of `TIME <TIME> BYTES <OCTETS>` *messages*, where `<TIME>` indicates the elapsed time in nanoseconds and `<OCTETS>` indicates the number of octets received by the server since the reception of the `PUT` *command*. After the reception of the last data chunk — indicated by the Termination Byte (see section 3.4.1) — the server sends the final `TIME <TIME>` *message*, where `<TIME>` indicates the duration of the measurement in nanoseconds (`duration_server_ns`). After the `TIME` *message* the server sends an `ACCEPT` *message* indicating its readiness for further *commands*. Figure 3.8 visualizes the Uplink phase as MSC.

See algorithm 3.6 for an algorithmic description of the Uplink phase.

3.5 Measurement Algorithm

Algorithm 3.1 describes how a single, client-initiated measurement is conducted on the client side, as pseudo code. The actual measurement is performed in the function `runTest()` (see algorithm 3.2) which runs on multiple threads in parallel. Error handling is out of the scope of this algorithm, and not included in the pseudo code.

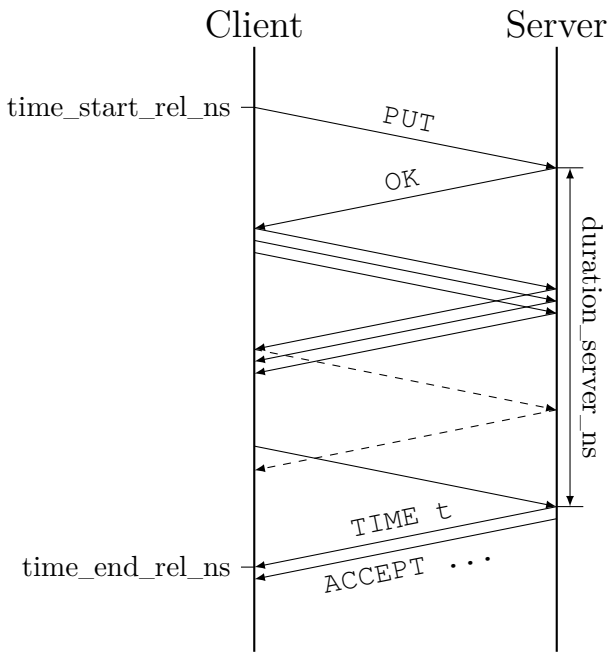


Figure 3.8: Message Sequence Chart of the Uplink Phase

The following functions are used in the algorithms:

`now()` returns a monotonically increasing time counter in seconds since some unspecified starting point, with a resolution of milliseconds or better.

`rttTcpPayload()` performs a latency measurement over the TCP connection. For details, see section 3.4.4.

`barrierAwait()` is used to synchronize the measurement threads. This function only completes if all threads have called this function, in which case the barrier is released and reset.

`requestChunks(numberOfChunks)` requests numberOfChunks chunks of size chunkSize from the measurement server.

`sendChunks(numberOfChunks)` sends numberOfChunks chunks of size chunkSize to the server.

Algorithm 3.1: Measurement algorithm

```
1  numFlows ⟵ max(cnf_dl_num_flows, cnf_ul_num_flows);
2  for f ← 0 to numFlows do
3  |  start new thread f;
4  |  execute asynchronously on thread f: runTest(f);
5  end
6  wait for all started threads to finish;
7  collect results from all threads;
8  combine results according to equation (3.4);
```

Algorithm 3.2: `runTest(threadId)`

```
1  Function runTest(threadId)
2  |  connect to measurement server;
   |  // Downlink pre-test
3  |  barrierAwait();
4  |  downlinkPretest(); // see algorithm 3.3
   |  // Latency test
5  |  barrierAwait();
6  |  if threadId == 0 then // only one thread
7  |  |  for i ← 0 to cnf_rtt_tcp_payload_num do
8  |  |  |  rttTcpPayload();
9  |  |  end
10 |  end
   |  // Downlink test
11 |  barrierAwait();
12 |  downlinkTest(); // see algorithm 3.4
   |  // Uplink pre-test
13 |  barrierAwait();
14 |  uplinkPretest(); // see algorithm 3.5
   |  // Uplink test
15 |  barrierAwait();
16 |  uplinkTest(); // see algorithm 3.6
17 |  close connection to measurement server;
18 end
```

Algorithm 3.3: `downlinkPretest()`

```
1 Function downlinkPretest()
2     numberOfChunks ⟵ 1;
3     startTime ⟵ now();
4     repeat
5         requestChunks(numberOfChunks);
6         numberOfChunks ⟵ numberOfChunks ∗ 2;
7     until startTime + cnf_dl_pretest_duration_s < now();
8 end
```

Algorithm 3.4: `downlinkTest()`

```
1 Function downlinkTest()
2     startTime ⟵ now();
3     request download stream from server;
4     while startTime + cnf_dl_duration_s ≥ now() do
5         receive download stream and record timestamps of chunks;
6     end
7     if reconnect required then
8         terminate connection;
9         connect to measurement server;
10    end
11 end
```

Algorithm 3.5: `uplinkPretest()`

```
1 Function uplinkPretest()
2     numberOfChunks ⟵ 1;
3     startTime ⟵ now();
4     repeat
5         sendChunks(numberOfChunks);
6         numberOfChunks ⟵ numberOfChunks ∗ 2;
7     until startTime + cnf_ul_pretest_duration_s < now();
8 end
```

Algorithm 3.6: `uplinkTest()`

```
1 Function uplinkTest()
2     startTime ⟵ now();
3     while startTime + cnf_ul_duration_s ≥ now() do
4         send upload stream to measurement server and receive measured
            timestamps from server asynchronously;
5     end
6 end
```

3.5.1 Aggregate Throughput

As several TCP flows are transmitting at the same time, the throughput calculation has to take all flows into consideration and the amount of data received by each of the flows has to be aggregated.

Let n be the number of TCP flows used for the measurement and $F := \{1, 2, 3, \ldots, n\}$ be the set of these flows. All transmissions start at the same time, which is denoted as relative time 0.

The data is sent in chunks. Let m_f be the total number of chunks sent over TCP flow f and $C_f := \{1, 2, 3, \ldots, m_f\}$ be the set of all chunks received for this flow.

For each TCP flow $f \in F$, the client records — for every chunk $j \in C_f$ received — the relative time $t_f^{(j)}$ of the completed reception and the total amount $b_f^{(j)}$ of data received on this flow from time 0 to $t_f^{(j)}$ $\forall\, j \in C_f$.

Let $t_f^{(0)} := 0$, $b_f^{(0)} := 0\ \forall\, f \in F$ and let m_f be the number of pairs $\left(t_f^{(j)}, b_f^{(j)}\right)$ which have been recorded for TCP flow f.

$$t^* := \min\left(\{t_f^{(m_f)}\ \forall\, f \in F\}\right) \tag{3.1}$$

t^* being the relative time of the reception of the last chunk on flow f, which has been the earliest flow to stop receiving data. This will be our true measurement duration.

$$l_f := \min\left(\{t_f^{(j)} \geqslant t^*\ \forall\, j \in C_f\}\right)\ \forall\, f \in F \tag{3.2}$$

l_f being the index of the chunk received on thread f at t^* or right after t^*.

Then the amount b_f of data received over TCP flow f from time 0 to time t^* is approximately

$$b_f :\approx b_f^{(l_f-1)} + \frac{t^* - t_f^{(l_f-1)}}{t_f^{(l_f)} - t_f^{(l_f-1)}}\left(b_f^{(l_f)} - b_f^{(l_f-1)}\right) \tag{3.3}$$

To get the aggregated throughput, we take the sum of all calculated data amounts and divide by the true measurement duration.

$$T := \frac{1}{t^*} \sum_{f=1}^{n} b_f \tag{3.4}$$

T is used as the aggregated throughput of all TCP flows combined.

3.6 Measurement Output

An RMBT measurement run produces a plethora of result data. The newly implemented, C based RMBT client, was improved regarding the collection of result and meta data, in comparison to the original, Java based client.

The output data consists of three result categories:

1. The **summary** data, which is a set of the most relevant measurement result data. It consists of the measurement metadata and the aggregated throughput calculations (see section 3.5.1). For detailed information about the summary data, see section 4.5.1.

2. The **flows** data, which consists of the detailed time series, collected individually for each TCP flow, as explained in section 3.4. Detailed information about the flows data can be found in section 4.5.2.

3. The **stats** data, which consists of `TCP_INFO` samples during the whole measurement run. For detailed information about the stats data, see section 4.5.3.

3.7 MONROE platform

The EU *Horizon 2020* research programme-funded MONROE project[13] is a European transnational open platform for independent, multi-homed, large-scale mobile broadband measurements and experiments. It utilizes fixed and mobile hardware measurement nodes distributed over Norway, Sweden, Spain and Italy [8].

MONROE nodes are capable of running most measurement and experiment tasks, including demanding applications like adaptive video streaming [6, 42]. The nodes are connected to up to three MBB providers, which makes it possible to conduct a wide range of measurements and experiments that compare the performance of different networks. In addition to information about the network, time and location for experiments, MONROE nodes also provide rich context information about the connection.

[13] `https://www.monroe-project.eu/`

A MONROE node consists of two small Single-board computers (SBCs) (*PC Engines apu2d4*[14] with three 3G/4G *MC7455 PCI Express Mini Card*[15] modems using Long Term Evolution (LTE) CAT6. An *apu2d4* has an *AMD Embedded G series GX-412TC* 1 GHz *quad core* processor and 4 GB of RAM.

The software on the nodes was based on Debian GNU/Linux "stretch"[16] at the time the measurements presented in this thesis were conducted.

Experiments running on the platform utilize Docker containers to provide a consistent measurement environment and to simplify development and testing [4].

MONROE allows for automated experiments and data collection. The *MONROE scheduler* is used to schedule, deploy and run experiments. Authenticated user access to the MONROE scheduler is possible through a web portal [7]. This enables exclusive access to nodes (i.e. no two experiments run on the node at the same time). The results from each experiment are periodically transferred from the nodes to a repository at a backend server, while the scheduler also sets data quotas to ensure fairness among users.

The work developed in this thesis adapted the RMBT client to run on MONROE nodes. The measurement tool was successfully deployed in an extensive set of large-scale measurements, which allowed to collect results on a variety of network operators and environments.

Moreover, the MONROE project adopted RMBT as part of their experiment base and several research institutions benefit from the measurement tool and the results that it offers in their own research.

Figure 3.9 shows how the RMBT measurements were performed in MONROE.

1. The RMBT measurements are scheduled with the help of the MONROE scheduler, running on the *MONROE Server*. The *Scheduling Data* is pulled by the *MONROE Scheduling Client*, running on the *MONROE Node*.
2. The *MONROE Scheduling Client* takes care of starting the *RMBT Client Container* with the necessary *Measurement Parameters*.
3. The *Measurement* is run against an *RMBT Server*.
4. After the measurement, the *Measurement Results* are stored on the node.
5. The script *monroe-rsync-results*, running on the node, takes care of transmitting the result data to the *MONROE Server* where they can be downloaded and analyzed.

All measurements presented in this thesis were run on MONROE nodes. The measurements in the first campaigns were run directly on the node in development mode, without

[14] https://www.pcengines.ch/apu2d4.htm
[15] https://techship.com/products/sierra-wireless-mc7455-lte-cat6/
[16] https://www.debian.org/releases/stretch/

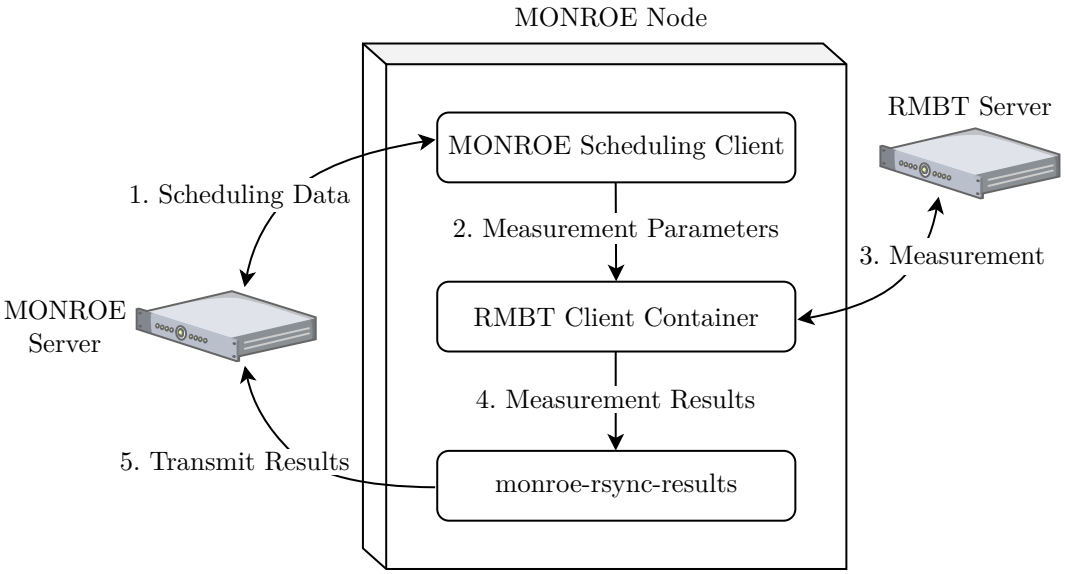


Figure 3.9: Running RMBT on a MONROE node

the help of the scheduler. Later, the large-scale campaigns, utilizing more than one node, were run with the MONROE scheduler. Chapter 5 presents the measurement campaigns and evaluation and analysis of selected results.

CHAPTER 4

Implementation

This chapter describes the implemented software in more detail. As discussed in chapter 3, RMBT was used as a basis for this implementation. Section 4.1 gives a short overview of the resulting source code and explains how to build it. The server part of the software was taken mostly unaltered, as described in section 4.2. In section 4.3, we briefly describe how the measurements were run. Section 4.4 documents the configuration parameters in detail, while section 4.5 provides detailed information about the result formats and the interpretation.

4.1 RMBT Client

As discussed in section 3.2, the RMBT client was reimplemented in the work of this thesis. Section 4.1.1 gives an overview of the resulting source code, and section 4.1.2 explains how the source code can be compiled to an executable.

4.1.1 Source Code

The resulting source code from this thesis work is released as free and open-source software under the Apache License. The relevant source code files and links to the open-source repositories can be found in appendix B.

Figure 4.1 gives a Unified Modeling Language (UML) based overview of the implemented source code. The C code (`.c`) and header (`.h`) files, as well as defined `structs` are represented as UML classes, showing the defined functions and fields. The dotted lines represent the dependencies between the files. The main files (`rmbt.c` and `rmbt.h`) are not part of the diagram, to avoid making it incomprehensible, as they have a dependency to all other files and don't expose any functions and fields to other files.

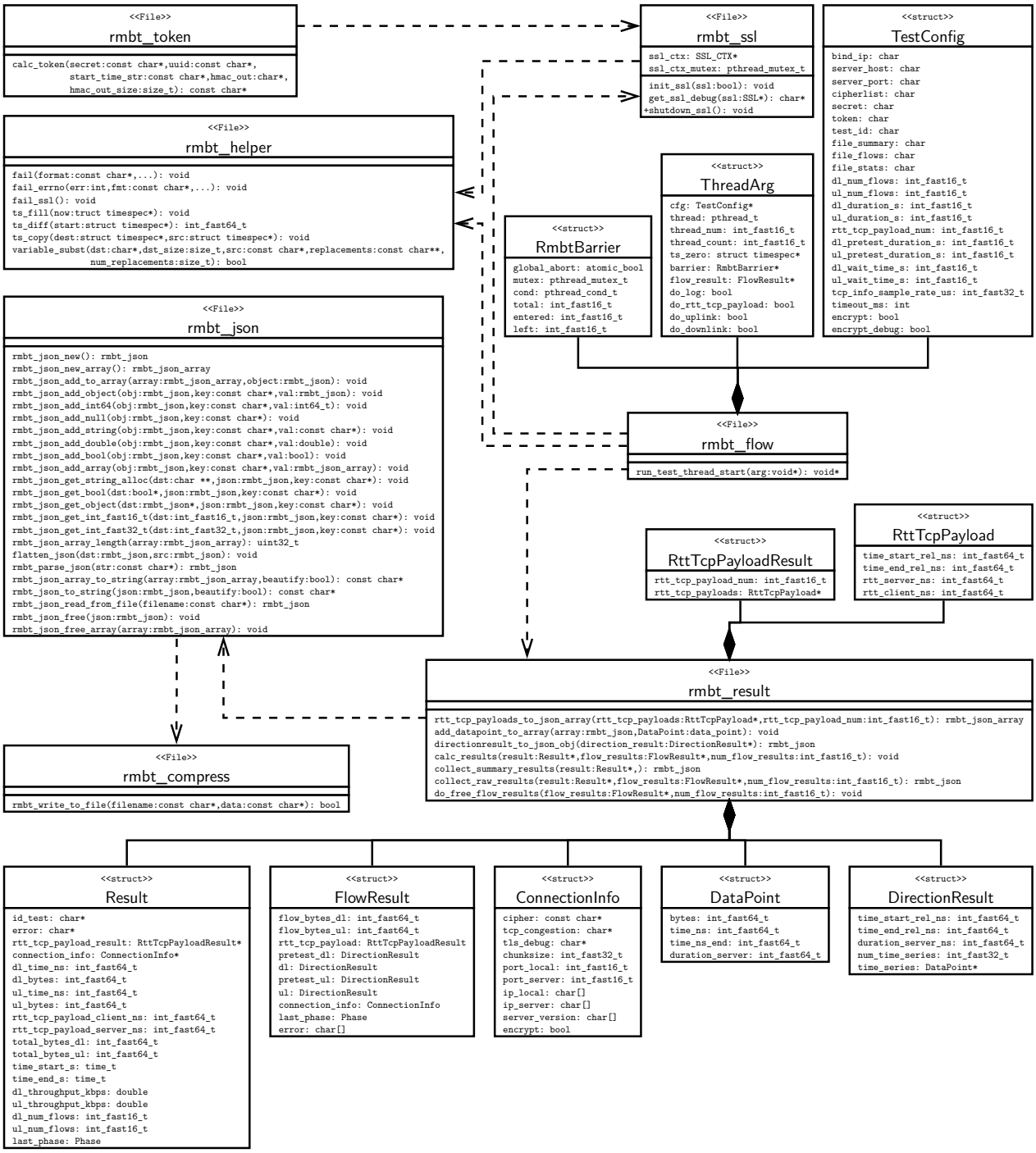


Figure 4.1: Overview of RMBT source code

The following list gives a short description of the individual source code files.

`rmbt_common.h` is the main header file which defines the necessary `#include` statements to include the needed library header files (see listing B.1).

`rmbt.h`, `rmbt.c` are the main files for the client application and include the `main()` function which provides the entry point of the application. The parsing of the command line arguments and the JSON configuration file is done here. After the configuration is read completely, the necessary number of threads is started with the help of the `pthreads` library functions. After the measurements are finished, the main thread waits for all measurement threads to end and collects the results. The main function ends by outputting the results and calling the required shutdown and cleanup functions. (see listings B.2 and B.3)

`rmbt_compress.h`, `rmbt_compress.c` provide the function to write a compressed file to disk. The `lzma` library is utilized for the compression to the efficient xz format. Other formats such as `gzip` and `bzip2` have been tested inplace of the xz format but generated much larger output files for the JSON based output files. (see listings B.4 and B.5)

`rmbt_flow.h`, `rmbt_flow.c` provide the function `run_test_thread_start()` which is run in parallel for each measurement thread. Here the RMBT protocol and the measurements are implemented. For more information about the protocol and the measurement algorithm, see sections 3.4 and 3.5. (see listings B.6 and B.7)

`rmbt_helper.h`, `rmbt_helper.c` define several helper functions that are used in several places in the code. The main focus is on error handling functions, time calculations and variable substitution. (see listings B.8 and B.9)

`rmbt_result.h`, `rmbt_result.c` implement functions for calculating aggregated results and for conversion of internal result representations to the specified JSON output format. (see listings B.10 and B.11)

`rmbt_json.h`, `rmbt_json.c` define the functions that are interfacing with the JSON library to parse the input configuration data and to produce the output files. (see listings B.12 and B.13)

`rmbt_ssl.h`, `rmbt_ssl.c` provide the functions that interface with the *OpenSSL* library to be able to use TLS secured measurement connections. These functions help with the complex initialization of the library. See also section 3.4. (see listings B.14 and B.15)

`rmbt_stats.h`, `rmbt_stats.c` implement the function to interface with the syscall `getsockopt` which collects the `TCP_INFO` metrics. For more information, see section 4.5.3. (see listings B.16 and B.17)

`rmbt_token.h`, `rmbt_token.c` provides the function to calculate the token used in the RMBT protocol. For more information, see section 3.4.2. (see listings B.18 and B.19)

`autobuild.sh`, `configure.ac`, and `Makefile.am` are the source files for the `autoconf` and `automake` utilities. These utilities help with the build environment and produce the `configure` script and the `Makefiles`. To generate the necessary files the `autoreconf -i` and `configure` commands can be used. The helper script, `autobuild.sh`, can be used to run the necessary command to build everything. (see listings B.20 to B.22)

`Dockerfile` is a Docker build file to allow for building a Docker image of the RMBT client.

4.1.2 Building the executable

To be able to build the source code into an executable file the following libraries are required:

- `libc` for interfacing with kernel syscalls,
- `libuuid` for generating UUIDs,
- `libssl` and `libcrypto` for TLS support and for calculating the Hash-based Message Authentication Code (HMAC) of the token,
- `libjson-c` for parsing input and generating output JSON, and
- optionally `liblzma` for compressing output files more efficiently with the xz format

The following additional components are needed for the compilation:

- `autoconf` for generating the `configure` script,
- `automake` for generating the `Makefiles`,
- `make` for automating the compilation and linking steps, and
- `gcc`, `clang` or another C compiler for compiling the source code

Building the source code is straightforward thanks to the utilized tools. The script `autobuild.sh` (see listing B.23) calls the necessary tools and fully automates the compilation to an executable file.

4.2 RMBT Server

During the reimplementation we made sure to keep compatibility with the protocol used by RMBT [103]. Therefore, it was possible to use the RMBT server code from the free and open-source Open-RMBT project [95].

The server code — which was already written in C — was used unaltered.[1] The only change to the configuration in the file `config.h` was to disable the verification of the token by changing the define `CHECK_TOKEN` from 1 to 0. With this change, the server will not verify the token and will accept any token. Therefore, the RMBT client can be run with the parameter `cnf_secret` set to the empty string (i.e. "").

The following commands were used to fetch, build and run the server on the default ports of 8081 and 8082:

```
$ git clone https://github.com/rtr-nettest/rmbt-server
$ cd rmbt-server/
$ sed -i 's|#define CHECK_TOKEN 1|#define CHECK_TOKEN 0|' config.h
$ openssl req -x509 -newkey rsa:4096 -keyout server.key -out
↪  server.crt -nodes -subj '/CN=localhost' -sha256 -days 10000
$ make server-prod
$ make run
```

4.3 Running Measurements

The RMBT client does a single measurement run for each execution. A measurement run consists of all the phases described in section 3.4.

The client supports a number of configuration parameters, detailed in section 4.4. Most parameters have default values, which can be displayed by invoking

```
$ rmbt -?
```

Some basic parameters can be specified as command line parameters, but all can be specified with a JSON config file with the help of the command line parameter `-c`:

```
$ rmbt -c config.json
```

The following parameters have no default values and need to be specified:

- `cnf_server_host` or `-h`

[1] The repository `https://github.com/rtr-nettest/rmbt-server` was used with commit 0f4c58698af612de1032dff4cc4c57257470053e.

- `cnf_server_port` or `-p`
- either `cnf_token` (`-t`) or `cnf_secret` (`-s`)

A sample invocation could be:

```
$ rmbt -h rmbt.example.com -p 8081 -s ''
```

Measurement campaigns conducted in the course of this thesis work were run by automated calls to the RMBT client. Some campaigns were executed via simple scripts run on the MONROE nodes, where the investigated parameters (e.g. the number of TCP flows) as well as the wait times between measurement batches were randomized. The large-scale campaigns, utilizing more than one node, were run with the MONROE scheduler.

For more information about MONROE, see section 3.7. For details about the measurement campaigns, see chapter 5.

4.4 Configuration Parameters

One of the key features of RMBT, which is missing in other existing tools, is the flexibility. It offers a wide range of test configurations to tailor the measurements to the need of the target scenario.

In order to automatize the running of the measurement client, it is vital to be able to provide it with the needed measurement parameters. As the MONROE experiments use JSON as the primary data format for configuration and data output [89], it is a natural decision to use the same data format for the reimplementation of the RMBT client. The JSON file has to comply with RFC 7159 [17]. Listing 4.1 shows an example configuration input file that we use for the client.

All configuration parameters are prefixed with `cnf_` to have a clear distinction between input and output fields. A missing configuration parameter indicates the use of the default value. All parameters except for `cnf_server_host` and `cnf_server_port` are optional and have a default value. To connect successfully to a server additionally either `cnf_secret` or `cnf_token` is required.

In the course of this thesis work, we configured the RMBT measurement servers to disable the token check (see section 4.2). Therefore, we could use an empty secret (i.e. empty string as configuration parameter `cnf_secret`) to easily measure against our server.

The following list describes each input parameter and the corresponding defaults. The *emphasized* types refer to the types described in RFC 7159 [17]:

`cnf_server_host` Required parameter without a default. *String* of the hostname or IP address of the RMBT server to connect to.

```
{
  "cnf_server_host": "192.0.2.1",
  "cnf_server_port": 8081,
  "cnf_bind_ip": null,
  "cnf_encrypt": false,
  "cnf_encrypt_debug": false,
  "cnf_cipherlist": null,
  "cnf_timeout_s": 30,
  "cnf_secret": "",
  "cnf_rtt_tcp_payload_num": 11,
  "cnf_dl_num_flows": 5,
  "cnf_ul_num_flows": 5,
  "cnf_dl_duration_s": 10,
  "cnf_ul_duration_s": 10,
  "cnf_dl_pretest_duration_s": 1,
  "cnf_ul_pretest_duration_s": 1,
  "cnf_dl_wait_time_s": 20,
  "cnf_ul_wait_time_s": 20,
  "cnf_tcp_info_sample_rate_us": 0,
  "cnf_file_summary": "{time}_{id_test}_summary.json",
  "cnf_file_flows": "{time}_{id_test}_flows.json.xz",
  "cnf_file_stats": "{time}_{id_test}_stats.json.xz",
  "cnf_add_to_result": {
  }
}
```

Listing 4.1: config.example.json

`cnf_server_port` Required parameter without a default. *String* of the port name or number of the RMBT server to connect to.

`cnf_bind_ip` *String* of the local IP of an interface to bind to. The default value is `null` indicating the use of the system default IP.

`cnf_encrypt` *Boolean* value (`true`/`false`) indicating whether or not to use TLS encryption. The default value is `false`.

`cnf_encrypt_debug` *Boolean* value (`true`/`false`) indicating whether or not to enable the output of TLS debug information (i.e. client random and master key) to the flows output file. This debug information can be used to decrypt the captured encrypted TLS connections afterwards. Only relevant if `cnf_encrypt` is `true`, otherwise it will be ignored. The default value is `false`.

`cnf_cipherlist` *String* of an OpenSSL cipher list[2] to use when connecting with TLS to a server. Only relevant if `cnf_encrypt` is `true`, otherwise it will be ignored. The default value is `null` indicating the use of the OpenSSL default cipher list.

`cnf_timeout_s` *Number* of seconds of no transmission on a TCP connection after which the measurement will abort and fail. The default value is 30 s.

[2]See the relevant OpenSSL man page with “man 1ssl ciphers” (section “CIPHER LIST FORMAT”) for a description of the format.

`cnf_secret` *String* with the secret that needs to be shared with the RMBT server. If set, the client will generate the necessary access token with this secret. The default value is `null`. Either `cnf_secret` or `cnf_token` (but not both) needs to be non-`null`.

We set this parameter to an empty string (i.e. "") as we disabled the token check in our measurement servers (see section 4.2). This effectively sets the *key* of the HMAC function to the empty string. For more information about the token generation, see section 3.4.2.

`cnf_token` *String* with the token that is used by the client to connect to the RMBT server. The default value is `null`. Either `cnf_token` or `cnf_secret` (but not both) needs to be non-`null`.

`cnf_rtt_tcp_payload_num` *Number* of TCP RTT measurements the client should do. The default value is 11.

`cnf_dl_num_flows` *Number* of TCP flows to the RMBT server the client should open in parallel and use for DL measurements. The default value is 5.

`cnf_ul_num_flows` *Number* of TCP flows to the RMBT server the client should open in parallel and use for UL measurements. The default value is 5.

`cnf_dl_duration_s` *Number* of nominal seconds for the DL measurement. The default value is 10 s.

`cnf_ul_duration_s` *Number* of nominal seconds for the UL measurement. The default value is 10 s.

`cnf_dl_pretest_duration_s` *Number* of minimal seconds for the DL pretest measurement. After reaching this number of seconds, the pretest will finish the current chunks. The default value is 2 s.

`cnf_ul_pretest_duration_s` *Number* of minimal seconds for the UL pretest measurement. After reaching this number of seconds, the pretest will finish the current chunks. The default value is 2 s.

`cnf_dl_wait_time_s` *Number* indicating the maximum number of seconds to wait in the DL measurement phase for all threads to finish transmitting the last chunk. The default value is 20 s.

`cnf_ul_wait_time_s` *Number* indicating the maximum number of seconds to wait in the UL measurement phase for all threads to finish transmitting the last chunk and waiting for the server measurement results. The default value is 20 s.

`cnf_tcp_info_sample_rate_us` *Number* of microseconds indicating the sample rate of the socket option `TCP_INFO`. The default value is `null`, which disables the sampling of `TCP_INFO`, as does the value 0.

`cnf_file_summary` *String* of a filename to write the summary JSON file to. For more information about the contents of the summary file, see section 4.5. The summary file is also written to the standard output, regardless of this parameter. If the filename ends with ".xz" and the RMBT client is compiled with `liblzma`, the file is compressed with the xz format. The default value is `null`, indicating no summary file should be written.

`cnf_file_flows` *String* of a filename to write the flows JSON file to. For more information about the contents of the flows file, see section 4.5. If the filename ends with ".xz" and the RMBT client is compiled with `liblzma`, the file is compressed with the xz format. The default value is `null`, indicating no flows file should be written.

`cnf_file_stats` *String* of a filename to write the stats JSON file to. For more information about the contents of the stats file, see section 4.5. If the filename ends with ".xz" and the RMBT client is compiled with `liblzma`, the file is compressed with the xz format. For this file to contain meaningful data, the option `cnf_tcp_info_sample_rate_us` needs to be set to a reasonable value. The default value is `null`, indicating no stats file should be written.

`cnf_add_to_result` *Object* whose content will be included in the summary and flows JSON output files. This parameter can be used to add entries to the result files. The default value is `null`, which indicates that nothing additional will be written to the result files.

4.5 Result Format

Similar to the configuration parameters, we use JSON as the data format for the result output. As we want to include a large number of time series and also samples of the socket option `TCP_INFO`, we decided to split the result output into multiple — partially optional — output files (see section 4.4 for the relevant parameters to generate these output files).

The three output files are as follows:

summary.json The summary files includes the *generic* output fields such as basic input parameters, the test identifier (ID), main timestamps, status, versions, server and connection info, total bytes transferred, and system information. Additionally to the *generic* fields, it also includes specific or calculated values such as DL and UL throughput, actual number of flows, and the median TCP payload RTTs. The contents of the summary file are always output to the standard output by the RMBT client, regardless of the creation of the summary file on disk. An example summary output file can be found in listing 4.2 (`summary.json`). In section 4.5.1, we describe the fields of this file in more detail.

flows.json The flows file includes the *generic* output fields and additionally all the raw per-flow time series for all measurement phases (init, latency, pretest_dl, dl, pretest_ul, ul). It does not duplicate the values which are calculated from the raw data. A trimmed example flows output file can be found in listing 4.3 (`flows.json`). The dots (...) are not part of the files but rather indicate left out parts. In section 4.5.2, we describe the fields of this file in more detail.

stats.json If activated in the configuration, the stats file includes samples of the socket option `TCP_INFO` every `cnf_tcp_info_sample_rate_us` µs for each TCP flow. We found 10 ms (i.e. 100 Hz) to be a reasonable sample rate, balancing resolution against load and result file size. A trimmed example showing only one sample can be found in listing 4.5 (`stats.json`). In section 4.5.3, we describe the fields of this file in more detail.

4.5.1 Summary Output File

The summary file is a JSON file and can be generated by setting the configuration parameter `cnf_file_summary` (see section 4.4). Additionally to the optional generation of the file, the RMBT client will output the contents of this file to the standard output. An example of this file can be found in listing 4.2.

The following list describes each field found in the summary file. The *emphasized* types refer to the types described in RFC 7159 [17]:

`res_id_test` *String* containing the UUID [46] of the test, which is also part of the token and uniquely identifies a single measurement.

`res_time_start_s` *Number* of seconds since the epoch (as defined by sections 3.150 and 4.16 of POSIX.1-2008 [34]) indicating the start of the measurement run See also glossary entry of "UNIX timestamp".

`res_time_end_s` *Number* of seconds since the epoch indicating the end of the measurement run.

`res_status` *String* indicating if the measurement run was successful. Its value is `"success"` if the run was successful, or `"fail_<phase>"` with `<phase>` indicating the measurement phase in which the run failed, in case of failure.

`res_status_msg` *String* indicating the reason of the failure. Its value is `null` if there was no failure.

`res_version_client` *String* indicating the version of the client.

`res_version_server` *String* indicating the version of the server.

`res_server_ip` *String* indicating the IP of the server to which the client connected.

```
{
  "res_id_test": "cdd28b7e-82a5-45f1-911a-1c4fd698ad28",
  "res_time_start_s": 1501568268,
  "res_time_end_s": 1501568298,
  "res_status": "success",
  "res_status_msg": null,
  "res_version_client": "v1.0-11-gcd544e53c",
  "res_version_server": "RMBTv0.3",
  "res_server_ip": "192.0.2.1",
  "res_server_port": 10080,
  "res_encrypt": false,
  "res_chunksize": 4096,
  "res_tcp_congestion": "cubic",
  "res_total_bytes_dl": 80343013,
  "res_total_bytes_ul": 11334715,
  "res_uname_sysname": "Linux",
  "res_uname_nodename": "af871b0b2239",
  "res_uname_release": "4.9.0-2-amd64",
  "res_uname_version": "#1 SMP Debian 4.9.18-1 (2017-03-30)",
  "res_uname_machine": "x86_64",
  "res_rtt_tcp_payload_num": 11,
  "res_rtt_tcp_payload_client_ns": 61378328,
  "res_rtt_tcp_payload_server_ns": 66762222,
  "res_dl_num_flows": 5,
  "res_dl_time_ns": 10544056131,
  "res_dl_bytes": 78895006,
  "res_dl_throughput_kbps": 59859.321703,
  "res_ul_num_flows": 5,
  "res_ul_time_ns": 11960828535,
  "res_ul_bytes": 9637026,
  "res_ul_throughput_kbps": 6445.724707
}
```


Listing 4.2: summary.json

res_server_port *Number* indicating the port of the server to which the client connected.

res_encrypt *Boolean* value (`true`/`false`) indicating whether encryption was used for the connection to the server.

res_chunksize *Number* of bytes, indicating the chunksize used during the measurement. See also section 3.4.1 for more details.

res_tcp_congestion *String* containing the used TCP congestion control algorithm [9] on the client side. This value is acquired by reading the TCP socket option `TCP_CONGESTION`.

res_total_bytes_dl *Number* of bytes the client has received over the TCP resp. TLS connection in total during the whole measurement run. (i.e., the number of bytes on OSI layer 7 are counted [37].)

res_total_bytes_ul *Number* of bytes the client has transmitted over the TCP resp. TLS connection in total during the whole measurement run. (i.e., the number of bytes on OSI layer 7 are counted [37].)

res_uname_sysname *String* containing the field `sysname` from the `uname` syscall.

res_uname_nodename *String* containing the field `nodename` from the `uname` syscall.

res_uname_release *String* containing the field `release` from the `uname` syscall.

res_uname_version *String* containing the field `version` from the `uname` syscall.

res_uname_machine *String* containing the field `machine` from the `uname` syscall.

res_rtt_tcp_payload_num *Number* of RTT measurements that were performed over TCP. See section 3.4.4 for details about the latency phase.

res_rtt_tcp_payload_client_ns *Number* indicating the median of the measured RTT over TCP from the client side. See section 3.4.4 for details about the latency phase.

res_rtt_tcp_payload_server_ns *Number* indicating the median of the measured RTT over TCP from the server side. See section 3.4.4 for details about the latency phase.

res_dl_num_flows *Number* of TCP flows that were used in parallel for the DL measurement. See section 3.4.5 for details about the downlink phase.

res_dl_time_ns *Number* of nanoseconds calculated according to equation (3.1) for the DL measurement (see section 3.5.1). See section 3.4.5 for details about the downlink phase.

res_dl_bytes *Number* of bytes calculated according to equation (3.3) for the DL measurement. See section 3.4.5 for details about the downlink phase.

res_dl_throughput_kbps *Number* indicating the calculated DL throughput according to equation (3.4) in kbit/s (see section 3.5.1). See section 3.4.5 for details about the downlink phase.

res_ul_num_flows *Number* of TCP flows that were used in parallel for the UL measurement. See section 3.4.7 for details about the downlink phase.

res_ul_time_ns *Number* of nanoseconds calculated according to equation (3.1) for the DL measurement (see section 3.5.1). See section 3.4.7 for details about the uplink phase.

res_ul_bytes *Number* of bytes calculated according to equation (3.3) for the UL measurement (see section 3.5.1). See section 3.4.7 for details about the uplink phase.

```
{
  "res_id_test": "cdd28b7e-82a5-45f1-911a-1c4fd698ad28",
  "res_time_start_s": 1501568268,
  "res_time_end_s": 1501568298,
  "res_status": "success",
  "res_status_msg": null,
  "res_version_client": "v1.0-11-gcd544e53c",
  "res_version_server": "RMBTv0.3",
  "res_server_ip": "192.0.2.1",
  "res_server_port": 10080,
  "res_encrypt": false,
  "res_chunksize": 4096,
  "res_tcp_congestion": "cubic",
  "res_total_bytes_dl": 80343013,
  "res_total_bytes_ul": 11334715,
  "res_uname_sysname": "Linux",
  "res_uname_nodename": "af871b0b2239",
  "res_uname_release": "4.9.0-2-amd64",
  "res_uname_version": "#1 SMP Debian 4.9.18-1 (2017-03-30)",
  "res_uname_machine": "x86_64",
  "res_details": { ... }
}
```

Listing 4.3: flows.json

res_ul_throughput_kbps *Number* indicating the calculated UL throughput according to equation (3.4) in kbit/s (see section 3.5.1). See section 3.4.7 for details about the uplink phase.

4.5.2 Flows Output File

The flows file is a JSON file and can be generated by setting the configuration parameter `cnf_file_flows` (see section 4.4). An example of this file can be found in listing 4.3.

The *object* "res_details" can be found in listing 4.4. As the flows files tend to get very large, the example provided in this thesis is trimmed to show only the general structure of the file. The dots (. . .) indicate cut out parts.

The following lists describe the fields found in the flows file.

The following fields are identical to the summary file. The fields from the summary file — which can be calculated using the raw data in `res_details` — are not duplicated in the flows file. For more information on these fields, see section 4.5.1.

res_id_test
res_time_start_s
res_time_end_s
res_status
res_status_msg
res_version_client
res_version_server

```
{
  "rtt_tcp_payload": {
    "values": [
      {
        "time_start_rel_ns": 1540754595,
        "rtt_server_ns": 66762222,
        "rtt_client_ns": 61378328,
        "time_end_rel_ns": 1669114294
      },...
    ]
  },
  "init": [
    {
      "client_port": 60433
    },...
  ],
  "pretest_dl": [
    {
      "time_start_rel_ns": 454937517,
      "time_end_rel_ns": 1540181544,
      "time_series": [
        {
          "t_begin": 751,
          "b": 8192,
          "t_end": 207189496,
          "d_server": 203109861
        },...
      ]
    },...
  ],
  "dl": [
    {
      "time_start_rel_ns": 4290124771,
      "time_end_rel_ns": 14939031644,
      "duration_server_ns": 10655883562,
      "time_series": [
        {
          "t": 69256776,
          "b": 4096
        },...
      ]
    },...
  ],
  "pretest_ul": [
    {
      "time_start_rel_ns": 15127533066,
      "time_end_rel_ns": 16439620688,
      "time_series": [
        {
          "t_begin": 77954672,
          "b": 8192,
          "t_end": 165685497,
          "d_server": 88485637
        },...
      ]
    },...
  ],
  "ul": [
    {
      "time_start_rel_ns": 16440146277,
      "time_end_rel_ns": 29817469811,
      "time_series": [
        {
          "t": 134584836,
          "b": 1408
        },...
      ]
    }
  ]
}
```


Listing 4.4: "res_details" of flows.json

res_server_ip
res_server_port
res_encrypt
res_chunksize
res_tcp_congestion
res_total_bytes_dl
res_total_bytes_ul
res_uname_sysname
res_uname_nodename
res_uname_release
res_uname_version
res_uname_machine

The *object* **res_details** consists of the following keys. The *emphasized* types refer to the types described in RFC 7159 [17]. See also section 3.4 for more information on the fields.

rtt_tcp_payload *Object* with an entry `values` containing an *array* of *objects* with the following entries. Each *array* entry represents a single RTT measurement over TCP. For more information regarding the fields, see also figure 3.5:

- **time_start_rel_ns** *Number* representing the start time of this RTT measurement as relative time in nanoseconds since `res_time_start_s`.
- **rtt_server_ns** *Number* in nanoseconds representing the measured RTT over TCP from the server's point of view.
- **rtt_client_ns** *Number* in nanoseconds representing the measured RTT over TCP from the client's point of view.
- **time_end_rel_ns** *Number* representing the end time of this RTT measurement as relative time in nanoseconds since `res_time_start_s`.

init *Array* of *objects* containing the following field. There is one *object* per TCP flow:

- **client_port** *Number* representing the TCP port number on the client side.

pretest_dl *Array* of *objects* with the following fields. There is one *object* per TCP flow used for the pretest DL measurement. For more information regarding the fields, see also figure 3.4:

- **time_start_rel_ns** *Number* representing the start time of this pretest DL measurement as relative time in nanoseconds since `res_time_start_s`.
- **time_end_rel_ns** *Number* representing the end time of this pretest DL measurement as relative time in nanoseconds since `res_time_start_s`.

time_series *Array* of *objects* with the following fields. Each *object* represents a single block during the pretest DL measurement on one TCP flow:

t_begin *Number* representing the time in nanoseconds since the start of the pretest DL measurement (represented by `time_start_rel_ns`) indicating the beginning of this block. The timestamp is collected on the client side.

b *Number* of bytes the client has received between `t_begin` and `t_end`.

t_end *Number* representing the time in nanoseconds since the start of the pretest DL measurement (represented by `time_start_rel_ns`) indicating the end of this block. The timestamp is collected on the client side.

d_server *Number* representing the amount of time the transmission of this block took from the view of the server, in nanoseconds. The corresponding duration on the client can be calculated by subtracting `t_begin` from `t_end`.

dl *Array* of *objects* with the following fields. There is one *object* per TCP flow used for the DL measurement. For more information regarding the fields, see also figure 3.6:

time_start_rel_ns *Number* representing the start time of this DL measurement as relative time in nanoseconds since `res_time_start_s`.

time_end_rel_ns *Number* representing the end time of this DL measurement as relative time in nanoseconds since `res_time_start_s`.

duration_server_ns *Number* representing the duration of the whole DL measurement as seen by the server. The corresponding time on the client side and the total number of bytes transmitted can be found by looking at the last sample in `time_series`.

time_series *Array* of *objects* with the following fields. Each *object* represents a single sample during the DL measurement on one TCP flow:

t *Number* in nanoseconds since the start of the DL measurement (represented by `time_start_rel_ns`) for this sample. The timestamp is collected on the client side.

b *Number* of bytes the client has received at time `t`.

pretest_ul *Array* of *objects* with the following fields. There is one *object* per TCP flow used for the pretest UL measurement. For more information regarding the fields, see also figure 3.7:

time_start_rel_ns *Number* representing the start time of this pretest UL measurement as relative time in nanoseconds since `res_time_start_s`.

time_end_rel_ns *Number* representing the end time of this pretest UL measurement as relative time in nanoseconds since `res_time_start_s`.

time_series *Array* of *objects* with the following fields. Each *object* represents a single block during the pretest UL measurement on one TCP flow:

- **t_begin** *Number* representing the time in nanoseconds since the start of the pretest UL measurement (represented by `time_start_rel_ns`) indicating the beginning of this block. The timestamp is collected on the client side.
- **b** *Number* of bytes the client has sent between `t_begin` and `t_end`.
- **t_end** *Number* representing the time in nanoseconds since the start of the pretest UL measurement (represented by `time_start_rel_ns`) indicating the end of this block. The timestamp is collected on the client side.
- **d_server** *Number* representing the time the transmission of this block took from the view of the server in nanoseconds. The corresponding time on the client can be calculated by subtracting `t_begin` from `t_end`.

ul *Array* of *objects* with the following fields. There is one *object* per TCP flow used for the UL measurement. For more information regarding the fields, see also figure 3.8:

- **time_start_rel_ns** *Number* representing the start time of this UL measurement as relative time in nanoseconds since `res_time_start_s`.
- **time_end_rel_ns** *Number* representing the end time of this UL measurement as relative time in nanoseconds since `res_time_start_s`.
- **time_series** *Array* of *objects* with the following fields. Each *object* represents a single sample during the UL measurement on one TCP flow. These samples are measured and reported by the server:
  - **t** *Number* in nanoseconds since the start of the UL measurement (represented by `time_start_rel_ns`) for this sample. The timestamp is collected on the server side.
  - **b** *Number* of bytes the server has received at time `t`.

4.5.3 Stats Output File

The stats file is a JSON file which can be generated by setting the configuration parameter `cnf_file_stats` (see section 4.4). An example of this file can be found in listing 4.5. As the stats files tend to get very large, the example provided in this thesis is trimmed to show only a single entry. The full file has an entry every `cnf_tcp_info_sample_rate_us` µs for every TCP flow.

Each entry in the stats file consists of a sample of the `struct tcp_info`[3] which is acquired by repeated calls to `getsockopt` with the `optname` `TCP_INFO` for each TCP flow.

[3] see `struct tcp_info` in `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/tcp.h`

```
[
  {
    "flow_id": 0,
    "timestamp_ns": 63131809,
    "tcpi_state": 2,
    "tcpi_ca_state": 0,
    "tcpi_retransmits": 0,
    "tcpi_probes": 0,
    "tcpi_backoff": 0,
    "tcpi_options": 0,
    "tcpi_snd_wscale": 0,
    "tcpi_rcv_wscale": 0,
    "tcpi_delivery_rate_app_limited": 0,
    "tcpi_rto": 1000000,
    "tcpi_ato": 0,
    "tcpi_snd_mss": 524,
    "tcpi_rcv_mss": 88,
    "tcpi_unacked": 1,
    "tcpi_sacked": 0,
    "tcpi_lost": 0,
    "tcpi_retrans": 0,
    "tcpi_fackets": 0,
    "tcpi_last_data_sent": 2483277360,
    "tcpi_last_ack_sent": 0,
    "tcpi_last_data_recv": 2483277360,
    "tcpi_last_ack_recv": 2483277360,
    "tcpi_pmtu": 1500,
    "tcpi_rcv_ssthresh": 29200,
    "tcpi_rtt": 0,
    "tcpi_rttvar": 250000,
    "tcpi_snd_ssthresh": 2147483647,
    "tcpi_snd_cwnd": 10,
    "tcpi_advmss": 1460,
    "tcpi_reordering": 3,
    "tcpi_rcv_rtt": 0,
    "tcpi_rcv_space": 0,
    "tcpi_total_retrans": 0,
    "tcpi_pacing_rate": -1,
    "tcpi_max_pacing_rate": -1,
    "tcpi_bytes_acked": 0,
    "tcpi_bytes_received": 0,
    "tcpi_segs_out": 1,
    "tcpi_segs_in": 0,
    "tcpi_notsent_bytes": 0,
    "tcpi_min_rtt": 4294967295,
    "tcpi_data_segs_in": 0,
    "tcpi_data_segs_out": 0,
    "tcpi_delivery_rate": 0
  },
  ...
]
```


Listing 4.5: stats.json

The following list describes each field found in the stats file. All fields are of RFC 7159 [17] type *Number*. The fields are described as they can be retrieved from the Linux kernel. Other systems (e.g. Berkeley Software Distribution (BSD) derivatives) also support the `TCP_INFO` socket option, but may only support a subset of the fields and fields may have a different meaning as the TCP implementations differ. Paths refer to files in the Linux kernel sources as of version 4.13.

`flow_id` ID of the TCP flow the sample was collected for. The first flow gets ID 0. If a TCP flow needs to be re-opened during a measurement run (see algorithm 3.4), it also gets a new ID. With this information, it can be determined if and which flow was reused in the uplink phase.

`timestamp_ns` Timestamp of the acquired sample in nanoseconds, relative to the start of the measurement (field `res_time_start_s`; see section 4.5.1).

`tcpi_state` State of the TCP connection.
The connection states are defined in `include/net/tcp_states.h`[4]:

- `TCP_ESTABLISHED`
- `TCP_SYN_SENT`
- `TCP_SYN_RECV`
- `TCP_FIN_WAIT1`
- `TCP_FIN_WAIT2`
- `TCP_TIME_WAIT`
- `TCP_CLOSE`
- `TCP_CLOSE_WAIT`
- `TCP_LAST_ACK`
- `TCP_LISTEN`
- `TCP_CLOSING`
- `TCP_NEW_SYN_RECV`

`tcpi_ca_state` Bit field describing the state of the congestion control.
See `enum tcp_ca_state` in `include/uapi/linux/tcp.h`[5]. The following bits can be set:

- `TCPF_CA_Open`
- `TCPF_CA_Disorder`

[4] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/net/tcp_states.h`

[5] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/tcp.h`

- TCPF_CA_CWR
- TCPF_CA_Recovery
- TCPF_CA_Loss

tcpi_retransmits Number of unrecovered Retransmit Timeouts (RTOs).
See icsk_retransmits in struct inet_connection_sock in include/net/inet_connection_sock.h[6].

tcpi_probes Unanswered 0 window probes.
See icsk_probes_out in struct inet_connection_sock in include/net/inet_connection_sock.h[6].

tcpi_backoff Exponential backoff.
See icsk_backoff in include/net/inet_connection_sock.h[6].

tcpi_options Bit field consisting of the following fields (defined in include/uapi/linux/tcp.h[5]):

- TCPI_OPT_TIMESTAMPS
- TCPI_OPT_SACK
- TCPI_OPT_WSCALE
- TCPI_OPT_ECN
- TCPI_OPT_ECN_SEEN
- TCPI_OPT_SYN_DATA

tcpi_snd_wscale Window scaling received from sender.
See snd_wscale in struct tcp_options_received in include/linux/tcp.h[7].

tcpi_rcv_wscale Window scaling to send to receiver.
Only set if TCPI_OPT_WSCALE is set in tcpi_options (i.e. Selective Acknowledgment (SACK) was seen on SYN packet). See rcv_wscale in struct tcp_options_received in include/linux/tcp.h[7].

tcpi_delivery_rate_app_limited A *Boolean* value (true/false) indicating if the tcpi_delivery_rate was measured when the socket's throughput was limited by the sending application.
See function tcp_rate_check_app_limited in net/ipv4/tcp_rate.c[8].

[6] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/net/inet_connection_sock.h
[7] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/tcp.h
[8] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/tcp_rate.c

tcpi_rto Retransmit Timeout (RTO) in µs.
See `icsk_rto` in `struct inet_connection_sock` in `include/net/inet_connection_sock.h`[6].

tcpi_ato Acknowledgment (ACK) Timeout (ATO).
See `ato` in `icsk_ack` in `struct inet_connection_sock` in `include/net/inet_connection_sock.h`[6].

tcpi_snd_mss Cached effective MSS, not including SACKs. See `mss_cache` in `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_rcv_mss MSS used for delayed ACK decisions.
See `rcv_mss` in `icsk_ack` in `struct inet_connection_sock` in `include/net/inet_connection_sock.h`[6].

tcpi_unacked Packets which are "in flight".
See `packets_out` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_sacked SACK'd packets [58].
See `sacked_out` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_lost Lost packets.
See `lost_out` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_retrans Retransmitted packets out.
See `retrans_out` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_fackets Forward Acknowledgment (FACK)'d packets.
See `fackets_out` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_last_data_sent Timestamp of last sent data packet in µs.
See `lsndtime` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_last_ack_sent Not implemented in Linux.
See `include/uapi/linux/tcp.h`[5].

tcpi_last_data_recv Timestamp of last received data packet in µs.
See `lrcvtime` in `icsk_ack` in `struct inet_connection_sock` in `include/net/inet_connection_sock.h`[6].

tcpi_last_ack_recv Timestamp of last received ACK in µs.
See `rcv_tstamp` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_pmtu Last Path MTU Discovery (PMTU) seen by socket.
See `icsk_pmtu_cookie` in `struct inet_connection_sock` in `include/net/inet_connection_sock.h`[6].

tcpi_rcv_ssthresh Current window clamp.
See `rcv_ssthresh` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_rtt Smoothed RTT in µs.
See `srtt_us` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_rttvar RTT medium deviation.
See `mdev_us` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_snd_ssthresh Slow start size threshold.
See `snd_ssthresh` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_snd_cwnd Sending congestion window.
See `snd_cwnd` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_advmss Advertised MSS.
See `advmss` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_reordering Packet reordering metric.
See `reordering` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_rcv_rtt Estimated receiver side RTT in µs.
See `rcv_rtt_est` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_rcv_space Receiver queue space.
See `space` of `rcvq_space` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_total_retrans Total retransmits for entire connection.
See `total_retrans` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_pacing_rate Pacing rate[9] in B/s.
See `sk_pacing_rate` of `struct sock` in `include/net/sock.h`[10].

tcpi_max_pacing_rate Maximum pacing rate[9].
See `sk_max_pacing_rate` of `struct sock` in `include/net/sock.h`[10].

tcpi_bytes_acked Total number of ACK'd bytes.
`tcpEStatsAppHCThruOctetsAcked` as defined in RFC 4898 [57].
See `bytes_acked` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_bytes_received Total number of received bytes.
`tcpEStatsAppHCThruOctetsReceived` as defined in RFC 4898 [57].
See `bytes_received` of `struct tcp_sock` in `include/linux/tcp.h`[7].

tcpi_segs_out Total number of segments sent.
`tcpEStatsPerfSegsOut` as defined in RFC 4898 [57].
See `segs_out` of `struct tcp_sock` in `include/linux/tcp.h`[7].

[9] Pacing is a technology to reduce the burstiness of TCP. By default, TCP pacing is effectively disabled in the Linux kernel.

[10] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/net/sock.h`

tcpi_segs_in Total number of segments received.
tcpEStatsPerfSegsIn as defined in RFC 4898 [57].
See segs_in of struct tcp_sock in include/linux/tcp.h[7].

tcpi_notsent_bytes Number of bytes in send buffer.
Calculated as write_seq – snd_nxt.
See write_seq and snd_nxt of struct tcp_sock in include/linux/tcp.h[7].

tcpi_min_rtt Minimum observed RTT in µs.
See function tcp_min_rtt in include/net/tcp.h[11].

tcpi_data_segs_in Total number of data segments received.
tcpEStatsDataSegsIn as defined in RFC 4898 [57]. See data_segs_in of struct tcp_sock in include/linux/tcp.h[7].

tcpi_data_segs_out Total number of data segments sent.
tcpEStatsDataSegsOut as defined in RFC 4898 [57]. See data_segs_out of struct tcp_sock in include/linux/tcp.h[7].

tcpi_delivery_rate Delivery rate in bytes per second.
The most recent throughput, as measured by tcp_rate_gen() in net/ipv4/tcp_rate.c[8].

[11] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/net/tcp.h

CHAPTER 5

Measurements and Evaluation

As introduced in chapter 1, one of the main objectives of this thesis is to test the implemented client on operational MBB networks and deploy a set of automated measurements. These measurements allow to (i) verify the performance of the implementation, and (ii) to prove the suitability of such a measurement tool.

We make use of the MONROE platform (see section 3.7) to perform an extensive set of measurement campaigns in order to investigate a number of parameters listed in chapter 2.

This chapter introduces the measurement campaigns, each of them targeting different objectives and analyzes the influence of different parameters on the measurement of achievable network throughput.

5.1 Measurement Campaign Overview

In the course of this work, we have conducted numerous measurement campaigns, ranging from a few samples to thousands of samples in size. This section gives an overview of a chosen subset of these campaigns, which includes only the most relevant campaigns with at least 100 samples. We go into more detail regarding the investigated topics in Section 5.3.

Figure 5.1 presents how the MONROE platform was used to conduct the RMBT measurements. The RMBT client ran on MONROE nodes for all the campaigns mentioned in this thesis. For more information about MONROE, see section 3.7.

Table 5.1 presents the list of the selected measurement campaigns. Mentioned dates and times are in Coordinated Universal Time (UTC). The table consists of the following columns:

ID	Start Date	BS	TS	Batches	Samples	Technology	Operator	Server	Flows	Dur.
1	2017-03-29	2	13 h	996	1993	Ethernet	N/A	T430s	3	7 s
2	2017-03-31	2	13 h	999	1998	Ethernet	N/A	T430s	3	7 s
3	2017-04-03	2	13 h	997	1995	Ethernet	N/A	T430s	3	7 s
4	2017-04-05	2	12 h	999	1999	Ethernet	N/A	Node	3	7 s
5	2017-04-26	2	14 h	50	100	4G	NR	.se	3	7 s
6	2017-04-27	7	17 h	30	210	4G	NA	.se	1-7	7 s
7	2017-04-28	5	26 h	59	269	4G	NA	.se	3-7	10 s
8	2017-05-03	8	17 h	38	304	4G	NA	.de, .se	3,5,7,9	15 s
9	2017-05-04	8	22 h	49	389	4G	NR	.de, .se	3,5,7,9	15 s
10	2017-05-10	4	9 h	35	140	Fiber	NK	.se	3,5,7,9	10 s
11	2017-05-12	7	49 h	50	353	Fiber	NK	.se	3,10,15,25, 50,100,199	10 s
12	2017-05-17	8	50 h	38	317	4G	NA	.de, .se	3,5,7,9	15 s
13	2017-05-19	12	10 h	10	100	4G	NA	.de, .se	1,3,4,5,7,9	15 s
14	2017-05-24	12	95 h	266	4785	4G	S3,NA,SA, NI,NR,SR	1.no, .se	1,3,4,5,7,9	15 s
15	2017-06-08	12	99 h	176	1504	4G	S3,NA,SA, NI,NR,SR	1.no, .se	1,3,4,5,7,9	15 s
16	2017-07-04	4	300 h	508	2151	4G	S3,NA,SA, NI,NR,SR	.se	1,3,5,7	15 s
17	2017-07-22	4	375 h	714	2973	4G	S3,NA,SA, NI,NR,SR	.se	1,3,5,7	15 s
18	2017-11-07	12	240 h	638	6668	4G	S3,NA,SA, NI,NR,SR	2.no, .se	3,5,7,9, 12,20	15 s

Table 5.1: Measurement Campaigns

ID The number used throughout this thesis to uniquely identify the measurement campaign.

Start Date The date of the first measurement of the campaign.

BS (Batch size) The number of measurements per batch. A batch consist of one complete set of measurements. For instance, in the case of the comparison of different number of flows and servers, a batch includes one sample for each number of flows for each server.

TS (Time Span) The approximate time span in hours between the first and the last measurement of the campaign.

Batches The number of batches in the campaign. Incomplete batches due to failed single measurements are discarded and are not included in this number.

Samples Total number of successful measurements in this campaign. This number also includes measurements that are part of incomplete batches, which are discarded for the analysis.

Technology The connection technology that was used by the measurement client while conducting the measurements. Can be *Ethernet* for a direct Ethernet connection between the respective interfaces of two nodes, *Fiber* for optical fiber providing

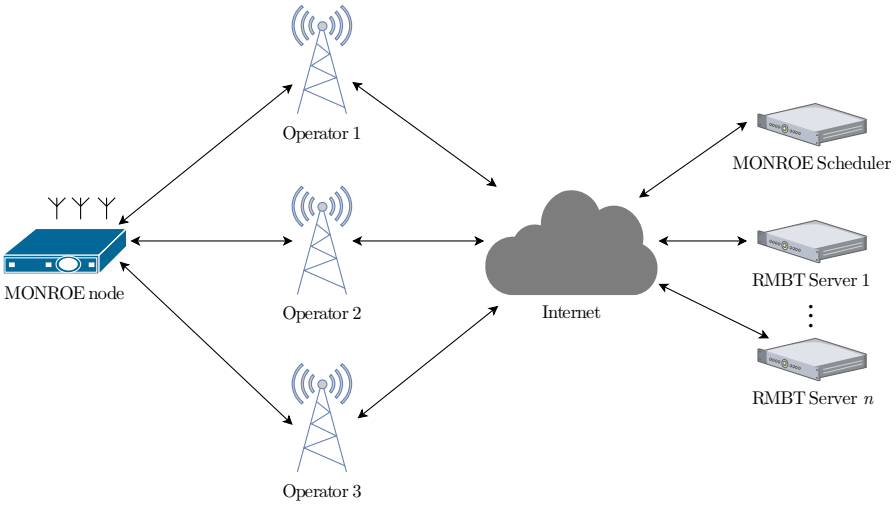


Figure 5.1: Overview of the measurement setup within the MONROE project

a 1 Gbit/s connection over the Internet, or *4G* for 4G/LTE connection over the Internet.

Operator An abbreviation for the Mobile Network Operator (MNO) that was used on the client side for the connection to the Internet. *N/A* for "not applicable" in the case of a direct Ethernet connection, not involving the Internet.

Server The distinct server(s) used in the campaign. The Top-Level Domain (TLD) of the location of the server is used. *.de* is used for the server in Germany, *1.no* for the first server in Norway, *2.no* for the second server in Norway and *.se* for the server in Sweden. See `cnf_server_host` in section 4.4 for details.

Flows List of distinct number of TCP flows for DL and UL measurements used in the campaign. See `cnf_dl_num_flows` and `cnf_ul_num_flows` in section 4.4 for details.

Dur. The nominal measurement duration used for DL and UL measurements. See `cnf_dl_duration_s` and `cnf_ul_duration_s` in section 4.4 for details.

The campaigns 1–4 gather measurements in a controlled environment with known ground truth. The client and the server were connected via a direct 1 Gbit/s Ethernet connection. No other network equipment like switches, routers or similar was used. The nominal measurement duration was set to 7 s and 3 parallel TCP flows were used. All the campaigns include 1000 batches consisting of 2 runs of the measurement client: One run with encryption disabled and another run with encryption enabled. In campaigns 1–3, a Lenovo ThinkPad T430s, running Ubuntu 17.04, acted as RMBT server, running solely

the server. All other unneeded services were stopped. In campaign 4, another MONROE node acted as a server.

- The purpose of campaigns **1** and **2** was to investigate the effect of background services running on the MONROE node. Campaign 1 ran with background services enabled while campaign 2 ran with background services disabled. Section 5.3.1 covers the influence of background services in more detail.
- The purpose of campaign **3** was to investigate the effect of the TCP Segmentation Offload (TSO) feature of the Ethernet network card of the MONROE node. Therefore, this campaign was run with TSO turned off on the node (i.e. client side). Section 5.3.2 goes into detail regarding the overhead of Ethernet, IP, and TCP.
- In campaign **4**, we investigated the overhead added by encryption and both client and server were MONROE nodes. Section 5.3.4 covers the overhead of encryption in more detail.

For all campaigns starting from campaign 5, the measurement server(s) were connected to the Internet. A server was set up in Sweden (denoted *.se*), hosted by *Karlstad University*, to be used as the primary measurement server for all future campaigns. During the campaigns the server was solely targeted by our measurements, therefore making the need for further server-side scheduling of the measurements unnecessary, to avoid overload scenarios, which could influence the measurement result [66, 98].

The purpose of campaign **5** was to investigate the effect of running MobileInsight [51] in the background. It was also the first larger campaign run over a commercial mobile network.

Campaign **6** was the first campaign with the intent of investigating the influence of the number of parallel TCP flows on the result. All possible flows from 1–7 were used in random order per batch. The nominal measurement duration was still set to 7 s. Section 5.3.5 covers the influence of the number of parallel TCP flows and the measure duration in more detail.

In campaign **7**, the nominal measurement duration was increased to 10 s. All possible flows from 3–7 were used in random order per batch.

For further campaigns, another server in Germany (denoted by *.de*) was installed on a dedicated server hosted by *Hetzner Online GmbH* to be used as an alternative target. Section 5.3.6 covers the influence of the server location in more detail.

As we saw in previous campaigns that a nominal measurement duration of 10 s might not be enough to fully saturate the bandwidth and reach a stable throughput — especially with lower number of flows — the nominal measurement duration for most of the following campaigns was increased to 15 s.

Campaigns **8** and **9** were the first campaigns to utilize multiple servers. The measurements were done alternating to the servers in Sweden and Germany, denoted by .se and .de in the overview table, respectively. The set of flows was set to 3, 5, 7, and 9 to decrease the data quota used on the mobile subscriptions.

During all mobile measurement campaigns, the maximum available data quota was always a limiting factor on the extent of parameters we could possibly investigate. Unlimited mobile subscriptions would have been optimal for the course of this work. Unfortunately, we had no access to unlimited mobile subscriptions.

Campaigns **10** and **11** were run over a fixed line network connected over a 1 Gbit/s fiber connection provided by *Kvantel AS* to the *Simula Research Laboratory* premises. As we saw faster stabilization of the throughput on fixed line networks, the nominal measurement duration was set to 10 s for these campaigns. In campaign 10, the set of flows was still set to 3, 5, 7, and 9. For campaign 11, this set was set to 3, 10, 15, 25, 50, 100, and 199 to investigate the effect of a very high number of parallel flows. 199 was used as the highest number as the server in Sweden was at that time limited to a (default) total number of 200 flows. It was made sure that the server was not in use by other measurements at the same time.

Campaign **12** was run with the same set of parameters as campaign 8, but over a longer timespan.

Campaign **13** was a short campaign to investigate a different set of parallel TCP flows. In contrast to campaign 12, the set of flows was increased to 1, 3, 4, 5, 7, and 9.

For further measurements, a server in Norway (denoted by *1.no*) was installed on a virtual machine hosted at *Simula Research Laboratory* and connected by *University of Oslo*.

Campaigns **14** and **15** were the first campaigns to include more than one operator on the client side and were run concurrently in Sweden and in Norway. The campaigns utilized nodes with Subscriber Identity Module (SIM) cards of three Swedish and three Norwegian operators. All measurements were done against both the servers in Sweden and in Norway. The set of investigated parallel TCP flows was 1, 3, 4, 5, 7, and 9. The measurements were scheduled in a random fashion. Each measurement batch, consisting of the whole set of different flows and servers, was run back to back without any artificial delay between the measurement runs. The order inside the batch was randomized to avoid any systematic bias. After each batch there was a random delay between 1800 s and 3600 s to minimize the effect of any temporal patterns. See section 5.3.7 for a discussion of temporal patterns in MBB networks.

We decided to run the next campaigns over a longer time period than previous ones. So far campaign 15 was the longest running campaign with a timespan of around 100 hours (i.e. around 4 days). Campaigns **16** and **17** were run over a timespan of around 300 resp. 375 hours (i.e. around 13 resp. 16 days). To be able to keep the data quota usage at an

acceptable level, we decided to reduce the number of investigated parameters. Therefore it was run with a set of 1, 3, 5, and 7 flows and only using the Swedish server.

For the campaign **18**, another server in Norway (denoted by *2.no*) was installed on a virtual machine hosted at *Simula Research Laboratory* and connected by *University of Oslo*, to be used as the measurement server for Norwegian nodes. The investigated set of flows was increased to 3, 5, 7, 9, 12, and 20.

More detailed data for each of the campaigns presented in this chapter can be found in appendix A.

5.2 Measurement Characteristics

We have analyzed the measurements conducted over operational MBB networks to give some insights into the characteristics of a measurement. All measurement campaigns listed in table 5.1 except for campaigns 1–4, 10, and 11 were conducted over operational MBB networks.

Between April and November 2017, we conducted a total of 19 871 measurements in 4G/LTE networks in the course of these campaigns. The mean total duration of a single measurement is 42.27 s. The mean duration of the DL resp. UL phase is 16.60 s resp. 17.50 s.

Six different MBB networks were used to conduct these measurements, namely: *Telenor*, *Telia*, and *ice.net* in Norway and *Telenor*, *Telia*, and *3* in Sweden.

In total, we used 2907 GB of mobile data (without overhead; see section 5.3.2). A single measurement had a mean data usage of 108 MB in the DL and 39 MB in the UL direction.

The minimum and maximum observed DL throughput are 60 kbit/s and 196.46 Mbit/s, respectively. The minimum and maximum observed UL throughput are 50 kbit/s and 48.44 Mbit/s, respectively. The median observed DL throughput is 43.63 Mbit/s and the median observed UL throughput is 14.6 Mbit/s.

In all measurements conducted in the course of this thesis work, we left the TCP congestion control algorithm at the current Linux default of *CUBIC* for both, client and server [47]. The version of the Linux kernel utilized for the measurements was “4.9.0”.

The total, compressed result data of a single measurement adds up to around 2 MB, which includes the detailed time series (see section 4.5.2) and `TCP_INFO` data with a resolution of 10 ms (see section 4.5.3).

In total, we collected around 33 GB of result data.

Failed measurements (i.e. connection problems during the measurement) were not counted in the numbers represented above.

5.3 Result Evaluation

5.3.1 Influence of Background Services

The MONROE nodes have a number of services running in the background. Examples of these services include the scheduling client, *metadata broadcasting service*, *Tstat* passive probe software,[1] and various system tools like *cron*[2] [6].

In campaigns 1 and 2, we were investigating the effect of these background services on the measurements in a controlled setup. These campaigns were therefore performed with a dedicated, direct 1 Gbit/s Ethernet connection between the server and the client not involving any shared network.

In campaign 1, all background services were running, while in campaign 2, all not strictly necessary services where stopped by running `systemctl stop`.

The following services were stopped for campaign 2:

```
marvind
cron
watchdog
network-listener
metadata-exporter
wd_keepalive
metadata-timeloop
dlb
usb-monitor
multi
autotunnel
stunnel4
nginx
```

Figure 5.2 shows the Empirical Cumulative Distribution Function (ECDF) of the resulting measurements with and without background services. Figure 5.3 shows the same measurements over time in scatter plots. We can see that the background services have a influence on the measurement result. In the case without background services, we can observe more stable throughput results. Although we are able to reach the maximum observed throughput also with running background services, the median and especially the lowest results are well below the theoretical maximum.

In light of these results, all further measurement campaigns, within a controlled setup, were run with the background services stopped during the measurements.

5.3.2 Network Overhead

The RMBT client uses TCP or TLS (over TCP) for performing throughput measurements. This means that the actual amount of transmitted data depends on the underlying network

[1] https://github.com/MONROE-PROJECT/mPlane
[2] https://github.com/MONROE-PROJECT/Scheduler

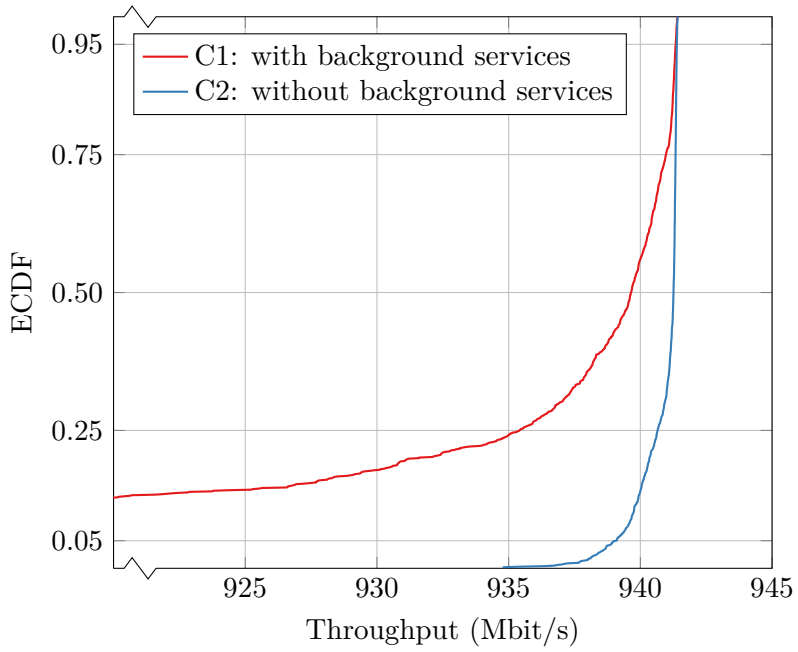
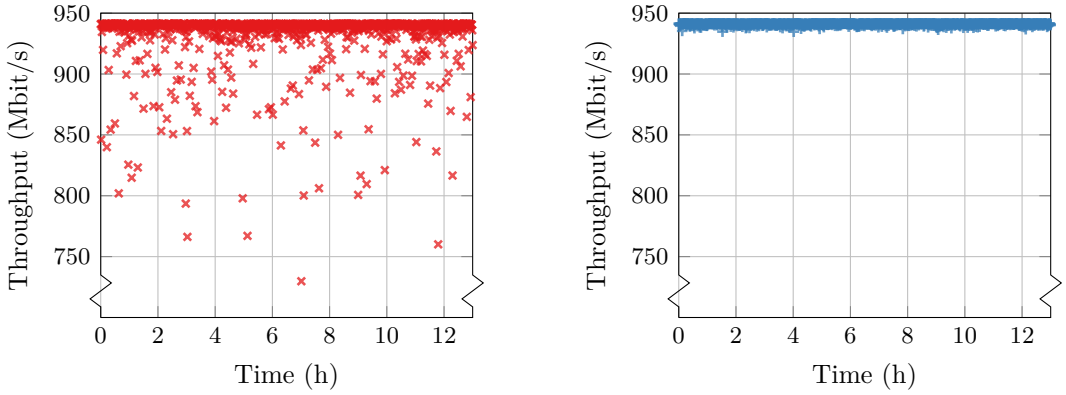


Figure 5.2: Campaigns 1+2: ECDF of throughput; with and without background services (1999 samples)



(a) Campaign 1: with background services

(b) Campaign 2: without background services

Figure 5.3: Scatterplots of measured throughput over time

stack. Additionally to the data useful for the measurement, this will include at least the TCP and IP headers plus the overhead from underlying network layers, as shown in figure 5.4. In the design of RMBT, we decided that the result of the measurement would be computed just taking into account the useful data without the overhead introduced by lower layers. It therefore measures the throughput on Open Systems Interconnection (OSI) layer 5 [37] and is closer to what the end user or an application would experience.

To understand the difference between the measured throughput and the gross bit rate of the network connection we need to calculate the overhead introduced by the different network protocols.

We define the efficiency E to be the measured throughput, T, divided by the gross bit rate, R, which includes overhead:

$$E = \frac{T}{R} \tag{5.1}$$

The throughput, T, is the length of the received data, D_T, divided by the time, t, it took to receive this data.

$$T = \frac{D_T}{t} \tag{5.2}$$

The gross bit rate, R, is the total number of physically transmitted bits on the communication link, D_R, divided by the time, t, it took for the transmission.

$$R = \frac{D_R}{t} \tag{5.3}$$

So it follows from equation (5.1) that:

$$E = \frac{\frac{D_T}{t}}{\frac{D_R}{t}} = \frac{D_T}{D_R} \tag{5.4}$$

Figure 5.4 shows the composition of one data frame transmitted over our Ethernet connection. The topmost part represents the header of the Ethernet frame. According to the Institute of Electrical and Electronics Engineers (IEEE) 802.3 standard [83], this header has a length L_{Eth} of 22 octets.

The next part is the IP header. In our case, this is an IPv4 header which has a length L_{IPv4} of 20 octets according to RFC 791 [91].

After the IP header the TCP header follows, it has also a length L_{TCP} of 20 octets according to RFC 793 [92].

In our case, the Linux kernel also included the *TCP timestamps option* header by default. This was determined by capturing the data transmission with *tcpdump*[3] and analyzing the

[3] https://www.tcpdump.org/

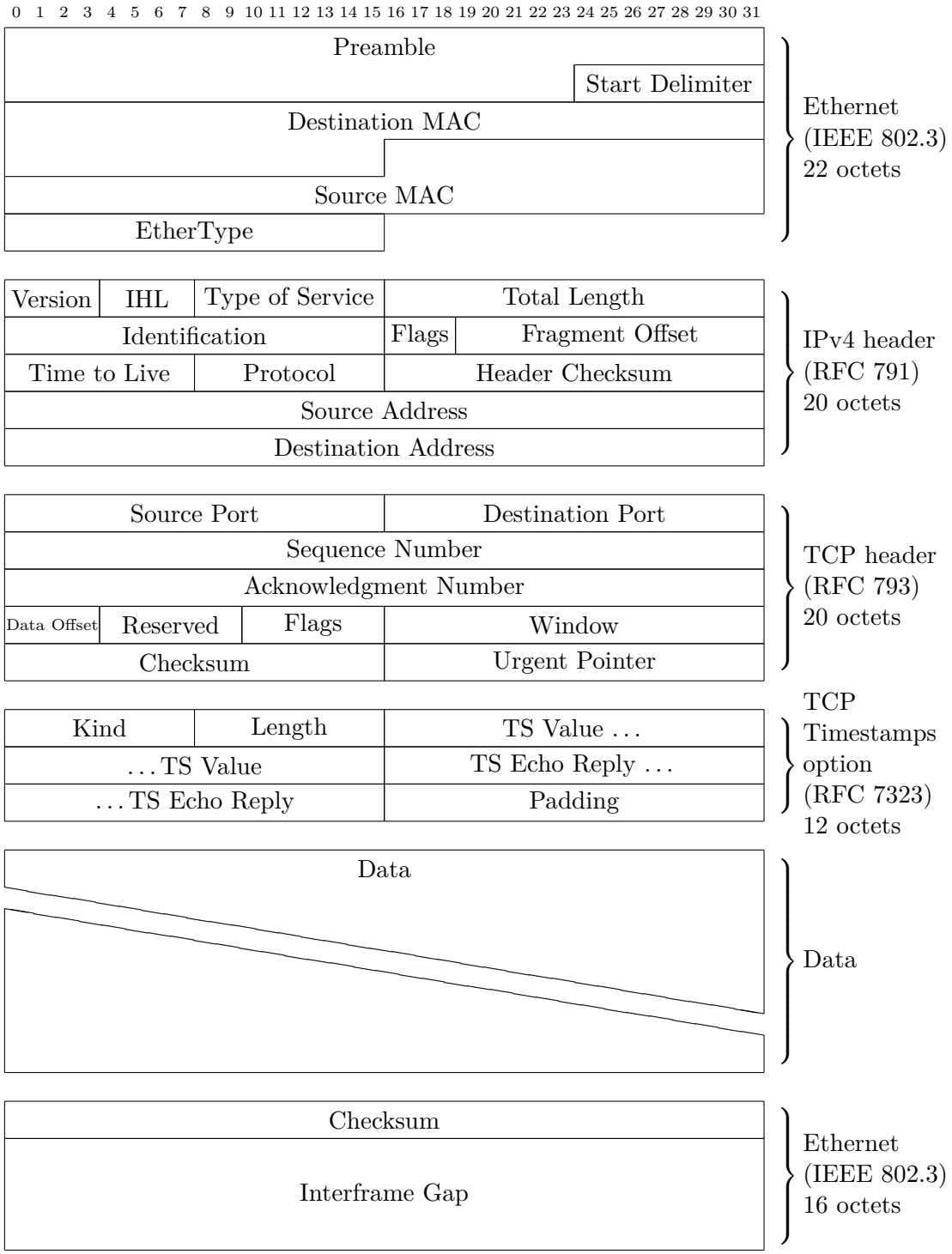


Figure 5.4: Overhead of Ethernet, IPv4 and TCP with Timestamps option

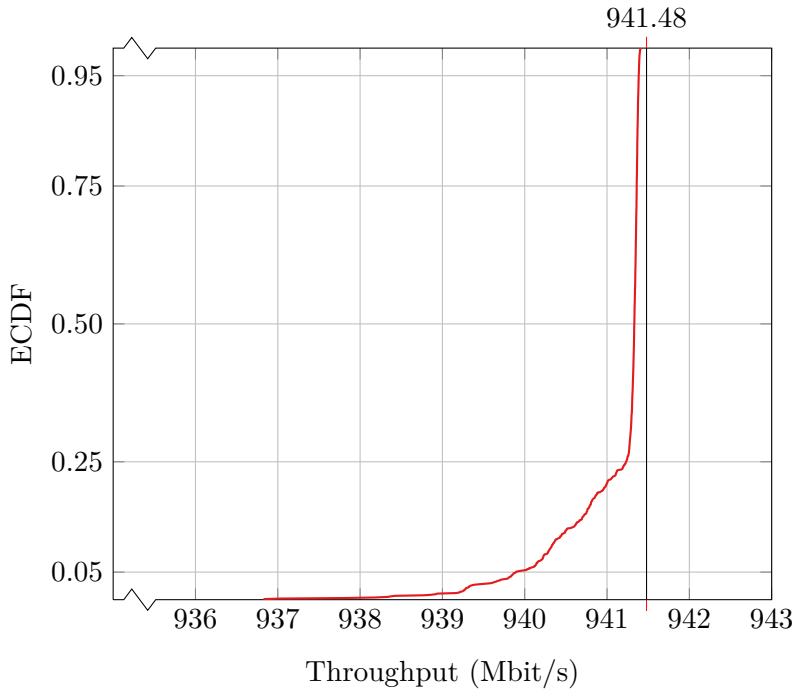


Figure 5.5: Campaign 4: ECDF of throughput in 1 Gbit/s Ethernet (1000 samples)

data with *Wireshark*[4]. The length of this option is 10 octets according to RFC 7323 [15].

According to RFC 793, if a TCP option does not have a length that is a multiple of 4 octets, padding needs to be added. Therefore 2 octets of padding are added, which gives a total length L_{TSopt} of 12 octets.

Further on, the data follows. IEEE 802.3 defines the Maximum Transmission Unit (MTU) (MTU) to be 1500. The data after the Ethernet header (including IP and TCP header in our case) can therefore be from 0 to 1500 octets in length.

After the data, the checksum of the Ethernet header with a length L_{EthCS} of 4 bytes follows. Before the next Ethernet frame can be transmitted, there is a minimum Interframe Gap (IFG) which is equivalent in size to transmitting 96 bit or 12 octets (L_{IFG}).

The length of the usable data, D_T, in a packet with the maximum size without the overhead depends on the MTU and the lengths L of the respective headers:

$$D_T = MTU - L_{IPv4} - L_{TCP} - L_{TSopt} \tag{5.5}$$

[4] https://www.wireshark.org/

The total transmitted data length is the MTU plus the Ethernet overhead:

$$D_R = MTU + L_{Eth} + L_{EthCS} + L_{IFG} \tag{5.6}$$

If we insert this into equation (5.4), we get:

$$E = \frac{MTU - L_{IPv4} - L_{TCP} - L_{TSopt}}{MTU + L_{Eth} + L_{EthCS} + L_{IFG}} \tag{5.7}$$

After finishing the implementation of the RMBT client, we investigated the measurement accuracy in a controlled environment. The first four measurement campaigns presented in table 5.1 were performed with a dedicated, direct 1 Gbit/s Ethernet connection between the server and the client not involving any shared network. No unneeded services were running on the client or server.

These campaigns were performed over a IEEE 802.3ab *1000BASE-T* Ethernet with a gross bit rate of 1 Gbit/s. IPv4 was the used IP protocol.

Then, if we insert the known lengths into equation (5.7) we can calculate the efficiency of our Ethernet connection. We assume that transmitted packets are of maximum length and there is no idle time during the transmissions. The RMBT client sends data to the Operating System (OS) kernel faster than the kernel can send the data. Therefore, the kernel buffers should be filled constantly, allowing for sending packets of maximum size back to back.

$$E = \frac{1500 - 20 - 20 - 12}{1500 + 22 + 4 + 12} = \frac{1448}{1538} \approx 0.94148 \tag{5.8}$$

As we are measuring over a 1 Gbit/s connection, this leads to a theoretical maximum throughput of:

$$1\,\text{Gbit/s} * 0.94148 = 941.48\,\text{Mbit/s} \tag{5.9}$$

Figure 5.5 shows the ECDF of 1000 measurement runs from campaign 4. The median measured throughput is 941.33 Mbit/s. Therefore the result of our measurement proofs to be accurate as it reaches 99.98 % of the theoretical maximum throughput.

5.3.3 Effect of TCP Segmentation Offload

The MONROE nodes used for measurements include an Intel I210-AT Network Interface Controller (NIC) which has support for TCP Segmentation Offload (TSO) [35]. TSO is a technology, implemented in various NICs, to reduce the Central Processing Unit (CPU) usage while sending TCP packets. It works by offloading the segmentation of the data

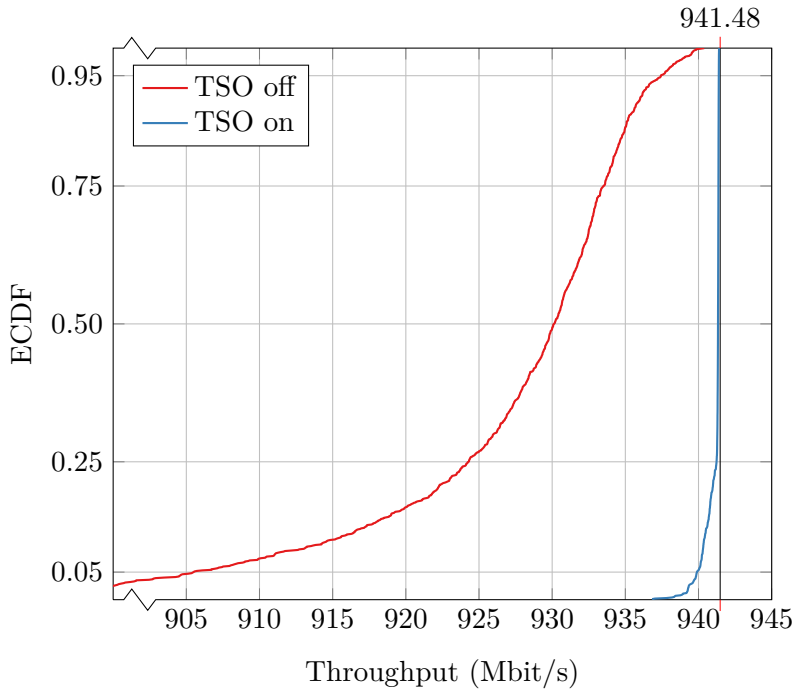


Figure 5.6: Campaigns 3+4: ECDF of throughput; with and without TSO (1998 samples)

into individual TCP packets, to the NIC, therefore reducing the number of times, the kernel has to directly interact with the NIC while sending large amounts of data.

This functionality is beneficial for our scenario with relatively high speeds, as we are dealing with nodes with relatively low powered hardware and benefit from lower CPU load, as shown in section 5.3.1.

With a gross bit rate of 1 Gbit/s and a total packet size of 1538 octets (see section 5.3.2), this leads to approximately 650 000 packets per second. TSO helps to reduce the CPU load for the sending side significantly here.

Figure 5.6 shows the difference between the measurements conducted in campaign 3 with disabled TSO versus campaign 4 with TSO enabled as it is per default.

The median throughput drops from 941.33 Mbit/s to 930.14 Mbit/s without TSO.

5.3.4 Encryption Overhead

The RMBT protocol allows for measurements on a direct TCP connection or optionally further encapsulated in a TLS connection (see section 3.4).

The computationally and time-wise expensive TLS handshake is performed at the initialization phase (see section 3.4.2) and is therefore not influencing the result of the

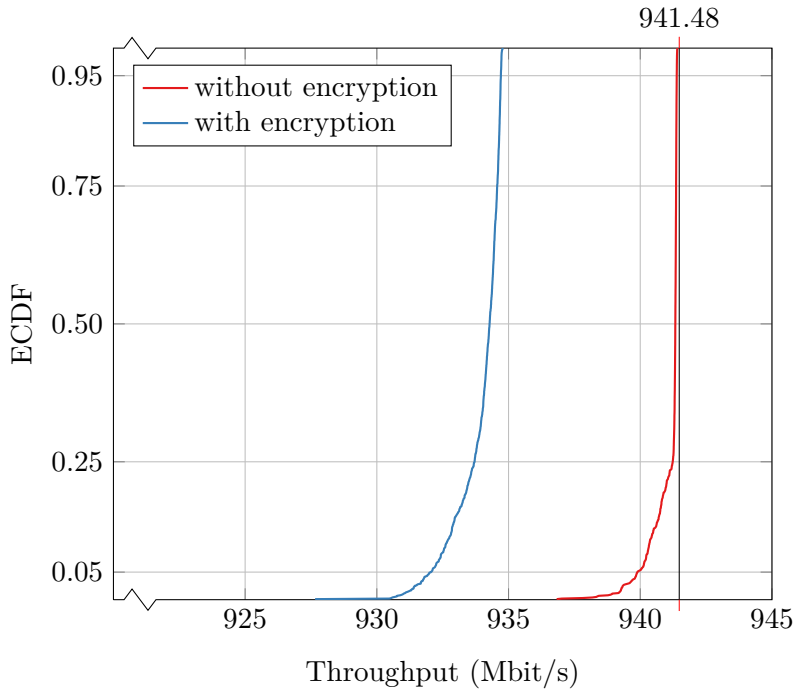


Figure 5.7: Campaign 4: ECDF of throughput; with and without encryption (1999 samples)

measurement. However TLS can add an overhead to the measurement due to the higher CPU load caused by the encryption and the usage of HMAC or Authenticated Encryption with Associated Data (AEAD) [60].

The MONROE hardware nodes have support for Advanced Encryption Standard New Instructions (AES-NI) and therefore keeping CPU overhead to a minimum if Advanced Encryption Standard (AES) is used for encryption.

Campaign 4 was performed to investigate the influence of using TLS. Figure 5.7 shows the results with and without TLS. The median throughput drops from 941.38 Mbit/s to 933.93 Mbit/s with the usage of TLS.

In this campaign TLS version 1.2 was used with the cipher suite `AES256-GCM-SHA384`. The drop in throughput can be explained by the size of the TLS header and the usage of an AEAD cipher which adds an authentication code to the resulting cipher text.

5.3.5 Number of Flows and Measurement Duration

Next, we investigate the effect of number of flows and measurement duration on the reported throughput.

There are two main concerns in the choice of the number of flows:

- The TCP **slow start** causes the rate of the flow to converge to the available bandwidth only after some time, meaning that the more TCP flows we aggregate, the faster we can capture the available bandwidth or an estimate thereof [9, 102].
- Although theoretically possible, for **fairness** reasons (in order to be fair to the other users of the links along our end-to-end path), and because of limited resources on the measurement hardware, we cannot open an infinite number of flows between our client and server. Also network operator equipment might limit the number of connections that can be opened by a client. This is especially the case with Carrier-Grade Network Address Translation (CGN) equipment which is widely in use by MBB operators [77, 88, 56, 53].

The choice of measurement duration is important for the following reasons:

- Reducing the overall time required for an accurate measurement of network throughput is resource-friendly.
- The longer our measurements take, the more data they consume from (end-users' or the experimentation platform's) mobile subscriptions. Data volume consumption for a 10 s measurement can go up to 50 MB in 3G (assuming 42 Mbit/s) and 200 MB (assuming 150 Mbit/s[5]) in 4G networks (see also section 5.2). This is undesirable.
- We would like our measurement tool to be usable in mobility scenarios as well. But under mobility the larger span of the measurement decreases reliability because of the change in network conditions.

The ideal measurement tool, specially for MBB networks, would provide an accurate result with the shortest measurement duration and consuming as little data as possible.

Due to the collection of detailed time series during the measurement, it is possible to simulate a shorter measurement duration from the collected data. This was done by applying the throughput calculation (see section 3.5.1 and equation (3.4)) to the time series of a 15 s measurement, but only using the first n seconds of the data. This effectively equals the result of the measurement, if it would have been stopped after n seconds. Therefore we deemed a longer-than-necessary measurement duration more suitable.

We focus on measurement campaigns 14 and 15 where we have six different number of flows and we evaluate the impact of the number of flows on the measured throughput. Figure 5.8 shows the results of running measurements over six operational MBB networks — three in Sweden and three in Norway — with 1, 3, 4, 5, 7, and 9 TCP flows. The nominal measurement duration was set to 15 s. The figure shows the median throughput reported

[5] 150 Mbit/s is the theoretical maximum of LTE Cat-4. We measured effective throughputs up to almost 200 Mbit/s and the latest LTE standard at the time of this writing (Cat-16) reaches a theoretical 1 Gbit/s, with 5G networks having theoretical peak bit rates of up to 20 Gbit/s [38].

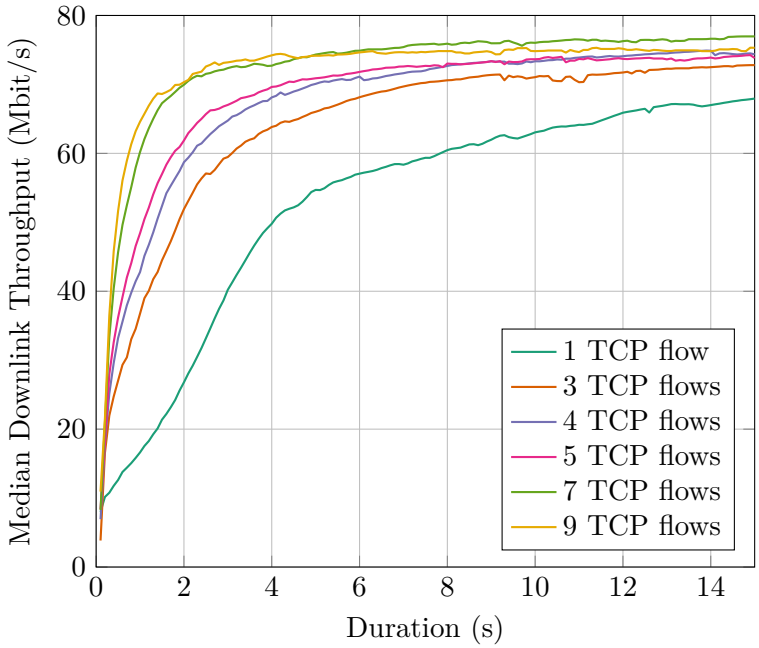


Figure 5.8: Campaign 14+15: Effect of number of flows and measurement duration (DL)

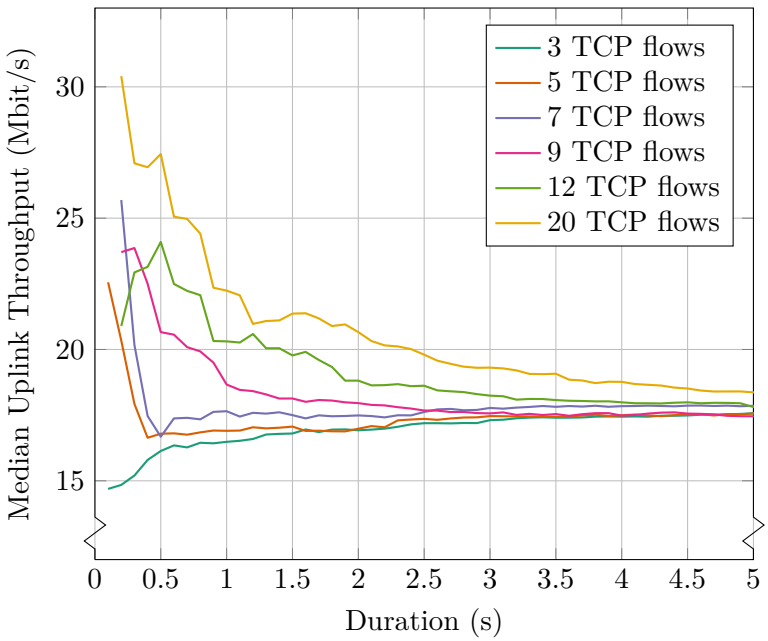


Figure 5.9: Campaign 18: Effect of number of flows and measurement duration (UL)

by the measurements, if they stop at the duration indicated by the x-axis. We have the possibility to undertake these calculations, because we collect the time series of all the separate flows in the form of the *Flows* output file (see section 4.5.2). For each run, we first compute the reported throughput using the calculation in section 3.5.1 for every 10 ms between 0 s and 15 s as t^*. We then used the median of these values, per t^*, to plot the curves in the figures.

The figures show that the reported throughput depends on the number of flows and the measurement duration at the same time. In figure 5.8 we see that the maximum DL throughput can be achieved with a higher number of parallel TCP flows. After 15 s the maximum throughput can be reached even with a lower number of flows (≥ 3), but can be reached much quicker with a high number of flows.

We can reach a reasonably accurate approximation limiting the duration of the measurement and saving resources by using a higher number of flows.

Figure 5.9 shows the median UL throughput of the measurements of campaign 18. We can see that these measurements reach higher throughputs for shorter measurement durations, especially with higher number of flows, ≥ 5 flows. This means that with short burst of data the client is able to reach higher throughputs, while longer data transmissions lead to a lower, capped throughput. The effect is stronger with a higher number of flows.

Figures 5.10 and 5.11 refer to the same dataset, but with a different representation, comparing the number of flows or duration.

Figure 5.10 shows the ECDF of the measurements of campaign 17 for 1, 3, 5, and 7 flows if they would have been stopped at 1 s, 3 s, 5 s, 7 s, 9 s, 11 s, 13 s, and for the full 15 s measurement duration.

Figure 5.11 shows the same data set but compares 1 s, 5 s, 9 s, and 13 s of a fixed number of flows (1, 3, 5, and 7) in every plot.

We see a general trend of increasing reported throughput with increased number of flows. For the evaluation, we quantify the similarity of time series using Minkowski distance [99]. Table 5.2 presents the percentage of the median throughput at 15 s (assumed to be the saturation throughput) that the curves capture at 2 s, 4 s, 6 s, 8 s, and 10 s, as well as the distance of each curve to the one indicating 9 flows (referred to as the final curve).

We see that for the aggregate case, 7 flows are adequate to capture more than 99 % of the saturation throughput for ≥ 8 s measurement duration. For specific operators, 7 flows with 10 s duration respectively yield: 94.2 %, 99.7 %, 99.3 %, 98.1 %, 93.5 % and 99.0 % of the saturation throughput. Considering the balance of data consumption and accuracy, we suggest 7 **flows** with 8 s **duration**, which provides ≥ 90 % of the saturation throughput.

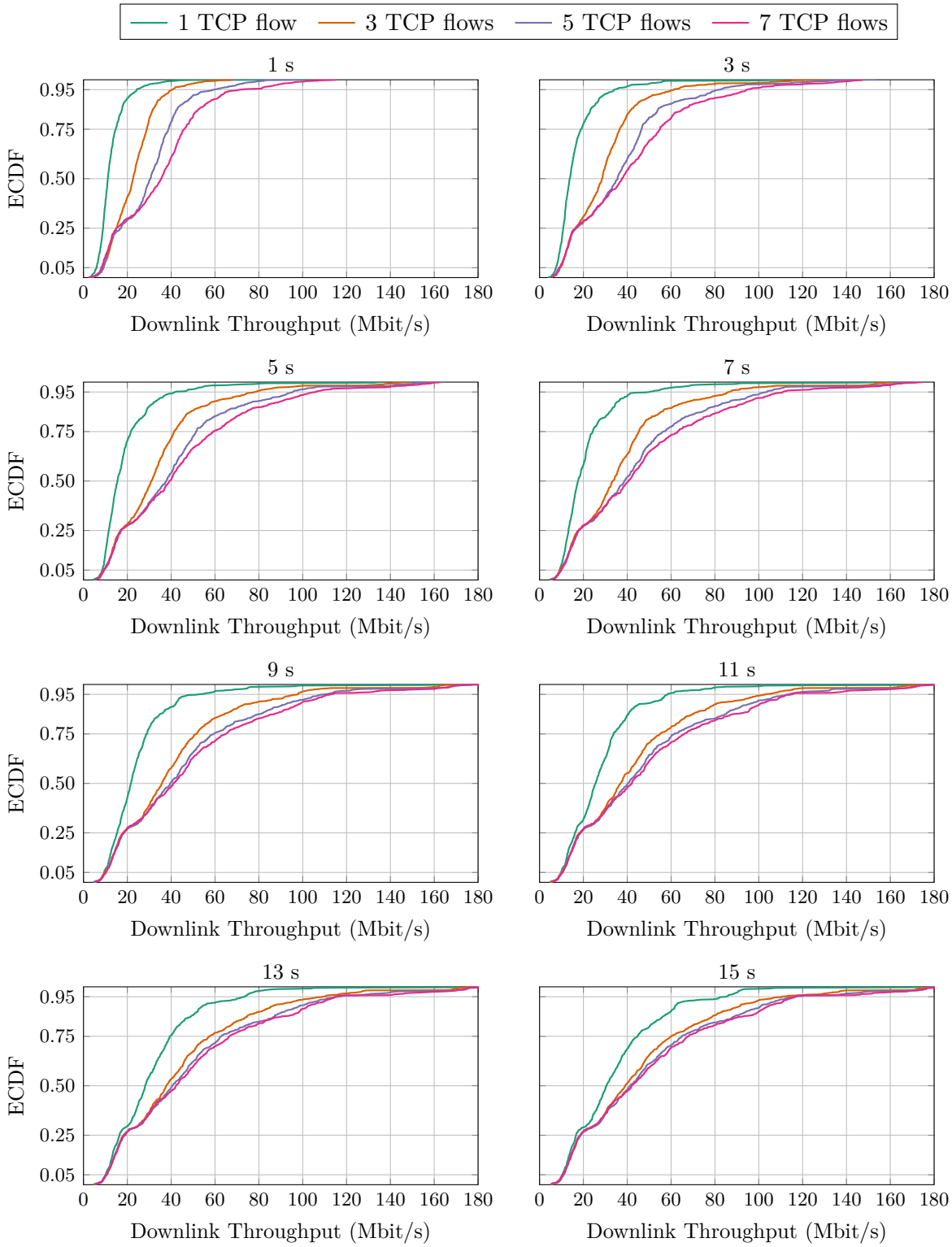


Figure 5.10: Campaign 17: ECDF of simulated throughput for different measurement durations (2840 samples)

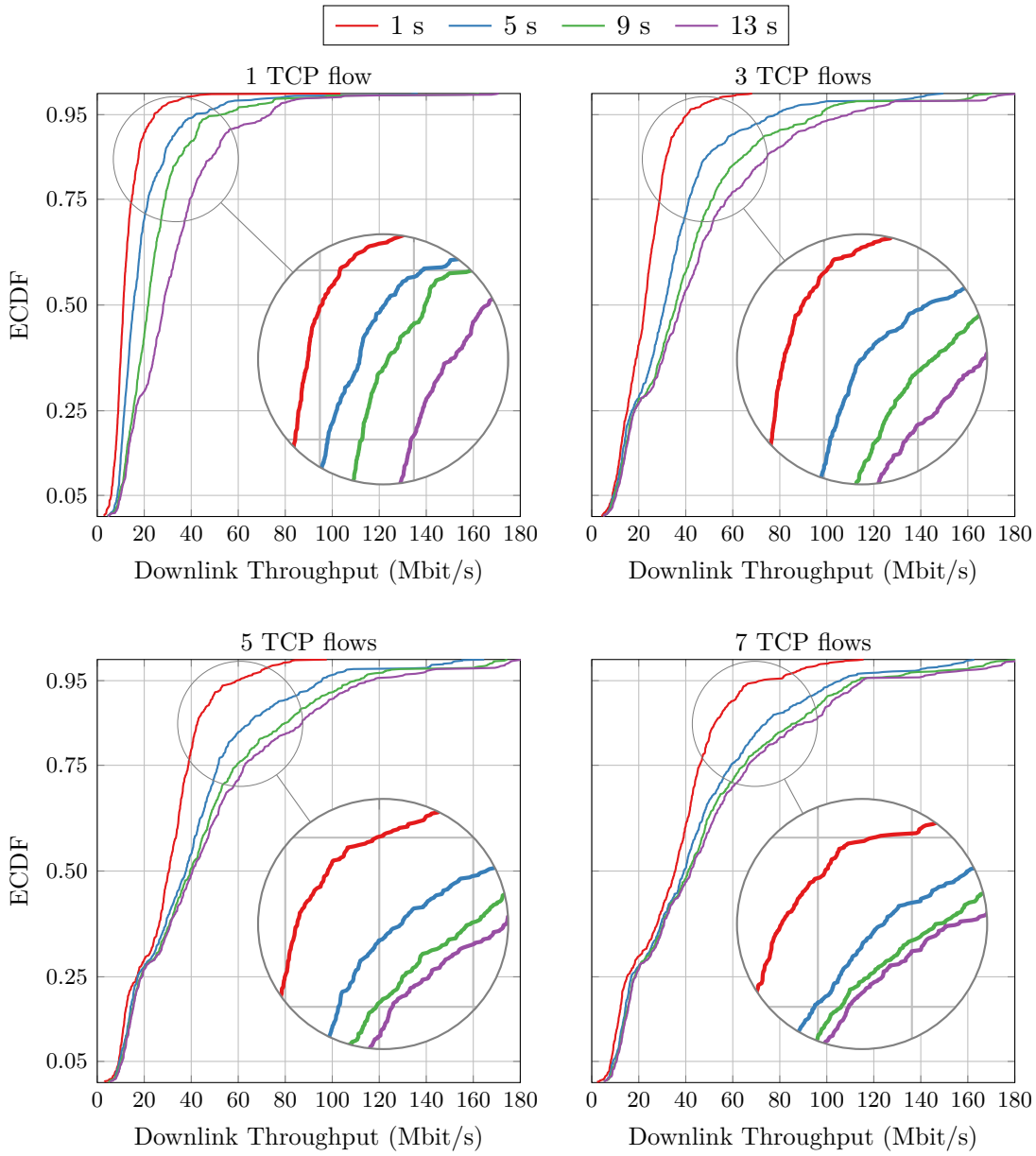


Figure 5.11: Campaign 17: ECDF of simulated throughput for different number of flows (2840 samples)

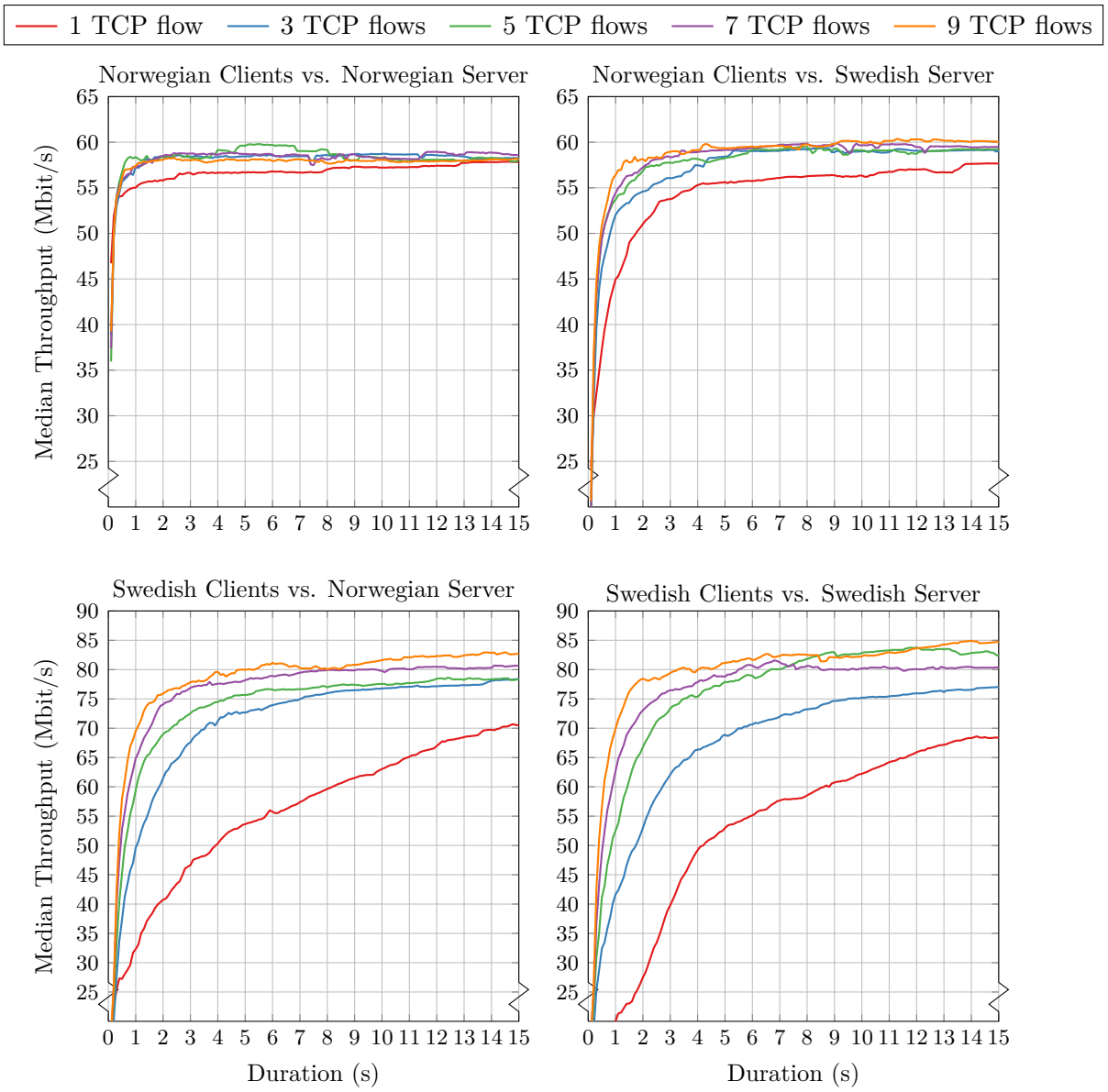


Figure 5.12: Campaigns 14+15: Influence of server location (2840 samples)

Table 5.2: The impact of number of flows on measured throughput

	% of Saturation Throughput					Distance to Last Curve		
Flows	**2s**	**4s**	**6s**	**8s**	**10s**	**Manhattan**	**Euclidean**	**Chebyshev**
1	67.5	78.6	82.5	84.9	89.4	3206.0	56.6	31.8
3	73.3	88.6	92.9	96.2	98.2	1179.0	34.3	18.8
4	78.3	91.6	95.6	97.5	98.2	700.2	26.5	15.3
5	83.9	94.2	96.9	97.6	99.0	603.2	24.6	13.0
7	91.0	96.4	98.3	99.0	99.1	257.1	16.0	6.28
9	92.8	95.6	97.8	98.1	99.0	NA	NA	NA

5.3.6 Influence of Server Location

Next, we investigate the effect of server location. For this, we compare measurements that were run against servers in different countries.

Figure 5.12 presents the results of campaigns 14 and 15. The plots show the effect of number of flows and measurement duration equivalent to figure 5.8 (see section 5.3.5). The top row is filtered by Norwegian clients and the bottom row by Swedish clients, the left column by measurements against the Norwegian server and the right column by measurements against the Swedish server.

In general, and without further information on the network configurations, the closer the server, the higher the throughput and the faster it can be achieved.

5.3.7 Temporal Patterns

The extensive set of measurements spanning several days, allows us to further observe daily patterns in throughput for some operators, indicating that the spread between curves observed in sections 5.3.5 and 5.3.6 might not be constant throughout a day. Figure 5.13 shows the scatter plot of throughput vs. relative time for one operator from campaign 14, for the first 50 hours of the campaign. There is a clear 24 h trend in throughput for all number of flows visible in the plot.

In addition, the MONROE nodes collect metadata in the background. This metadata includes, among other things, various signal strengths, the frequency band, cell id, and location [87]. We collected this metadata during the measurement campaigns to allow for further analysis.

To illustrate with an example the influence of the temporal variations in MBB, figure 5.14a shows the Reference Signals Received Power (RSRP) and Received Signal Strength Indicator (RSSI) of one stationary device during the measurements conducted in the course of campaign 8. We can observe that the device switched the frequency band twice in the time span of the approx. 16 h. Figures 5.14b and 5.14c show the scatter plots of the individual DL resp. UL measurements of the same device during the same time span.

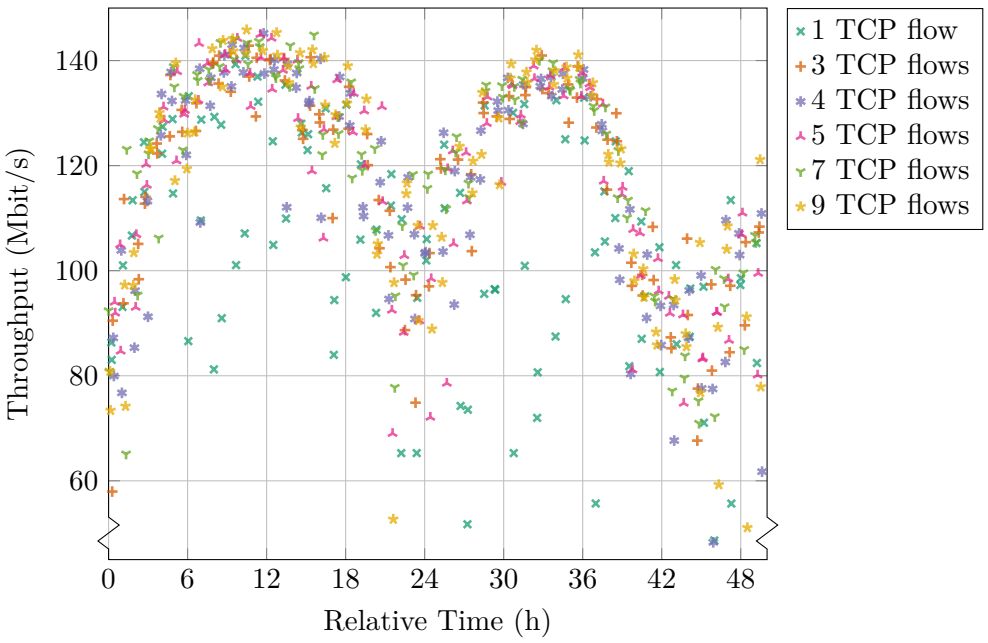
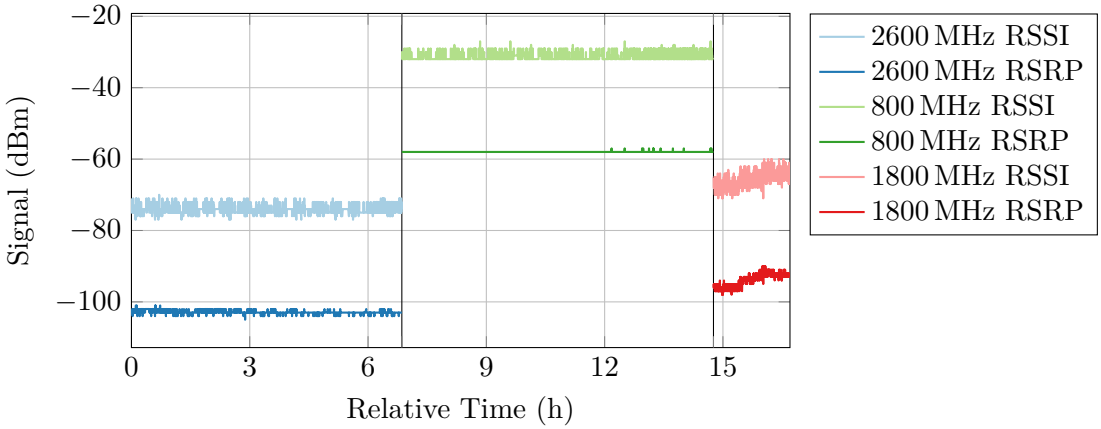
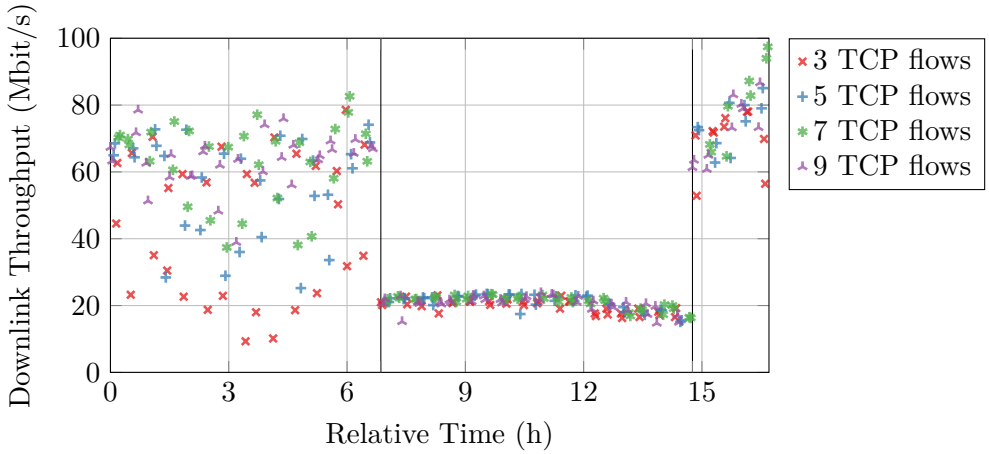


Figure 5.13: Campaign 14: Daily patterns in reported throughput

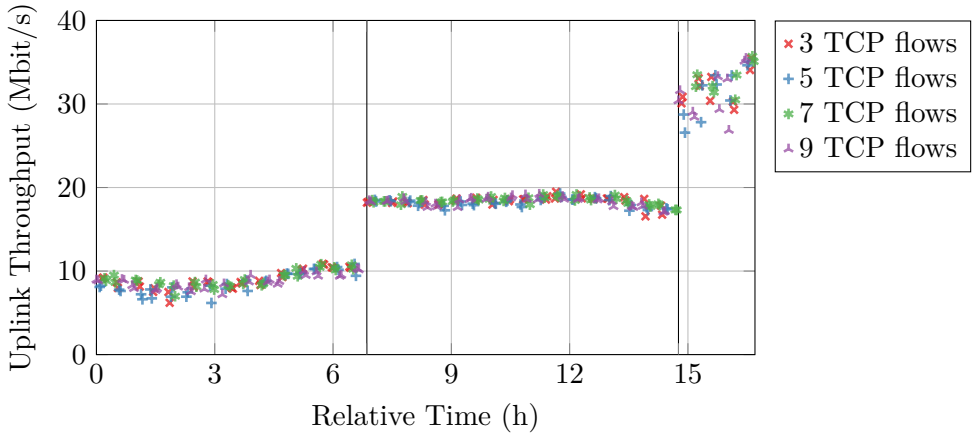
The changes in frequency band of this stationary device have a clear influence on the measured throughputs.



(a) Signal



(b) Downlink



(c) Uplink

Figure 5.14: Campaign 8: Changes in throughput for a stationary node due to changing frequency bands

CHAPTER 6

Conclusion

In this chapter, we present the summary of the contributions of this thesis work in section 6.1. In section 6.2, we list potentially interesting future research topics connected with this work. Finally, section 6.3 lists refereed publications that were contributed in the course of this thesis work and links to the resulting source code.

6.1 Summary of Contributions

What is the optimal measurement design for collecting performance metrics on MBB networks? — As stated in section 1.3 this is the research question that motivated this thesis work. We addressed it by studying the possibilities to measure performance and by developing a flexible and configurable measurement tool, which will enable further research.

Specifically the contributions of this work can be summarized as:

- We give an introduction into measuring mobile broadband networks and show currently available tools for this task.
- We present our rationale for selecting RMBT / RTR-Netztest as basis for our own implementation. The available, Java based client, is deemed as being too resource hungry for our large-scale measurement campaigns. We therefore reimplement the client in C.
- We give an overview of the source code and explain how to build the code and perform measurements. We also give a very detailed list of all configuration parameters and the output formats.

- Furthermore, we describe the RMBT protocol in comprehensive detail and introduce the MONROE platform, that was utilized to perform a number of large-scale measurement campaigns.
- Our implementation adds very high detail to the output format. Additional to the metrics collected by existing implementations, we collect detailed time series of all measurements, separated for each of the TCP flows. We also add an implementation to collect detailed TCP metrics, with the use of the `TCP_INFO` socket option. Our tool can efficiently sample these metrics with a high resolution and output them.
- The resulting implementation is released as free and open-source software to the research community, to serve as a powerful tool for performing measurements in current and future generations of networks. The resulting could also be of interest for regulatory authorities, network operators, and end users.
- As the measurement client is completely open, well documented, highly configurable, and reports the results in high detail, it can easily be modified or extended for other measurement needs. The tool is very versatile and independent of the underlying network technology.
- After the completed implementation, the tool was adopted under the name *"MONROE-Nettest"* [65] as a MONROE base experiment, that is run regularly on all active MONROE nodes [55][1]. The experiment can also be scheduled with custom parameters easily by all MONROE users.
- Furthermore, we present an overview of all the major measurement campaigns conducted with our newly implemented tool. We pick some interesting findings and present and analyze them in detail.
- We test the influence of the background services, normally running on all MONROE nodes. The background services seem to have an influence on very high speed measurements conducted over direct 1 Gbit/s Ethernet links.
- To figure out if our tool is reporting accurate results, we perform a large number of measurements on a setup with known ground truth. As the reported throughput of our tool does not account for the overhead of underlying network protocols, we take a close look on the sizes of these headers and can confirm that our tool reports 99.98 % of the expected throughput.
- We also test the influence of disabled TSO on our measurements and find out that we measure lower and less stable results without TSO.
- The implemented RMBT client also supports encryption of the measurement connections. Our measurements indicate, that the reported throughput is slightly lower with enabled encryption, as we need to account for the additional overhead.

[1] `https://github.com/MONROE-PROJECT/Experiments/tree/master/experiments/nettest`

- We perform several large-scale measurement campaigns over commercial mobile networks and collect a large data set with detailed time series.
- We investigate the influence of measurement parameters on the results. We vary the number of TCP flows, used to perform the measurement. Thanks to the detailed time series we collect, we can calculate the theoretically reported throughput, if we would have set the nominal measurement time to a shorter time. Therefore, we set our nominal measurement time for our large-scale measurement campaigns to a relatively high number of 15 s and later calculate the theoretical results of shorter measurement durations.
- We find out that a measurement duration of 8 s with 7 parallel TCP flows is a reasonable measurement parameter configuration for 4G mobile networks.
- Finally, we also take a quick look at the influence of the server location and present observed temporal patterns in our measurement campaigns.

6.2 Future Work

The resulting tool and the collected data present a valuable asset for future work. In the following, we list a number of potentially interesting topics of research:

- The implemented RMBT client could easily be adapted to compile on other platforms. It could supersede the Java measurement core in Android based clients, to allow for a more detailed collection of additional metrics.
- A limitation of the measurement campaigns over operational MBB networks was the usage of SIM cards with limited data quota in the MONROE nodes. To perform larger campaigns with a higher number of samples and more measurements per time period, it would be beneficial to have access to SIM cards with unlimited data quota and without data rate caps.
- All measurements campaigns, presented in this thesis, were run on stationary nodes. It would be interesting to investigate results for moving clients.
- The implemented RMBT client was tested in a controlled setup, with known ground truth, consisting of a direct 1 Gbit/s Ethernet connection. It would also be very interesting to perform measurements in controlled 4G/5G networks without any other influencing traffic, while also recording and analyzing various network parameters [5].
- TCP parameters were left at their respective default values for the measurement campaigns conducted for this thesis. This was done to investigate the throughput, as it would be experienced by the typical end user, who does not change the default parameters. The implemented tool could easily be adapted to vary certain

parameters, such as the used TCP congestion control algorithm or initial window sizes, and investigate the influence on the results [9].

- The collected, detailed TCP metrics could be analyzed, to get a deeper understanding of the measurement results. These metrics show for example retransmissions, the currently observed RTT, the slow start threshold, window sizes, the status of the buffers, and the number of sent and acknowledged bytes. This information can also be used to eliminate the influence of the protocol overhead of TLS, as the number of transmitted bytes on top of the TCP connection can directly be observed.

6.3 Publications and Resources

The following refereed publications were produced in the course of this thesis work:

- **L. Wimmer**, C. Midoglu, A. Lutu, Ö. Alay, and C. Griwodz. „Concept and Implementation of a Configurable Nettest Tool for Mobile Broadband“. In: *2018 IEEE Wireless Communications and Networking Conference (WCNC): IEEE WCNC2018 Student Program (IEEE WCNC 2018 Students)*. Barcelona, Spain, 2018-04
- C. Midoglu, **L. Wimmer**, A. Lutu, Ö. Alay, and C. Griwodz. „MONROE-Nettest: A Configurable Tool for Dissecting Speed Measurements in Mobile Broadband Networks“. In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2018-04, pp. 342–347. DOI: 10.1109/INFCOMW.2018.8406836 (Won the *Best Paper Award*)

The resulting source code of the RMBT client can be found in appendix B and publicly available under these URLs:

- https://github.com/lwimmer/rmbt-client
- https://gitlab.com/lwimmer/rmbt-client
- https://github.com/MONROE-PROJECT/Experiments/tree/master/experiments/nettest

APPENDIX A

Measurement Campaigns

This chapter provides detailed information about the conducted measurement campaigns. For a thorough description and analysis see chapter 5.

A.1 Campaign 1

First measurement: 2017-03-29 17:01:59 UTC
Last measurement: 2017-03-30 06:27:39 UTC
Nominal number of samples per batch: 2
Number of complete batches: 996
Total number of samples: 1993
Utilized technology: 1 Gbit/s Ethernet (direct connection)
Server: Lenovo ThinkPad T430s with Ubuntu 17.04
Client: MONROE node 298 (APU2), local usage in development mode
Tested number of parallel TCP flows: 3
Nominal measurement duration: 7 s
Investigated parameters: encryption / no encryption
Mean Downlink duration: 7.06 s
Mean Uplink duration: 7.03 s
Median measured Downlink: 927.79 Mbit/s
Median measured Uplink: 930.79 Mbit/s
Total data usage: 4104.59 GB

A.2 Campaign 2

First measurement: 2017-03-31 11:23:15 UTC
Last measurement: 2017-04-01 00:53:58 UTC
Nominal number of samples per batch: 2
Number of complete batches: 996
Total number of samples: 1998
Utilized technology: 1 Gbit/s Ethernet (direct connection)
Server: Lenovo ThinkPad T430s with Ubuntu 17.04

Client: MONROE node 298 (APU2), local usage in development mode
Tested number of parallel TCP flows: 3
Nominal measurement duration: 7 s
Investigated parameters: encryption / no encryption
Mean Downlink duration: 7.04 s
Mean Uplink duration: 7.04 s
Median measured Downlink: 929.45 Mbit/s
Median measured Uplink: 934.78 Mbit/s
Total data usage: 4211.00 GB

A.3 Campaign 3

First measurement: 2017-04-03 16:10:49 UTC
Last measurement: 2017-04-04 04:55:59 UTC
Nominal number of samples per batch: 2
Number of complete batches: 997
Total number of samples: 1995
Utilized technology: 1 Gbit/s Ethernet (direct connection)
Server: Lenovo ThinkPad T430s with Ubuntu 17.04
Client: MONROE node 298 (APU2), local usage in development mode
Tested number of parallel TCP flows: 3
Nominal measurement duration: 7 s
Investigated parameters: encryption / no encryption
Mean Downlink duration: 7.01 s
Mean Uplink duration: 7.00 s
Median measured Downlink: 931.07 Mbit/s
Median measured Uplink: 882.34 Mbit/s
Total data usage: 3967.93 GB

A.4 Campaign 4

First measurement: 2017-04-05 13:40:56 UTC
Last measurement: 2017-04-06 01:04:31 UTC
Nominal number of samples per batch: 2
Number of complete batches: 999
Total number of samples: 1999
Utilized technology: 1 Gbit/s Ethernet
Server: MONROE node 298 (APU2), local usage in development mode
Client: MONROE node 303 (APU2), local usage in development mode
Tested number of parallel TCP flows: 3
Nominal measurement duration: 7 s
Investigated parameters: encryption / no encryption
Mean Downlink duration: 7.04 s
Mean Uplink duration: 7.03 s
Median measured Downlink: 934.76 Mbit/s
Median measured Uplink: 936.83 Mbit/s
Total data usage: 4258.99 GB

A.5 Campaign 5

First measurement: 2017-04-26 16:42:33 UTC
Last measurement: 2017-04-27 06:31:16 UTC
Nominal number of samples per batch: 2
Number of complete batches: 50
Total number of samples: 100
Utilized technology: 4G/LTE
Operator: Telenor Norway
Server: Sweden (*Karlstad University*)
Client: MONROE node 301 (APU2), local usage in development mode
Tested number of parallel TCP flows: 3
Nominal measurement duration: 7 s
Investigated parameters: with MobileInsight / without MobileInsight
Mean Downlink duration: 7.72 s
Mean Uplink duration: 7.51 s
Median measured Downlink: 21.87 Mbit/s
Median measured Uplink: 18.71 Mbit/s
Total data usage: 4.47 GB

A.6 Campaign 6

First measurement: 2017-04-27 15:15:54 UTC
Last measurement: 2017-04-28 07:43:34 UTC
Nominal number of samples per batch: 7
Number of complete batches: 30
Total number of samples: 210
Utilized technology: 4G/LTE
Operator: Telia Norway
Server: Sweden (*Karlstad University*)
Client: MONROE node 301 (APU2), local usage in development mode
Tested number of parallel TCP flows: 1–7
Nominal measurement duration: 7 s
Investigated parameters: Number of flows
Mean Downlink duration: 7.73 s
Mean Uplink duration: 7.61 s
Median measured Downlink: 22.68 Mbit/s
Median measured Uplink: 19.86 Mbit/s
Total data usage: 9.98 GB

A.7 Campaign 7

First measurement: 2017-04-28 14:52:17 UTC
Last measurement: 2017-04-29 16:43:12 UTC
Nominal number of samples per batch: 5
Number of complete batches: 59
Total number of samples: 271
Utilized technology: 4G/LTE
Operator: Telia Norway
Server: Sweden (*Karlstad University*)
Client: MONROE node 301 (APU2), local usage in development mode

Tested number of parallel TCP flows: 3–7
Nominal measurement duration: 10 s
Investigated parameters: Number of flows
Mean Downlink duration: 10.64 s
Mean Uplink duration: 12.78 s
Median measured Downlink: 76.29 Mbit/s
Median measured Uplink: 10.19 Mbit/s
Total data usage: 35.20 GB

A.8 Campaign 8

First measurement: 2017-05-03 15:17:49 UTC
Last measurement: 2017-05-04 07:56:34 UTC
Nominal number of samples per batch: 8
Number of complete batches: 38
Total number of samples: 304
Utilized technology: 4G/LTE
Operator: Telia Norway
Servers: Sweden (*Karlstad University*), Germany (*Hetzner Online GmbH*)
Client: MONROE node 301 (APU2), local usage in development mode
Tested number of parallel TCP flows: 3, 5, 7, 9
Nominal measurement duration: 15 s
Investigated parameters: Number of flows and server location
Mean Downlink duration: 16.51 s
Mean Uplink duration: 18.58 s
Median measured Downlink: 23.47 Mbit/s
Median measured Uplink: 17.96 Mbit/s
Total data usage: 39.85 GB

A.9 Campaign 9

First measurement: 2017-05-04 16:00:12 UTC
Last measurement: 2017-05-05 13:56:12 UTC
Nominal number of samples per batch: 8
Number of complete batches: 49
Total number of samples: 389
Utilized technology: 4G/LTE
Operator: Telenor Norway
Servers: Sweden (*Karlstad University*), Germany (*Hetzner Online GmbH*)
Client: MONROE node 301 (APU2), local usage in development mode
Tested number of parallel TCP flows: 3, 5, 7, 9
Nominal measurement duration: 15 s
Investigated parameters: Number of flows and server location
Mean Downlink duration: 15.87 s
Mean Uplink duration: 15.94 s
Median measured Downlink: 89.53 Mbit/s
Median measured Uplink: 44.68 Mbit/s
Total data usage: 109.08 GB

A.10 Campaign 10

First measurement: 2017-05-10 21:48:06 UTC
Last measurement: 2017-05-11 06:48:53 UTC
Nominal number of samples per batch: 4
Number of complete batches: 35
Total number of samples: 140
Utilized technology: 1 Gbit/s Fiber
Operator: Kvantel AS
Server: Sweden (*Karlstad University*)
Client: MONROE node 301 (APU2), local usage in development mode
Tested number of parallel TCP flows: 3, 5, 7, 9
Nominal measurement duration: 10 s
Investigated parameters: Number of flows
Mean Downlink duration: 10.12 s
Mean Uplink duration: 10.06 s
Median measured Downlink: 619.39 Mbit/s
Median measured Uplink: 466.40 Mbit/s
Total data usage: 214.739 GB

A.11 Campaign 11

First measurement: 2017-05-12 15:37:59 UTC
Last measurement: 2017-05-14 16:11:23 UTC
Nominal number of samples per batch: 7
Number of complete batches: 50
Total number of samples: 353
Utilized technology: 1 Gbit/s Fiber
Operator: Kvantel
Server: Sweden (*Karlstad University*)
Client: MONROE node 301 (APU2), local usage in development mode
Tested number of parallel TCP flows: 3, 10, 15, 25, 50, 100, 199
Nominal measurement duration: 10 s
Investigated parameters: Number of flows
Mean Downlink duration: 10.21 s
Mean Uplink duration: 10.27 s
Median measured Downlink: 530.42 Mbit/s
Median measured Uplink: 503.07 Mbit/s
Total data usage: 569.39 GB

A.12 Campaign 12

First measurement: 2017-05-17 06:08:28 UTC
Last measurement: 2017-05-19 08:01:55 UTC
Nominal number of samples per batch: 8
Number of complete batches: 38
Total number of samples: 317
Utilized technology: 4G/LTE
Operator: Telia Norway
Servers: Sweden (*Karlstad University*), Germany (*Hetzner Online GmbH*)
Client: MONROE node 301 (APU2), local usage in development mode

Tested number of parallel TCP flows: 3, 5, 7, 9
Nominal measurement duration: 15 s
Investigated parameters: Number of flows and server location
Mean Downlink duration: 16.94 s
Mean Uplink duration: 17.12 s
Median measured Downlink: 24.47 Mbit/s
Median measured Uplink: 19.09 Mbit/s
Total data usage: 31.99 GB

A.13 Campaign 13

First measurement: 2017-05-19 13:32:01 UTC
Last measurement: 2017-05-19 23:50:21 UTC
Nominal number of samples per batch: 12
Number of complete batches: 10
Total number of samples: 100
Utilized technology: 4G/LTE
Operator: Telia Norway
Servers: Sweden (*Karlstad University*), Germany (*Hetzner Online GmbH*)
Client: MONROE node 301 (APU2), local usage in development mode
Tested number of parallel TCP flows: 1, 3, 4, 5, 7, 9
Nominal measurement duration: 15 s
Investigated parameters: Number of flows and server location
Mean Downlink duration: 16.53 s
Mean Uplink duration: 22.59 s
Median measured Downlink: 26.92 Mbit/s
Median measured Uplink: 4.03 Mbit/s
Total data usage: 14.11 GB

A.14 Campaign 14

First measurement: 2017-05-24 15:23:06 UTC
Last measurement: 2017-05-28 14:46:48 UTC
Nominal number of samples per batch: 12
Number of complete batches: 266
Total number of samples: 4785
Utilized technology: 4G/LTE
Operators: 3 Sweden, Telia Norway, Telia Sweden, .ice Norway, Telenor Norway, Telenor Sweden
Servers: Norway (*University of Oslo*), Sweden (*Karlstad University*)
Clients: MONROE nodes in Norway (470, 471) and Sweden (478, 479)
Tested number of parallel TCP flows: 1, 3, 4, 5, 7, 9
Nominal measurement duration: 15 s
Investigated parameters: Number of flows and server location
Mean Downlink duration: 15.62 s
Mean Uplink duration: 16.33 s
Median measured Downlink: 65.99 Mbit/s
Median measured Uplink: 13.29 Mbit/s
Total data usage: 669.53 GB

A.15 Campaign 15

First measurement: 2017-06-08 08:14:17 UTC
Last measurement: 2017-06-12 11:27:34 UTC
Nominal number of samples per batch: 12
Number of complete batches: 176
Total number of samples: 1504
Utilized technology: 4G/LTE
Operators: 3 Sweden, Telia Norway, Telia Sweden, .ice Norway, Telenor Norway, Telenor Sweden
Servers: Norway (*University of Oslo*), Sweden (*Karlstad University*)
Clients: MONROE nodes in Norway (470, 471) and Sweden (478, 479)
Tested number of parallel TCP flows: 1, 3, 4, 5, 7, 9
Nominal measurement duration: 15 s
Investigated parameters: Number of flows and server location
Mean Downlink duration: 17.57 s
Mean Uplink duration: 16.91 s
Median measured Downlink: 66.45 Mbit/s
Median measured Uplink: 39.82 Mbit/s
Total data usage: 285.26 GB

A.16 Campaign 16

First measurement: 2017-07-04 08:16:15 UTC
Last measurement: 2017-07-16 21:17:07 UTC
Nominal number of samples per batch: 4
Number of complete batches: 508
Total number of samples: 2151
Utilized technology: 4G/LTE
Operators: 3 Sweden, Telia Norway, Telia Sweden, .ice Norway, Telenor Norway, Telenor Sweden
Server: Sweden (*Karlstad University*)
Clients: MONROE nodes in Norway (470, 471) and Sweden (478, 479)
Tested number of parallel TCP flows: 1, 3, 5, 7
Nominal measurement duration: 15 s
Investigated parameters: Number of flows
Mean Downlink duration: 16.70 s
Mean Uplink duration: 17.92 s
Median measured Downlink: 33.46 Mbit/s
Median measured Uplink: 12.41 Mbit/s
Total data usage: 244.96 GB

A.17 Campaign 17

First measurement: 2017-07-21 23:01:36 UTC
Last measurement: 2017-08-06 20:40:54 UTC
Nominal number of samples per batch: 4
Number of complete batches: 714
Total number of samples: 2973
Utilized technology: 4G/LTE
Operators: 3 Sweden, Telia Norway, Telia Sweden, .ice Norway, Telenor Norway, Telenor Sweden
Server: Sweden (*Karlstad University*)
Clients: MONROE nodes in Norway (470, 471) and Sweden (478, 479)

Tested number of parallel TCP flows: 1, 3, 5, 7
Nominal measurement duration: 15 s
Investigated parameters: Number of flows
Mean Downlink duration: 16.67 s
Mean Uplink duration: 17.99 s
Median measured Downlink: 36.20 Mbit/s
Median measured Uplink: 13.99 Mbit/s
Total data usage: 382.34 GB

A.18 Campaign 18

First measurement: 2017-11-07 17:24:16 UTC
Last measurement: 2017-11-17 15:03:30 UTC
Nominal number of samples per batch: 12
Number of complete batches: 638
Total number of samples: 6668
Utilized technology: 4G/LTE
Operators: 3 Sweden, Telia Norway, Telia Sweden, .ice Norway, Telenor Norway, Telenor Sweden
Servers: Second server in Norway (*University of Oslo*), Sweden (*Karlstad University*)
Clients: MONROE nodes in Norway (470, 471) and Sweden (478, 479)
Tested number of parallel TCP flows: 3, 5, 7, 9, 12, 20
Nominal measurement duration: 15 s
Investigated parameters: Number of flows and server location
Mean Downlink duration: 17.34 s
Mean Uplink duration: 18.24 s
Median measured Downlink: 30.23 Mbit/s
Median measured Uplink: 15.54 Mbit/s
Total data usage: 922.12 GB

APPENDIX B

Source Code

This chapter presents listings of the C code produced in the course of this thesis work. Chapter 4 goes into detail regarding the implementation, configuration parameters and the output format.

The source code can also be found under:

`https://github.com/lwimmer/rmbt-client` or
`https://gitlab.com/lwimmer/rmbt-client`

Listing B.1: rmbt_common.h

```
1  /*******************************************************************************
2   * Copyright 2017 Leonhard Wimmer
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *  http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
15  ******************************************************************************/
16
17 #ifndef SRC_RMBT_COMMON_H_
18 #define SRC_RMBT_COMMON_H_
19
20 #if HAVE_CONFIG_H
21 # include <config.h>
22 #endif
23
24 #ifndef _GNU_SOURCE
25 # define _GNU_SOURCE 1
26 #endif
```

```
27  // #define _POSIX_C_SOURCE 200809L
28  // #define _ISOC11_SOURCE
29  /* #define _BSD_SOURCE */
30
31  #include <stddef.h>
32  #include <stdio.h>
33  #include <stdlib.h>
34  #include <time.h>
35  #include <pthread.h>
36  #include <unistd.h>
37  #include <inttypes.h>
38  #include <stdbool.h>
39  #include <string.h>
40
41  #endif /* SRC_RMBT_COMMON_H_ */
```

Listing B.2: rmbt.h

```
1  /*****************************************************************************
2   * Copyright 2017 Leonhard Wimmer
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *  http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
15  *****************************************************************************/
16
17  #ifndef SRC_RMBT_H_
18  #define SRC_RMBT_H_
19
20  #include "rmbt_common.h"
21
22  #endif /* SRC_RMBT_H_ */
```

Listing B.3: rmbt.c

```
1  /*****************************************************************************
2   * Copyright 2017 Leonhard Wimmer
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *  http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
15  *****************************************************************************/
```

```
16
17  #include "rmbt.h"
18
19  #include <uuid.h>
20  #include <signal.h>
21
22  #include "rmbt_config.h"
23  #include "rmbt_helper.h"
24  #include "rmbt_token.h"
25  #include "rmbt_flow.h"
26  #include "rmbt_ssl.h"
27  #include "rmbt_stats.h"
28  #include "rmbt_json.h"
29  #include "rmbt_compress.h"
30
31  #define MAX_TO_FREE 64
32
33  static void *to_free[MAX_TO_FREE];
34  static uint_fast16_t to_free_cnt;
35
36  static void print_help(void) {
37    fprintf(stderr, "command line arguments:\n\n"
38        " -c     json config file; use \"-\" to read from stdin\n"
39        " -b     local ip to bind\n"
40        " -h     host to connect to\n"
41        " -p     port to connect to\n"
42        " -e     connect using SSL/TLS\n"
43        " -t     token to use (either -t or -s is needed)\n"
44        " -s     secret for token generation\n"
45        " -f     number of flows\n"
46        " -d     measurement duration for downlink\n"
47        " -u     measurement duration for uplink\n"
48        " -n     number of rtt_tcp_payloads\n\n"
49        "Default config:\n"
50        "%s\n", DEFAULT_CONFIG);
51  }
52
53  static void remember_to_free(void *ptr) {
54    if (to_free_cnt >= MAX_TO_FREE)
55      fail("to_free limit reached");
56    to_free[to_free_cnt++] = ptr;
57  }
58
59  static void do_free(void) {
60    for (uint_fast16_t i = 0; i < to_free_cnt; i++) {
61      free(to_free[i]);
62      to_free[i] = NULL;
63    }
64    to_free_cnt = 0;
65  }
66
67  static void my_json_get_string(char **dst, rmbt_json json, const char *key) {
68    char *value = NULL;
69    rmbt_json_get_string_alloc(&value, json, key);
70    if (value != NULL) {
71      remember_to_free(value);
72      *dst = value;
73    }
74  }
75
76  static void read_config(TestConfig *c, rmbt_json json) {
77    my_json_get_string(&c->bind_ip, json, "cnf_bind_ip");
```

```
78    my_json_get_string(&c->server_host, json, "cnf_server_host");
79    my_json_get_string(&c->server_port, json, "cnf_server_port");
80    rmbt_json_get_bool(&c->encrypt, json, "cnf_encrypt");
81    rmbt_json_get_bool(&c->encrypt_debug, json, "cnf_encrypt_debug");
82    my_json_get_string(&c->cipherlist, json, "cnf_cipherlist");
83    my_json_get_string(&c->secret, json, "cnf_secret");
84    my_json_get_string(&c->token, json, "cnf_token");
85
86    int_fast16_t timeout_s = 0;
87    rmbt_json_get_int_fast16_t(&timeout_s, json, "cnf_timeout_s");
88    if (timeout_s > 0)
89      c->timeout_ms = (int) (timeout_s * 1000);
90
91    rmbt_json_get_int_fast16_t(&c->dl_num_flows, json, "cnf_dl_num_flows");
92    rmbt_json_get_int_fast16_t(&c->ul_num_flows, json, "cnf_ul_num_flows");
93    rmbt_json_get_int_fast16_t(&c->dl_duration_s, json, "cnf_dl_duration_s");
94    rmbt_json_get_int_fast16_t(&c->ul_duration_s, json, "cnf_ul_duration_s");
95    rmbt_json_get_int_fast16_t(&c->rtt_tcp_payload_num, json,
      ↪  "cnf_rtt_tcp_payload_num");
96    rmbt_json_get_int_fast16_t(&c->dl_pretest_duration_s, json,
      ↪  "cnf_dl_pretest_duration_s");
97    rmbt_json_get_int_fast16_t(&c->ul_pretest_duration_s, json,
      ↪  "cnf_ul_pretest_duration_s");
98    rmbt_json_get_int_fast16_t(&c->dl_wait_time_s, json, "cnf_dl_wait_time_s");
99    rmbt_json_get_int_fast16_t(&c->ul_wait_time_s, json, "cnf_ul_wait_time_s");
100   rmbt_json_get_int_fast32_t(&c->tcp_info_sample_rate_us, json,
      ↪  "cnf_tcp_info_sample_rate_us");
101
102   my_json_get_string(&c->file_summary, json, "cnf_file_summary");
103   my_json_get_string(&c->file_flows, json, "cnf_file_flows");
104   my_json_get_string(&c->file_stats, json, "cnf_file_stats");
105 }
106
107 static char *read_stdin(void) {
108   size_t size = 512, min = 128, len = 0;
109   char *p, *input = malloc(size);
110
111   while (!feof(stdin)) {
112     if (size - len <= min) {
113       size *= 2;
114       p = realloc(input, size);
115       if (p == NULL) {
116         free(input);
117         return NULL;
118       }
119       input = p;
120     }
121     size_t num_read = fread(input + len, 1, size - len - 1, stdin); /* -1: reserve
      ↪  space for '\0' */
122     len += num_read;
123   }
124   *(input + len++) = '\0';
125   return input;
126 }
127
128 int main(int argc, char **argv) {
129   fprintf(stderr, "==== rmbt %s ====\n", RMBT_VERSION);
130
131   TestConfig config = { 0 };
132
133   struct sigaction action = { .sa_handler = SIG_IGN };
134   if (sigaction(SIGPIPE, &action, NULL) != 0)
```

```
135       fail("sigaction");
136
137     rmbt_json default_json = rmbt_parse_json(DEFAULT_CONFIG);
138     if (default_json == NULL)
139       fail("could not read default config");
140     read_config(&config, default_json);
141     rmbt_json_free(default_json);
142
143     rmbt_json add_to_result = NULL;
144     rmbt_json json;
145     char *input;
146     int c, r;
147     while ((c = getopt(argc, argv, "?c:b:h:p:et:s:f:d:u:n:v")) != -1)
148       switch (c) {
149
150       case 'c':
151         if (strcmp("-", optarg) == 0) { /* read from stdin */
152           input = read_stdin();
153           if (input == NULL)
154             fail("could not read config from stdin");
155           json = rmbt_parse_json(input);
156           free(input);
157         } else {
158           json = rmbt_json_read_from_file(optarg);
159           if (json == NULL)
160             fail("could not read config file '%s'", optarg);
161           printf("%s\n", rmbt_json_to_string(json, true));
162         }
163         read_config(&config, json);
164         if (rmbt_json_get_object(&add_to_result, json, "cnf_add_to_result"))
165           json_object_get(add_to_result);
166         rmbt_json_free(json);
167         break;
168       case 'b':
169         config.bind_ip = optarg;
170         break;
171
172       case 'h':
173         config.server_host = optarg;
174         break;
175
176       case 'p':
177         config.server_port = optarg;
178         break;
179
180       case 'e':
181         config.encrypt = true;
182         break;
183
184       case 't':
185         if (config.secret != NULL)
186           fail("arguments -t and -s are mutually exclusive");
187         config.token = optarg;
188         break;
189
190       case 's':
191         if (config.token != NULL)
192           fail("arguments -t and -s are mutually exclusive");
193         config.secret = optarg;
194         break;
195
196       case 'f':
```

```
197         r = sscanf(optarg, "%" PRIdFAST16, &config.dl_num_flows);
198         if (r <= 0)
199           fail("could not parse argument to -%c: %s", c, optarg);
200         config.ul_num_flows = config.dl_num_flows;
201         break;
202
203       case 'd':
204         r = sscanf(optarg, "%" PRIdFAST16, &config.dl_duration_s);
205         if (r <= 0)
206           fail("could not parse argument to -%c: %s", c, optarg);
207         break;
208
209       case 'u':
210         r = sscanf(optarg, "%" PRIdFAST16, &config.ul_duration_s);
211         if (r <= 0)
212           fail("could not parse argument to -%c: %s", c, optarg);
213         break;
214
215       case 'n':
216         r = sscanf(optarg, "%" PRIdFAST16, &config.rtt_tcp_payload_num);
217         if (r <= 0)
218           fail("could not parse argument to -%c: %s", c, optarg);
219         break;
220
221       case 'v':
222         fprintf(stderr, "rmbt version: %s\n", RMBT_VERSION);
223         return EXIT_SUCCESS;
224
225       default:
226       case '?':
227         print_help();
228         return EXIT_SUCCESS;
229       }
230
231     if (config.server_host == NULL)
232       fail("host is required (either via config file or -h)");
233
234     if (config.server_port == 0)
235       fail("port is required (either via config file or -p)");
236
237     if (config.secret == NULL && config.token == NULL)
238       fail("either token or secret is required (either via config file, -s or -t)");
239
240     char time_str[20];
241     snprintf(time_str, sizeof(time_str), "%ld", time(NULL));
242
243     if (config.token == NULL) {
244       /* need to generate token */
245       uuid_t uuid;
246       uuid_generate_random(uuid);
247       char uuid_str[37];
248       uuid_unparse_lower(uuid, uuid_str);
249
250       char hmac_str[EVP_MAX_MD_SIZE * 2];
251       calc_token(config.secret, uuid_str, time_str, hmac_str, sizeof(hmac_str));
252
253       int size = snprintf(config.token, 0, "%s_%s_%s", uuid_str, time_str,
          ↪ hmac_str);
254       if (size < 0)
255         fail("error while generating token");
256       size++;
257       config.token = malloc((size_t) size);
```

```
258       remember_to_free(config.token);
259       snprintf(config.token, (size_t) size, "%s_%s_%s", uuid_str, time_str,
          ↪  hmac_str);
260     }
261
262     char uuid_str[37];
263     if (sscanf(config.token, "%36[0-9a-f-]", uuid_str) != 1)
264       fail("could not get uuid from token");
265
266     init_ssl(config.encrypt);
267
268     Result result = { .id_test = uuid_str };
269
270     int_fast16_t num_threads = config.dl_num_flows;
271     if (config.ul_num_flows > num_threads)
272       num_threads = config.ul_num_flows;
273
274     ThreadArg thread_arg[num_threads];
275     memset(thread_arg, 0, sizeof(thread_arg));
276     FlowResult flow_results[num_threads];
277     memset(flow_results, 0, sizeof(flow_results));
278     StatsThreadEntry stats_entries[num_threads];
279     memset(stats_entries, 0, sizeof(stats_entries));
280     RmbtBarrier barrier = RMBT_BARRIER_INITIALIZER;
281     barrier.total = num_threads;
282
283     result.time_start_s = time(NULL);
284
285     struct timespec ts_zero;
286     ts_fill(&ts_zero);
287
288     for (int_fast16_t t = 0; t < num_threads; t++) {
289       thread_arg[t].cfg = &config;
290       thread_arg[t].thread_num = t;
291       thread_arg[t].thread_count = num_threads;
292       thread_arg[t].ts_zero = &ts_zero;
293       thread_arg[t].flow_result = &flow_results[t];
294       thread_arg[t].barrier = &barrier;
295       thread_arg[t].do_log = t == 0;
296       thread_arg[t].do_rtt_tcp_payload = t == 0;
297       thread_arg[t].do_downlink = t < config.dl_num_flows;
298       thread_arg[t].do_uplink = t < config.ul_num_flows;
299       r = pthread_create(&thread_arg[t].thread, NULL, &run_test_thread_start,
          ↪  &thread_arg[t]);
300       if (r != 0)
301         fail_errno(r, "could not create thread");
302     }
303
304     pthread_t stats_thread;
305     StatsThreadArg starg = { .ts_zero = &ts_zero, .length = (size_t)num_threads,
          ↪  .entries = stats_entries, .tcp_info_sample_rate_us =
          ↪  config.tcp_info_sample_rate_us };
306     if (config.tcp_info_sample_rate_us > 0) {
307       stats_set_arg(&starg);
308       fprintf(stderr, "starting stats thread (sampling rate: % "PRIdFAST32 "us)\n",
          ↪  config.tcp_info_sample_rate_us);
309       r = pthread_create(&stats_thread, NULL, &stats_thread_start, NULL);
310       if (r != 0)
311         fail_errno(r, "could not create stats thread");
312     }
313
314     for (int_fast16_t t = 0; t < num_threads; t++) {
```

```
315       r = pthread_join(thread_arg[t].thread, NULL);
316       if (r != 0)
317         fail_errno(r, "could not join thread");
318     }
319
320     if (config.tcp_info_sample_rate_us > 0) {
321       fprintf(stderr, "stopping stats thread\n");
322       r = pthread_cancel(stats_thread);
323       if (r != 0)
324         fail_errno(r, "could not cancel stats thread");
325       r = pthread_join(stats_thread, NULL);
326       if (r != 0)
327         fail_errno(r, "could not join stats thread");
328     }
329
330     result.time_end_s = time(NULL);
331
332     calc_results(&result, flow_results, num_threads);
333
334     fprintf(stderr, "dl_throughput_mbps = %.6f\n", result.dl_throughput_kbps /
        ↪  1000);
335     fprintf(stderr, "ul_throughput_mbps = %.6f\n", result.ul_throughput_kbps /
        ↪  1000);
336
337     rmbt_json result_json = collect_summary_results(&result);
338     flatten_json(result_json, add_to_result);
339     printf("%s\n", rmbt_json_to_string(result_json, true));
340
341     const char *replacements[] = { \
342         "id_test", result.id_test, \
343         "time", time_str };
344     size_t num_replacements = sizeof(replacements) / sizeof(char*) / 2;
345
346     char buf[512];
347     if (config.file_summary != NULL) {
348       bool ok = variable_subst(buf, sizeof(buf), config.file_summary, replacements,
        ↪  num_replacements);
349       rmbt_write_to_file(ok ? buf : config.file_summary,
        ↪  rmbt_json_to_string(result_json, false));
350     }
351     rmbt_json_free(result_json);
352
353     if (config.file_flows != NULL) {
354       bool ok = variable_subst(buf, sizeof(buf), config.file_flows, replacements,
        ↪  num_replacements);
355
356       rmbt_json raw_result_json = collect_raw_results(&result, flow_results,
        ↪  num_threads);
357       flatten_json(raw_result_json, add_to_result);
358       rmbt_write_to_file(ok ? buf : config.file_flows,
        ↪  rmbt_json_to_string(raw_result_json, false));
359       rmbt_json_free(raw_result_json);
360     }
361
362     if (config.file_stats != NULL) {
363         bool ok = variable_subst(buf, sizeof(buf), config.file_stats, replacements,
        ↪  num_replacements);
364
365         rmbt_json_array stats_json = get_stats_as_json_array(&starg);
366         rmbt_write_to_file(ok ? buf : config.file_stats,
        ↪  rmbt_json_to_string(stats_json, false));
367         rmbt_json_free_array(stats_json);
```

```
368       }
369
370     if (add_to_result != NULL)
371       rmbt_json_free(add_to_result);
372
373     shutdown_ssl();
374
375     for (int_fast16_t t = 0; t < num_threads; t++)
376       free(stats_entries[t].tcp_infos);
377     do_free_flow_results(flow_results, num_threads);
378     do_free();
379
380     fprintf(stderr, "Exiting.\n");
381     return 0;
382   }
```

Listing B.4: rmbt_compress.h

```
1   /*******************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
14   * limitations under the License.
15   *******************************************************************************/
16
17  #ifndef SRC_RMBT_COMPRESS_H_
18  #define SRC_RMBT_COMPRESS_H_
19
20  #include "rmbt_common.h"
21
22  bool rmbt_write_to_file(const char *filename, const char *data);
23
24  #endif /* SRC_RMBT_COMPRESS_H_ */
```

Listing B.5: rmbt_compress.c

```
1   /*******************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
```

```
14   * limitations under the License.
15   ******************************************************************************/
16
17  #include "rmbt_compress.h"
18
19  #include <string.h>
20  #include <errno.h>
21
22  #ifdef HAVE_LZMA
23
24  #define RMBT_COMPRESS_EXT ".xz"
25
26  #include <lzma.h>
27
28  #define LZMA_PRESET 4
29
30  static bool rmbt_compress(const char *input_data, size_t input_data_length,
31      FILE *f) {
32
33    lzma_stream strm = LZMA_STREAM_INIT;
34    lzma_ret ret = lzma_easy_encoder(&strm, LZMA_PRESET, LZMA_CHECK_CRC32);
35    if (ret != LZMA_OK) {
36      fprintf(stderr, "error in lzma_easy_encoder");
37      return false;
38    }
39
40    uint8_t outbuf[BUFSIZ];
41    strm.next_in = (const uint8_t *) input_data;
42    strm.avail_in = input_data_length;
43    strm.next_out = outbuf;
44    strm.avail_out = sizeof(outbuf);
45
46    for (;;) {
47      ret = lzma_code(&strm, LZMA_FINISH);
48      if (strm.avail_out == 0 || ret == LZMA_STREAM_END) {
49        size_t write_size = sizeof(outbuf) - strm.avail_out;
50        if (fwrite(outbuf, 1, write_size, f) != write_size) {
51          fprintf(stderr, "error while writing to file: %s",
52              strerror(errno));
53          return false;
54        }
55        if (ret == LZMA_STREAM_END)
56          break;
57        strm.next_out = outbuf;
58        strm.avail_out = sizeof(outbuf);
59
60      }
61      if (ret != LZMA_OK) {
62        fprintf(stderr, "error in lzma_code");
63        return false;
64      }
65    }
66
67    lzma_end(&strm);
68
69    return true;
70  }
71
72  #endif /* HAVE_LZMA */
73
74  bool rmbt_write_to_file(const char *filename, const char *data) {
75
```

```
76    FILE *f = fopen(filename, "w");
77    if (f == NULL) {
78      fprintf(stderr, "error while opening %s: %s", filename,
79          strerror(errno));
80      return false;
81    }
82
83    size_t data_len = strlen(data);
84
85    bool done = false;
86  #ifdef RMBT_COMPRESS_EXT
87    char *dot = strrchr(filename, '.');
88    if (dot && !strcmp(dot, RMBT_COMPRESS_EXT)) {
89      if (!rmbt_compress(data, data_len, f)) {
90        fprintf(stderr, "error while compressing %s", filename);
91        return false;
92      }
93      done = true;
94    }
95  #endif
96    if (!done) {
97      if (fwrite(data, 1, data_len, f) != data_len) {
98        fprintf(stderr, "error while writing to %s: %s", filename,
99            strerror(errno));
100       return false;
101     }
102   }
103
104   fclose(f);
105
106   return true;
107 }
```

Listing B.6: rmbt_flow.h

```
1   /*******************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
14   * limitations under the License.
15   ******************************************************************************/
16
17  #ifndef SRC_RMBT_FLOW_H_
18  #define SRC_RMBT_FLOW_H_
19
20  #include "rmbt_common.h"
21  #include "rmbt_result.h"
22
23  #include <stdatomic.h>
24
25  typedef struct {
```

```
26    char *bind_ip, *server_host, *server_port, *cipherlist, *secret, *token,
      ↪  *test_id, *file_summary, *file_flows, *file_stats;
27    int_fast16_t dl_num_flows, ul_num_flows, dl_duration_s, ul_duration_s,
      ↪  rtt_tcp_payload_num, dl_pretest_duration_s, ul_pretest_duration_s,
      ↪  dl_wait_time_s,
28        ul_wait_time_s;
29    int_fast32_t tcp_info_sample_rate_us;
30    int timeout_ms;
31    bool encrypt, encrypt_debug;
32  } TestConfig;
33
34  typedef struct {
35    atomic_bool global_abort;
36    pthread_mutex_t mutex;
37    pthread_cond_t cond;
38    int_fast16_t total;
39    int_fast16_t entered;
40    int_fast16_t left;
41  } RmbtBarrier;
42
43  #define RMBT_BARRIER_INITIALIZER { false, PTHREAD_MUTEX_INITIALIZER,
    ↪  PTHREAD_COND_INITIALIZER, 0, 0, 0 }
44
45  typedef struct {
46    TestConfig *cfg;
47    pthread_t thread;
48    int_fast16_t thread_num;
49    int_fast16_t thread_count;
50    struct timespec *ts_zero;
51    RmbtBarrier *barrier;
52    FlowResult *flow_result;
53    bool do_log;
54    bool do_rtt_tcp_payload;
55    bool do_uplink;
56    bool do_downlink;
57  } ThreadArg;
58
59  #define RETURN_IF_NOK(x) if (!(x)) return false;
60
61  void *run_test_thread_start(void *arg);
62
63  #endif /* SRC_RMBT_FLOW_H_ */
```

Listing B.7: rmbt_flow.c

```
1   /*******************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
14   * limitations under the License.
15   *******************************************************************************/
```

```
16
17  #include "rmbt_flow.h"
18
19  #include <stdbool.h>
20  #include <sys/types.h>
21  #include <sys/socket.h>
22  #include <netinet/in.h>
23  #include <netinet/tcp.h>
24  #include <fcntl.h>
25  #include <netdb.h>
26  #include <arpa/inet.h>
27  #include <poll.h>
28
29  #include "rmbt_helper.h"
30  #include "rmbt_ssl.h"
31  #include "rmbt_stats.h"
32
33  #define MAX_CHUNKSIZE  65536
34  #define DATAPOINT_INCREMENT_PRETEST 32
35  #define DATAPOINT_INCREMENT_MAIN 51200
36
37  #define NL_C       '\n'
38  #define NL         "\n"
39  #define WHITESPACE     " \t"
40  #define EMPTY       ""
41  #define GETCHUNKS     "GETCHUNKS"
42  #define GETTIME       "GETTIME"
43  #define PUT         "PUT"
44  #define PUTNORESULT     "PUTNORESULT"
45  #define PING       "PING"
46  #define PONG       "PONG"
47  #define QUIT       "QUIT"
48  #define OK         "OK"
49  #define ERR         "ERR"
50  #define BYE         "BYE"
51  #define ACCEPT       "ACCEPT"
52  #define CHUNKSIZE     "CHUNKSIZE"
53  #define TOKEN       "TOKEN"
54  #define RMBTv       "RMBTv"
55
56  #define BYTE_CONTINUE  0x00
57  #define BYTE_END    0xff
58
59  #define M_TOKEN       0x0001
60  #define M_QUIT       0x0002
61  #define M_GETCHUNKS     0x0004
62  #define M_GETTIME     0x0008
63  #define M_PUT       0x0010
64  #define M_PUTNORESULT  0x0020
65  #define M_PING       0x0040
66  #define M_OK       0x0100
67  #define M_ERR       0x0200
68  #define M_BYE       0x0400
69
70  #define MASK_IS_SET(mask,bit)   ((mask & bit) == bit)
71  #define IS_OK(mask)     MASK_IS_SET(mask,M_OK)
72  #define IS_ERR(mask)   MASK_IS_SET(mask,M_ERR)
73
74  #define BUF_SIZE    512
75
76  #define NUM_ERRORS    16
77
```

```
78  #define I_1E9 (int_fast64_t) 1000000000
79
80  #define BARRIER RETURN_IF_NOK(barrier_wait(s))
81
82  #define IS_SSL_WANT_READ_OR_WRITE(x)  ((x == SSL_ERROR_WANT_READ) || (x ==
    ↪  SSL_ERROR_WANT_WRITE))
83
84  #define NEED_POLL_READ  -2
85  #define NEED_POLL_WRITE  -3
86  #define IS_NEED_POLL(x) (x == NEED_POLL_READ || x == NEED_POLL_WRITE)
87
88  #if !defined(TCP_CORK) && defined(TCP_NOPUSH) /* hack for now to make it work on
    ↪  *BSD */
89  #define TCP_CORK TCP_NOPUSH
90  #endif
91
92  typedef uint_fast16_t Mask;
93
94  typedef struct {
95    const TestConfig *config;
96    ThreadArg *targ;
97    unsigned char *buf_chunk;
98    SSL *ssl;
99    int socket_fd;
100   long unread_buf_s;
101   long unread_buf_e;
102   Mask mask;
103   char unread_buf[BUF_SIZE];
104   char *error[NUM_ERRORS];
105   bool have_err;
106   bool need_reconnect;
107 } State;
108
109 __attribute__ ((hot)) inline static int_fast64_t get_relative_time_ns(State *s) {
110   return ts_diff(s->targ->ts_zero);
111 }
112
113 __attribute__ ((format (printf, 2, 3))) static bool add_error(State *s, const char
    ↪  *fmt, ...) {
114   s->targ->barrier->global_abort = true;
115   pthread_cond_broadcast(&s->targ->barrier->cond);
116
117   s->have_err = true;
118   for (uint_fast16_t i = 0; i < NUM_ERRORS; i++) {
119     if (s->error[i] == NULL) {
120       va_list ap;
121       va_start(ap, fmt);
122       int r = vasprintf(&(s->error[i]), fmt, ap);
123       va_end(ap);
124       if (r == -1)
125         perror("could not add_error");
126       break;
127     }
128   }
129   return false;
130 }
131
132 static void collect_ssl_errors(State *s) {
133   const char *file = NULL;
134   const char *data = NULL;
135   int line;
136   int flags = ERR_TXT_STRING;
```

```
137
138   unsigned long err;
139   while ((err = ERR_get_error_line_data(&file, &line, &data, &flags)) != 0) {
140     char buf[256];
141     ERR_error_string_n(err, buf, sizeof(buf)); // ERR_error_string is NOT thread
      ↪  safe (as it seems)
142     add_error(s, "%s:%s:%d:%s", ERR_error_string(err, NULL), file, line, data);
143   }
144 }
145
146 __attribute__ ((format (printf, 2, 3))) static void my_log_force(__attribute__
    ↪  ((unused)) State *s, const char *fmt, ...) {
147   va_list ap;
148   va_start(ap, fmt);
149   vfprintf(stderr, fmt, ap);
150   va_end(ap);
151   fprintf(stderr, "\n");
152 }
153
154 __attribute__ ((format (printf, 2, 3))) static void my_log(State *s, const char
    ↪  *fmt, ...) {
155   if (s->targ->do_log) {
156     va_list ap;
157     va_start(ap, fmt);
158     vfprintf(stderr, fmt, ap);
159     va_end(ap);
160     fprintf(stderr, "\n");
161   }
162 }
163
164 static void get_errors(State *s, char *dst, size_t dst_size) {
165   collect_ssl_errors(s);
166   size_t len = 0;
167   for (uint_fast16_t i = 0; i < NUM_ERRORS; i++) {
168     if (s->error[i] != NULL) {
169       int res = snprintf(dst + len, dst_size - len, len == 0 ? "%s" : "; %s",
        ↪  s->error[i]);
170       if (res > 0)
171         len += (size_t) res;
172       if (res < 0 || dst_size - len <= 1)
173         return;
174     }
175   }
176 }
177
178 static void print_errors(State *s, FILE *stream, bool clear) {
179   collect_ssl_errors(s);
180   for (uint_fast16_t i = 0; i < NUM_ERRORS; i++) {
181     if (s->error[i] != NULL) {
182       fprintf(stream, "%s\n", s->error[i]);
183       if (clear) {
184         free(s->error[i]);
185         s->error[i] = NULL;
186       }
187     }
188   }
189   if (clear)
190     s->have_err = false;
191 }
192
193 /*
194  * We don't use pthread_barrier, as it is not available on Android.
```

```
195   * Also this way we can abort all threads more easily if one fails.
196   */
197  static bool barrier_wait(State *s) {
198    RmbtBarrier *b = s->targ->barrier;
199    pthread_mutex_lock(&b->mutex);
200    if (b->global_abort) {
201      pthread_cond_broadcast(&b->cond);
202      pthread_mutex_unlock(&b->mutex);
203      return false;
204    }
205
206    while (! b->global_abort && b->entered == b->total) /* not all threads have left
       ↪  the last barrier */
207      pthread_cond_wait(&b->cond, &b->mutex);
208
209    if (++b->entered == b->total)
210      pthread_cond_broadcast(&b->cond); /* if I was the last one, tell the others */
211
212    while (! b->global_abort && b->entered < b->total) /* the barrier. waiting for
       ↪  the others */
213      pthread_cond_wait(&b->cond, &b->mutex);
214
215    if (++b->left == b->total) { /* I was the last one to leave, cleanup */
216      b->entered = b->left = 0;
217      pthread_cond_broadcast(&b->cond); /* tell threads potentially waiting for the
         ↪  next barrier */
218    }
219
220    bool result = ! b->global_abort;
221    pthread_mutex_unlock(&b->mutex);
222    return result;
223  }
224
225  static inline void set_nodelay(State *s, int value) {
226    setsockopt(s->socket_fd, IPPROTO_TCP, TCP_NODELAY, &value, sizeof(value));
227  }
228
229  static inline void set_cork(State *s, int value) {
230    setsockopt(s->socket_fd, IPPROTO_TCP, TCP_CORK, &value, sizeof(value));
231  }
232
233  static inline void set_throughput(State *s) {
234    set_nodelay(s, 0);
235    set_cork(s, 1);
236  }
237
238  static inline void set_low_delay(State *s) {
239    set_cork(s, 0);
240    set_nodelay(s, 1);
241  }
242
243  static inline int my_poll(State *s, bool read, bool write, int ssl_err) {
244    if (ssl_err != 0) {
245      switch (ssl_err) {
246      case SSL_ERROR_WANT_READ:
247        read = true;
248        break;
249      case SSL_ERROR_WANT_WRITE:
250        write = true;
251        break;
252      default:
253        break;
```

```
254       }
255     }
256     short int events = 0;
257     if (read)
258       events |= POLLIN;
259     if (write)
260       events |= POLLOUT;
261     if (events == 0) {
262       add_error(s, "no events to monior in poll");
263       return -1;
264     }
265     struct pollfd pfd = { .fd = s->socket_fd, .events = events };
266     int ret = poll(&pfd, 1, s->config->timeout_ms);
267     if (ret == 0)
268       add_error(s, "timeout");
269     else if (ret < 0)
270       add_error(s, "error in poll: %s", strerror(errno));
271     return ret;
272   }
273
274   static inline ssize_t my_write(State *s, unsigned char *buf, ssize_t size) {
275     ssize_t num_written, total_written = 0;
276     bool again;
277     do {
278       again = false;
279       if (s->ssl != NULL) {
280         num_written = SSL_write(s->ssl, buf + total_written, (int) (size -
          ↪  total_written));
281         if (num_written <= 0) {
282           int ssl_err = SSL_get_error(s->ssl, (int) num_written);
283           if (IS_SSL_WANT_READ_OR_WRITE(ssl_err)) {
284             int poll_res = my_poll(s, false, false, ssl_err);
285             if (poll_res > 0)
286               again = true;
287             else
288               return -1;
289           } else {
290             add_error(s, "error during SSL_write");
291             return -1;
292           }
293         }
294       } else {
295         num_written = write(s->socket_fd, buf + total_written, (size_t) (size -
          ↪  total_written));
296         if (num_written < 0) {
297           if (errno == EAGAIN) {
298             int poll_res = my_poll(s, false, true, 0);
299             if (poll_res > 0)
300               again = true;
301             else
302               return -1;
303           } else {
304             add_error(s, "error during write: %s", strerror(errno));
305             return -1;
306           }
307         }
308       }
309       if (num_written >= 0) {
310         total_written += num_written;
311         if (total_written == size)
312           return total_written;
313       }
```

```
314     } while (total_written < size || again);
315     add_error(s, "unknown error in write");
316     return -1;
317   }
318
319   static inline bool unread_empty(State *s) {
320     return s->unread_buf_s == s->unread_buf_e;
321   }
322
323   /*
324    * returns NEED_POLL_READ on EAGAIN or SSL_ERROR_WANT_READ if nonblocking
325    * returns NEED_POLL_WRITE on SSL_read if SSL_ERROR_WANT_WRITE
326    */
327   static inline long my_read(State *s, unsigned char *buf, ssize_t size, bool
      ↪  nonblocking) {
328     if (unread_empty(s)) {
329       bool again;
330       do {
331         again = false;
332         ssize_t num_read;
333         if (s->ssl != NULL) {
334           num_read = SSL_read(s->ssl, buf, (int) size);
335           if (num_read <= 0) {
336             int ssl_err = SSL_get_error(s->ssl, (int) num_read);
337             if (IS_SSL_WANT_READ_OR_WRITE(ssl_err)) {
338               if (nonblocking)
339                 return ssl_err == SSL_ERROR_WANT_WRITE ? NEED_POLL_WRITE :
                    ↪  NEED_POLL_READ;
340               int poll_res = my_poll(s, false, false, ssl_err);
341               if (poll_res > 0)
342                 again = true;
343               else
344                 return -1;
345             } else {
346               add_error(s, "error during SSL_read");
347               return -1;
348             }
349           }
350         } else {
351           num_read = read(s->socket_fd, buf, (size_t) size);
352           if (num_read < 0) {
353             if (errno == EAGAIN) {
354               if (nonblocking)
355                 return NEED_POLL_READ;
356               int poll_res = my_poll(s, true, false, 0);
357               if (poll_res > 0)
358                 again = true;
359               else
360                 return -1;
361             } else {
362               add_error(s, "error during read: %s", strerror(errno));
363               return -1;
364             }
365           }
366         }
367         if (num_read >= 0) {
368           s->targ->flow_result->flow_bytes_dl += num_read;
369           return num_read;
370         }
371       } while (again);
372       add_error(s, "unknown error in read");
373       return -1;
```

```
374     } else {
375       long bs = sizeof(s->unread_buf);
376       long unread_size = s->unread_buf_e - s->unread_buf_s;
377       if (unread_size < 0)
378         unread_size += bs;
379       if (unread_size < size)
380         size = unread_size;
381       long first_copy = size, second_copy = 0;
382       long max_first = bs - s->unread_buf_s;
383       if (first_copy > max_first) {
384         second_copy = first_copy - max_first;
385         first_copy = max_first;
386       }
387       memcpy(buf, s->unread_buf + s->unread_buf_s, (size_t) first_copy);
388       if (second_copy != 0)
389         memcpy(buf + first_copy, s->unread_buf, (size_t) second_copy);
390       s->unread_buf_s += size;
391       if (s->unread_buf_s >= bs)
392         s->unread_buf_s -= bs;
393       if (s->unread_buf_s == s->unread_buf_e)
394         s->unread_buf_s = s->unread_buf_e = 0;
395       return size;
396     }
397   }
398
399   static void my_unread(State *s, unsigned char *buf, long size) {
400     long bs = sizeof(s->unread_buf);
401     long max = s->unread_buf_s - s->unread_buf_e;
402     if (max <= 0)
403       max += bs;
404     if (max < size)
405       size = max;
406     long first_copy = size, second_copy = 0;
407     long max_first = bs - s->unread_buf_e;
408     if (first_copy > max_first) {
409       second_copy = first_copy - max_first;
410       first_copy = max_first;
411     }
412     memcpy(s->unread_buf + s->unread_buf_e, buf, (size_t) first_copy);
413     if (second_copy != 0)
414       memcpy(s->unread_buf, buf + first_copy, (size_t) second_copy);
415     s->unread_buf_e += size;
416     if (s->unread_buf_e >= bs)
417       s->unread_buf_e -= bs;
418   }
419
420   /*
421    * may return NEED_POLL_READ / NEED_POLL_WRITE
422    */
423   static inline long my_readline(State *s, unsigned char *buf, int size, bool
      ↪  nonblocking) {
424     unsigned char *buf_ptr = buf;
425     ssize_t size_remain = size;
426     long r;
427     unsigned char *nl_ptr = NULL;
428
429     do {
430       r = my_read(s, (void*) buf_ptr, size_remain, nonblocking);
431       if (nonblocking && IS_NEED_POLL(r)) {
432         if (buf_ptr > buf) {
433   //         printf("unread nb: %ld:%s\n", buf_ptr - buf, buf);
434           my_unread(s, buf, buf_ptr - buf);
```

```
435         }
436         return r;
437       }
438       if (r > 0) {
439         nl_ptr = memchr(buf_ptr, NL_C, (size_t) r);
440         buf_ptr += r;
441         size_remain -= r;
442       }
443     } while (r > 0 && nl_ptr == NULL && size_remain > 0);
444     if (nl_ptr == NULL && size_remain <= 0)
445       return -1;
446     if (nl_ptr != NULL) {
447       *nl_ptr = '\0';
448       if (nl_ptr + 1 < buf_ptr) {
449         //printf("unread: %ld:%s\n", buf_ptr - nl_ptr - 1, nl_ptr + 1);
450         my_unread(s, nl_ptr + 1, buf_ptr - nl_ptr - 1);
451       }
452       return nl_ptr - buf + 1;
453     } else
454       return buf_ptr - buf;
455   }
456
457   /*
458    * may return NEED_POLL_READ / NEED_POLL_WRITE
459    */
460   static long read_time_bytes(State *s, int_fast64_t *time, bool *got_bytes,
      ↪  int_fast64_t *bytes, bool nonblocking) {
461     unsigned char buf[BUF_SIZE];
462     long r = my_readline(s, buf, sizeof(buf), nonblocking);
463     if (nonblocking && IS_NEED_POLL(r))
464       return r;
465     if (r <= 0)
466       return add_error(s, "could not read TIME from server");
467     if (bytes != NULL)
468       r = sscanf((char*) buf, "TIME %" SCNdFAST64 " BYTES %"SCNdFAST64, time,
        ↪  bytes);
469     else
470       r = sscanf((char*) buf, "TIME %" SCNdFAST64, time);
471     if (r <= 0)
472       return add_error(s, "could not parse TIME from server: %s", buf);
473     if (got_bytes != NULL)
474       (*got_bytes) = (r == 2);
475     return true;
476   }
477
478   static inline bool read_time(State *s, int_fast64_t *time) {
479     return read_time_bytes(s, time, NULL, NULL, false);
480   }
481
482   static bool read_ok(State *s) {
483     unsigned char buf[BUF_SIZE];
484     long r = my_readline(s, buf, sizeof(buf), false);
485     if (r <= 0)
486       return add_error(s, "could not read OK from server");
487     if (strcmp(OK, (char*) buf) != 0)
488       return add_error(s, "expected server to send OK");
489     return true;
490   }
491
492   static bool read_ok_accept(State *s) {
493     unsigned char buf[BUF_SIZE];
494     s->mask = 0;
```

```
495     while (true) {
496       long r = my_readline(s, buf, sizeof(buf), false);
497       if (r <= 0)
498         return add_error(s, "could not read from server");
499
500       char *saveptr, *first, *rest;
501       first = strtok_r((char*) buf, WHITESPACE, &saveptr);
502       rest = strtok_r(NULL, EMPTY, &saveptr);
503
504       if (strcmp(OK, first) == 0) {
505         s->mask |= M_OK;
506       } else if (strcmp(ACCEPT, first) == 0) {
507         char *str, *part;
508         for (str = rest;; str = NULL) {
509           part = strtok_r(str, WHITESPACE, &saveptr);
510           if (part == NULL)
511             break;
512           if (strcmp(TOKEN, part) == 0)
513             s->mask |= M_TOKEN;
514           else if (strcmp(QUIT, part) == 0)
515             s->mask |= M_QUIT;
516           else if (strcmp(GETCHUNKS, part) == 0)
517             s->mask |= M_GETCHUNKS;
518           else if (strcmp(GETTIME, part) == 0)
519             s->mask |= M_GETTIME;
520           else if (strcmp(PUT, part) == 0)
521             s->mask |= M_PUT;
522           else if (strcmp(PUTNORESULT, part) == 0)
523             s->mask |= M_PUTNORESULT;
524           else if (strcmp(PING, part) == 0)
525             s->mask |= M_PING;
526         }
527         return true;
528       } else if (strcmp(CHUNKSIZE, first) == 0) {
529         int_fast32_t chunksize;
530         sscanf(rest, "%" SCNdFAST32, &chunksize);
531         if (chunksize <= 0 || chunksize > MAX_CHUNKSIZE)
532           return add_error(s, "server sent illegal CHUNKSIZE: %"PRIdFAST32" (max:
           ↪  %"PRIdFAST32")", chunksize, (int_fast32_t) MAX_CHUNKSIZE);
533
534         s->targ->flow_result->connection_info.chunksize = chunksize;
535         free(s->buf_chunk);
536         s->buf_chunk = calloc((size_t)
           ↪  (s->targ->flow_result->connection_info.chunksize), 1);
537       } else if (strcmp(ERR, first) == 0) {
538         s->mask |= M_ERR;
539         return add_error(s, "server responded with: %s", buf);
540       } else if (strcmp(BYE, first) == 0) {
541         s->mask |= M_BYE;
542         return true;
543       } else if (strncmp(RMBTv, first, strlen(RMBTv)) == 0) {
544         if (strlen(first) <
           ↪  sizeof(s->targ->flow_result->connection_info.server_version))
545           strncpy(s->targ->flow_result->connection_info.server_version, first,
             ↪  sizeof(s->targ->flow_result->connection_info.server_version));
546       } else
547         return add_error(s, "could not parse line from server: %s", buf);
548     }
549   }
550
551   __attribute__ ((format (printf, 2, 3))) static bool write_to_server(State *s,
    ↪  const char *fmt, ...) {
```

```
552     unsigned char buf[BUF_SIZE];
553     va_list ap;
554     va_start(ap, fmt);
555     ssize_t num = vsnprintf((char*) buf, sizeof(buf), fmt, ap);
556     if (num <= 0)
557       return add_error(s, "error while writing to server (vsnprintf)");
558     va_end(ap);
559     num = my_write(s, buf, num);
560     if (num <= 0)
561       return add_error(s, "error while writing to server");
562     s->targ->flow_result->flow_bytes_ul += num;
563     return true;
564   }
565
566   static void extract_ip_port_from_sockaddr(struct sockaddr_storage *addr, in_port_t
      ↪  *port, const void **ip_addr) {
567     if (addr->ss_family == AF_INET) { // IPv4
568       struct sockaddr_in *si = (struct sockaddr_in *) addr;
569       *port = si->sin_port;
570       *ip_addr = &si->sin_addr;
571     } else if (addr->ss_family == AF_INET6) { // IPv6
572       struct sockaddr_in6 *si = (struct sockaddr_in6 *) addr;
573       *port = si->sin6_port;
574       *ip_addr = &si->sin6_addr;
575     }
576   }
577
578   static bool connect_to_server(State *s) {
579
580     struct addrinfo *res, *rp, hints = { .ai_family = AF_UNSPEC, .ai_socktype =
      ↪  SOCK_STREAM, .ai_protocol = IPPROTO_TCP };
581     int gai_err = getaddrinfo(s->config->server_host, s->config->server_port, &hints,
      ↪  &res);
582     if (gai_err != 0)
583       return add_error(s, "error in getaddrinfo (for connect): %s",
        ↪  gai_strerror(gai_err));
584
585     int sfd = -1;
586     for (rp = res; rp != NULL; rp = rp->ai_next) {
587       sfd = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
588       if (sfd == -1)
589         continue;
590
591       if (s->config->bind_ip != NULL) {
592         struct addrinfo *res_bind, *rp_bind, hints_bind = { .ai_family =
          ↪  rp->ai_family, .ai_socktype = rp->ai_socktype, .ai_protocol =
          ↪  rp->ai_protocol };
593         gai_err = getaddrinfo(s->config->bind_ip, NULL, &hints_bind, &res_bind);
594         if (gai_err != 0)
595           return add_error(s, "error in getaddrinfo (for bind) (ip: %s): %s",
            ↪  s->config->bind_ip, gai_strerror(gai_err));
596         for (rp_bind = res_bind; rp_bind != NULL; rp_bind = rp_bind->ai_next) {
597           if (bind(sfd, rp_bind->ai_addr, rp_bind->ai_addrlen) == 0)
598             break; /* Success */
599         }
600         if (rp_bind == NULL)
601           return add_error(s, "could not bind to specified ip: %s",
            ↪  s->config->bind_ip);
602         freeaddrinfo(res_bind);
603       }
604
605       fcntl(sfd, F_SETFL, O_NONBLOCK);
```

```
606       if (connect(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
607         break; /* Success */
608       if (errno == EINPROGRESS) { // is nonblocking
609         int poll_res = my_poll(s, true, true, 0);
610         if (poll_res > 0) { // socket ready
611           int err;
612           socklen_t err_len = sizeof(err);
613           int r = getsockopt(sfd, SOL_SOCKET, SO_ERROR, &err, &err_len);
614           if (r == -1)
615             return add_error(s, "error in getsockopt: %s", strerror(errno));
616           if (err != 0)
617             return add_error(s, "error while connecting: %s", strerror(err));
618           break;
619         }
620       }
621       my_log_force(s, "could not connect: %s", strerror(errno));
622       close(sfd);
623     }
624     if (rp == NULL || sfd == -1)
625       return add_error(s, "could not connect to server");
626     freeaddrinfo(res);
627
628     s->socket_fd = sfd;
629
630     stats_thread_set_sfd(s->targ->thread_num, sfd);
631
632     char buf[64];
633     socklen_t buf_len = sizeof(buf);
634     if (getsockopt(sfd, 6, TCP_CONGESTION, &buf, &buf_len) == 0)
635       if (asprintf(&s->targ->flow_result->connection_info.tcp_congestion, "%s", buf)
       ↪  <= 0)
636         s->targ->flow_result->connection_info.tcp_congestion = NULL;
637
638     set_low_delay(s);
639
640     if (s->config->encrypt) {
641
642       pthread_mutex_lock(&ssl_ctx_mutex);
643       SSL *ssl = SSL_new(ssl_ctx);
644       if (ssl == NULL) {
645         pthread_mutex_unlock(&ssl_ctx_mutex);
646         return add_error(s, "error in SSL_new");
647       }
648
649       if (s->config->cipherlist != NULL && strlen(s->config->cipherlist) > 0) {
650         int r = SSL_set_cipher_list(ssl, s->config->cipherlist);
651         if (r <= 0) {
652           pthread_mutex_unlock(&ssl_ctx_mutex);
653           return add_error(s, "error while setting cipherlist: %s",
           ↪  s->config->cipherlist);
654         }
655       }
656
657       SSL_set_fd(ssl, sfd);
658
659       BIO_set_nbio(SSL_get_rbio(ssl), 1);
660       BIO_set_nbio(SSL_get_wbio(ssl), 1);
661
662       SSL_set_connect_state(ssl);
663
664       int ssl_ret, ssl_err, poll_ret;
665       do {
```

```
666         poll_ret = 0;
667         ERR_clear_error();
668         ssl_ret = SSL_connect(ssl);
669         if (ssl_ret == 1)
670           break;
671         ssl_err = SSL_get_error(ssl, ssl_ret);
672         if (IS_SSL_WANT_READ_OR_WRITE(ssl_err))
673           poll_ret = my_poll(s, false, false, ssl_err);
674         else {
675           pthread_mutex_unlock(&ssl_ctx_mutex);
676           return add_error(s, "error during SSL_connect");
677         }
678       } while (poll_ret > 0);
679       pthread_mutex_unlock(&ssl_ctx_mutex);
680
681       s->ssl = ssl;
682       s->targ->flow_result->connection_info.encrypt = true;
683       s->targ->flow_result->connection_info.cipher = SSL_get_cipher(ssl);
684       if (s->config->encrypt_debug)
685         s->targ->flow_result->connection_info.tls_debug = get_ssl_debug(ssl);
686     }
687     RETURN_IF_NOK(read_ok_accept(s));
688
689     struct sockaddr_storage addr;
690     socklen_t addr_len = sizeof(addr);
691     const void *ip_addr = NULL;
692     in_port_t port = 0;
693
694     if (getsockname(sfd, (struct sockaddr*) &addr, &addr_len) == 0) {
695       extract_ip_port_from_sockaddr(&addr, &port, &ip_addr);
696       inet_ntop(addr.ss_family, ip_addr,
          ↪  s->targ->flow_result->connection_info.ip_local,
          ↪  sizeof(s->targ->flow_result->connection_info.ip_local));
697       s->targ->flow_result->connection_info.port_local = ntohs(port);
698     }
699
700     if (getpeername(sfd, (struct sockaddr*) &addr, &addr_len) == 0) {
701       extract_ip_port_from_sockaddr(&addr, &port, &ip_addr);
702       inet_ntop(addr.ss_family, ip_addr,
          ↪  s->targ->flow_result->connection_info.ip_server,
          ↪  sizeof(s->targ->flow_result->connection_info.ip_server));
703       s->targ->flow_result->connection_info.port_server = ntohs(port);
704     }
705
706     return true;
707   }
708
709   static bool send_token(State *s) {
710     if (!MASK_IS_SET(s->mask, M_TOKEN))
711       return add_error(s, "expected server to accept TOKEN");
712
713     RETURN_IF_NOK(write_to_server(s, TOKEN " %s\n", s->config->token));
714     return read_ok_accept(s);
715   }
716
717   static bool connect_and_send_token(State *s) {
718     RETURN_IF_NOK(connect_to_server(s));
719     return send_token(s);
720   }
721
722   static bool disconnect(State *s) {
723     if (s->ssl != NULL) {
```

```
724       int err = SSL_shutdown(s->ssl);
725       if (err < 0)
726         add_error(s, "error in SSL_shutdown");
727       SSL_free(s->ssl);
728       s->ssl = NULL;
729     }
730     close(s->socket_fd);
731     stats_thread_set_sfd(s->targ->thread_num, -1);
732     return true;
733   }
734
735   static bool do_getchunks(State *s, int_fast32_t chunks, struct timespec *ts_zero,
      ↪  DataPoint *data_point) {
736     if (!MASK_IS_SET(s->mask, M_GETCHUNKS))
737       return add_error(s, "expected server to accept PING");
738
739     data_point->time_ns = ts_diff(ts_zero); // t_begin
740
741     RETURN_IF_NOK(write_to_server(s, GETCHUNKS " %" PRIuFAST32 NL, chunks));
742
743     int_fast64_t totalRead = 0;
744     long read;
745     unsigned char lastByte = 0;
746     const int_fast32_t chunksize = s->targ->flow_result->connection_info.chunksize;
747     do {
748       read = my_read(s, s->buf_chunk, chunksize, false);
749       if (read > 0) {
750         int_fast64_t posLast = chunksize - 1 - (totalRead % chunksize);
751         if (read > posLast)
752           lastByte = s->buf_chunk[posLast];
753         totalRead += read;
754       }
755     } while (read > 0 && lastByte != BYTE_END);
756
757     data_point->time_ns_end = ts_diff(ts_zero); // t_end
758
759     RETURN_IF_NOK(write_to_server(s, OK NL));
760
761     RETURN_IF_NOK(read_time(s, &data_point->duration_server));
762
763     return read_ok_accept(s);
764   }
765
766   static bool do_rtt_tcp_payload(State *s) {
767     unsigned char buf[BUF_SIZE];
768     RttTcpPayloadResult *rtt_tcp_payload_result =
      ↪  &s->targ->flow_result->rtt_tcp_payload;
769
770     int_fast16_t rtt_tcp_payload_num = s->config->rtt_tcp_payload_num;
771     RttTcpPayload *rtt_tcp_payloads = calloc((size_t) rtt_tcp_payload_num,
      ↪  sizeof(RttTcpPayload));
772
773     for (int_fast16_t i = 0; i < rtt_tcp_payload_num; i++) {
774       if (!MASK_IS_SET(s->mask, M_PING))
775         return add_error(s, "expected server to accept PING");
776
777       rtt_tcp_payloads[i].time_start_rel_ns = get_relative_time_ns(s);
778
779       struct timespec ts_start;
780       ts_fill(&ts_start);
781       RETURN_IF_NOK(write_to_server(s, PING NL));
782
```

```
783       long r = my_readline(s, buf, sizeof(buf), false);
784       rtt_tcp_payloads[i].rtt_client_ns = ts_diff(&ts_start);
785
786       if (r <= 0)
787         return add_error(s, "could not read PONG from server");
788       if (strcmp(PONG, (char *) buf) != 0)
789         return add_error(s, "expected PING, server sent: %s", buf);
790
791       RETURN_IF_NOK(write_to_server(s, OK NL));
792
793       RETURN_IF_NOK(read_time(s, &rtt_tcp_payloads[i].rtt_server_ns));
794
795       rtt_tcp_payloads[i].time_end_rel_ns = get_relative_time_ns(s);
796
797       RETURN_IF_NOK(read_ok_accept(s));
798     }
799
800     rtt_tcp_payload_result->rtt_tcp_payloads = rtt_tcp_payloads;
801     rtt_tcp_payload_result->rtt_tcp_payload_num = rtt_tcp_payload_num;
802
803     return true;
804   }
805
806   static bool do_pretest(State *s, int_fast16_t duration, DirectionResult *res, bool
      ↪  (*do_chunks)(State *, int_fast32_t, struct timespec *, DataPoint *)) {
807     int_fast32_t max_datapoints = DATAPOINT_INCREMENT_PRETEST;
808     DataPoint *time_series = malloc((size_t) max_datapoints * sizeof(DataPoint));
809     int_fast32_t ts_idx = 0;
810
811     int_fast32_t chunks = 1;
812     struct timespec ts_end, ts_zero;
813     res->time_start_rel_ns = get_relative_time_ns(s);
814     ts_fill(&ts_zero);
815     ts_copy(&ts_end, &ts_zero);
816     ts_end.tv_sec += duration;
817     int_fast64_t timediff;
818     do {
819       RETURN_IF_NOK(do_chunks(s, chunks, &ts_zero, &time_series[ts_idx]));
820       chunks *= 2;
821
822       if (ts_idx >= max_datapoints) {
823         max_datapoints += DATAPOINT_INCREMENT_PRETEST;
824         time_series = realloc(time_series, (size_t) max_datapoints *
          ↪  sizeof(DataPoint));
825       }
826       time_series[ts_idx++].bytes = chunks *
          ↪  s->targ->flow_result->connection_info.chunksize;
827
828       timediff = ts_diff(&ts_end);
829     } while (timediff < 0);
830
831     res->time_series = time_series;
832     res->num_time_series = ts_idx;
833
834     res->time_end_rel_ns = get_relative_time_ns(s);
835     return true;
836   }
837
838   static inline bool do_pretest_downlink(State *s) {
839     return do_pretest(s, s->config->dl_pretest_duration_s,
      ↪  &s->targ->flow_result->pretest_dl, do_getchunks);
840   }
```

```
841
842 __attribute__ ((flatten,hot)) static bool do_downlink(State *s) {
843   if (!MASK_IS_SET(s->mask, M_GETTIME))
844     return add_error(s, "expected server to accept GETTIME");
845
846   DirectionResult *res = &s->targ->flow_result->dl;
847
848   res->time_start_rel_ns = get_relative_time_ns(s);
849
850   struct timespec ts_start;
851   ts_fill(&ts_start);
852   RETURN_IF_NOK(write_to_server(s, GETTIME " %" PRIdFAST16 "\n",
      ↪  s->config->dl_duration_s));
853
854   int_fast64_t totalRead = 0;
855   int_fast64_t read;
856   unsigned char lastByte = 0;
857   int_fast64_t timediff_ns = 0;
858   int_fast64_t max_timediff_ns = (s->config->dl_duration_s +
      ↪  s->config->dl_wait_time_s) * I_1E9;
859
860   int_fast32_t max_datapoints = DATAPOINT_INCREMENT_MAIN;
861   DataPoint *time_series = malloc((size_t) max_datapoints * sizeof(DataPoint));
862   int_fast32_t ts_idx = 0;
863
864   const int_fast32_t chunksize = s->targ->flow_result->connection_info.chunksize;
865
866   do {
867     read = my_read(s, s->buf_chunk, chunksize, false);
868     if (read > 0) {
869       int_fast64_t posLast = chunksize - 1 - (totalRead % chunksize);
870       if (read > posLast)
871         lastByte = (unsigned char) s->buf_chunk[posLast];
872       totalRead += read;
873
874       timediff_ns = ts_diff(&ts_start);
875
876       if (ts_idx >= max_datapoints) {
877         max_datapoints += DATAPOINT_INCREMENT_MAIN;
878         time_series = realloc(time_series, (size_t) max_datapoints *
          ↪  sizeof(DataPoint));
879       }
880       time_series[ts_idx].bytes = totalRead;
881       time_series[ts_idx].time_ns = timediff_ns;
882       time_series[ts_idx].duration_server = 0;
883       time_series[ts_idx++].time_ns_end = 0;
884     }
885   } while (read > 0 && lastByte != BYTE_END && timediff_ns < max_timediff_ns);
886
887   if (read <= 0)
888     return add_error(s, "error during do_downlink");
889
890   res->time_series = time_series;
891   res->num_time_series = ts_idx;
892
893   res->time_end_rel_ns = get_relative_time_ns(s);
894
895   /* need to reconnect */
896   if (lastByte != BYTE_END) {
897     my_log_force(s, "need reconnect");
898     s->need_reconnect = true;
899     res->duration_server_ns = 0;
```

```
900       return true;
901     }
902
903     RETURN_IF_NOK(write_to_server(s, OK NL));
904
905     RETURN_IF_NOK(read_time(s, &res->duration_server_ns));
906
907     return read_ok_accept(s);
908   }
909
910   static bool do_putchunks(State *s, int_fast32_t chunks, struct timespec *ts_zero,
      ↪  DataPoint *data_point) {
911     if (!MASK_IS_SET(s->mask, M_PUTNORESULT))
912       return add_error(s, "expected server to accept PUTNORESULT");
913
914     RETURN_IF_NOK(write_to_server(s, PUTNORESULT NL));
915
916     RETURN_IF_NOK(read_ok(s));
917
918     data_point->time_ns = ts_diff(ts_zero); // t_begin
919
920     const int_fast32_t chunksize = s->targ->flow_result->connection_info.chunksize;
921
922     s->buf_chunk[chunksize - 1] = BYTE_CONTINUE;
923
924     set_throughput(s);
925
926     int_fast64_t total_written = 0;
927     for (int_fast32_t i = 0; i < chunks; i++) {
928       if (i == chunks - 1) // for last chunk
929         s->buf_chunk[chunksize - 1] = BYTE_END;
930       ssize_t num = my_write(s, s->buf_chunk, chunksize);
931       if (num < 0)
932         return add_error(s, "error while writing to server in do_putchunks");
933       total_written += num;
934     }
935     s->targ->flow_result->flow_bytes_ul += total_written;
936
937     set_low_delay(s);
938
939     RETURN_IF_NOK(read_time(s, &data_point->duration_server));
940
941     data_point->time_ns_end = ts_diff(ts_zero); // t_end
942
943     return read_ok_accept(s);
944   }
945
946   static inline bool do_pretest_uplink(State *s) {
947     return do_pretest(s, s->config->ul_pretest_duration_s,
      ↪  &s->targ->flow_result->pretest_ul, do_putchunks);
948   }
949
950   __attribute__ ((flatten,hot)) static bool do_uplink(State *s) {
951     if (!MASK_IS_SET(s->mask, M_PUT))
952       return add_error(s, "expected server to accept PUT");
953
954     DirectionResult *res = &s->targ->flow_result->ul;
955
956     res->time_start_rel_ns = get_relative_time_ns(s);
957
958     RETURN_IF_NOK(write_to_server(s, PUT NL));
959
```

```
960     RETURN_IF_NOK(read_ok(s));
961
962     BARRIER; // barrier with other flow do_uplink()s
963
964     const int_fast32_t chunksize = s->targ->flow_result->connection_info.chunksize;
965
966     s->buf_chunk[chunksize - 1] = BYTE_CONTINUE; // set last byte to continue value
967
968     int_fast32_t max_datapoints = DATAPOINT_INCREMENT_MAIN;
969     DataPoint *time_series = malloc((size_t) max_datapoints * sizeof(DataPoint));
970     int_fast32_t ts_idx = 0;
971
972     struct timespec ts_start;
973     ts_fill(&ts_start);
974
975     int_fast64_t total_bytes_written = 0;
976     int_fast64_t timediff_ns, max_timediff_ns = s->config->ul_duration_s * I_1E9;
977     int_fast64_t cutoff_timediff_ns = (s->config->ul_duration_s +
        ↪  s->config->ul_wait_time_s) * I_1E9;
978
979     struct pollfd pfd = { .fd = s->socket_fd };
980
981     set_throughput(s);
982
983     bool last_chunk = false, stop_writing = false;
984     bool ssl_need_read = false, ssl_need_write = false;
985     bool poll_read = false, poll_write = false;
986     do {
987       // we always want to read
988       timediff_ns = ts_diff(&ts_start);
989       if (poll_read && (poll_write || (!ssl_need_write && stop_writing))) {
990         if (s->ssl != NULL && SSL_pending(s->ssl) > 0)
991           poll_read = false;
992         else {
993           pfd.events = POLLIN | (poll_write ? POLLOUT : 0);
994           int poll_ret = poll(&pfd, 1, s->config->timeout_ms);
995           if (poll_ret != 1) // timeout or error
996             return add_error(s, "error in poll of do_uplink: %s", poll_ret == 0 ?
                ↪  "timeout" : strerror(errno));
997           if (pfd.revents & POLLIN)
998             poll_read = false;
999           if (pfd.revents & POLLOUT)
1000            poll_write = false;
1001        }
1002      }
1003
1004      if (!stop_writing && (!poll_write || (ssl_need_read && !poll_read))) { // can
         ↪  write
1005        ssize_t offset = total_bytes_written % chunksize;
1006        ssize_t written;
1007        bool poll_needed = false;
1008        if (s->ssl != NULL) {
1009          written = SSL_write(s->ssl, s->buf_chunk + offset, (int) (chunksize -
             ↪  offset));
1010          int ssl_err = SSL_get_error(s->ssl, (int) written);
1011          if (written <= 0 && IS_SSL_WANT_READ_OR_WRITE(ssl_err)) {
1012            if (ssl_err == SSL_ERROR_WANT_READ) {
1013              poll_read = true;
1014              ssl_need_read = true;
1015            } else if (ssl_err == SSL_ERROR_WANT_WRITE) {
1016              poll_write = true;
1017            }
```

```
1018            poll_needed = true;
1019          }
1020        } else {
1021          written = write(s->socket_fd, s->buf_chunk + offset, (size_t) (chunksize -
              ↪  offset));
1022          if (written == -1 && errno == EAGAIN) {
1023            poll_write = true;
1024            poll_needed = true;
1025          }
1026        }
1027        if (!poll_needed) {
1028          ssl_need_read = false;
1029          if (written <= 0)
1030            return add_error(s, "error while writing to server in do_uplink");
1031          total_bytes_written += written;
1032
1033          if (last_chunk && total_bytes_written % chunksize == 0) {
1034            set_low_delay(s);
1035            stop_writing = true;
1036          }
1037
1038          if (timediff_ns >= max_timediff_ns && !last_chunk) {
1039            s->buf_chunk[chunksize - 1] = BYTE_END; // set last byte to termination
              ↪  value
1040            last_chunk = true;
1041          }
1042        }
1043      }
1044
1045      if (!poll_read || (ssl_need_write && !poll_write)) { // can read
1046        bool got_bytes = false;
1047        if (ts_idx >= max_datapoints) {
1048          max_datapoints += DATAPOINT_INCREMENT_MAIN;
1049          time_series = realloc(time_series, (size_t) max_datapoints *
              ↪  sizeof(DataPoint));
1050        }
1051
1052        long ret = read_time_bytes(s, &time_series[ts_idx].time_ns, &got_bytes,
        ↪  &time_series[ts_idx].bytes, true);
1053        // ret == NEED_POLL_READ on EAGAIN/SSL_ERROR_WANT_READ
1054        // ret == NEED_POLL_WRITE on SSL_ERROR_WANT_WRITE
1055        if (IS_NEED_POLL(ret)) {
1056          if (ret == NEED_POLL_READ) {
1057            poll_read = true;
1058          } else if (ret == NEED_POLL_WRITE) {
1059            poll_write = true;
1060            ssl_need_write = true;
1061          }
1062          continue;
1063        }
1064        if (ret != true)
1065          return false;
1066
1067        ssl_need_write = false;
1068
1069        time_series[ts_idx].time_ns_end = 0;
1070        time_series[ts_idx++].duration_server = 0;
1071
1072        if (!got_bytes) { // got end result; end result has no bytes set
1073          time_series[ts_idx - 1].bytes = total_bytes_written;
1074          s->targ->flow_result->ul.time_series = time_series;
1075          s->targ->flow_result->ul.num_time_series = ts_idx;
```

```
1076            break;
1077          }
1078        }
1079      } while (timediff_ns < cutoff_timediff_ns);
1080
1081      set_low_delay(s);
1082
1083      s->targ->flow_result->flow_bytes_ul += total_bytes_written;
1084
1085      res->time_end_rel_ns = get_relative_time_ns(s);
1086
1087      /* need to reconnect */
1088      if (timediff_ns >= cutoff_timediff_ns) {
1089        my_log_force(s, "cutoff time reached");
1090        s->need_reconnect = true;
1091        return true;
1092      }
1093
1094      return read_ok_accept(s);
1095    }
1096
1097    static bool check_for_reconnect(State *s) {
1098      if (s->need_reconnect) {
1099        RETURN_IF_NOK(disconnect(s));
1100        RETURN_IF_NOK(connect_and_send_token(s));
1101        s->need_reconnect = false;
1102      }
1103      return true;
1104    }
1105
1106    static inline bool quit(State *s) {
1107      if (!MASK_IS_SET(s->mask, M_QUIT))
1108        return add_error(s, "expected server to accept QUIT");
1109      return write_to_server(s, QUIT NL);
1110    }
1111
1112    static inline void set_phase(State *s, Phase phase) {
1113      s->targ->flow_result->last_phase = phase;
1114    }
1115
1116    static bool run_test(State *s) {
1117      set_phase(s, PH_init);
1118
1119      my_log(s, "connecting...");
1120
1121      RETURN_IF_NOK(connect_and_send_token(s));
1122
1123      BARRIER;
1124      my_log(s, "connected with %" PRIuFAST16 " flow(s) for dl; %" PRIuFAST16 "
          ↪  flow(s) for ul", s->config->dl_num_flows, s->config->ul_num_flows);
1125
1126      /* pretest downlink */
1127      set_phase(s, PH_pretest_dl);
1128      my_log(s, "pretest downlink start... (min %" PRIuFAST16 "s)",
          ↪  s->config->dl_pretest_duration_s);
1129      if (s->targ->do_downlink)
1130        RETURN_IF_NOK(do_pretest_downlink(s));
1131      BARRIER;
1132      my_log(s, "pretest downlink end.");
1133
1134      /* rtt_tcp_payload */
1135      set_phase(s, PH_rtt_tcp_payload);
```

```
1136    my_log(s, "rtt_tcp_payload start... (%" PRIuFAST16 " times)",
        ↪  s->config->rtt_tcp_payload_num);
1137    if (s->targ->do_rtt_tcp_payload) /* only one thread does rtt_tcp_payload */
1138      RETURN_IF_NOK(do_rtt_tcp_payload(s));
1139    BARRIER;
1140    my_log(s, "rtt_tcp_payload end.");
1141
1142    /* downlink */
1143    set_phase(s, PH_dl);
1144    my_log(s, "downlink test start... (%" PRIuFAST16 "s)",
        ↪  s->config->dl_duration_s);
1145    if (s->targ->do_downlink) {
1146      RETURN_IF_NOK(do_downlink(s));
1147      RETURN_IF_NOK(check_for_reconnect(s));
1148    }
1149    BARRIER;
1150    my_log(s, "downlink test end.");
1151
1152    /* pretest uplink */
1153    set_phase(s, PH_pretest_ul);
1154    my_log(s, "pretest uplink start... (min %" PRIuFAST16 "s)",
        ↪  s->config->ul_pretest_duration_s);
1155    if (s->targ->do_uplink)
1156      RETURN_IF_NOK(do_pretest_uplink(s));
1157    BARRIER;
1158    my_log(s, "pretest uplink end.");
1159
1160    /* uplink */
1161    set_phase(s, PH_ul);
1162    my_log(s, "uplink test start... (%" PRIuFAST16 "s)", s->config->ul_duration_s);
1163    if (s->targ->do_uplink) {
1164      RETURN_IF_NOK(do_uplink(s));
1165    } else
1166      BARRIER; // there is a BARRIER in do_uplink
1167    BARRIER;
1168    my_log(s, "uplink test end.");
1169
1170    /* end */
1171    set_phase(s, PH_end);
1172    my_log(s, "disconnecting.");
1173    RETURN_IF_NOK(quit(s));
1174    return true;
1175  }
1176
1177  void *run_test_thread_start(void *arg) {
1178    State state = { .targ = arg, .config = ((ThreadArg *) arg)->cfg };
1179
1180    bool ok = run_test(&state);
1181    if (!ok || state.have_err)
1182      get_errors(&state, state.targ->flow_result->error,
        ↪  sizeof(state.targ->flow_result->error));
1183
1184    disconnect(&state);
1185
1186    print_errors(&state, stderr, true);
1187
1188    free(state.buf_chunk);
1189    state.buf_chunk = NULL;
1190
1191  #if !defined(HAVE_ERR_REMOVE_THREAD_STATE_DEPRECATED) && \
1192    defined(HAVE_ERR_REMOVE_THREAD_STATE)
1193    ERR_remove_thread_state(NULL);
```

```
1194  #endif
1195
1196    return NULL;
1197  }
```

Listing B.8: rmbt_helper.h

```
1   /*******************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
14   * limitations under the License.
15   ******************************************************************************/
16
17  #ifndef SRC_RMBT_HELPER_H_
18  #define SRC_RMBT_HELPER_H_
19
20  #include "rmbt_common.h"
21
22  void fail(const char *format, ...) __attribute__ ((noreturn,format (printf, 1,
    ↪  2)));
23  void fail_errno(int err, const char *fmt, ...) __attribute__ ((noreturn,format
    ↪  (printf, 2, 3)));
24  void fail_ssl(void) __attribute__ ((noreturn));
25  void ts_fill(struct timespec *now) __attribute__ ((hot));
26  int_fast64_t ts_diff(struct timespec *start) __attribute__ ((hot));
27  void ts_copy(struct timespec *dest, struct timespec *src);
28  bool variable_subst(char *dst, size_t dst_size, const char *src, const char
    ↪  **replacements, size_t num_replacements);
29
30  #endif /* SRC_RMBT_HELPER_H_ */
```

Listing B.9: rmbt_helper.c

```
1   /*******************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
14   * limitations under the License.
15   ******************************************************************************/
```

```
16
17  #include "rmbt_helper.h"
18
19  #include <stdarg.h>
20  #include "rmbt_ssl.h"
21
22  #define I_1E9 (int_fast64_t)1000000000
23
24  void fail(const char *fmt, ...) {
25    va_list ap;
26    va_start(ap, fmt);
27    vfprintf(stderr, fmt, ap);
28    va_end(ap);
29    fprintf(stderr, "\n");
30    exit(EXIT_FAILURE);
31  }
32
33  void fail_errno(int err, const char *fmt, ...) {
34    va_list ap;
35    va_start(ap, fmt);
36    vfprintf(stderr, fmt, ap);
37    va_end(ap);
38    errno = err;
39    perror("error: ");
40    fprintf(stderr, "\n");
41    exit(EXIT_FAILURE);
42  }
43
44  void fail_ssl() {
45    fprintf(stderr, "ssl error:\n");
46    ERR_print_errors_fp(stderr);
47    exit(EXIT_FAILURE);
48  }
49
50  void ts_fill(struct timespec *now) {
51    int rc = clock_gettime(CLOCK_MONOTONIC, now);
52    if (rc == -1)
53      fail("error during clock_gettime");
54  }
55
56  int_fast64_t ts_diff(struct timespec *start) {
57    struct timespec end;
58    ts_fill(&end);
59
60    if ((end.tv_nsec - start->tv_nsec) < 0) {
61      end.tv_sec = end.tv_sec - start->tv_sec - 1;
62      end.tv_nsec = I_1E9 + end.tv_nsec - start->tv_nsec;
63    } else {
64      end.tv_sec = end.tv_sec - start->tv_sec;
65      end.tv_nsec = end.tv_nsec - start->tv_nsec;
66    }
67    return end.tv_nsec + (int_fast64_t) end.tv_sec * I_1E9 ;
68  }
69
70  void ts_copy(struct timespec *dest, struct timespec *src) {
71    memcpy(dest, src, sizeof(struct timespec));
72  }
73
74  bool variable_subst(char *dst, size_t dst_size, const char *src, const char
    ↪  **replacements, size_t num_replacements) {
75    size_t len = 0;
76    const char *start = src;
```

```
77    for (;;) {
78      const char *delim_start = index(start, '{');
79
80      if (delim_start == NULL) {
81        size_t size = strlen(start);
82        if (size + 1 > dst_size - len)
83          return false;
84        memcpy(dst + len, start, size + 1);
85        return true;
86      }
87
88      size_t size = (size_t) (delim_start - start);
89      if (size > dst_size - len)
90        return false;
91      memcpy(dst + len, start, size);
92      len += size;
93      const char *delim_end = index(delim_start, '}');
94      if (delim_end != NULL) {
95        bool found = false;
96        for (size_t i = 0; i < num_replacements * 2; i += 2) {
97          if (strncmp(delim_start + 1, replacements[i], (size_t) (delim_end -
          ↪  delim_start - 1)) == 0) {
98            size = strlen(replacements[i + 1]);
99            if (size > dst_size - len)
100             return false;
101           memcpy(dst + len, replacements[i + 1], size);
102           len += size;
103           found = true;
104           break;
105         }
106       }
107       if (!found) {
108         size = (size_t) (delim_end - delim_start + 1);
109         if (size > dst_size - len)
110           return false;
111         memcpy(dst + len, delim_start, size);
112         len += size;
113       }
114       start = delim_end + 1;
115     }
116   }
117   return true;
118 }
```

Listing B.10: rmbt_result.h

```
1  /*******************************************************************************
2   * Copyright 2017 Leonhard Wimmer
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *  http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
```

```
15   ********************************************************************************/
16
17  #ifndef SRC_RMBT_RESULT_H_
18  #define SRC_RMBT_RESULT_H_
19
20  #include <arpa/inet.h>
21  #include <stdbool.h>
22  #include <time.h>
23
24  #include "rmbt_json.h"
25
26  #define FOREACH_PHASE(PHASE) \
27          PHASE(init)    \
28          PHASE(pretest_dl)   \
29          PHASE(rtt_tcp_payload)    \
30          PHASE(dl)   \
31      PHASE(pretest_ul)   \
32      PHASE(ul)   \
33      PHASE(end)
34
35  #define GENERATE_ENUM(ENUM)  PH_##ENUM,
36  #define GENERATE_STRING(STRING) #STRING,
37
38  typedef enum {
39    FOREACH_PHASE(GENERATE_ENUM)
40  } Phase;
41
42  typedef struct {
43    const char *cipher;
44    char *tcp_congestion;
45    char *tls_debug;
46    int_fast32_t chunksize;
47    int_fast16_t port_local, port_server;
48    char ip_local[INET6_ADDRSTRLEN];
49    char ip_server[INET6_ADDRSTRLEN];
50    char server_version[64];
51    bool encrypt;
52  } ConnectionInfo;
53
54  typedef struct {
55    int_fast64_t time_start_rel_ns, time_end_rel_ns;
56    int_fast64_t rtt_server_ns, rtt_client_ns;
57  } RttTcpPayload;
58
59  typedef struct {
60    int_fast16_t rtt_tcp_payload_num;
61    RttTcpPayload *rtt_tcp_payloads;
62  } RttTcpPayloadResult;
63
64  typedef struct {
65    char *id_test;
66    char *error;
67    RttTcpPayloadResult *rtt_tcp_payload_result;
68    ConnectionInfo *connection_info;
69    int_fast64_t dl_time_ns, dl_bytes;
70    int_fast64_t ul_time_ns, ul_bytes;
71    int_fast64_t rtt_tcp_payload_client_ns;
72    int_fast64_t rtt_tcp_payload_server_ns;
73    int_fast64_t total_bytes_dl, total_bytes_ul;
74    time_t time_start_s;
75    time_t time_end_s;
76    double dl_throughput_kbps;
```

```
77    double ul_throughput_kbps;
78    int_fast16_t dl_num_flows, ul_num_flows;
79    Phase last_phase;
80  } Result;
81
82  typedef struct {
83    int_fast64_t bytes;
84    int_fast64_t time_ns;
85    int_fast64_t time_ns_end;
86    int_fast64_t duration_server;
87  } DataPoint;
88
89  typedef struct {
90    int_fast64_t time_start_rel_ns, time_end_rel_ns, duration_server_ns;
91    int_fast32_t num_time_series;
92    DataPoint *time_series;
93  } DirectionResult;
94
95  typedef struct {
96    int_fast64_t flow_bytes_dl;
97    int_fast64_t flow_bytes_ul;
98    RttTcpPayloadResult rtt_tcp_payload;
99    DirectionResult pretest_dl, dl, pretest_ul, ul;
100   ConnectionInfo connection_info;
101   Phase last_phase;
102   char error[512];
103 } FlowResult;
104
105 rmbt_json_array rtt_tcp_payloads_to_json_array(RttTcpPayload *rtt_tcp_payloads,
      ↪  int_fast16_t rtt_tcp_payload_num);
106 void add_datapoint_to_array(rmbt_json array, DataPoint data_point);
107 rmbt_json directionresult_to_json_obj(DirectionResult *direction_result);
108
109 void calc_results(Result *result, FlowResult *flow_results, int_fast16_t
      ↪  num_flow_results);
110
111 rmbt_json collect_summary_results(Result *result);
112 rmbt_json collect_raw_results(Result *result, FlowResult *flow_results,
      ↪  int_fast16_t num_flow_results);
113 void do_free_flow_results(FlowResult *flow_results, int_fast16_t
      ↪  num_flow_results);
114
115 #endif /* SRC_RMBT_RESULT_H_ */
```

Listing B.11: rmbt_result.c

```
1  /*******************************************************************************
2   * Copyright 2017 Leonhard Wimmer
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *  http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
```

```
15   ********************************************************************************/
16
17  #include "rmbt_result.h"
18
19  #include <stdio.h>
20  #include <string.h>
21
22  #include "rmbt_config.h"
23  #include "rmbt_stats.h"
24
25  #define SUCCESS    "success"
26  #define FAIL    "fail_"
27
28  #define FREE_AND_SET_NULL(ptr) if (ptr != NULL) { free(ptr); ptr = NULL; }
29
30  static const char *PHASE_STRING[] = { FOREACH_PHASE(GENERATE_STRING) };
31
32  rmbt_json_array rtt_tcp_payloads_to_json_array(RttTcpPayload rtt_tcp_payloads[],
    ↪  int_fast16_t rtt_tcp_payload_num) {
33    rmbt_json_array result = rmbt_json_new_array();
34    for (int_fast16_t i = 0; i < rtt_tcp_payload_num; i++) {
35      rmbt_json point = rmbt_json_new();
36      rmbt_json_add_int64(point, "time_start_rel_ns", (*(rtt_tcp_payloads +
        ↪  i)).time_start_rel_ns);
37      rmbt_json_add_int64(point, "rtt_server_ns", (int64_t) (*(rtt_tcp_payloads +
        ↪  i)).rtt_server_ns);
38      rmbt_json_add_int64(point, "rtt_client_ns", (*(rtt_tcp_payloads +
        ↪  i)).rtt_client_ns);
39      rmbt_json_add_int64(point, "time_end_rel_ns", (*(rtt_tcp_payloads +
        ↪  i)).time_end_rel_ns);
40      rmbt_json_add_to_array(result, point);
41    }
42    return result;
43  }
44
45  static rmbt_json_array time_series_to_json_array(DataPoint data_points[],
    ↪  int_fast32_t num_data_points) {
46    rmbt_json_array result = rmbt_json_new_array();
47    for (int_fast32_t i = 0; i < num_data_points; i++) {
48      DataPoint data_point = *(data_points + i);
49      rmbt_json point = rmbt_json_new();
50      if (data_point.time_ns_end == 0) {
51        rmbt_json_add_int64(point, "t", data_point.time_ns);
52        rmbt_json_add_int64(point, "b", data_point.bytes);
53      } else {
54        rmbt_json_add_int64(point, "t_begin", data_point.time_ns);
55        rmbt_json_add_int64(point, "b", data_point.bytes);
56        rmbt_json_add_int64(point, "t_end", data_point.time_ns_end);
57      }
58      if (data_point.duration_server != 0)
59        rmbt_json_add_int64(point, "d_server", data_point.duration_server);
60      rmbt_json_add_to_array(result, point);
61    }
62    return result;
63  }
64
65  rmbt_json directionresult_to_json_obj(DirectionResult *direction_result) {
66    rmbt_json result_json = rmbt_json_new();
67    rmbt_json_add_int64(result_json, "time_start_rel_ns",
      ↪  direction_result->time_start_rel_ns);
68    rmbt_json_add_int64(result_json, "time_end_rel_ns",
      ↪  direction_result->time_end_rel_ns);
```

```
69    if (direction_result->duration_server_ns != 0)
70      rmbt_json_add_int64(result_json, "duration_server_ns",
        ↪  direction_result->duration_server_ns);
71    rmbt_json_add_array(result_json, "time_series",
      ↪  time_series_to_json_array(direction_result->time_series,
      ↪  direction_result->num_time_series));
72    return result_json;
73  }
74
75  static void calc_direction_results(double *result, int_fast64_t *result_time,
    ↪  int_fast64_t *result_bytes, int_fast16_t *num_flows,
76      DirectionResult *direction_results[], int_fast16_t num_direction_results) {
77
78    int_fast64_t target_time = INT_FAST64_MAX;
79    int_fast16_t target_time_idx = -1;
80
81    *num_flows = 0;
82
83    for (int_fast16_t i = 0; i < num_direction_results; i++) {
84      DirectionResult *direction_result = direction_results[i];
85      if (direction_result->time_series != NULL) {
86        (*num_flows)++;
87        int_fast64_t last_time =
          ↪  direction_result->time_series[direction_result->num_time_series -
          ↪  1].time_ns;
88        if (target_time_idx == -1 || last_time < target_time) {
89          target_time_idx = i;
90          target_time = last_time;
91        }
92      }
93    }
94
95    int_fast64_t total_bytes = 0;
96    /* could be optimized by using binary search */
97    for (int_fast16_t i = 0; i < num_direction_results; i++) {
98      DirectionResult *direction_result = direction_results[i];
99      int_fast32_t target_idx_a = -1, target_idx_b = -1;
100     for (int_fast32_t j = direction_result->num_time_series - 1; j >= 0; j--) {
101       if (direction_result->time_series[j].time_ns <= target_time) {
102         target_idx_a = target_idx_b = j;
103         if (direction_result->time_series[j].time_ns != target_time && j !=
            ↪  direction_result->num_time_series - 1)
104           target_idx_b++;
105         break;
106       }
107     }
108     if (target_idx_a == target_idx_b && target_idx_a != -1)
109       total_bytes += direction_result->time_series[target_idx_a].bytes;
110     else {
111       int_fast64_t bytes_a = target_idx_a == -1 ? 0 :
          ↪  direction_result->time_series[target_idx_a].bytes;
112       int_fast64_t bytes_b = target_idx_b == -1 ? 0 :
          ↪  direction_result->time_series[target_idx_b].bytes;
113       int_fast64_t time_a = target_idx_a == -1 ? 0 :
          ↪  direction_result->time_series[target_idx_a].time_ns;
114       int_fast64_t time_b = target_idx_b == -1 ? 0 :
          ↪  direction_result->time_series[target_idx_b].time_ns;
115
116       total_bytes += bytes_a;
117       if (time_b - time_a != 0)
118         total_bytes += (bytes_b - bytes_a) * (target_time - time_a) / (time_b -
          ↪  time_a);
```

```
119       }
120     }
121     *result = (double) total_bytes / (double) target_time * 8e6;
122     *result_time = target_time;
123     *result_bytes = total_bytes;
124   }
125
126   static int cmp_int_fast64_p(const void *p1, const void *p2) {
127     int_fast64_t diff = *(const int_fast64_t *) p2 - *(const int_fast64_t *) p1;
128     if (diff > 0)
129       return 1;
130     if (diff < 0)
131       return -1;
132     return 0;
133   }
134
135   void calc_results(Result *result, FlowResult *flow_results, int_fast16_t
      ↪  num_flow_results) {
136
137     /* rtt_tcp_payload */
138     if (result->rtt_tcp_payload_result == NULL) {
139       for (int_fast16_t i = 0; i < num_flow_results; i++) {
140         FlowResult *flow_result = flow_results + i;
141         if (flow_result->rtt_tcp_payload.rtt_tcp_payloads != NULL) {
142           result->rtt_tcp_payload_result = &flow_result->rtt_tcp_payload;
143           break;
144         }
145       }
146     }
147     if (result->rtt_tcp_payload_result != NULL) {
148       int_fast16_t rtt_tcp_payload_num =
          ↪  result->rtt_tcp_payload_result->rtt_tcp_payload_num;
149       int_fast64_t rtt_tcp_payloads_client[rtt_tcp_payload_num];
150       int_fast64_t rtt_tcp_payloads_server[rtt_tcp_payload_num];
151       for (int_fast16_t i = 0; i < rtt_tcp_payload_num; i++) {
152         rtt_tcp_payloads_client[i] =
            ↪  result->rtt_tcp_payload_result->rtt_tcp_payloads->rtt_client_ns;
153         rtt_tcp_payloads_server[i] =
            ↪  result->rtt_tcp_payload_result->rtt_tcp_payloads->rtt_server_ns;
154       }
155
156       qsort(rtt_tcp_payloads_client, (size_t) rtt_tcp_payload_num,
          ↪  sizeof(int_fast64_t), cmp_int_fast64_p);
157       qsort(rtt_tcp_payloads_server, (size_t) rtt_tcp_payload_num,
          ↪  sizeof(int_fast64_t), cmp_int_fast64_p);
158
159       /*
160        rtt_tcp_payload_result->rtt_client_shortest_ns = rtt_tcp_payloads_client[0];
161        rtt_tcp_payload_result->rtt_server_shortest_ns = rtt_tcp_payload_server[0];
162        */
163
164       if (rtt_tcp_payload_num == 1) {
165         result->rtt_tcp_payload_client_ns = rtt_tcp_payloads_client[0];
166         result->rtt_tcp_payload_server_ns = rtt_tcp_payloads_server[0];
167       } else if (rtt_tcp_payload_num > 1) {
168         int_fast16_t idx_median = rtt_tcp_payload_num / 2;
169         if (rtt_tcp_payload_num % 2 == 0) {
170           result->rtt_tcp_payload_client_ns = (rtt_tcp_payloads_client[idx_median] +
              ↪  rtt_tcp_payloads_client[idx_median + 1]) / 2;
171           result->rtt_tcp_payload_server_ns = (rtt_tcp_payloads_server[idx_median] +
              ↪  rtt_tcp_payloads_server[idx_median + 1]) / 2;
172         } else {
```

```
173           result->rtt_tcp_payload_client_ns = rtt_tcp_payloads_client[idx_median];
174           result->rtt_tcp_payload_server_ns = rtt_tcp_payloads_server[idx_median];
175         }
176       }
177     }
178
179     /* dl / ul */
180     DirectionResult *direction_result_dl[num_flow_results];
181     DirectionResult *direction_result_ul[num_flow_results];
182     result->total_bytes_dl = result->total_bytes_ul = 0;
183     for (int_fast16_t i = 0; i < num_flow_results; i++) {
184       FlowResult *flow_result = flow_results + i;
185
186       if (result->connection_info == NULL)
187         result->connection_info = &flow_result->connection_info;
188
189       direction_result_dl[i] = &flow_result->dl;
190       direction_result_ul[i] = &flow_result->ul;
191
192       result->total_bytes_dl += flow_result->flow_bytes_dl;
193       result->total_bytes_ul += flow_result->flow_bytes_ul;
194     }
195     calc_direction_results(&result->dl_throughput_kbps, &result->dl_time_ns,
       ↪  &result->dl_bytes, &result->dl_num_flows, direction_result_dl,
       ↪  num_flow_results);
196     calc_direction_results(&result->ul_throughput_kbps, &result->ul_time_ns,
       ↪  &result->ul_bytes, &result->ul_num_flows, direction_result_ul,
       ↪  num_flow_results);
197
198     /* status */
199     for (int_fast16_t i = 0; i < num_flow_results; i++) {
200       FlowResult *flow_result = flow_results + i;
201       if (flow_result->error[0] != '\0') {
202         result->error = flow_result->error;
203         result->last_phase = flow_result->last_phase;
204         break;
205       }
206     }
207   }
208
209   static void add_common_results(Result *result, rmbt_json result_json) {
210
211     if (result->id_test != NULL)
212       rmbt_json_add_string(result_json, "res_id_test", result->id_test);
213     rmbt_json_add_int64(result_json, "res_time_start_s", result->time_start_s);
214     rmbt_json_add_int64(result_json, "res_time_end_s", result->time_end_s);
215     if (result->error == NULL) {
216       rmbt_json_add_string(result_json, "res_status", SUCCESS);
217       rmbt_json_add_null(result_json, "res_status_msg");
218     } else {
219       char status[sizeof(result->error) + strlen(FAIL)];
220       snprintf(status, sizeof(status), "%s%s", FAIL,
       ↪  PHASE_STRING[result->last_phase]);
221       rmbt_json_add_string(result_json, "res_status", status);
222       rmbt_json_add_string(result_json, "res_status_msg", result->error);
223     }
224     rmbt_json_add_string(result_json, "res_version_client", RMBT_VERSION);
225
226     if (result->connection_info != NULL) {
227       if (strlen(result->connection_info->server_version) > 0)
228         rmbt_json_add_string(result_json, "res_version_server",
         ↪  result->connection_info->server_version);
```

```
229       if (strlen(result->connection_info->ip_server) > 0)
230         rmbt_json_add_string(result_json, "res_server_ip",
            ↪  result->connection_info->ip_server);
231       if (result->connection_info->port_server != 0)
232         rmbt_json_add_int64(result_json, "res_server_port",
            ↪  result->connection_info->port_server);
233       rmbt_json_add_bool(result_json, "res_encrypt",
          ↪  result->connection_info->encrypt);
234       if (result->connection_info->cipher != NULL)
235         rmbt_json_add_string(result_json, "res_cipher",
            ↪  result->connection_info->cipher);
236       if (result->connection_info->chunksize > 0)
237         rmbt_json_add_int64(result_json, "res_chunksize",
            ↪  result->connection_info->chunksize);
238       if (result->connection_info->tcp_congestion != NULL)
239         rmbt_json_add_string(result_json, "res_tcp_congestion",
            ↪  result->connection_info->tcp_congestion);
240     }
241
242     if (result->total_bytes_dl > 0)
243       rmbt_json_add_int64(result_json, "res_total_bytes_dl",
          ↪  result->total_bytes_dl);
244     if (result->total_bytes_ul > 0)
245       rmbt_json_add_int64(result_json, "res_total_bytes_ul",
          ↪  result->total_bytes_ul);
246
247     get_uname(result_json);
248   }
249
250   rmbt_json collect_summary_results(Result *result) {
251     rmbt_json result_json = rmbt_json_new();
252
253     add_common_results(result, result_json);
254
255     if (result->rtt_tcp_payload_result != NULL &&
        ↪  result->rtt_tcp_payload_result->rtt_tcp_payload_num > 0)
256       rmbt_json_add_int64(result_json, "res_rtt_tcp_payload_num",
          ↪  result->rtt_tcp_payload_result->rtt_tcp_payload_num);
257     if (result->rtt_tcp_payload_client_ns > 0)
258       rmbt_json_add_int64(result_json, "res_rtt_tcp_payload_client_ns",
          ↪  result->rtt_tcp_payload_client_ns);
259     if (result->rtt_tcp_payload_server_ns > 0)
260       rmbt_json_add_int64(result_json, "res_rtt_tcp_payload_server_ns",
          ↪  result->rtt_tcp_payload_server_ns);
261
262     if (result->dl_bytes > 0) {
263       rmbt_json_add_int64(result_json, "res_dl_num_flows", result->dl_num_flows);
264       rmbt_json_add_int64(result_json, "res_dl_time_ns", result->dl_time_ns);
265       rmbt_json_add_int64(result_json, "res_dl_bytes", result->dl_bytes);
266       rmbt_json_add_double(result_json, "res_dl_throughput_kbps",
          ↪  result->dl_throughput_kbps);
267     }
268     if (result->ul_bytes > 0) {
269       rmbt_json_add_int64(result_json, "res_ul_num_flows", result->ul_num_flows);
270       rmbt_json_add_int64(result_json, "res_ul_time_ns", result->ul_time_ns);
271       rmbt_json_add_int64(result_json, "res_ul_bytes", result->ul_bytes);
272       rmbt_json_add_double(result_json, "res_ul_throughput_kbps",
          ↪  result->ul_throughput_kbps);
273     }
274
275     return result_json;
276   }
```

```
277
278 static void add_json_array_if_nonempty(rmbt_json object, const char *key,
    ↪ rmbt_json_array json_array) {
279   if (rmbt_json_array_length(json_array) > 0)
280     rmbt_json_add_array(object, key, json_array);
281   else
282     rmbt_json_free_array(json_array);
283 }
284
285 rmbt_json collect_raw_results(Result *result, FlowResult *flow_results,
    ↪ int_fast16_t num_flow_results) {
286   rmbt_json result_json = rmbt_json_new();
287
288   add_common_results(result, result_json);
289
290   rmbt_json json_details = rmbt_json_new();
291
292   if (result->rtt_tcp_payload_result != NULL) {
293     rmbt_json json_rtt_tcp_payload = rmbt_json_new();
294     rmbt_json_add_array(json_rtt_tcp_payload, "values",
295         rtt_tcp_payloads_to_json_array(result->rtt_tcp_payload_result->rtt_tcp_payloads,
          ↪ result->rtt_tcp_payload_result->rtt_tcp_payload_num));
296     rmbt_json_add_object(json_details, "rtt_tcp_payload", json_rtt_tcp_payload);
297   }
298
299   rmbt_json_array json_init = rmbt_json_new_array();
300   rmbt_json_array json_pretest_dl = rmbt_json_new_array();
301   rmbt_json_array json_dl = rmbt_json_new_array();
302   rmbt_json_array json_pretest_ul = rmbt_json_new_array();
303   rmbt_json_array json_ul = rmbt_json_new_array();
304
305   for (int_fast16_t i = 0; i < num_flow_results; i++) {
306     FlowResult *flow_result = flow_results + i;
307
308     rmbt_json init_obj = rmbt_json_new();
309
310     if (flow_result->connection_info.port_local != 0)
311       rmbt_json_add_int64(init_obj, "client_port",
          ↪ flow_result->connection_info.port_local);
312     if (flow_result->connection_info.cipher != NULL)
313       rmbt_json_add_string(init_obj, "cipher",
          ↪ flow_result->connection_info.cipher);
314     if (flow_result->connection_info.tls_debug != NULL)
315       rmbt_json_add_string(init_obj, "tls_debug",
          ↪ flow_result->connection_info.tls_debug);
316     rmbt_json_add_to_array(json_init, init_obj);
317
318     if (flow_result->pretest_dl.time_series != NULL)
319       rmbt_json_add_to_array(json_pretest_dl,
          ↪ directionresult_to_json_obj(&flow_result->pretest_dl));
320     if (flow_result->dl.time_series != NULL)
321       rmbt_json_add_to_array(json_dl,
          ↪ directionresult_to_json_obj(&flow_result->dl));
322     if (flow_result->pretest_ul.time_series != NULL)
323       rmbt_json_add_to_array(json_pretest_ul,
          ↪ directionresult_to_json_obj(&flow_result->pretest_ul));
324     if (flow_result->ul.time_series != NULL)
325       rmbt_json_add_to_array(json_ul,
          ↪ directionresult_to_json_obj(&flow_result->ul));
326   }
327
328   add_json_array_if_nonempty(json_details, "init", json_init);
```

```
329     add_json_array_if_nonempty(json_details, "pretest_dl", json_pretest_dl);
330     add_json_array_if_nonempty(json_details, "dl", json_dl);
331     add_json_array_if_nonempty(json_details, "pretest_ul", json_pretest_ul);
332     add_json_array_if_nonempty(json_details, "ul", json_ul);
333
334     rmbt_json_add_object(result_json, "res_details", json_details);
335
336     return result_json;
337   }
338
339   void do_free_flow_results(FlowResult *flow_results, int_fast16_t num_flow_results)
      ↪ {
340     for (int_fast16_t i = 0; i < num_flow_results; i++) {
341       FlowResult *flow_result = flow_results + i;
342
343       FREE_AND_SET_NULL(flow_result->connection_info.tcp_congestion);
344       FREE_AND_SET_NULL(flow_result->connection_info.tls_debug);
345       FREE_AND_SET_NULL(flow_result->pretest_dl.time_series);
346       FREE_AND_SET_NULL(flow_result->dl.time_series);
347       FREE_AND_SET_NULL(flow_result->pretest_ul.time_series);
348       FREE_AND_SET_NULL(flow_result->ul.time_series);
349       FREE_AND_SET_NULL(flow_result->rtt_tcp_payload.rtt_tcp_payloads);
350     }
351   }
```

Listing B.12: rmbt_json.h

```
1   /*****************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
14   * limitations under the License.
15   *****************************************************************************/
16
17  #ifndef SRC_RMBT_JSON_H_
18  #define SRC_RMBT_JSON_H_
19
20  #include "rmbt_common.h"
21
22  #ifdef HAVE_JSONC
23
24  #include <json.h>
25  typedef struct json_object *rmbt_json;
26  typedef struct json_object *rmbt_json_array;
27
28  #else
29
30  #error need a JSON library
31
32  #endif /* HAVE_JSONC */
33
```

```
34  rmbt_json rmbt_json_new(void);
35  rmbt_json_array rmbt_json_new_array(void);
36
37  void rmbt_json_add_to_array(rmbt_json_array array, rmbt_json object);
38
39  void rmbt_json_add_object(rmbt_json obj, const char *key, rmbt_json val);
40  void rmbt_json_add_int64(rmbt_json obj, const char *key, int64_t val);
41  void rmbt_json_add_null(rmbt_json obj, const char *key);
42  void rmbt_json_add_string(rmbt_json obj, const char *key, const char *val);
43  void rmbt_json_add_double(rmbt_json obj, const char *key, double val);
44  void rmbt_json_add_bool(rmbt_json obj, const char *key, bool val);
45  void rmbt_json_add_array(rmbt_json obj, const char *key, rmbt_json_array val);
46
47  void rmbt_json_get_string_alloc(char **dst, rmbt_json json, const char *key);
48
49  void rmbt_json_get_bool(bool *dst, rmbt_json json, const char *key);
50
51  bool rmbt_json_get_object(rmbt_json *dst, rmbt_json json, const char *key);
52
53  void rmbt_json_get_int_fast16_t(int_fast16_t *dst, rmbt_json json, const char
    ↪  *key);
54  void rmbt_json_get_int_fast32_t(int_fast32_t *dst, rmbt_json json, const char
    ↪  *key);
55
56  uint32_t rmbt_json_array_length(rmbt_json_array array);
57
58  void flatten_json(rmbt_json dst, rmbt_json src);
59
60  rmbt_json rmbt_parse_json(const char *str);
61
62  const char *rmbt_json_array_to_string(rmbt_json_array array, bool beautify);
63  const char *rmbt_json_to_string(rmbt_json json, bool beautify);
64
65  rmbt_json rmbt_json_read_from_file(const char *filename);
66
67  void rmbt_json_free(rmbt_json json);
68  void rmbt_json_free_array(rmbt_json_array array);
69
70  #endif /* SRC_RMBT_JSON_H_ */
```

Listing B.13: rmbt_json.c

```
1   /*******************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
14   * limitations under the License.
15   ******************************************************************************/
16
17  #include "rmbt_json.h"
18
```

```
19  #include <errno.h>
20
21  #include "rmbt_compress.h"
22
23  #ifdef HAVE_JSONC
24
25  #define IS_JSON_NULL(x)  (json_object_get_type(x) == json_type_null)
26
27  rmbt_json rmbt_json_new(void) {
28    return json_object_new_object();
29  }
30
31  rmbt_json rmbt_json_new_array(void) {
32    return json_object_new_array();
33  }
34
35  void rmbt_json_add_to_array(rmbt_json_array array, rmbt_json object) {
36    json_object_array_add(array, object);
37  }
38
39  void rmbt_json_add_object(rmbt_json obj, const char *key, rmbt_json val) {
40    json_object_object_add(obj, key, val);
41  }
42
43  void rmbt_json_add_int64(rmbt_json obj, const char *key, int64_t val) {
44    json_object_object_add(obj, key, json_object_new_int64(val));
45  }
46
47  void rmbt_json_add_null(rmbt_json obj, const char *key) {
48    json_object_object_add(obj, key, NULL);
49  }
50
51  void rmbt_json_add_string(rmbt_json obj, const char *key, const char *val) {
52    json_object_object_add(obj, key, json_object_new_string(val));
53  }
54
55  void rmbt_json_add_double(rmbt_json obj, const char *key, double val) {
56    json_object_object_add(obj, key, json_object_new_double(val));
57  }
58
59  void rmbt_json_add_bool(rmbt_json obj, const char *key, bool val) {
60    json_object_object_add(obj, key, json_object_new_boolean(val));
61  }
62
63  void rmbt_json_add_array(rmbt_json obj, const char *key, rmbt_json_array val) {
64    json_object_object_add(obj, key, val);
65  }
66
67  void rmbt_json_get_string_alloc(char **dst, rmbt_json json, const char *key) {
68    rmbt_json value;
69    if (json_object_object_get_ex(json, key, &value) && !IS_JSON_NULL(value)) {
70      *dst = strdup(json_object_get_string(value));
71    }
72  }
73
74  void rmbt_json_get_bool(bool *dst, rmbt_json json, const char *key) {
75    rmbt_json value;
76    if (json_object_object_get_ex(json, key, &value) && !IS_JSON_NULL(value))
77      *dst = json_object_get_boolean(value);
78  }
79
80  bool rmbt_json_get_object(rmbt_json *dst, rmbt_json json, const char *key) {
```

```
81    rmbt_json value;
82    if (json_object_object_get_ex(json, key, &value)) {
83      *dst = value;
84      return true;
85    }
86    return false;
87  }
88
89  void rmbt_json_get_int_fast16_t(int_fast16_t *dst, rmbt_json json, const char
      ↪ *key) {
90    rmbt_json value;
91    if (json_object_object_get_ex(json, key, &value) && !IS_JSON_NULL(value))
92      *dst = json_object_get_int(value);
93  }
94
95  void rmbt_json_get_int_fast32_t(int_fast32_t *dst, rmbt_json json, const char
      ↪ *key) {
96    rmbt_json value;
97    if (json_object_object_get_ex(json, key, &value) && !IS_JSON_NULL(value))
98      *dst = json_object_get_int(value);
99  }
100
101 uint32_t rmbt_json_array_length(rmbt_json_array array) {
102   return (uint32_t)json_object_array_length(array);
103 }
104
105 rmbt_json rmbt_parse_json(const char *str) {
106   return json_tokener_parse(str);
107 }
108
109 const char *rmbt_json_to_string(rmbt_json json, bool beautify) {
110   return json_object_to_json_string_ext(json, beautify ? JSON_C_TO_STRING_PRETTY :
      ↪ JSON_C_TO_STRING_PLAIN);
111 }
112
113 const char *rmbt_json_array_to_string(rmbt_json_array array, bool beautify) {
114   return rmbt_json_to_string(array, beautify);
115 }
116
117 rmbt_json rmbt_json_read_from_file(const char *filename) {
118   return json_object_from_file(filename);
119 }
120
121 void rmbt_json_free(rmbt_json json) {
122   json_object_put(json);
123 }
124
125 void rmbt_json_free_array(rmbt_json_array array) {
126   json_object_put(array);
127 }
128
129 #pragma GCC diagnostic push  // require GCC 4.6
130 #pragma GCC diagnostic ignored "-Wcast-qual" // json_object_object_foreachC
    ↪ otherwise leads to warnings
131 void flatten_json(rmbt_json dst, rmbt_json src) {
132   if (src == NULL || dst == NULL)
133     return;
134   struct json_object_iter iter;
135   json_object_object_foreachC(src, iter)
136   {
137     json_object_get(iter.val);
138     json_object_object_add(dst, iter.key, iter.val);
```

```
139     }
140   }
141   #pragma GCC diagnostic pop   // require GCC 4.6
142
143   #endif /* HAVE_JSONC */
```

Listing B.14: rmbt_ssl.h

```
1   /*****************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
14   * limitations under the License.
15   *****************************************************************************/
16
17  #ifndef SRC_RMBT_SSL_H_
18  #define SRC_RMBT_SSL_H_
19
20  #include "rmbt_common.h"
21
22  #define OPENSSL_THREAD_DEFINES
23  #include <openssl/opensslconf.h>
24  #if !defined(OPENSSL_THREADS)
25  #error no thread support in openssl
26  #endif
27
28  #include <openssl/bio.h> /* BIO objects for I/O */
29  #include <openssl/crypto.h>
30  #include <openssl/ssl.h> /* SSL and SSL_CTX for SSL connections */
31  #include <openssl/err.h> /* Error reporting */
32  #include <openssl/hmac.h>
33
34  #include "rmbt_config.h"
35
36  extern SSL_CTX *ssl_ctx;
37  extern pthread_mutex_t ssl_ctx_mutex;
38
39  #if OPENSSL_VERSION_NUMBER >= 0x10002003L && \
40    OPENSSL_VERSION_NUMBER <= 0x10002FFFL && \
41    !defined(OPENSSL_NO_COMP)
42  #define HAVE_SSL_COMP_FREE_COMPRESSION_METHODS 1
43  #endif
44
45  #if (OPENSSL_VERSION_NUMBER >= 0x10000000L)
46  #define HAVE_ERR_REMOVE_THREAD_STATE 1
47  #endif
48
49  #if (OPENSSL_VERSION_NUMBER >= 0x10100000L) && /* OpenSSL 1.1.0+ */ \
50    !defined(LIBRESSL_VERSION_NUMBER)
51  #define HAVE_ERR_REMOVE_THREAD_STATE_DEPRECATED 1
52  #define HAVE_SSL_SESSION_GET_MASTER_KEY 1
```

```
53  #endif
54
55  void init_ssl(bool ssl);
56
57  char *get_ssl_debug(SSL *ssl);
58
59  void shutdown_ssl(void);
60
61  #endif /* SRC_RMBT_SSL_H_ */
```

Listing B.15: rmbt_ssl.c

```
1   /*******************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
14   * limitations under the License.
15   ******************************************************************************/
16
17  #include "rmbt_ssl.h"
18
19  #include "rmbt_helper.h"
20
21  static pthread_mutex_t *lockarray = NULL;
22  SSL_CTX *ssl_ctx = NULL;
23  pthread_mutex_t ssl_ctx_mutex = PTHREAD_MUTEX_INITIALIZER;
24
25  #pragma GCC diagnostic push  // require GCC 4.6
26  #pragma GCC diagnostic ignored "-Wunused-function"
27  static void lock_callback(int mode, int type, const char *file, int line) {
28    (void) file;
29    (void) line;
30    if (mode & CRYPTO_LOCK) {
31      pthread_mutex_lock(&(lockarray[type]));
32    } else {
33      pthread_mutex_unlock(&(lockarray[type]));
34    }
35  }
36  #pragma GCC diagnostic pop  // require GCC 4.6
37
38  void init_ssl(bool ssl) {
39    int i;
40
41    lockarray = (pthread_mutex_t *) calloc((size_t) CRYPTO_num_locks(),
    ↪  sizeof(pthread_mutex_t));
42    for (i = 0; i < CRYPTO_num_locks(); i++)
43      pthread_mutex_init(&(lockarray[i]), NULL);
44
45    SSL_library_init(); /* load encryption & hash algorithms for SSL */
46    SSL_load_error_strings(); /* we also want some error msg without using ssl */
47
```

```
48    CRYPTO_set_locking_callback(lock_callback);
49
50    if (ssl) {
51      ssl_ctx = SSL_CTX_new(SSLv23_method());
52      if (ssl_ctx == NULL)
53        fail_ssl();
54      SSL_CTX_set_options(ssl_ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_SSLv3);
55      SSL_CTX_set_mode(ssl_ctx, SSL_MODE_AUTO_RETRY);
56    }
57  }
58
59  static ssize_t to_hex(unsigned char *dst, size_t dst_len, unsigned char *src,
    ↪  size_t src_len) {
60    ssize_t wr = 0;
61    for (size_t i = 0; i < src_len; i++) {
62      int w = snprintf((char *) dst + wr, dst_len - (size_t) wr, "%02X", src[i]);
63      if (w <= 0)
64        return -1;
65      wr += (ssize_t) w;
66    }
67    return wr;
68  }
69
70  char *get_ssl_debug(SSL *ssl) {
71    //    SSL_SESSION_print_fp(stderr, SSL_get_session(ssl));
72
73    ssize_t r;
74    unsigned char buf_client_random[256];
75    unsigned char buf_master_key[256];
76    SSL_SESSION *ssl_session = SSL_get_session(ssl);
77
78  #ifdef HAVE_SSL_SESSION_GET_MASTER_KEY
79    unsigned char buf_raw[256];
80    size_t ssl_r = SSL_get_client_random(ssl, buf_raw, sizeof(buf_raw));
81    if (ssl_r <= 0)
82      return NULL;
83    r = to_hex(buf_client_random, sizeof(buf_client_random), buf_raw, ssl_r);
84    if (r <= 0)
85      return NULL;
86
87    ssl_r = SSL_SESSION_get_master_key(ssl_session, buf_raw, sizeof(buf_raw));
88    if (ssl_r <= 0)
89      return NULL;
90    r = to_hex(buf_master_key, sizeof(buf_master_key), buf_raw, ssl_r);
91    if (r <= 0)
92      return NULL;
93  #else
94    r = to_hex(buf_client_random, sizeof(buf_client_random), ssl->s3->client_random,
    ↪  sizeof(ssl->s3->client_random));
95    if (r <= 0)
96      return NULL;
97    r = to_hex(buf_master_key, sizeof(buf_master_key), ssl_session->master_key,
    ↪  sizeof(ssl_session->master_key));
98    if (r <= 0)
99      return NULL;
100 #endif
101   char *result;
102   r = asprintf(&result, "CLIENT_RANDOM %s %s", buf_client_random, buf_master_key);
103   if (r > 0)
104     return result;
105   return NULL;
106 }
```

```
107
108 /* mainly to make valgrind usable */
109 void shutdown_ssl(void) {
110   if (ssl_ctx != NULL)
111     SSL_CTX_free(ssl_ctx);
112   ssl_ctx = NULL;
113 #if !defined(HAVE_ERR_REMOVE_THREAD_STATE_DEPRECATED) && \
114   defined(HAVE_ERR_REMOVE_THREAD_STATE)
115   ERR_remove_thread_state(NULL);
116 #endif
117   CRYPTO_cleanup_all_ex_data();
118   ERR_free_strings();
119 #if !defined(HAVE_ERR_REMOVE_THREAD_STATE) &&
    ↪  !defined(HAVE_ERR_REMOVE_THREAD_STATE)
120   ERR_remove_state(0);
121 #endif
122   EVP_cleanup();
123 #ifdef HAVE_SSL_COMP_FREE_COMPRESSION_METHODS
124   SSL_COMP_free_compression_methods();
125 #endif
126   CRYPTO_set_locking_callback(NULL);
127   for (int i = 0; i < CRYPTO_num_locks(); i++)
128     pthread_mutex_destroy(&(lockarray[i]));
129   free(lockarray);
130   lockarray = NULL;
131 }
```

Listing B.16: rmbt_stats.h

```
1  /*******************************************************************************
2   * Copyright 2017 Leonhard Wimmer
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *  http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
15  ******************************************************************************/
16
17 #ifndef SRC_RMBT_STATS_H_
18 #define SRC_RMBT_STATS_H_
19
20 #include "rmbt_common.h"
21
22 #include <sys/socket.h>
23
24 #include "rmbt_json.h"
25
26 /*
27  * We use our own version of tcp_info, as we might run on a kernel that is more
    ↪  recent
28  * than the headers we are compiling against.
29  * The "introduced in" comments refer to git commit ids of the linux kernel
    ↪  sources.
```

```
30    * */
31   struct rmbt_tcp_info {
32     uint8_t  tcpi_state;
33     uint8_t  tcpi_ca_state;
34     uint8_t  tcpi_retransmits;
35     uint8_t  tcpi_probes;
36     uint8_t  tcpi_backoff;
37     uint8_t  tcpi_options;
38     uint8_t  tcpi_snd_wscale : 4, tcpi_rcv_wscale : 4;
39     /* next one introduced in eb8329e0a04db0061f714f033b4454326ba147f4 (4.9) */
40     uint8_t  tcpi_delivery_rate_app_limited:1;
41
42     uint32_t  tcpi_rto;
43     uint32_t  tcpi_ato;
44     uint32_t  tcpi_snd_mss;
45     uint32_t  tcpi_rcv_mss;
46
47     uint32_t  tcpi_unacked;
48     uint32_t  tcpi_sacked;
49     uint32_t  tcpi_lost;
50     uint32_t  tcpi_retrans;
51     uint32_t  tcpi_fackets;
52
53     /* Times. */
54     uint32_t  tcpi_last_data_sent;
55     uint32_t  tcpi_last_ack_sent;
56     uint32_t  tcpi_last_data_recv;
57     uint32_t  tcpi_last_ack_recv;
58
59     /* Metrics. */
60     uint32_t  tcpi_pmtu;
61     uint32_t  tcpi_rcv_ssthresh;
62     uint32_t  tcpi_rtt;
63     uint32_t  tcpi_rttvar;
64     uint32_t  tcpi_snd_ssthresh;
65     uint32_t  tcpi_snd_cwnd;
66     uint32_t  tcpi_advmss;
67     uint32_t  tcpi_reordering;
68
69     uint32_t  tcpi_rcv_rtt;
70     uint32_t  tcpi_rcv_space;
71
72     uint32_t  tcpi_total_retrans;
73
74     uint64_t  tcpi_pacing_rate;
75     uint64_t  tcpi_max_pacing_rate;
76     /* introduced in 0df48c26d8418c5c9fba63fac15b660d70ca2f1c (4.1) */
77     uint64_t  tcpi_bytes_acked;
78     /* introduced in bdd1f9edacb5f5835d1e6276571bbbe5b88ded48 (4.1) */
79     uint64_t  tcpi_bytes_received;
80     /* introduced in 2efd055c53c06b7e89c167c98069bab9afce7e59 (4.2) */
81     uint32_t  tcpi_segs_out;
82     uint32_t  tcpi_segs_in;
83
84     /* introduced in cd9b266095f422267bddbec88f9098b48ea548fc (4.6) */
85     uint32_t  tcpi_notsent_bytes;
86     uint32_t  tcpi_min_rtt;
87     /* introduced in a44d6eacdaf56f74fad699af7f4925a5f5ac0e7f (4.6) */
88     uint32_t  tcpi_data_segs_in;
89     uint32_t  tcpi_data_segs_out;
90
91     /* introduced in eb8329e0a04db0061f714f033b4454326ba147f4 (4.9) */
```

```
92    uint64_t  tcpi_delivery_rate;
93
94    /* introduced in efd90174167530c67a54273fd5d8369c87f9bd32 (4.10) */
95    uint64_t  tcpi_busy_time;
96    uint64_t  tcpi_rwnd_limited;
97    uint64_t  tcpi_sndbuf_limited;
98  };
99
100 /* macro to check if specified member was actually returned by getsockopt
101  * (i.e. kernel supports it) */
102 #define  IS_IN_RMBT_TCP_INFO(len, m)  (offsetof(struct rmbt_tcp_info, m) +
    ↪  sizeof(((struct rmbt_tcp_info *)0)->m) <= len)
103 #define JSON_ADD_OBJ_TCP_INFO(len, obj, m)  if (IS_IN_RMBT_TCP_INFO(len, m)) { \
104     rmbt_json_add_int64(obj, #m, (int64_t)i->m); }
105 #define JSON_ADD_OBJ_TCP_INFO_BITFIELD(len, obj, m, next_memb)  \
106     if (offsetof(struct rmbt_tcp_info, next_memb) <= len) { \
107       rmbt_json_add_int64(obj, #m, (int64_t)i->m); }
108
109 typedef struct {
110   int_fast64_t ts;
111   socklen_t tcp_info_length;
112   struct rmbt_tcp_info tcp_info;
113 } TcpInfoEntry;
114
115 typedef struct {
116   int sfd;
117   TcpInfoEntry *tcp_infos;
118   size_t tcp_infos_size;
119   size_t tcp_infos_length;
120 } StatsThreadEntry;
121
122 typedef struct {
123   struct timespec *ts_zero;
124   StatsThreadEntry *entries;
125   size_t length;
126   int_fast32_t tcp_info_sample_rate_us;
127 } StatsThreadArg;
128
129 void get_uname(rmbt_json obj);
130
131 void stats_set_arg(StatsThreadArg *arg);
132 rmbt_json_array get_stats_as_json_array(StatsThreadArg* e);
133 void stats_thread_set_sfd(int_fast16_t tid, int sfd);
134 void *stats_thread_start(void *arg) __attribute__ ((noreturn));
135
136 #endif /* SRC_RMBT_STATS_H_ */
```

Listing B.17: rmbt_stats.c

```
1  /*******************************************************************************
2   * Copyright 2017 Leonhard Wimmer
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *  http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
```

```
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13   * See the License for the specific language governing permissions and
14   * limitations under the License.
15   ******************************************************************************/
16
17  #include "rmbt_stats.h"
18
19  #include <time.h>
20  #include <sys/types.h>
21  #include <sys/utsname.h>
22  #include <sys/socket.h>
23  #include <netinet/in.h>
24  #include <netinet/tcp.h>
25
26  #include "rmbt_helper.h"
27
28  #define RMBT_STATS_INCREMENT  512
29
30  static pthread_mutex_t stats_mtx = PTHREAD_MUTEX_INITIALIZER;
31  static StatsThreadArg *stats_arg = NULL;
32
33   void get_uname(rmbt_json obj) {
34    struct utsname n;
35    if (uname(&n) < 0)
36      return;
37    rmbt_json_add_string(obj, "res_uname_sysname", n.sysname);
38    rmbt_json_add_string(obj, "res_uname_nodename", n.nodename);
39    rmbt_json_add_string(obj, "res_uname_release", n.release);
40    rmbt_json_add_string(obj, "res_uname_version", n.version);
41    rmbt_json_add_string(obj, "res_uname_machine", n.machine);
42  }
43
44  static void tcp_info_set_json(rmbt_json obj, struct rmbt_tcp_info *i, socklen_t
    ↪  i_len) {
45    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_state);
46    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_ca_state);
47    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_retransmits);
48    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_probes);
49    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_backoff);
50    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_options);
51    JSON_ADD_OBJ_TCP_INFO_BITFIELD(i_len, obj, tcpi_snd_wscale, tcpi_rto);
52    JSON_ADD_OBJ_TCP_INFO_BITFIELD(i_len, obj, tcpi_rcv_wscale, tcpi_rto);
53
54    /* tcpi_busy_time because tcpi_delivery_rate_app_limited was introduced with
      ↪  tcpi_delivery_rate */
55    JSON_ADD_OBJ_TCP_INFO_BITFIELD(i_len, obj, tcpi_delivery_rate_app_limited,
      ↪  tcpi_busy_time);
56
57    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_rto);
58    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_ato);
59    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_snd_mss);
60    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_rcv_mss);
61
62    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_unacked);
63    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_sacked);
64    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_lost);
65    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_retrans);
66    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_fackets);
67
68
69    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_last_data_sent);
70    JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_last_ack_sent);
```

```
71      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_last_data_recv);
72      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_last_ack_recv);
73
74      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_pmtu);
75      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_rcv_ssthresh);
76      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_rtt);
77      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_rttvar);
78      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_snd_ssthresh);
79      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_snd_cwnd);
80      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_advmss);
81      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_reordering);
82
83      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_rcv_rtt);
84      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_rcv_space);
85
86      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_total_retrans);
87
88      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_pacing_rate);
89      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_max_pacing_rate);
90      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_bytes_acked);
91      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_bytes_received);
92      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_segs_out);
93      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_segs_in);
94
95      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_notsent_bytes);
96      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_min_rtt);
97      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_data_segs_in);
98      JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_data_segs_out);
99
100     JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_delivery_rate);
101
102     JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_busy_time);
103     JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_rwnd_limited);
104     JSON_ADD_OBJ_TCP_INFO(i_len, obj, tcpi_sndbuf_limited);
105 }
106
107 static rmbt_json get_tcp_info_entry_as_json(TcpInfoEntry* e) {
108   rmbt_json obj = rmbt_json_new();
109   tcp_info_set_json(obj, &e->tcp_info, e->tcp_info_length);
110   rmbt_json_add_int64(obj, "timestamp_ns", e->ts);
111   return obj;
112 }
113
114 rmbt_json_array get_stats_as_json_array(StatsThreadArg* e) {
115   rmbt_json_array arr = rmbt_json_new_array();
116   for (size_t i = 0; i < e->length; i++) {
117     StatsThreadEntry *ste = &e->entries[i];
118     for (size_t j = 0; j < ste->tcp_infos_length; j++) {
119       rmbt_json obj = get_tcp_info_entry_as_json(&ste->tcp_infos[j]);
120       rmbt_json_add_int64(obj, "flow_id", (int64_t)i);
121       rmbt_json_add_to_array(arr, obj);
122     }
123   }
124   return arr;
125 }
126
127 static void rmbt_add_tcp_info(StatsThreadEntry *e) {
128   /* make sure there is enough space */
129   if (e->tcp_infos_length >= e->tcp_infos_size) {
130     e->tcp_infos_size += RMBT_STATS_INCREMENT;
131     e->tcp_infos = realloc(e->tcp_infos, e->tcp_infos_size *
      ↪  sizeof(TcpInfoEntry));
```

```
132       memset(&e->tcp_infos[e->tcp_infos_length], 0, (e->tcp_infos_size -
          ↪  e->tcp_infos_length) * sizeof(TcpInfoEntry));
133     }
134     e->tcp_infos[e->tcp_infos_length].ts = ts_diff(stats_arg->ts_zero);
135     struct rmbt_tcp_info *info = &e->tcp_infos[e->tcp_infos_length].tcp_info;
136     e->tcp_infos[e->tcp_infos_length].tcp_info_length = sizeof(struct
          ↪  rmbt_tcp_info);
137     if (getsockopt(e->sfd, IPPROTO_TCP, TCP_INFO, info,
          ↪  &e->tcp_infos[e->tcp_infos_length].tcp_info_length) == 0)
138       e->tcp_infos_length++;
139   }
140
141   void stats_thread_set_sfd(int_fast16_t tid, int sfd) {
142     pthread_mutex_lock(&stats_mtx);
143     if (stats_arg != NULL && tid < (int_fast16_t)stats_arg->length)
144       stats_arg->entries[tid].sfd = sfd;
145     pthread_mutex_unlock(&stats_mtx);
146   }
147
148   void stats_set_arg(StatsThreadArg *arg) {
149     pthread_mutex_lock(&stats_mtx);
150     stats_arg = arg;
151     for (size_t i = 0; i < stats_arg->length; i++)
152       stats_arg->entries[i].sfd = -1;
153     pthread_mutex_unlock(&stats_mtx);
154   }
155
156   void *stats_thread_start(__attribute__((unused)) void *arg) {
157     struct timespec sleep_time;
158     sleep_time.tv_sec = stats_arg->tcp_info_sample_rate_us / 1000000;
159     sleep_time.tv_nsec = stats_arg->tcp_info_sample_rate_us % 1000000 * 1000;
160     while(true) { // thread will be canceled; clock_nanosleep is a cancellation
          ↪  point
161       pthread_mutex_lock(&stats_mtx);
162       for (size_t i = 0; i < stats_arg->length; i++) {
163         int sfd = stats_arg->entries[i].sfd;
164         if (sfd >= 0)
165           rmbt_add_tcp_info(&stats_arg->entries[i]);
166       }
167       pthread_mutex_unlock(&stats_mtx);
168       int r = clock_nanosleep(CLOCK_REALTIME, 0, &sleep_time, NULL);
169       if (r != 0)
170         pthread_exit(NULL);
171     }
172   }
```

Listing B.18: rmbt_token.h

```
1   /*******************************************************************************
2    * Copyright 2017 Leonhard Wimmer
3    *
4    * Licensed under the Apache License, Version 2.0 (the "License");
5    * you may not use this file except in compliance with the License.
6    * You may obtain a copy of the License at
7    *
8    *  http://www.apache.org/licenses/LICENSE-2.0
9    *
10   * Unless required by applicable law or agreed to in writing, software
11   * distributed under the License is distributed on an "AS IS" BASIS,
12   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
 * See the License for the specific language governing permissions and
 * limitations under the License.
 ******************************************************************************/

#ifndef SRC_RMBT_TOKEN_H_
#define SRC_RMBT_TOKEN_H_

#include "rmbt_common.h"

const char *calc_token(const char *secret, const char *uuid, const char
↪  *start_time_str, char *hmac_out, size_t hmac_out_size);

#endif /* SRC_TOKEN_H_ */
```

Listing B.19: rmbt_token.c

```
/******************************************************************************
 * Copyright 2017 Leonhard Wimmer
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *  http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 ******************************************************************************/

#include "rmbt_token.h"

#include "rmbt_ssl.h"

static const char *base64(const char *input, int ilen, char *output, size_t *olen)
↪  {
  BIO *bmem, *b64;
  BUF_MEM *bptr;

  b64 = BIO_new(BIO_f_base64());
  bmem = BIO_new(BIO_s_mem());
  b64 = BIO_push(b64, bmem);
  BIO_write(b64, input, ilen);
  BIO_flush(b64);
  BIO_get_mem_ptr(b64, &bptr);

  if (bptr->length > *olen) {
    BIO_free_all(b64);
    return NULL;
  } else {
    memcpy((void *) output, bptr->data, bptr->length);
    output[bptr->length - 1] = '\0';
    *olen = bptr->length;
    BIO_free_all(b64);
    return output;
  }
}

```

```c
const char *calc_token(const char *secret, const char *uuid, const char *start_time_str, char *hmac_out, size_t hmac_out_size) {
  unsigned char md_buf[EVP_MAX_MD_SIZE];
  unsigned int md_size = sizeof(md_buf);

  unsigned char msg[128];
  int r;
  r = snprintf((char *) msg, sizeof(msg), "%s_%s", uuid, start_time_str);
  if (r < 0)
    return 0;

  unsigned char *md = HMAC(EVP_sha1(), secret, (int) strlen(secret), msg, strnlen((char *) msg, sizeof(msg)), md_buf, &md_size);
  if (md == NULL)
    return NULL;
  return base64((char *) md, (int) md_size, hmac_out, &hmac_out_size);
}
```

Listing B.20: configure.ac

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ([2.61])
AC_INIT(rmbt-client,
  m4_esyscmd_s([git describe --abbrev=9 --dirty --always --tags --long 2> /dev/null || echo unknown]),
  [https://github.com/lwimmer/rmbt-client/issues],,
  [https://github.com/lwimmer/rmbt-client])
AM_INIT_AUTOMAKE([foreign std-options -Wall -Werror -Wno-portability])
m4_ifdef([AM_SILENT_RULES], [AM_SILENT_RULES([yes])])
AC_CONFIG_SRCDIR([src/rmbt.c])
AC_CONFIG_HEADERS([config.h])

AC_USE_SYSTEM_EXTENSIONS

AC_PROG_CC([cc gcc clang])

# Checks for libraries.
PKG_CHECK_MODULES([UUID], [uuid], [AC_DEFINE([HAVE_UUID], [1], [Use libuuid])])
PKG_CHECK_MODULES([SSL], [libssl libcrypto], [AC_DEFINE([HAVE_LIBSSL], [1], [Use libssl])])
PKG_CHECK_MODULES([JSON], [json-c], [AC_DEFINE([HAVE_JSONC], [1], [Use json-c])])
PKG_CHECK_MODULES([LZMA], [liblzma], [AC_DEFINE([HAVE_LZMA], [1], [Use liblzma])], [true])

AC_CONFIG_FILES([Makefile src/Makefile])
AC_OUTPUT
```

Listing B.21: Makefile.am

```
AUTOMAKE_OPTIONS = foreign
SUBDIRS = src
EXTRA_DIST = README.md LICENSE config.example.json autobuild.sh
```

Listing B.22: src/Makefile.am

```
AUTOMAKE_OPTIONS = foreign

GIT_VERSION := $(shell git describe --abbrev=9 --dirty --always --tags --long 2>
↪  /dev/null)

WARNINGS =  -Wall -Wextra -Wformat=2 -Wswitch-default -Wcast-align -Wpointer-arith
↪  \
    -Wbad-function-cast -Wstrict-prototypes -Winline -Wundef -Wnested-externs \
    -Wcast-qual -Wshadow -Wwrite-strings -Wconversion -Wunreachable-code \
    -pedantic -Wdisabled-optimization -Winit-self -Wmissing-declarations
    ↪  -Wmissing-include-dirs \
    -Wmissing-prototypes -Wparentheses -Wredundant-decls -Wsequence-point \
    -Wsign-compare -Wuninitialized -Wno-format-nonliteral -Wmissing-noreturn
OPTS =    -fno-common -fstrict-aliasing -fmessage-length=0

AM_CFLAGS = -std=gnu11 -O3 -g $(WARNINGS) $(OPTS)

AM_CFLAGS += $(if $(GIT_VERSION),-DGIT_VERSION=\"$(GIT_VERSION)\",)

AM_LDFLAGS = -pthread

bin_PROGRAMS = rmbt
rmbt_SOURCES = rmbt.c \
    rmbt_common.h \
    rmbt_compress.c \
    rmbt_compress.h \
    rmbt_config.h \
    rmbt_flow.c \
    rmbt_flow.h \
    rmbt.h \
    rmbt_helper.c \
    rmbt_helper.h \
    rmbt_json.c \
    rmbt_json.h \
    rmbt_result.c \
    rmbt_result.h \
    rmbt_ssl.c \
    rmbt_ssl.h \
    rmbt_stats.c \
    rmbt_stats.h \
    rmbt_token.c \
    rmbt_token.h

rmbt_CFLAGS = $(JSON_CFLAGS) $(UUID_CFLAGS) $(SSL_CFLAGS) $(LZMA_CFLAGS)
↪  $(AM_CFLAGS)
rmbt_LDADD = $(JSON_LIBS) $(UUID_LIBS) $(SSL_LIBS) $(LZMA_LIBS)
```

Listing B.23: autobuild.sh

```
#!/bin/sh
set -e
[ -e configure ] || autoreconf -i
[ -e Makefile ]  || ./configure
make ${GIT_VERSION:+GIT_VERSION="$GIT_VERSION"}
```

Listing B.24: Dockerfile

```
1  #####
2  FROM alpine AS build-env
3  RUN apk add --no-cache util-linux-dev gcc autoconf automake make openssl-dev
   ↪  json-c-dev musl-dev
4
5  ADD . /work
6  WORKDIR /work
7
8  ARG GIT_VERSION
9  RUN GIT_VERSION=${GIT_VERSION:-unknown} ./autobuild.sh
10
11 #####
12 FROM alpine
13 ARG GIT_VERSION
14 LABEL version=${GIT_VERSION:-unknown}
15 RUN apk add --no-cache libuuid json-c
16 COPY --from=build-env /work/src/rmbt /bin/rmbt
17 ENTRYPOINT ["/bin/rmbt"]
```

List of Figures

List of Tables

List of Algorithms

List of Listings

Glossary

2G second generation mobile networks (GSM, CDMA,...). 9, 165

3G third generation mobile networks (UMTS, HSPA/WCDMA,...). 2, 165

4G forth generation mobile networks (LTE, LTE-Advanced). ix, 2, 165

5G fifth generation mobile networks (currently being standardized). 8, 165

Android is a prominent platform for smartphones, a mobile operating system based on the Linux kernel. 12, 18, 87

C is a rather low-level, general-purpose, imperative computer programming language. 18, 33, 37, 40, 41, 85, 97

Hypertext Transfer Protocol (HTTP) is a stateless protocol for transfer of data on the application layer over a computer network. It is primarily used to transfer website data, requested by a web browser. 13, 166

Internet Protocol (IP) is the primary communication protocol of the Internet for transmitting datagrams. It is defined in RFC 791 (STD 5) [91]. 2, 166

iOS is a mobile operating system by Apple Inc. for iPhone, iPad, and iPod touch. 12

Java is a high-level, general-purpose, object-oriented computer programming language. 18, 33, 85, 87

Linux is a free and open-source Unix-like computer operating system kernel. 69, 163

Measuring Mobile Broadband Networks in Europe (MONROE) (MONROE) is a European transnational open platform for independent, multi-homed, large-scale monitoring and assessment of performance of MBB networks in heterogeneous environments [71]. See section 3.7. ix, xi, 3, 166

One-Way Delay (OWD) is the time a packet takes through a network from source to destination. (cf. Round Trip Time). 164, 167

Round Trip Time (RTT) is the time a packet takes through a network from source to destination and back to source. (cf. One-Way Delay). 10, 25, 163, 167

Secure Sockets Layer (SSL) see Transport Layer Security. 167

TCP Segmentation Offload (TSO) is a technique for offloading the segmentation of egress data into TCP packets to the NIC. The operating system can send a multipacket buffer to the NIC thus reducing the CPU load on the sending system. 64, 72, 168

Transmission Control Protocol (TCP) is a reliable, ordered, error-checked stream protocol, mostly used on top of IP. It is defined in RFC 793 (STD 7) [92] and is one of the most used protocols on the Internet. 3, 168

Transport Layer Security (TLS) formerly known as SSL, is a set of cryptographic protocols for secure data communication over insecure networks such as the Internet. It is defined in RFC 5246 (TLS 1.2) and RFC 8446 (TLS 1.3). 10, 164, 168

UNIX timestamp is the elapsed time since the epoch (usually in seconds). The epoch is the time at 1970-01-01 00:00:00 UTC. See sections 3.150 and 4.16 of POSIX.1-2008 [34]. 23, 46

User Datagram Protocol (UDP) is a simplistic, connectionless, datagram-based network protocol, mostly used on top of IP. It is defined in RFC 768 (STD 6) [90]. 10, 168

xz is a lossless data compression file format which uses the Lempel–Ziv–Markov chain algorithm (LZMA) compression algorithm. 39, 40, 45

Acronyms

2G Second Generation. 9, *Glossary:* 2G

3G Third Generation. 2, 9, 75, *Glossary:* 3G

4G Fourth Generation. ix, xi, 2, 9, 63, 66, 75, 87, *Glossary:* 4G

5G Fifth Generation. 8, 9, 75, 87, *Glossary:* 5G

ACK Acknowledgment. 10, 21, 57, 165

AEAD Authenticated Encryption with Associated Data. 74

AES Advanced Encryption Standard. 74

AES-NI Advanced Encryption Standard New Instructions. 74

ASCI Advertising Standards Council of India. 3

ASCII American Standard Code for Information Interchange. 20

ATO Acknowledgment (ACK) Timeout. 57

BEREC The Body of European Regulators for Electronic Communications. 17

BSD Berkeley Software Distribution. 55

CDMA Code Division Multiple Access. 163

CDN Content Delivery Network. 13

CGN Carrier-Grade Network Address Translation. 75

CPU Central Processing Unit. 72–74, 164

DL Downlink. 7, 13, 26, 44, 45, 48, 51, 52, 63, 66, 76, 77, 81, 157

DNS Domain Name System. 10

DSL Digital Subscriber Line. 24

ECDF Empirical Cumulative Distribution Function. 67, 72, 77

EU European Union. 11, 33

FACK Forward Acknowledgment. 57

FCC Federal Communications Commission. 2

GSM Global System for Mobile Communications. 163

HMAC Hash-based Message Authentication Code. 40, 44, 74

HSPA High Speed Packet Access. 163

HTTP Hypertext Transfer Protocol. 10, 13, 166, *Glossary:* Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure. 18

ID Identifier. 45, 55

IEEE Institute of Electrical and Electronics Engineers. 69–72

IETF Internet Engineering Task Force. 10

IFG Interframe Gap. 71

IP Internet Protocol. 2, 8, 9, 42, 43, 46, 64, 69, 71, 72, 164, 166, *Glossary:* Internet Protocol

JSON JavaScript Object Notation. 16, 39–42, 45, 46, 49, 53

LTE Long Term Evolution. 34, 63, 66, 163

LZMA Lempel–Ziv–Markov chain algorithm. 164

MBB Mobile Broadband. ix, 2–5, 7, 9, 11, 33, 61, 65, 66, 75, 81, 85, 87, 163

MNO Mobile Network Operator. 63

MONROE Measuring Mobile Broadband Networks in Europe. ix, xi, 3–5, 18, 19, 33–35, 42, 61, 64, 67, 72, 74, 81, 86, 87, 166, *Glossary:* Measuring Mobile Broadband Networks in Europe

MSC Message Sequence Chart. 20, 25–28

MSS Maximum Segment Size. 10, 21, 57, 58

MTU Maximum Transmission Unit. 57, 71, 72, 167

NAT Network Address Translation. 75, 165

NDT Network Diagnostic Tool. 11

NIC Network Interface Controller. 72, 73, 164

NRA National Regulatory Authority. ix, 2, 11, 13

OECD Organisation for Economic Co-operation and Development. 17

OS Operating System. 72

OSI Open Systems Interconnection. 47, 48, 69

OWD One-Way Delay. 164, 167, *Glossary:* One-Way Delay

PMTU Path MTU Discovery. 57

QoS Quality of Service. 12

QUIC Quick UDP Internet Connections. 10

RAM Random Access Memory. 18, 34

RFC Requests for Comments. 8, 21, 42, 46, 51, 55, 58, 59, 69–71, 163, 164

RMBT RTR Multithreaded Broadband Test. 2, 14, 17–19, 22, 24, 25, 33, 34, 37, 39–46, 61, 63, 67, 69, 72, 73, 85–88, 159

RSRP Reference Signals Received Power. 81

RSSI Received Signal Strength Indicator. 81

RTO Retransmit Timeout. 56, 57

RTR Austrian Regulatory Authority for Broadcasting and Telecommunications (RTR-GmbH). ix, 2, 12, 14, 17, 167

RTT Round Trip Time. 10, 25, 26, 44, 45, 48, 51, 58, 59, 88, 163, 167, *Glossary:* Round Trip Time

SACK Selective Acknowledgment. 56, 57

SBC Single-board computer. 34

SI International System of Units / Système international (d'unités). 7

SIM Subscriber Identity Module. 65, 87

SSL Secure Sockets Layer. 164, 167, *Glossary:* Secure Sockets Layer

TCP Transmission Control Protocol. 3, 5, 7, 10, 13, 15, 18, 20–23, 25, 29, 32, 33, 42–48, 51–53, 55, 58, 63–67, 69, 71–73, 75, 77, 86–96, 164, 168, *Glossary:* Transmission Control Protocol

TLD Top-Level Domain. 63

TLS Transport Layer Security. 10, 20–23, 39, 40, 43, 47, 48, 67, 73, 74, 88, 164, 168, *Glossary:* Transport Layer Security

TSO TCP Segmentation Offload. 64, 72, 73, 86, 168, *Glossary:* TCP Segmentation Offload

UDP User Datagram Protocol. 10, 167, 168, *Glossary:* User Datagram Protocol

UL Uplink. 7, 13, 44, 45, 48, 49, 52, 53, 63, 66, 76, 77, 81, 157

UML Unified Modeling Language. 37

UMTS Universal Mobile Telecommunications System. 163

URL Uniform Resource Locator. 1, 88

UTC Coordinated Universal Time. 61

UUID Universally Unique Identifier. 23, 40, 46

WCDMA Wideband Code Division Multiple Access. 163

Bibliography

[1] 3rd Generation Partnership Project (3GPP). *Medium Access Control (MAC) protocol specification (Release 11) / TS 36.321*. Version V11.1.0. 2012-12. URL: http://www.qtc.jp/3GPP/Specs/36321-b10.pdf (visited on 2019-01-27).

[2] S. Aadeetya. *Airtel vs Reliance Jio vs Ookla: What They Said About Speed Tests*. The Quint. 2017-03-21. URL: https://www.thequint.com/tech-and-auto/tech-news/airtel-reliance-jio-and-ookla-fight-it-out-over-fast-internet-speed-test-claims-in-india (visited on 2017-11-09).

[3] G. Aceto, A. Botta, W. De Donato, P. Marchetta, A. Pescapé, and G. Ventre. „Open Source Platforms for Internet Monitoring and Measurement". In: *Signal Image Technology and Internet Based Systems (SITIS), 2012 Eighth International Conference on*. IEEE. 2012, pp. 563–570. DOI: 10.1109/SITIS.2012.87.

[4] Ö. Alay et al. „Demo: MONROE, a distributed platform to measure and assess mobile broadband networks". In: *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*. ACM. ACM, 2016, pp. 85–86. ISBN: 978-1-4503-4252-0. DOI: 10.1145/2980159.2980172.

[5] Ö. Alay, V. Mancuso, A. Brunstrom, S. Alfredsson, M. Mellia, G. Bernini, and H. Lonsethagen. „End to End 5G Measurements with MONROE: Challenges and Opportunities". In: *2018 IEEE 4th International Forum on Research and Technology for Society and Industry (RTSI)*. IEEE. 2018, pp. 1–6. DOI: 10.1109/RTSI.2018.8548510.

[6] O. Alay et al. „Experience: An Open Platform for Experimentation with Commercial Mobile Broadband Networks". In: *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. ACM. 2017, pp. 70–78. DOI: 10.1145/3117811.3117812.

[7] Ö. Alay et al. „Measuring and Assessing Mobile Broadband Networks with MONROE". In: *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2016 IEEE 17th International Symposium on A*. IEEE. 2016, pp. 1–3. DOI: 10.1109/WoWMoM.2016.7523537.

[8] Ö. Alay et al. „MONROE: Measuring Mobile Broadband Networks in Europe“. In: *Proceesings of the IRTF & ISOC Workshop on Research and Applications of Internet Measurements (RAIM)*. 2015.

[9] M. Allman, V. Paxson, and E. Blanton. *TCP Congestion Control*. RFC 5681. RFC Editor, 2009-09. DOI: 10.17487/RFC5681.

[10] M. Bagnulo, P. Eardley, T. Burbridge, B. Trammell, and R. Winter. „Standardizing Large-scale Measurement Platforms“. In: *ACM SIGCOMM Computer Communication Review* 43.2 (2013), pp. 58–63. DOI: 10.1145/2479957.2479967.

[11] V. Bajpai and J. Schönwälder. „A Survey on Internet Performance Measurement Platforms and Related Standardization Efforts“. In: *IEEE Communications Surveys & Tutorials* 17.3 (2015), pp. 1313–1341. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2418435.

[12] D. Baltrūnas. „On Reliability in Mobile Broadband Networks“. PhD thesis. University of Oslo, 2017-07, p. 216. URL: https://www.simula.no/publications/reliability-mobile-broadband-networks (visited on 2017-11-09).

[13] S. Bauer, D. D. Clark, and W. Lehr. „Understanding broadband speed measurements“. In: *TPRC 2010* (2010-08).

[14] Body of European Regulators for Electronic Communications. *BEREC Net Neutrality Regulatory Assessment Methodology*. 2017-10-05. URL: https://berec.europa.eu/eng/document_register/subject_matter/berec/regulatory_best_practices/methodologies/7295-berec-net-neutrality-regulatory-assessment-methodology (visited on 2019-02-02).

[15] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. *TCP Extensions for High Performance*. RFC 7323. RFC Editor, 2014-09. DOI: 10.17487/RFC7323.

[16] R. Braden. *Requirements for Internet Hosts - Communication Layers*. STD 3. RFC 1122. RFC Editor, 1989-10. DOI: 10.17487/RFC1122.

[17] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. RFC Editor, 2014-03. DOI: 10.17487/RFC7159.

[18] J. Cainey, B. Gill, S. Johnston, J. Robinson, and S. Westwood. „Modelling Download Throughput of LTE Networks“. In: *Local Computer Networks Workshops (LCN Workshops), 2014 IEEE 39th Conference on*. IEEE. 2014, pp. 623–628. DOI: 10.1109/LCNW.2014.6927712.

[19] I. Canadi, P. Barford, and J. Sommers. „Revisiting Broadband Performance“. In: *Proceedings of the 2012 Internet Measurement Conference*. ACM. 2012, pp. 273–286. DOI: 10.1145/2398776.2398805.

[20] S. Cheshire. *It's the latency, stupid*. 1996. URL: http://www.stuartcheshire.org/rants/Latency.html (visited on 2019-03-11).

[21] Cisco Systems, Inc. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper*. Document ID: 1454457600805266. 2017-02-07. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.pdf (visited on 2017-11-09).

[22] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. RFC Editor, 2008-08. DOI: 10.17487/RFC5246.

[23] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, K. Prabhu, et al. *iperf3 source code*. URL: https://github.com/esnet/iperf (visited on 2017-07-31).

[24] Ericsson. *Mobility Report*. Tech. rep. Ericsson, 2018-11. URL: https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-november-2018.pdf (visited on 2019-02-15).

[25] FCC. *Measuring Mobile Broadband*. 2014-09-29. URL: https://www.fcc.gov/general/measuring-mobile-broadband-performance (visited on 2019-02-17).

[26] U. Goel, M. P. Wittie, K. C. Claffy, and A. Le. „Survey of End-to-End Mobile Network Measurement Testbeds, Tools, and Services“. In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 105–123. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2485979.

[27] T. J. Hacker, B. D. Athey, and B. Noble. „The End-to-End Performance Effects of Parallel TCP Sockets on a Lossy Wide-Area Network“. In: *Proceedings 16th International Parallel and Distributed Processing Symposium*. IEEE. 2002. DOI: 10.1109/IPDPS.2002.1015527.

[28] E. Halepovic, J. Pang, and O. Spatscheck. „Can You GET Me Now?: Estimating the Time-to-first-byte of HTTP Transactions with Passive Measurements“. In: *Proceedings of the 2012 Internet Measurement Conference*. ACM. 2012, pp. 115–122. DOI: 10.1145/2398776.2398789.

[29] E. He, J. Leigh, O. Yu, and T. A. DeFanti. „Reliable blast UDP: Predictable high performance bulk data transfer“. In: *Proceedings. IEEE International Conference on Cluster Computing*. IEEE. 2002, pp. 317–324. DOI: 10.1109/CLUSTR.2002.1137760.

[30] M. Hirth, T. Hoßfeld, M. Mellia, C. Schwartz, and F. Lehrieder. „Crowdsourced Network Measurements: Benefits and Best Practices“. In: *Computer Networks* 90 (2015), pp. 85–98. DOI: 10.1016/j.comnet.2015.07.003.

[31] S. Homayouni, V. Raida, P. Svoboda, and M. Rupp. „The impact of duration and settings of TCP measurements on available bandwidth estimation in mobile networks“. In: *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. IEEE. 2017-10, pp. 1–6. DOI: 10.1109/PIMRC.2017.8292522.

[32] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. „An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance“. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 363–374. ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486006.

[33] J. Huang et al. „Mobiperf: Mobile network measurement system“. In: *Technical Report. University of Michigan and Microsoft Research* (2011).

[34] IEEE. „Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. POSIX.1-2008“. In: *IEEE Std 1003.1TM, 2016 Edition* (2016-09), pp. 1–3957. DOI: 10.1109/IEEESTD.2016.7582338.

[35] Intel. *Ethernet Controller I210 Datasheet*. 2018. URL: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/i210-ethernet-controller-datasheet.pdf (visited on 2019-02-03).

[36] *iPerf*. URL: https://iperf.fr/ (visited on 2017-07-31).

[37] ISO. *Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model*. ISO/IEC 7498-1. Geneva, CH: ISO, 1994-11.

[38] ITU. *IMT Vision – Framework and overall objectives of the future development of IMT for 2020 and beyond*. Recommendation ITU-R M.2083-0. ITU, 2015-10-12. URL: https://www.itu.int/rec/R-REC-M.2083-0-201509-I/en (visited on 2019-02-15).

[39] M. Jain and C. Dovrolis. „End-to-end Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput“. In: *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. Vol. 32. SIGCOMM '02 4. Pittsburgh, Pennsylvania, USA: ACM, 2002, pp. 295–308. DOI: 10.1145/633025.633054.

[40] G. Jin and B. Tierney. „Netest: A Tool to Measure the Maximum Burst Size, Available Bandwidth and Achievable Throughput“. In: *International Conference on Information Technology: Research and Education, 2003. Proceedings. ITRE2003.* IEEE. 2003, pp. 578–582. DOI: 10.1109/ITRE.2003.1270685.

[41] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. RFC Editor, 2006-10. DOI: 10.17487/RFC4648.

[42] A. S. Khatouni et al. „Speedtest-like Measurements in 3G/4G Networks: The MONROE Experience“. In: *Teletraffic Congress (ITC 29), 2017 29th International*. Vol. 1. IEEE. 2017, pp. 169–177. DOI: 10.23919/ITC.2017.8064353.

[43] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. RFC Editor, 1997-02. DOI: 10.17487/RFC2104.

[44] C. Kreibich, N. Weaver, G. Maier, B. Nechaev, and V. Paxson. „Experiences from Netalyzr with Engaging Users in End-system Measurement“. In: *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack*. ACM, 2011, pp. 25–30. DOI: 10.1145/2018602.2018609.

[45] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. „Netalyzr: Illuminating the Edge Network“. In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 246–259. DOI: 10.1145/1879141.1879173.

[46] P. J. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. RFC Editor, 2005-07. DOI: 10.17487/RFC4122.

[47] D. J. Leith, R. N. Shorten, and G. McCullagh. „Experimental evaluation of Cubic-TCP“. In: *Proceedings of the 6th International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2008)*. 2008.

[48] F. Li, X. Jiang, J. W. Chung, and M. Claypool. „Who is the King of the Hill? Traffic Analysis over a 4G Network“. In: *2018 IEEE International Conference on Communications (ICC)*. IEEE. 2018, pp. 1–6. DOI: 10.1109/ICC.2018.8422958.

[49] W. Li, R. K. Mok, R. K. Chang, and W. W. Fok. „Appraising the Delay Accuracy in Browser-based Network Measurement“. In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC '13. ACM. ACM, 2013, pp. 361–368. DOI: 10.1145/2504730.2504760.

[50] W. Li, R. Mok, D. Wu, and R. Chang. „On the Accuracy of Smartphone-based Mobile Network Measurement“. In: *IEEE Conference on Computer Communications (INFOCOM)*. 2015-04, pp. 370–378. DOI: 10.1109/INFOCOM.2015.7218402.

[51] Y. Li, C. Peng, Z. Yuan, J. Li, H. Deng, and T. Wang. „MobileInsight: Extracting and Analyzing Cellular Network Information on Smartphones“. In: *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*. ACM, 2016, pp. 202–215. DOI: 10.1145/2973750.2973751.

[52] Linux Programmer's Manual, ed. *tcp(7)*. 2017-09-15.

[53] I. Livadariu, K. Benson, A. Elmokashfi, A. Dhamdhere, and A. Dainotti. „Inferring Carrier-Grade NAT Deployment in the Wild“. In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE. 2018, pp. 2249–2257. DOI: 10.1109/INFOCOM.2018.8486223.

[54] M-Lab. *The M-Lab MobiPerf Data Set*. 2019. URL: https://measurementlab.net/tests/mobiperf.

[55] V. Mancuso et al. „Results from running an experiment as a service platform for mobile broadband networks in Europe“. In: *Computer Communications* 133 (2019), pp. 89–101. DOI: 10.1016/j.comcom.2018.09.004.

[56] A. M. Mandalari, A. Lutu, A. Dhamdhere, M. Bagnulo, and K. Claffy. „Tracking the Big NAT across Europe and the U.S“. In: *arXiv:1704.01296* (2017).

[57] M. Mathis, J. Heffner, and R. Raghunarayan. *TCP Extended Statistics MIB*. RFC 4898. RFC Editor, 2007-05. DOI: 10.17487/RFC4898.

[58] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*. RFC 2018. RFC Editor, 1996-10. DOI: 10.17487/RFC2018.

[59] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. „The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm“. In: *ACM SIGCOMM Computer Communication Review* 27.3 (1997), pp. 67–82. DOI: 10.1145/263932.264023.

[60] J. Mattsson. *Overview and Analysis of Overhead Caused by TLS*. Internet-Draft draft-mattsson-uta-tls-overhead-01. Internet Engineering Task Force, 2014-10-27. URL: https://datatracker.ietf.org/doc/html/draft-mattsson-uta-tls-overhead-01 (visited on 2019-02-03).

[61] Measurement Lab. *M-Lab Platform Status*. URL: https://www.measurementlab.net/status/ (visited on 2017-11-09).

[62] P. Megyesi, Z. Krämer, and S. Molnár. „How quick is QUIC?“ In: *2016 IEEE International Conference on Communications (ICC)*. IEEE. 2016, pp. 1–6. DOI: 10.1109/ICC.2016.7510788.

[63] J. M. Mellor-Crummey and M. L. Scott. „Algorithms for Scalable Synchronization on Shared-memory Multiprocessors“. In: *ACM Transactions on Computer Systems (TOCS)* 9.1 (1991), pp. 21–65. DOI: 10.1145/103727.103729.

[64] C. Midoglu and P. Svoboda. „Opportunities and Challenges of Using Crowdsourced Measurements for Mobile Network Benchmarking“. In: *SAI Computing Conference (SAI), 2016*. IEEE. 2016-07, pp. 996–1005. DOI: 10.1109/SAI.2016.7556101.

[65] C. Midoglu, **L. Wimmer**, A. Lutu, Ö. Alay, and C. Griwodz. „MONROE-Nettest: A Configurable Tool for Dissecting Speed Measurements in Mobile Broadband Networks“. In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2018-04, pp. 342–347. DOI: 10.1109/INFCOMW.2018.8406836.

[66] C. Midoglu, **L. Wimmer**, and P. Svoboda. „Server Link Load Modeling and Request Scheduling for Crowdsourcing-Based Benchmarking Systems“. In: *Wireless Communications and Mobile Computing Conference (IWCMC), 2016 International*. IEEE. 2016-09, pp. 988–994. DOI: 10.1109/IWCMC.2016.7577193.

[67] G. Minshall, Y. Saito, J. C. Mogul, and B. Verghese. „Application performance pitfalls and TCP's Nagle algorithm“. In: *ACM SIGMETRICS Performance Evaluation Review* 27.4 (2000), pp. 36–44. ISSN: 0163-5999. DOI: 10.1145/346000.346012.

[68] A. R. Mishra. *Advanced Cellular Network Planning and Optimisation: 2G/2.5 G/3G... Evolution to 4G*. John Wiley & Sons, 2006. ISBN: 978-0-470-01471-4.

[69] *MobiPerf*. URL: https://sites.google.com/site/mobiperfdev/ (visited on 2017-07-31).

[70] MobiPerf. *MobiPerf source code*. 2011–2014. URL: https://github.com/Mobiperf/MobiPerf (visited on 2017-07-31).

[71] *MONROE project. Measuring Mobile Broadband Networks in Europe*. URL: https://www.monroe-project.eu/ (visited on 2017-07-31).

[72] J. Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. RFC Editor, 1984. DOI: 10.17487/RFC0896.

[73] K. Nepomuceno, I. N. de Oliveira, R. R. Aschoff, D. Bezerra, M. S. Ito, W. Melo, D. Sadok, and G. Szabó. „QUIC and TCP: A Performance Evaluation“. In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2018, pp. 00045–00051. DOI: 10.1109/ISCC.2018.8538687.

[74] *Netradar*. URL: https://www.netradar.org/ (visited on 2017-07-31).

[75] A. Nikravesh, D. R. Choffnes, E. Katz-Bassett, Z. M. Mao, and M. Welsh. „Mobile Network Performance from User Devices: A Longitudinal, Multidimensional Analysis.“ In: *PAM*. Vol. 14. Springer. 2014, pp. 12–22. DOI: 10.1007/978-3-319-04918-2_2.

[76] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao. „Mobilyzer: An Open Platform for Controllable Mobile Network Measurements“. In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 389–404. DOI: 10.1145/2742647.2742670.

[77] K. Nishizuka and D. Natsume. *Carrier-Grade-NAT (CGN) Deployment Considerations.* Internet-Draft draft-nishizuka-cgn-deployment-considerations-01. Work in Progress. Internet Engineering Task Force, 2013-09-27. 16 pp. URL: https://datatracker.ietf.org/doc/html/draft-nishizuka-cgn-deployment-considerations-01 (visited on 2019-02-03).

[78] OECD. „Access Network Speed Tests“. In: *OECD Digital Economy Papers* No. 237 (2014). OECD Publishing. DOI: 10.1787/5jz2m5mr66f5-en.

[79] Official Journal of the European Union. *Regulation (EU) 2015/2120 of the European Parliament and of the Council of 25 November 2015 laying down measures concerning open internet access and amending Directive 2002/22/EC on universal service and users' rights relating to electronic communications networks and services and Regulation (EU) No 531/2012 on roaming on public mobile communications networks within the Union (Text with EEA relevance)*. 2015-11-25. URL: https://publications.europa.eu/en/publication-detail/-/publication/8fdf5d08-93fc-11e5-983e-01aa75ed71a1/language-en (visited on 2019-02-02).

[80] Ookla. URL: http://www.speedtest.net/speedtest-servers-static.php (visited on 2017-11-09).

[81] Ookla. *Speedtest*. URL: http://www.speedtest.net/ (visited on 2017-07-31).

[82] *OpenSignal*. URL: https://opensignal.com/ (visited on 2017-07-31).

[83] K. Pahlavan and P. Krishnamurthy. „IEEE 802.3 Ethernet“. In: *Networking Fundamentals: Wide, Local and Personal Area Communications*. Wiley Telecom, 2009, pp. 656–. ISBN: 9780470779422. DOI: 10.1002/9780470779422.ch8.

[84] K. Pahlavan and A. H. Levesque. „Wireless Data Communications“. In: *Proceedings of the IEEE* 82.9 (1994), pp. 1398–1430. DOI: 10.1109/5.317085.

[85] V. Paxson. „Towards a Framework for Defining Internet Performance Metrics“. In: *Proc. INET'96*. Citeseer. 1996.

[86] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. *Framework for IP Performance Metrics*. RFC 2330. RFC Editor, 1998-05. DOI: 10.17487/RFC2330.

[87] M. Peón-Quirós et al. „Results from Running an Experiment As a Service Platform for Mobile Networks“. In: *Proceedings of the 11th Workshop on Wireless Network Testbeds, Experimental evaluation & CHaracterization*. WiNTECH '17. ACM. 2017, pp. 9–16. DOI: 10.1145/3131473.3131485.

[88] S. Perreault, I. Yamagata, S. Miyakawa, A. Nakagawa, and H. Ashida. *Common Requirements for Carrier-Grade NATs (CGNs)*. BCP 127. RFC Editor, 2013. URL: http://www.rfc-editor.org/rfc/rfc6888.txt.

[89] M. Peón-Quirós, Ö. Alay, V. Mancuso, T. Hirsch, and A. S. Khatouni. *MONROE Platform User Manual*. 2017-06-14. URL: https://github.com/MONROE-PROJECT/UserManual (visited on 2017-07-31).

[90] J. Postel. *User Datagram Protocol*. STD 6. RFC 768. RFC Editor, 1980-08. DOI: 10.17487/RFC0768.

[91] J. Postel. *Internet Protocol*. STD 5. RFC 791. RFC Editor, 1981-09. DOI: 10.17487/RFC0791.

[92] J. Postel. *Transmission Control Protocol*. STD 7. RFC 793. RFC Editor, 1981-09. DOI: 10.17487/RFC0793.

[93] S. Rosen, H. Yao, A. Nikravesh, Y. Jia, D. Choffnes, and Z. M. Mao. „Demo: Mapping Global Mobile Performance Trends with Mobilyzer and MobiPerf“. In: *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 353–353. ISBN: 978-1-4503-2793-0. DOI: 10.1145/2594368.2601469.

[94] Rundfunk und Telekom Regulierungs-GmbH. *72. Mobilregulierungsdialog. RTR-Netztest*. 2012-11-12. URL: https://www.rtr.at/de/inf/RegDialog12112012/29093_MRD_RTR-Netztest.pdf (visited on 2018-01-20).

[95] Rundfunk und Telekom Regulierungs-GmbH. *RTR-NetTest open-source repository*. 2013. URL: https://github.com/rtr-nettest/open-rmbt (visited on 2017-07-31).

[96] Rundfunk und Telekom Regulierungs-GmbH. *RTR-Netztest*. Austria. URL: https://www.netztest.at/ (visited on 2017-07-31).

[97] Rundfunk und Telekom Regulierungs-GmbH. *RTR-Netztest - Open Data Interface Specification*. 2019. URL: https://www.netztest.at/en/Opendata (visited on 2019-02-02).

[98] M. A. Sánchez, F. E. Bustamante, B. Krishnamurthy, and W. Willinger. „Experiment Coordination for Large-scale Measurement Platforms“. In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data*. ACM. 2015, pp. 21–26. DOI: 10.1145/2787394.2787401.

[99] J. Serrà and J. L. Arcos. „An Empirical Evaluation of Similarity Measures for Time Series Classification“. In: *Knowledge-Based Systems* 67 (2014), pp. 305–314. ISSN: 0950-7051. DOI: 10.1016/j.knosys.2014.04.035. arXiv: 1401.3973.

[100] S. Sonntag, L. Schulte, and J. Manner. „Mobile Network Measurements–It's not all about Signal Strength“. In: *Wireless Communications and Networking Conference (WCNC), 2013 IEEE*. IEEE. 2013-04, pp. 4624–4629. DOI: 10.1109/WCNC.2013.6555324.

[101] S. Sonntag, J. Manner, and L. Schulte. „Netradar - Measuring the Wireless World“. In: *Modeling & Optimization in Mobile, Ad Hoc & Wireless Networks (WiOpt), 2013 11th International Symposium on*. IEEE. 2013-05, pp. 29–34.

[102] S. Sundaresan, M. Allman, A. Dhamdhere, and K. Claffy. „TCP Congestion Signatures“. In: *Proceedings of the 2017 Internet Measurement Conference*. ACM. ACM, 2017, pp. 64–77. DOI: 10.1145/3131365.3131381.

[103] C. Sölder, **L. Wimmer**, D. Zlabinger, U. Latzenhofer, U. Prinzl, P. Sandner, L. Budryk, and U. Liener. *RTR Multithreaded Broadband Test (RMBT): Specification*. Version 0.8.0. 2015-10-16. URL: https://www.netztest.at/doc/ (visited on 2017-08-01).

[104] A. S. Tanenbaum and D. J. Wetherall. *Computer Networks*. 5th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2011. ISBN: 978-0-13-212695-3.

[105] Telenor ASA. *Best in Test - We are fastest in Norway!* 2017. URL: https://www.telenor.no/privat/dekning/ookla.jsp (visited on 2017-11-09).

[106] **L. Wimmer**, C. Midoglu, A. Lutu, Ö. Alay, and C. Griwodz. „Concept and Implementation of a Configurable Nettest Tool for Mobile Broadband“. In: *2018 IEEE Wireless Communications and Networking Conference (WCNC): IEEE WCNC2018 Student Program (IEEE WCNC 2018 Students)*. Barcelona, Spain, 2018-04.

[107] The Apache Software Foundation. *Apache License*. 2004. URL: https://www.apache.org/licenses/LICENSE-2.0 (visited on 2019-02-02).

[108] E. Yildirim and T. Kosar. „End-to-End Data-Flow Parallelism for Throughput Optimization in High-Speed Networks“. In: *Journal of Grid Computing* 10.3 (2012-09), pp. 395–418. ISSN: 1572-9184. DOI: 10.1007/s10723-012-9220-9.

[109] E. Yildirim, I. H. Suslu, and T. Kosar. „Which Network Measurement Tool is Right for You? A Multidimensional Comparison Study“. In: *2008 9th IEEE/ACM International Conference on Grid Computing*. IEEE. 2008, pp. 266–275. DOI: 10.1109/GRID.2008.4662808.