Improving Application QoE with Flow-Level, Interface-Level, and Device-Level Parallelism

by

Yihua Guo

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2017

Doctoral Committee:

Professor Z. Morley Mao, Chair
Professor Jason N. Flinn
Assistant Professor Feng Qian, Indiana University
Professor Ji Zhu

Yihua Guo

yhguo@umich.edu

ORCID iD: 0000-0001-9562-5481

© Yihua Guo 2017

All Rights Reserved

To Yiji, my grandparents, and my parents

ACKNOWLEDGEMENTS

Pain is inevitable. Suffering is optional.

Say you're running and you think, Man, this hurts, I can't take it anymore. *The* hurt *part is an unavoidable reality, but whether or not you can stand anymore is up to the runner himself.*

HARUKI MURAKAMI

Translated by PHILIP GABRIEL

Pursuing a Ph.D. is a *marathon*: the *runner* needs to motivate him or herself continuously to reach the finish line. This process requires enormous efforts. I am humbled to be one of those who accomplish the journey, with this dissertation as the finale. For me, the support of many individuals is indispensable to the birth of this dissertation. They make my journey much more enjoyable.

My advisor, Z. Morley Mao, provided endless support, inspiration, and foresight in my five years of Ph.D. study. I still remember her relentless efforts on editing our draft and focusing on every detail for my first paper submission in 2014. Her guidance has laid the foundation of my research, writing, and presentation skills. In research meetings, she always inspired me to think out of the box with her unique perspective on networking systems. Her candid feedbacks helped me overcome my weaknesses in the research; her insights, like lighthouses in the storm, helped me see the big picture. I'm appreciative of her belief in my research capability even in many times when it seemed like I might never finish, and I'm honored to work with her along to push the boundaries of the knowledge base of computer science.

I would like to thank my other committee members, Jason Flinn, Feng Qian, and Ji Zhu, for their insightful comments and valuable suggestions. They raise important questions on several aspects in this dissertation, asking me to focus on clarity and soundness. With their effort, this dissertation becomes complete and thorough.

Through remote collaboration, I greatly benefited from Feng Qian and Subhabrata (Shubho) Sen during my research career. They are engaged in every research discussion for most of my research projects. As a former AT&T Labs researcher and an Assistant Professor at Indiana University since 2015, Feng has edited all my papers from the research projects. I often discussed research questions with him for his advice. His devotion to research set an excellent example of the combination of passion, wisdom, and persistence for me. Shubho has inspired me in both research and life. I still remember the time when Shubho and I took a walk on the beach at Santa Monica, talking about my multipath project and his life experience. It's also a pleasure working with Feng and Shubho in person during my internship at AT&T in the summer of 2014.

Since Morley values the research collaboration with industry partners, I had the great opportunities of working with wonderful researchers from many companies besides AT&T. During my first year, I collaborated with Ulas Kozat, a then Docomo researcher, on improving cellular network performance. He introduced me to the world of cellular network research. In my fourth year, I got the chance of working on improving the VoLTE call quality with T-Mobile. The great researchers at T-Mobile including Jie Hui, Alex Yoon, Antoine Tran, and Kranthi Sontineni have expanded my vision into real problems in production networks. I have also received valuable perspective and suggestion on my multipath networking projects from Mondira Pant at Intel. I would like to express my gratitude to all of them.

I am very fortunate to have a few internship experiences in five years. During these internships, I have been working with knowledgeable and excellent researchers and engineers, including Mario Baldi, Sung-Ju Lee, Stanislav Miskovic, and Bruno Nardelli at

Narus (Sunnyvale, 2013), Vijay Gopalakrishnan, Emir Halepovic, and Oliver Spatscheck at AT&T Labs – Research (Bedminster, 2014), Andreas Terzis and Krishna Sayana at Google (Mountain View, 2015), Vinoth Chandar, Rajesh Mahindra, and Christopher Brauchli at Uber (San Francisco, 2017). I have enjoyed working with them. Many of their words have shaped my perspective on industrial research and product engineering.

My colleagues and friends at Michigan always make me feel like home here. Qiang Xu mentored me during my first year. He taught me many research basics. Junxian Huang included me in his in-depth study of LTE. It is my pleasure to work with him and make an important contribution to one of his major publication. Qi Alfred Chen and I entered the doctoral program at Michigan at the same time. I am grateful to be a friend of him, who elegantly combines idealism and realism. We always had a lot of things to chat about (his car's electrical system even shut down during midnight before we realized, after hours of discussion on research between us in his vehicle without the engine running!). I've had many other excellent graduate student collaborators as well, including Yuanyuan Tracy Zhou, Mehrdad Moradi, Ke David Hong, Sanae Rosen, Ashkan Nikravesh, Yunhan Jack Jia, and Xiao Shawn Zhu. I learned a lot from their different characters: Yuanyuan's creativity, Mehrdad's meticulousness, David's modesty, Sanae's enthusiasm, Ashkan's endurance, Jack's improvisation, and Xiao's diligence, to name a few.

My thanks also go to my other friends and colleagues at Michigan, including, but not limited to: Haokun Luo, Mark Gordon, Hongyi Yao, Amir Rahmati, Earlence Fernandes, Shichang Shawn Xu, Yuru Roy Shao, Yikai Lin, Chao Kong, Jie You, Jeremy Erickson, Yulong Cao, Yucheng Yin, Shengtuo Hu, *etc.* I benefited from their insightful comments. We also shared happiness and sorrow together, in research and life. I remember happy times with many of them in SIGBAAAAA, our foosball fun club. I am thankful for my other friends and people outside Michigan for appearing in and shaping my life: Zhe Wang, Xiaozhou Che, Wen Ma, Jingxing Wang, Yuxuan Zhang, Xiaoxiang Wang, Meng Wang, Tianyi Ma, Xiaofei Wen, Yushu Ma, Zhiyun Lu, Kuan Liu, Siji Quan, Yanrong Kang,

Xiaoyong Wu, Wei Li, Chunxu Xu, Yu Su, Hua He, Da Li, Sen Yang, Yu Xiang, Zheng Wang, Jian Zhang, Jie Jiang, Jacky Jiang, *etc.* In my spare time, I found a lot of joy playing the piano, with excellent guidance from my piano teacher, Daniel Drummond, since 2014.

It is hard for me to forget my undergraduate research advisor in Tsinghua University, Yong Cui. Professor Cui introduced me to the academic research, especially the research in computer networking. To prepare me with a necessary research background, he allowed me to participate in his graduate-level wireless networking course and research projects. I owe my special thanks to him for the great research experience I had in his research group and his strong recommendation of me to Morley.

Ann Arbor is the place I lived and worked for five years. The Computer Science and Engineering at Michigan is a great place for learning and research. The CSE staff makes everything smooth in the department. Their names include Dawn Freysinger, Ashley Andreae, Stephen Reger, Karen Liska, *etc.* I thank their efforts on making administrative things less concern for me during my doctoral study. I will remember people at CSE, the vibrant campus, and the Michigan Spirit. *Always Leading, Forever Valiant.*

Finally, I would like to express my deepest thanks to my dearest grandparents, Zhenquan Guo, Zhuo Cao, Deguang Chen, and Zhilin Wang, and my beloved parents, Fan Guo and Yi Chen for their unconditional support and love throughout my life. Your words and deeds have a lifelong impact on me. Yiji, you are always for me, during the highs and lows of my doctoral study. You are my friend, my love, and my life. This dissertation is dedicated to all of you.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES xi

LIST OF TABLES xv

ABSTRACT xvi

CHAPTER

I. Introduction 1

1.1 *Addressing Flow-Level Parallelism*: Reducing Cross-Traffic Interference on Mobile Devices 5
1.2 *Leveraging Interface-Level Parallelism*: Improving Multipath Transport with Flexible Architecture and Optimized Scheduler . . 7
1.3 *Embracing Device-level Parallelism*: Designing Wearable Network Management under Mobility for Real-Time Apps 9
1.4 Thesis Organization 10

II. Background 12

2.1 Multipath Networking 12
2.1.1 Multipath over Mobile Networks 12
2.1.2 Multipath TCP and its Schedulers 13
2.2 Android Wear OS 13

III. Understanding and Solving On-Device Bufferbloat in Cellular Networks 15

3.1 Introduction 15
3.2 Experimental Methodology 19
3.2.1 Controlled Local Experiments 19

3.2.2 Network Traces from a User Study 20
3.3 Upload Traffic Measurement 20
3.3.1 Traffic Volume 20
3.3.2 Flow Duration 21
3.3.3 Flow Rate 22
3.3.4 Flow Concurrency 23
3.3.5 RTT Dynamics 24
3.3.6 Impact of Upload on Download Latency 24
3.3.7 Summary 25
3.4 On-device Queuing Delay of Upload Traffic 25
3.4.1 Overall Delay Characterization 26
3.4.2 Root Cause of On-device Queuing 28
3.4.3 Prevalence across Carriers & Devices 31
3.4.4 Uplink Throughput Measurement 31
3.5 Impact of Upload on Mobile Application Performance 33
3.5.1 Impact of Upload on Bulk Download 33
3.5.2 Impact on Web Browsing 35
3.5.3 Impact on Video Streaming and VoIP 36
3.6 QCUT: Solution for On-device Bufferbloat 37
3.6.1 Inadequateness of Existing Solutions 37
3.6.2 QCUT Design 38
3.6.3 QCUT Implementation 44
3.7 Evaluation 45
3.7.1 Existing Solutions 45
3.7.2 Evaluation of QCUT 49
3.8 Discussion 53
3.9 Summary 53

IV. Improving Multipath Architecture for Mobile Networks 55

4.1 Introduction 55
4.2 MPFlex: A Flexible Architecture For Mobile Multipath 57
4.2.1 The MPFlex Architecture 57
4.2.2 MPFlex Design and Implementation 60
4.2.3 MPFlex Use Cases 63
4.3 Evaluation of MPFlex 65
4.3.1 File Download 66
4.3.2 Web Browsing 67
4.3.3 Applying Multipath Policies 68
4.3.4 Plugging-in Custom Schedulers 69
4.3.5 Impact of Proxy Location 70
4.3.6 System Overhead 72
4.4 Summary 72

V. Accelerating Multipath Transport through Balanced Subflow Completion . . . 74

5.1 Introduction . . . 74
5.2 Background and Motivation . . . 78
5.2.1 Can We Further Improve MinRTT? . . . 78
5.2.2 Ensuring Simultaneous Subflow Completion and its Challenges . . . 80
5.3 The DEMS Algorithm . . . 81
5.3.1 Chunk-based Data Transfer . . . 81
5.3.2 Simultaneous Subflow Completion . . . 82
5.3.3 Handling Variable Network Conditions . . . 86
5.3.4 Adaptive Reinjection . . . 88
5.3.5 Put Everything Together . . . 90
5.4 System Design . . . 92
5.5 Implementation . . . 94
5.6 Evaluation . . . 95
5.6.1 Experimental Setup and Methodology . . . 96
5.6.2 Stable Network Conditions . . . 97
5.6.3 Varying Network Conditions . . . 100
5.6.4 Field Test under Real-World Settings . . . 103
5.6.5 Compare with Other Schedulers . . . 105
5.6.6 Web Browsing Performance . . . 106
5.6.7 System Overhead . . . 108
5.7 Discussions . . . 109
5.8 Summary . . . 110

VI. A First Look at Android Wear Networking Performance under Mobility 111

6.1 Introduction . . . 111
6.2 Poor Handover Performance . . . 113
6.2.1 Impact of Handover on Real-time Apps . . . 113
6.2.2 Root Cause Analysis . . . 118
6.3 WearMan: Improving Handover Performance . . . 120
6.3.1 Solution Overview . . . 120
6.3.2 Implementation . . . 122
6.3.3 Evaluation of WearMan . . . 124
6.4 Summary . . . 125

VII. Related Work . . . 126

7.1 Improving TCP Performance over Cellular Networks . . . 126
7.2 Improving Multipath Transport . . . 128
7.3 Wearable Networking . . . 129

VIII. Conclusion and Future Work 130

8.1 Future Work 132
8.1.1 New Types of Applications: Beyond DEMS and Multipath over TCP 132
8.1.2 New Types of Devices: from Smartwatch to Internet of Things (IoT) and Autonomous Vehicles (AVs) 133

BIBLIOGRAPHY 135

LIST OF FIGURES

Figure

1.1 Dissertation organization. 5

3.1 Traffic volume distribution of user sessions. 21

3.2 Flow duration distributions. 22

3.3 Flow rate distributions. 22

3.4 TCP concurrency distribution. 23

3.5 Delay distributions. 24

3.6 On-device bufferbloat (in thick blue). 26

3.7 Overall latency characterization for a single TCP upload flow under two network conditions. 27

3.8 Packet processing and transmission on Android devices. 29

3.9 On-device queuing delay on diverse devices and cellular carriers. 31

3.10 Uplink throughput measurement error at different layers. 32

3.11 Impact of uplink traffic on downlink TCP/UDP throughput. 34

3.12 Impact of upload on PLT. The web browsing session begins δt after upload starts. “X” indicates the upload is completed before the web page is fully loaded. 36

3.13 The QCUT design. 39

3.14 Uplink throughput prediction error at different layers with 20ms prediction interval. 40

3.15 Radio firmware buffer occupancy estimation. 42

3.16 Impact of firmware buffer occupancy threshold of QCUT-B. The best threshold in each plot is in bold blue text. 43

3.17 Impact of TCP send buffer sizes on upload performance. 46

3.18 Impact of different TCP small queue (TSQ) sizes on upload. 46

3.19 Impact of different TCP CC on upload (with TSQ=128KB). 47

3.20 Effectiveness of CoDel on reducing latency. 48

3.21 Effectiveness of jointly applying multiple mitigation strategies on upload. 48

3.22 Compare TCP upload performance of different schemes. 50

3.23 Improvement of application performance brought by QCUT. 52

4.1 The MPFlex architecture. 58

4.2 Components within an MPFlex endpoint. 60

4.3 Single file download over MPTCP and MPFlex (best SPTCP results shown only for small downloads). 66

4.4 Transfer many short flows over MPTCP and MPFlex. 67

4.5 Fetch web pages over MPTCP and MPFlex. 68

4.6 Case study: MPTCP applies multipath to all traffic, while MPFlex does that selectively based on user policy. 69

4.7 Performance of MinRTT vs. TxDelay scheduler when the RTT difference between the two paths is large (20ms vs. 70ms). 70

4.8 Performance impact of the MPFlex proxy location. 71

5.1 Compare chunk download time (M: MinRTT, O: Optimal). 79

5.2 Compare subflow completion time on receiver (M: MinRTT, O: Optimal). 79

5.3 Key design decisions of DEMS. 81

5.4 Achieve simultaneous subflow completion (sender-side view). 84

5.5 Choose a subflow with an earlier estimated data arrival time (sender-side view). 85

5.6 Impact of inaccurate ΔOWD estimation. 86

5.7 A simple reinjection scheme. 87

5.8 The adaptive reinjection scheme. All OWD values are exaggerated in the plot for illustration purpose. 89

5.9 System diagram of DEMS. 92

5.10 Compare performance between DEMS and MinRTT on downloading files with different sizes (laptop, emulation). The WiFi and LTE bandwidth are 7040kbps and 9185kbps respectively. 98

5.11 Compare redundant data between DEMS-S and DEMS-F (laptop, emulation). 99

5.12 Compare subflow completion time difference (laptop, emulation, WiFi RTT 50ms, LTE RTT 270ms). 99

5.13 Download time reduction brought by DEMS-F compared to MinRTT under 36 bandwidth combinations (laptop, emulation, WiFi RTT 50ms, LTE RTT 70ms). 101

5.14 Compare different scheduling algorithms under varying network conditions (laptop, trace-driven emulation). 102

5.15 Compare performance of different scheduling algorithms (smartphone, real WiFi/LTE at five locations). 103

5.16 Downlink throughput of real WiFi and LTE at five locations. 104

5.17 Relative OWD and prediction error of real WiFi and LTE networks at five locations. 104

5.18 Compare performance among DEMS-F and four other schedulers (laptop, emulation). 105

5.19 Compare web page load time when using DEMS-F and MinRTT (smartphone, real WiFi/LTE). 107

5.20 Example waterfall diagrams for MinRTT and DEMS-F (some objects are omitted for better illustration). 108

6.1 Wearable measurement testbed. 115

6.2 Impact of BT-WiFi handover on the QoE of tinyCam security camera app on Huawei Watch. 117

6.3 Breakdown of handover delay. 119

6.4 BT download throughput under different signal strength. The zero throughput of 5th percentile for BT RSSI above -90dBm is due to BT sniff mode. 121

6.5 Download throughput of RTApp when using BT under different BT discovery settings. 123

6.6 Download throughput of RTApp when using WiFi under different BT discovery settings. 123

6.7 Comparison of network interruption time during handover under different handover strategies. 124

LIST OF TABLES

Table

1.1 Parallelism in mobile networking. 2

3.1 Impact of upload on download performance on different devices, vendors, and networks (C1, C2, and C3 refer to Carrier 1, 2, and 3 respectively). . 34

3.2 Summary of solutions for reducing queuing delay for upload traffic. "(✓)" means only limited support. 38

4.1 Comparison of three multipath proxy solutions. 59

4.2 Implementation overhead of different MPFlex plug-ins. 63

ABSTRACT

Smart device usage has witnessed rapid growth in the recent years, fueled by new mobile applications, faster mobile networks, and different form factors. There are three levels of parallelism in this growth that increase the diversity of mobile networking: (1) flow-level parallelism from a mix of both download and upload from multiple applications on the same device, (2) interface-level parallelism with which a mobile device can simultaneously use multiple network paths such as cellular and WiFi to accelerate data transfer, and (3) device-level parallelism where network communication on emerging wearable devices usually involves multiple smart devices for forwarding the network traffic. Despite seemingly bringing benefit to mobile applications at first glance, these levels of parallelism incur rather complex interactions with applications due to diverse traffic patterns and QoE requirements, potentially leading to severely degraded application QoE. My dissertation focuses on addressing this challenge, aiming at designing a networking stack on mobile systems that can efficiently use diverse network resources to improve application QoE without modifying the applications, based on the better understanding of the complex interactions between three levels of parallelism and applications.

Specifically, to understand the interference from flow-level parallelism, we conduct a comprehensive characterization of cellular upload traffic and investigate the on-device bufferbloat frequently incurred by uploads under diverse types of cellular networks on mobile devices. To mitigate this problem, we propose a system called QCUT to control the cellular firmware buffer, which incurs excessive queuing, from the OS kernel. To better leverage the interface-level parallelism, we propose a flexible software architecture called

MPFlex for multipath over mobile networks, which strategically employs multiplexing to improve multipath performance. Based on this flexible multipath architecture, we propose DEMS, a new multipath scheduler aiming at reducing the data chunk download time. DEMS is robust to diverse network conditions and brings significant performance boost compared to the default MPTCP scheduler. We also investigate the networking performance under mobility on a popular smartwatch OS, Android Wear, to understand the multi-device networking under device-level parallelism. Motivated by the finding that passive network switching and insufficient protocol support lead to undesirable handover performance, we further propose WearMan, a wearable network manager to switch networks proactively and provide seamless handover performance under mobility.

CHAPTER I

Introduction

Smart device usage has witnessed rapid growth, fueled by new mobile applications, faster mobile networks, and different form factors. Mobile network connection speeds grew more than 3-fold in 2016, and there were 325 million wearable devices in the same year, contributing to 63% growth of mobile data traffic over that period [4]. We observe three levels of parallelism in this growth that increase the diversity of mobile networking, summarized in Table 1.1:

- **Flow-level parallelism –** The mobile traffic paradigm is undergoing a shift from being dominated by download to a mix of both download and upload from multiple applications on the same device. The increasing upload is due to a wide range of emerging apps enabling user-generated traffic such as media content upload to social networks (*e.g.,* Facebook videos), cloud-based offloading, HD live video streaming, machine-to-machine (M2M) communication, *etc.* The prevalence of upload is further fueled by increasingly ubiquitous access to LTE networks, which provides uplink bandwidth of up to 20Mbps [78]. Besides, the frequent use and rich function of mobile devices give rise to multi-tasking, which enables users to interact with multiple applications at the same time (*a.k.a.* "multi-window"). Even without multitasking, a single foreground app can still trigger multiple concurrent TCP flows. A recent study [78] shows that around 28% of the time for each mobile user there are

Level of parallelism	**Definition**	**Example**	**Challenge**
Flow-level parallelism	Multiple TCP flows of download and upload transfer data on mobile networks at the same time.	One TCP flow downloads web pages for web browsing while another TCP flow uploads user-generated videos in the background.	The on-device queuing is sometimes outside the direct control of mobile OS, which makes reducing cross-flow interference difficult.
Interface-level parallelism	A mobile device simultaneously uses multiple network interfaces, *i.e.,* multipath, to download and upload data.	A smartphone uses both WiFi and LTE network at the same time to stream 1080p YouTube videos.	Flexible configuration of multipath usage and efficient multipath scheduler capable of adapting to different network scenarios and application requirements are not well supported in current MPTCP implementation.
Device-level parallelism	Multiple devices share all the network resources they have access to, in which one device depends on another device to forward network traffic.	A smartwatch runs a stand-alone app on its CPU and memory while leveraging paired smartphone's WiFi network through Bluetooth connection for accessing the Internet.	Multiple devices introduce new ways of utilizing diverse networks and increase the diversity of device configuration and application requirements, making network selection complex for different applications.

Table 1.1: Parallelism in mobile networking.

concurrent TCP flows.

- **Interface-level parallelism** – Simultaneously using multiple network paths such as cellular and WiFi to accelerate data transfer is an attractive feature on mobile devices. It is supported by many commercial products such as Apple Siri [12], Gigapath by Korean Telecom [31], and Samsung Download Booster [19]. Researchers have also devised multipath strategies for applications such as file download [84, 104], web browsing [68, 69], and video streaming [52, 70, 133]. Currently, the most widely used multipath solution is MPTCP [58], which enables unmodified applications to leverage multipath by adding a shim layer to the TCP interface. MPTCP establishes a *subflow* over each network path. The MPTCP sender distributes the data onto the subflows; the receiver reassembles the data into the original byte stream and delivers it to the app transparently.

- **Device-level parallelism** – Network communication on emerging wearable and IoT devices usually involves both the smart device and another networked device for forwarding the network traffic. Both categories of devices can handle computation and network communication independently while sharing all the network resources they have. For example, a smartwatch is capable of pairing with another smartphone using Bluetooth to leverage the phone's WiFi or cellular networks, while running stand-alone watch apps. This is due to constrained resources on wearable and IoT devices, *e.g.,* limited battery capacity, close-range wireless communication, *etc.* The smartwatch can also choose to use WiFi or even cellular networks directly if the phone is out of range. Beside network communication, computation on wearable and IoT devices such as data processing can also be offloaded to the phone or back-end servers. It aims at improving the app performance of smart devices by leveraging all sources of computation, not limited to single-device execution [62].

While seemingly bringing benefit to mobile applications at first glance, different levels

of parallelism require proper management in mobile networks and otherwise degrade application QoE if they are blindly used. Flow-level parallelism requires proper queuing management to prevent severe cross-flow interference. This is difficult as the on-device queuing is sometimes outside the direct control of mobile OS. Interface-level parallelism requires flexible configuration of multipath usage and efficient multipath scheduler to adapt to different network scenarios and application requirements, which are missing in current MPTCP implementation. Device-level parallelism requires proper coordination among multiple devices that share all the network resources. This is because multiple devices introduce new ways of utilizing diverse networks and increase the diversity of device configuration and application requirements. Coordinating network utilization among multiple devices is even more complicated than managing single-device multipath.

In reality, *the flow-level, interface-level, and device-level parallelism incur rather complex interactions with applications due to their diverse traffic patterns and QoE metrics*. Thus, application QoE can be severely degraded under such diverse mobile networking environment. We find that concurrent upload in cellular networks incurs significant user experience degradation on real applications, including web browsing (219% to 607% increase of page load time with concurrent upload), and video streaming (57% reduction of playback bitrate and numerous stalls), due to cross-traffic interference on device [63]. Based on our study, the widely-used multipath solution, MPTCP, provides suboptimal performance due to poor interaction with short flows, a lack of infrastructure support for multipath policy, and inefficient scheduling algorithm (up to 7.5x download time increase, 49% median increase compared to optimal scheduling in real-world networks) [64, 104]. Even worse, we observe that the handover between Bluetooth and WiFi on smartwatches takes a long time (more than 40 seconds) and interrupts network communication, when the watch moves out of the phone's Bluetooth coverage, due to poor network selection on two devices [95].

My dissertation is to address this challenge for mobile applications, demonstrating that **with a better understanding of the complex interaction between the flow-level,**

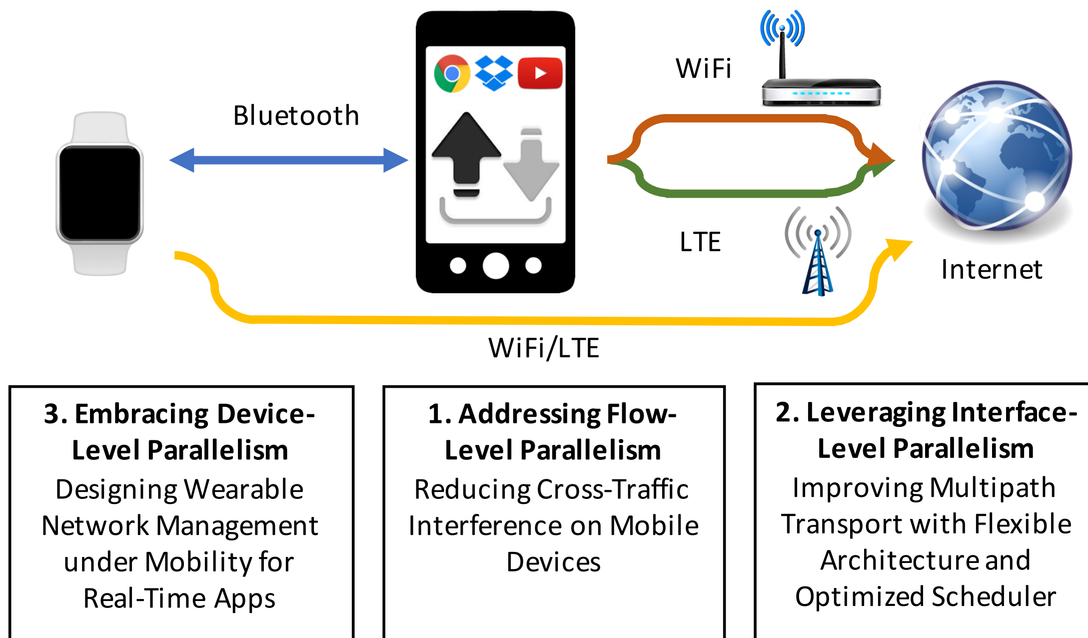


Figure 1.1: Dissertation organization.

interface-level, device-level parallelism and applications, the networking stack on mobile systems can efficiently use diverse network resources to improve application QoE without modifying the applications. We design an application-transparent networking stack that improves QoE of mobile applications by addressing complex interactions with applications incurred by flow-level, interface-level, and device-level parallelism paradigms. As shown in Figure 1.1, I elaborate the contributions of my dissertation in the following three sections.

1.1 *Addressing Flow-Level Parallelism*: Reducing Cross-Traffic Interference on Mobile Devices

Understanding the impact of flow-level parallelism on cellular traffic paradigm is the first necessary step towards addressing the potentially complex interaction between them. To achieve this goal, we conduct to our knowledge the first comprehensive, quantitative,

and cross-layer measurement study of cellular upload traffic and its interaction with concurrent traffic, using combined approaches of analyzing large network traces and conducting controlled lab experiments [63].

We characterize the cellular upload traffic by measuring its volume, duration, rate, concurrency, and impact on latency from a large network trace collected from an IRB-approved user study[1] involving 15 users for 33 months. We find that although the majority of today's smartphone traffic is still download, the upload traffic can be significant. We also find that large upload tends to have higher RTT, and upload traffic may also increase the RTT experienced by concurrent download.

For cellular upload, we find a significant fraction of the end-to-end latency occurs on the end-host device instead of in the network, due to the large buffer inside the mobile device. Specifically, we make two key observations. First, our findings suggest that bufferbloat for upload traffic can frequently occur on mobile devices accessing diverse types of networks (HSPA, LTE, and even Wi-Fi), across different devices. On-device bufferbloat can cause significant latency increase up to 4 seconds, or 100x of the network RTT, on off-the-shelf Android devices. This implies that when an upload is in progress, the on-device queuing delay, in fact, eclipses the network delay. We quantitatively demonstrate that concurrent upload incurs significant user experience degradation on real applications, including web browsing (219% to 607% increase of page load time with concurrent upload), video streaming (57% reduction of playback bitrate and numerous stalls), and VoIP.

Second, we identify the root cause of such excessive on-device queuing. It can happen at different layers including application buffer, OS TCP buffer, and the Queuing Discipline (Qdisc) in the OS kernel. In particular, we find excessive queuing also frequently happens at the *cellular firmware buffer*, whose occupancy can, for example, reach up to 300KB while accounting for 49% of the end-to-end delay when the uplink bandwidth is 2Mbps.

[1]This study was conducted entirely with data collected from active and passive measurements at the University of Michigan and was approved by the University of Michigan IRB (Institutional Review Board) approval number HUM00111075.

More importantly, the cellular firmware buffer distinguishes itself from other on-device buffers in that its occupancy plays a role in the cellular control plane which in turn affects base station's scheduling decision and achievable uplink throughput. Mobile OS does not usually have direct control over the firmware buffer, making it difficult to carry out queuing management.

Due to the uniqueness of on-device bufferbloat, we find that existing mitigation solutions, such as changing TCP congestion control, tuning TCP buffer size, reducing Qdisc sizes [23], prioritizing delay-sensitive traffic, and applying Active Queue Management (*e.g.,* CoDel [101] and PIE [109]), are not capable of effectively mitigating the excessive buffering. This is because *(i)* they cannot be directly implemented in the cellular device driver, and *(ii)* they are unaware of the interplay between the driver buffer and the cellular control plane.

Therefore, we design and implement a new solution called QCUT to control the firmware buffer occupancy from the OS kernel [63]. QCUT is general (independent of a particular driver implementation), lightweight, and effective. Our lab experiments show that QCUT effectively reduces the cellular firmware queuing delay by more than 96% while incurring little degradation of uplink throughput. We deploy QCUT on a user study involving 5 users for one week. The results indicate QCUT significantly improves the application QoE when concurrent upload is present.

1.2 *Leveraging Interface-Level Parallelism*: Improving Multipath Transport with Flexible Architecture and Optimized Scheduler

On mobile devices, besides flow-level parallelism, simultaneously using multiple network paths, *i.e.,* interface-level parallelism, has become increasingly popular. A well-known multipath solution, MPTCP, promises to bring better network performance to mobile devices. However, existing work found out that MPTCP suffers from a few limitations:

poor interaction with short/small flows, a lack of infrastructural support for multipath policy, and MPTCP extension often being blocked by middleboxes [48, 54, 104]. As a first step of improving multipath transport on mobile devices, we propose a flexible software architecture of mobile multipath called MPFlex that overcomes all the above limitations [104].

MPFlex has several prominent features. First, it performs transparent multiplexing for application traffic over multipath. Our multiplexing scheme reduces the number of handshakes from many (one per path) to zero, leading to significant improvement of bandwidth utilization for small flows. Second, MPFlex decouples the high-level scheduling algorithm and the low-level OS protocol implementation. Such a framework dramatically simplifies the development, deployment, and maintenance of multipath features. Third, MPFlex has visibility of all traffic on an end host, and thus provides an ideal vantage point for applying user-specified multipath policies. Fourth, MPFlex is middlebox-friendly as it does not use any Layer 3 or 4 protocol extensions which may be blocked by ISPs.

Compared to MPTCP, MPFlex reduces single file transfer time by up to 49%, improves bundled short flows' transfer time by up to 63%, and boosts real web page load speed by up to 20%, while incurring negligible overhead. We also demonstrate MPFlex's capability of flexibly plugging-in new features such as buffer-aware scheduling, smart packet reinjection, and per-application policies, which can be implemented in less than 70 lines of user-level code.

Besides designing a flexible multipath architecture for mobile devices, we also look into an important component of MPTCP, the schedulers. A multipath scheduler takes packets from applications and determines on which subflow to transmit each packet. The default scheduler, MinRTT, attempts to deliver the data as soon as possible by choosing a subflow with the smallest RTT unless its congestion window is full. We found that MinRTT does not achieve the optimal performance of downloading data chunks due to different subflow completion time at the receiver side. Based on this insight, we design, implement, and eval-

uate DEMS (**DE**coupled **M**ultipath **S**cheduler[2]), a new multipath packet scheduler aiming at reducing the data chunk download time over multipath [64]. The key idea behind DEMS is to achieve simultaneous subflow completion at the receiver side through strategic packet scheduling over decoupled subflows in order to minimize the chunk download time.

The key design decisions of DEMS include the following: (1) DEMS leverages a heuristic that treats all data in the meta buffer as a chunk, and it strategically decouples the paths for chunk delivery; (2) DEMS ensures simultaneous subflow completion at the receiver side by carefully introducing a timing offset at the sender side; (3) DEMS allows a path to trade a small amount of redundant data for performance. We implement DEMS on smartphones and evaluate it over both emulated and real cellular/WiFi networks. DEMS is robust to diverse network conditions and brings significant performance boost compared to the default MPTCP scheduler (*e.g.,* median download time reduction of 33%–48% for fetching files and median loading time reduction of 6%–43% for fetching web pages), and even more benefits compared to other state-of-the-art schedulers.

1.3 *Embracing Device-level Parallelism*: Designing Wearable Network Management under Mobility for Real-Time Apps

Wearable devices are becoming popular and common in addition to smartphones in our daily lives. Smartwatches have become the most popular smart wearables in the market. According to IDC, by volume, smartwatches account for the largest part of smart wearables and are expected to reach a total value of $17.8 billion dollars in 2020 [28]. Being network-connected devices, smartwatches promise to keep users connected with a wide range of features and apps such as receiving notifications, taking phone calls, monitoring fitness, remotely controlling the paired smartphone to take photos, *etc.* Wearable networking is different from, for example, smartphone networking that has been well studied in the past

[2]Note here "decoupled scheduling" is different from the decoupled congestion control in MPTCP.

decade. Due to the short range of Bluetooth (BT), network handover frequently occurs on a wearable: when it moves away from the phone, the BT session will be torn down, and the wearable has to use standalone WiFi or LTE to communicate with the external world.

Despite the promise, there lacks a thorough understanding of how smartwatch networking stack performs for smartwatch applications, especially under mobility. Motivated by this, we conduct to our knowledge a first investigation of the networking performance of Android Wear under mobility. Android Wear is one of the most popular OSes for wearables, which is a version of Android OS tailored to small-screen wearable devices. Our key finding is that, surprisingly, there exist serious performance issues under mobility regarding aforementioned aspects that distinguish wearable networking from smartphone networking. Due to passive network switching and insufficient protocol support for handovers, the BT-WiFi handover that frequently occurs on wearables may last more than 40 seconds, leading to significant disruption of application performance. To solve this problem, we propose a proactive handover approach, WearMan (**Wear**able Network **Man**ager), by performing preemptive handovers using the changes of BT signal strength as an indicator that tells in advance when a handover needs to be performed. WearMan ensures almost seamless handovers between BT and WiFi for our custom Android Wear app by strategically enabling and switching networks.

1.4 Thesis Organization

This dissertation is structured as follows. Chapter II provides sufficient background of multipath and Android Wear OS. In Chapter III, we describe our comprehensive characterization of cellular upload traffic and investigation of its interaction with other concurrent traffic. Motivated by the observations from this measurement study, we further systematically study a wide range of solutions for mitigating on-device bufferbloat and propose a system called QCUT to control the firmware buffer occupancy from the OS kernel. Then in Chapter IV, we move on from addressing flow-level parallelism to leveraging interface-

level parallelism on mobile networks, presenting a flexible software architecture for mobile multipath called MPFlex, which strategically employs multiplexing to improve multipath performance. Based on this flexible multipath architecture, we propose DEMS, a novel multipath scheduler aiming at reducing the data chunk download time in Chapter V. Next, in Chapter VI, we conduct the first in-depth investigation of the networking performance under mobility on Android Wear, one of the most popular OSes for wearables, and propose WearMan, a wearable network manager to switch networks proactively and provide seamless handover performance under mobility. We discuss related work in Chapter VII before concluding the thesis in Chapter VIII.

CHAPTER II

Background

This chapter provides sufficient background of multipath and Android Wear OS.

2.1 Multipath Networking

We first give a brief overview of the multipath over mobile networks and then describe multipath scheduler, the key component in multipath transport.

2.1.1 Multipath over Mobile Networks

Simultaneously using multiple network paths such as cellular and WiFi to accelerate data transfer is an attractive feature on mobile devices. It is supported by many commercial products such as Apple Siri [12], Gigapath by Korean Telecom [31], and Samsung Download Booster [19]. Researchers have also devised multipath strategies for applications such as file download [84, 104], web browsing [68, 69], and video streaming [52, 70, 133]. Currently the most widely used multipath solution is MPTCP [58], which enables unmodified applications to leverage multipath by adding a shim layer to the TCP interface. MPTCP establishes a *subflow* over each network path. The MPTCP sender distributes the data onto the subflows; the receiver reassembles the data into the original byte stream and delivers it to the app transparently.

2.1.2 Multipath TCP and its Schedulers

Multipath TCP (MPTCP [58]) enables simultaneous usage of multiple network paths (*a.k.a.* subflows). In the remainder of this dissertation, we primarily focus on two paths, as they correspond to the most common mobile multipath usage scenarios: jointly using WiFi and cellular on a smartphone or WiFi and Bluetooth on a wearable.

MPTCP has a complex transport protocol stack consisting of several components: subflow management, packet scheduling, congestion control, flow control, *etc.*, among which we focus on optimizing the *schedulers*. A multipath scheduler takes packets from applications (stored in the "meta buffer") and determines on which subflow to transmit each packet. MPTCP currently supports three schedulers: round-robin, ReMP (sending the same data to all subflows for better reliability), and MinRTT. Among them, MinRTT is the default scheduler aiming at reducing the overall data transfer time. As long as the congestion window allows, MinRTT favors the subflow with the smallest RTT so that the packet can be delivered as soon as possible. The MinRTT scheduler is simple and robust, and has registered wide usage in practical systems [12, 31].

2.2 Android Wear OS

Wearable devices are becoming popular and common in addition to smartphones in our daily lives. Smartwatches have become the most popular smart wearables in the market. Android Wear is one of the most popular OSes for wearables, which is a version of Android OS tailored to small-screen wearable devices. It is used by a wide range of smartwatches and potentially other wearables. The latest Android Wear 2.0 provides multiple networks for applications to access the Internet.

When close to the smartphone and within its Bluetooth (BT) coverage, the smartwatch can connect to the smartphone with a BT connection and use it as a "gateway" to access the Internet. In Android Wear, this transmission mechanism is called `COMPANION_PROXY`

mode. Some latest smartwatches also have standalone WiFi or cellular networks, with which the smartwatch can directly access the Internet without proxying traffic through a smartphone. To preserve the energy of network transmission, Android Wear proxies all application traffic through the smartphone if the BT connection is available, regardless of the connectivity of standalone WiFi, which is much more energy consuming. From a recent smartwatch user study, such a gateway mode accounts for 84% of the day time usage period [95].

To provide easy access to different networks including BT, WiFi, and cellular on the smartwatch, the latest Android Wear 2.0 OS provides a set of APIs for Wear applications to use active network interfaces and automatically handles transitions between networks [34]. Specifically, the `ConnectivityManager` [32] on Android Wear provides the information of the available and active networks. Applications can request access to an appropriate network based on its network requirement.

CHAPTER III

Understanding and Solving On-Device Bufferbloat in Cellular Networks

In this chapter, we start by understanding the impact of flow-level parallelism on cellular traffic paradigm. We provide an in-depth understanding of cross-flow interference from on-device bufferbloat, a performance problem we identify that is unique in cellular networks. As to be shown, on-device bufferbloat has an adverse impact on application QoE. To address this problem, we design, implement, and evaluate QCUT in the networking stack, a firmware-independent on-device queueing management system.

3.1 Introduction

The explosive growth of mobile devices and cellular networks shows no sign of slowing down. We notice two important trends not well explored in previous work, namely *user-generated traffic* and *multi-tasking*.

On one hand, the mobile traffic paradigm is undergoing a shift from being dominated by download to a mix of both download and upload, due to a wide range of emerging apps enabling user-generated traffic such as media content upload to social networks (*e.g.,* Facebook videos), background synchronization, cloud-based offloading, HD video chat, machine-to-machine (M2M), and device-to-device (D2D) communication, *etc.* The preva-

lence of upload is further fueled by increasingly ubiquitous access to LTE networks, which provides uplink bandwidth of up to 20Mbps.

On the other hand, the frequent use and rich function of mobile devices give rise to multi-tasking, which enables users to interact with multiple applications at the same time. Previously, due to the limitation of mobile operating system and the device processing power, older phones and Android systems only support a single application at foreground interacting with the user. Newer phones allow users to use multiple apps simultaneously (*a.k.a.* "multi-window"). Even without multi-tasking, a single foreground app can still trigger multiple concurrent TCP flows. A recent study [78] shows that around 28% of the time for each mobile user there are concurrent TCP flows.

Motivated by the above, in this chpater, we conduct to our knowledge the first comprehensive, quantitative, and cross-layer measurement study of cellular *upload* traffic and its interaction with concurrent traffic, using combined approaches of analyzing large network traces and conducting controlled lab experiments. Our contributions consist of the following.

Characterization of upload traffic (§3.3). We characterized the cellular upload traffic by measuring its volume, duration, rate, concurrency, and impact on latency from a large network trace collected from an IRB-approved user study[1] involving 15 users for 33 months. We found that although the majority of today's smartphone traffic is still download, the upload traffic can be significant. Upload can last for up to 11 minutes with 226 MB of data transferred. In particular, the upload speed can achieve up to 12.8Mbps (median 2.2Mbps for 10MB+ flows) in today's LTE networks, facilitating many applications that upload rich user-generated traffic. We also found that large upload tends to have higher RTT, and upload traffic may also increase the RTT experienced by concurrent download.

An anatomy of on-device queuing for upload traffic (§3.4.1-§3.4.3). For cellular up-

[1]This study was conducted entirely with data collected from active and passive measurements at the University of Michigan and was approved by the University of Michigan IRB (Institutional Review Board) approval number HUM00111075.

load, we found a significant fraction of the end-to-end latency occurs on the end-host device instead of in the network, due to the large buffer inside the mobile device. Specifically, we made two key observations. First, contrary to the common understanding [82, 125] that *(i)* excessive queuing delay (*a.k.a.* "*bufferbloat*") happens mostly inside the network (or near-edge network elements), and *(ii)* cellular upload traffic is less likely to incur queuing delay due to its low data rate, our findings suggest that *bufferbloat for upload traffic can frequently occur on mobile devices accessing diverse types of networks (HSPA, LTE, and even Wi-Fi), across different devices.* On-device bufferbloat can cause significant latency increase up to 4 seconds, or 100x of the network RTT, on off-the-shelf Android devices. This implies that when an upload is in progress, *the on-device queuing delay in fact eclipses the network delay.*

Second, we identified the root cause of such excessive on-device queuing. It can happen at different layers including application buffer, OS TCP buffer, and the Queuing Discipline (Qdisc) in the OS kernel. In particular, we found excessive queuing also frequently happens at the *cellular firmware buffer*, whose occupancy can, for example, reach up to 300KB while accounting for 49% of the end-to-end delay when the uplink bandwidth is 2Mbps. More importantly, the cellular firmware buffer distinguishes itself from other on-device buffers in that *its occupancy plays a role in the cellular control plane which in turn affects base station's scheduling decision and achievable uplink throughput.*

Accurate achievable uplink throughput estimation (§3.4.4). The excessive buffering at various layers makes accurate estimation of achievable cellular uplink throughput challenging. We found that surprisingly, using the same algorithm, the throughput estimated at upper layers (TCP and Qdisc) deviates from the lower-layer throughput estimation by 136% and 70% on average. We proposed a method that accurately infers the uplink throughput by leveraging lower-layer information from cellular control-plane messages.

Quantifying the impact of uplink bufferbloat (§3.5). We illustrated that large upload traffic significantly affects the performance of concurrent TCP download, whose average

RTT is increased by 91% and average throughput is reduced by 66%. We found *such severe performance degradation is predominantly caused by on-device buffers*, because the uplink ACK stream of TCP download shares the same Qdisc and cellular firmware buffers with concurrent upload, and is thus delayed mainly due to the on-device bufferbloat. We further quantitatively demonstrated that concurrent upload incurs significant user experience degradation on real applications, including web browsing (219% to 607% increase of page load time with concurrent upload), video streaming (57% reduction of playback bitrate and frequent stalls), and VoIP.

Mitigating on-device bufferbloat (§3.6,§3.7). Due to the uniqueness of on-device bufferbloat, we found existing mitigation solutions, such as changing TCP congestion control, tuning TCP buffer size, reducing Qdisc sizes [23], prioritizing delay-sensitive traffic, and applying Active Queue Management (*e.g.,* CoDel [101] and PIE [109]) are not capable of effectively mitigating the excessive buffering. This is because *(i)* they cannot be directly implemented at the cellular device driver, and *(ii)* they are unaware of the interplay between the driver buffer and the cellular control plane. We therefore design and implement a new solution called QCUT to control the firmware buffer occupancy from the OS kernel. QCUT is general (independent of a particular driver implementation), lightweight, and effective. Our lab experiments show that QCUT effectively reduces the cellular firmware queuing delay by more than 96% while incurring little degradation of uplink throughput. We deploy QCUT on a user study involving 5 users for one week. The results indicate that QCUT significantly improves the application QoE when concurrent upload is present.

Although we identified the on-device bufferbloat problem in today's HSPA/LTE networks, we anticipate it continues to affect future wireless technologies, whose uplink bandwidth will remain below the downlink bandwidth (*e.g.,* 50Mbps vs. 150Mbps for LTE Advanced [11]) causing the cellular uplink to often remain the end-to-end bottleneck link. More importantly, higher network speed and cheaper memory facilitate device vendors and cellular carriers to deploy larger buffers that exacerbate on-device (and in-network)

bufferbloat.

Chapter Organization. After describing the experimental methodology in §3.2, we conduct a measurement study of today's upload traffic in §3.3. We reveal the on-device queuing problem in §3.4, and quantify the impact of upload on mobile apps in §3.5. We then describe how QCUT mitigates on-device queuing in §3.6 and evaluate QCUT as well as existing mitigation strategies in §3.7. We conclude this chapter in §3.9.

3.2 Experimental Methodology

To comprehensively study cellular upload traffic, we carried out controlled lab experiments (§3.2.1) and analyzed data collected from a user study with 15 participants (§3.2.2).

3.2.1 Controlled Local Experiments

We conduct controlled experiments using off-the-shelf smartphones and commercial cellular networks. Our devices consist of the following: *(i)* Samsung Galaxy S3 running Android 4.4.4 with Linux kernel version 3.4.104, using Carrier 1's LTE network[2], *(ii)* Samsung Galaxy S4 running Android 4.2.2 with Linux kernel version 3.4.0, using Carrier 1's LTE network, *(iii)* Samsung Galaxy S3 running Android 4.0.4 with Linux kernel version 3.0.8, with access to Carrier 2's 3G network and *(iv)* Nexus 5 running Android 6.0.1 with Linux kernel version 3.4, using Carrier 1's LTE network. We also set up dedicated servers located at the University of Michigan with 64-core 2.6GHz CPU, 128GB memory, 64-bit Ubuntu 14.04 OS for experiments. Both mobile phones and the servers use TCP CUBIC [66], the default TCP variant for Linux/Android, unless otherwise mentioned. We conducted the experiments during off-peak hours. For each setting, we repeat the experiment for at least 10 times and report the average metrics unless otherwise noted.

For TCP throughput and RTT measurement, the mobile device first establishes a TCP connection to one of the dedicated servers. Then this TCP connection is used to transfer

[2] We anonymized three large U.S. cellular carriers' names as Carrier 1, 2, and 3.

random data without interruption. For bidirectional data transfer (*i.e.,* simultaneous upload and download) experiments, the mobile device establishes two TCP connections to two servers, one for download and the other for upload to eliminate server-specific bottlenecks. To measure throughput, we ignore the first 10 seconds to skip the slow start period and calculate the throughput every 500ms from transferred data.

3.2.2 Network Traces from a User Study

We also leveraged network traces collected from an IRB-approved smartphone user study by distributing instrumented Samsung Galaxy S3 phones to 15 students. Each of the 15 participants was also given unlimited LTE data plan. The phones were instrumented with data collection software (with very low runtime overhead). It continuously runs in the background and collects full packet traces in `tcpdump` format including both headers and payload. We collected 900GB of data in total from January 2013 to October 2015. We used an idle timing gap of 1 minute to separate *user sessions* of the same device.

3.3 Upload Traffic Measurement

In this section, we perform a measurement study of upload traffic in today's cellular networks, using the traces collected from our user study (§3.2.2).

3.3.1 Traffic Volume

Figure 3.1 plots the distributions of download, upload, and overall traffic volume of user sessions. We only consider TCP/UDP payload size when computing the session size. Today's mobile traffic is dominated by download, whose average size is about one order of magnitude larger than that of upload. About 2.2% of user sessions carry more than 1MB of downlink bytes, while only 0.4% upload more than 1MB data in the user study trace. We notice a major source of user-consumed traffic is video, which accounts for about half of download traffic.

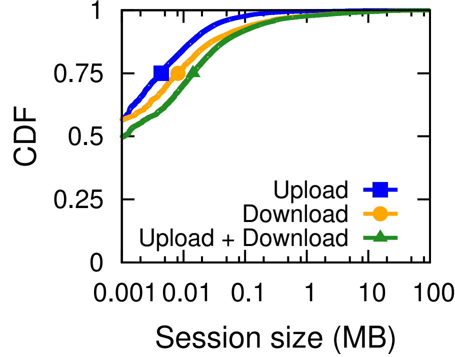
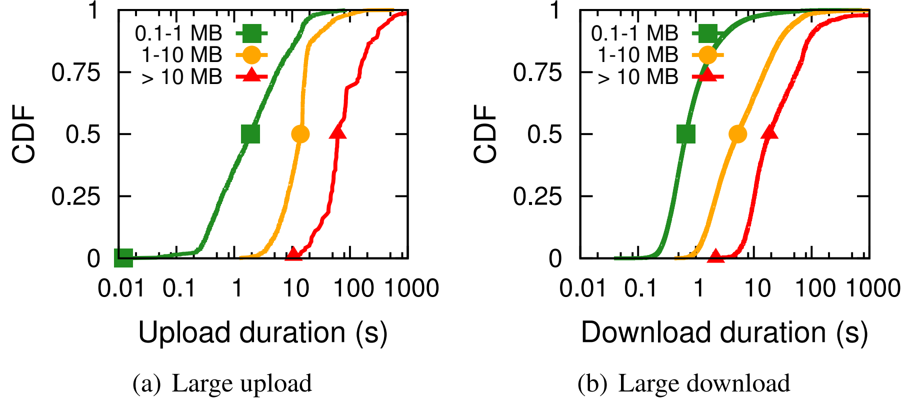


Figure 3.1: Traffic volume distribution of user sessions.

However, we observed that the fraction of upload bytes is indeed non-trivial. Within the top 20% of user sessions (in terms of their overall transferred bytes), the 25th, 50th, and 75th percentiles of the fractions of upload traffic are 9%, 20%, and 42%. Across all user sessions, the corresponding fractions are higher, *i.e.,* 19%, 50%, and 57%. The upload of one user session even lasts for 11 minutes with 226 MB of data transferred. We expect that in the future, the fraction of upload traffic will keep increasing because of the increasingly popular user-generated traffic. Compared to smartphones, wearable and IoT devices may incur even more upload traffic, due to their ubiquitous sensing capabilities.

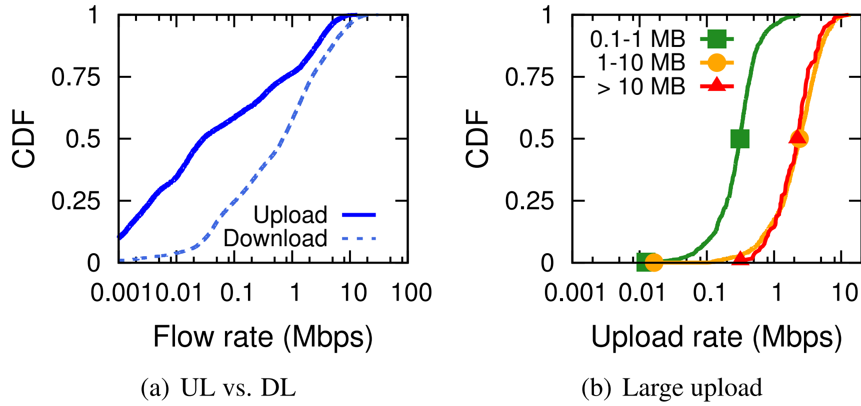
3.3.2 Flow Duration

We use inter-packet arrival time to divide a TCP flow into multiple segments with a threshold of 1 second. We also use the threshold of 1 second to eliminate idle period of a TCP flow. We then divide these segments into *upload bursts* that only have TCP payload in uplink, and *download bursts* with only TCP downlink payload, based on the direction of transferred TCP payload. Given a TCP flow, we define its *upload duration* as the total duration of all uplink bursts. Similarly, the *download duration* is the total duration of all downlink bursts. Figure 3.2(a) plots the upload durations for flows with large upload traffic volume (100KB to 1MB, 1MB to 10MB, and at least 10MB). As expected, larger flows tend to be longer in duration. For flows with large download traffic volume, as



(a) Large upload (b) Large download

Figure 3.2: Flow duration distributions.



(a) UL vs. DL (b) Large upload

Figure 3.3: Flow rate distributions.

shown in Figure 3.2(b), their download duration exhibits distributions qualitatively similar to those of upload duration, yet with the main difference being that the download duration is statistically shorter, largely due to the higher downlink bandwidth compared to the uplink bandwidth, as to be measured next.

3.3.3 Flow Rate

We compute the upload (download) rate of a TCP flow by dividing the total bytes of all upload (download) bursts by its upload (download) duration that is defined above. We only consider flows whose upload/download duration are longer than a threshold, which is empirically chosen to be 3 seconds, as the "rate" of a very short flow is not very meaningful. Figure 3.3(a) compares upload and download rates for the user study trace. Statistically,

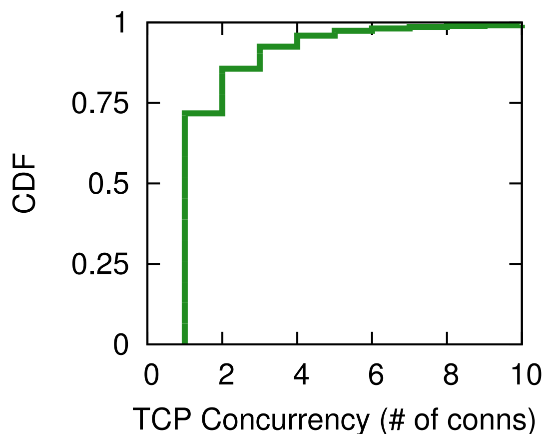
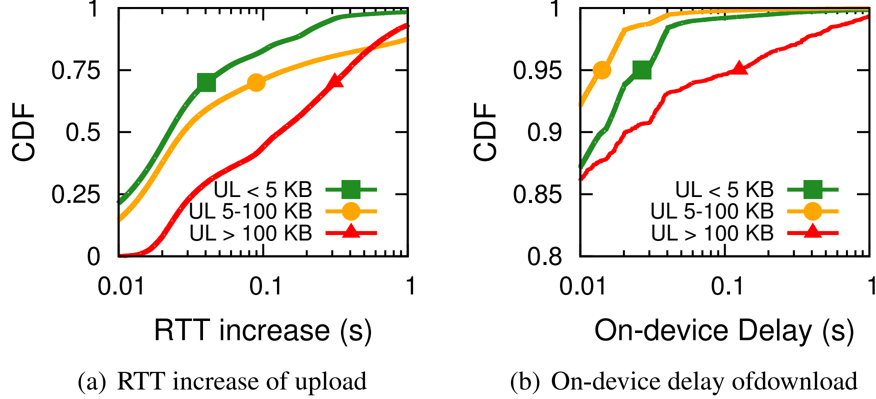


Figure 3.4: TCP concurrency distribution.

download is faster than upload, largely due to their differences in the underlying channel rates of the LTE radio access network. On the other hand, Figure 3.3(b) indicates larger upload flows (larger than 1MB) tend to have higher rates. In the user study dataset, for flows that upload 1 to 10 MB data, their 25%, 50%, and 75% percentiles of upload rates are about 1.4Mbps, 2.4Mbps, and 3.8Mbps, which are comparable or even higher than those of today's many residential broadband networks. For 10MB+ flows, the maximum achieved throughput are 12.8Mbps for the user study traces. Such high upload speed provides the infrastructural support for user-generated traffic.

3.3.4 Flow Concurrency

We explore the concurrency of TCP flows per user. The result is shown in Figure 3.4. For every one-second slot in each user session, we count the number of TCP flows that are transferring data. For the user study trace, for 28.2% of the time (*i.e.,* 28.2% of the one-second slots across all user sessions), there exist at least two TCP connections that perform either upload or download. The results indicate that concurrent TCP transfers are quite common on today's mobile devices. Motivated by this, we study the interplay between uplink and downlink traffic, a previously under-explored type of concurrency, in §3.5.



(a) RTT increase of upload (b) On-device delay ofdownload

Figure 3.5: Delay distributions.

3.3.5 RTT Dynamics

We next study the RTT dynamics of cellular upload from the user study trace. The RTT is measured by timing the timestamp difference between each uplink TCP data packet and its corresponding ACK packet captured by `tcpdump`. We then study the fluctuation of upload RTT using the following methodology. First, we split each user session into one-second slots and discard slots without uplink traffic. We also discard slots whose download traffic volume is non-trivial (using 5KB as a threshold). The purpose is to eliminate the impact of concurrent download on upload. Second, for each one-second slot s, we compute its *RTT increase* $I(s) = \text{mean}_p\{\text{RTT}(p) - \text{minRTT}(p.flow)\}$ over all data packets $\{p\}$ within the slot. $\text{RTT}(p)$ is the measured RTT, and $\text{minRTT}(p.flow)$ is the minimum RTT of upload stream of the TCP flow that p belongs to. Third, we plot in Figure 3.5(a) the distributions of $I(s)$ across all slots grouped by their upload size. As shown, the upload RTT is indeed highly "inflatable", and larger upload tends to incur much higher RTT. This resembles the "bufferbloat" effect that is well studied for download [82, 125], and motivates us to conduct a comprehensive investigation of bufferbloat for cellular upload in §3.4.

3.3.6 Impact of Upload on Download Latency

We are also interested in how upload impacts download latency, which is quantified as follows. We first generate one-second slots using a similar way as employed in Fig-

ure 3.5(a), but this time we only keep slots with *both* upload and download traffic. Then for every slot, we compute the average on-device delay of download. Note that since our user study traces were collected on client devices, we are only able to measure the *on-device* component of the download RTT *i.e.,* t_1 shown in Figure 3.6(b). Next, in Figure 3.5(b), we plot the distributions of the on-device download delay grouped by the size of per-slot upload size as is also done in Figure 3.5(a). We clearly observe that concurrent upload affects the on-device download delay because the ACK stream (for download) and the data stream (for upload) share several on-device buffers. We will conduct an in-depth investigation on this in §3.4 and §3.5.

3.3.7 Summary

Overall, we found that although the majority of today's smartphone traffic remains to be download, the upload traffic can still be large. In particular, the median upload speed is 2.2Mbps for 10MB+ flows and can achieve up to 12.8Mbps in today's LTE networks, enabling many applications to upload rich user-generated traffic. We also found that large upload tends to have higher RTT, and upload traffic may also increase the RTT experienced by concurrent download. Furthermore, it is quite common that multiple TCP flows are transferring data concurrently on a mobile device, leading to complex interactions possibly among uplink and downlink flows to be investigated soon.

3.4 On-device Queuing Delay of Upload Traffic

We conduct a thorough analysis of the latency characteristics for cellular upload traffic. We found a significant fraction of the latency happens on the end-host device instead of in the network (§3.4.1). In particular, in §3.4.2, we discover the root cause of large Qdisc and firmware buffers playing major roles in causing the excessive on-device delay, whose prevalence across devices and carriers are shown in §3.4.3. We also found in §3.4.4 that on-device queuing may significantly impact accurate uplink throughput estimation.

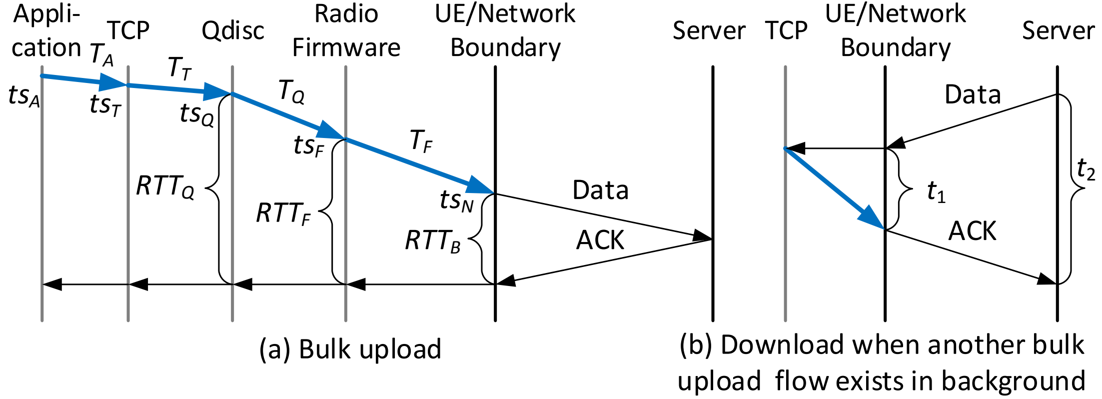


Figure 3.6: On-device bufferbloat (in thick blue).

3.4.1 Overall Delay Characterization

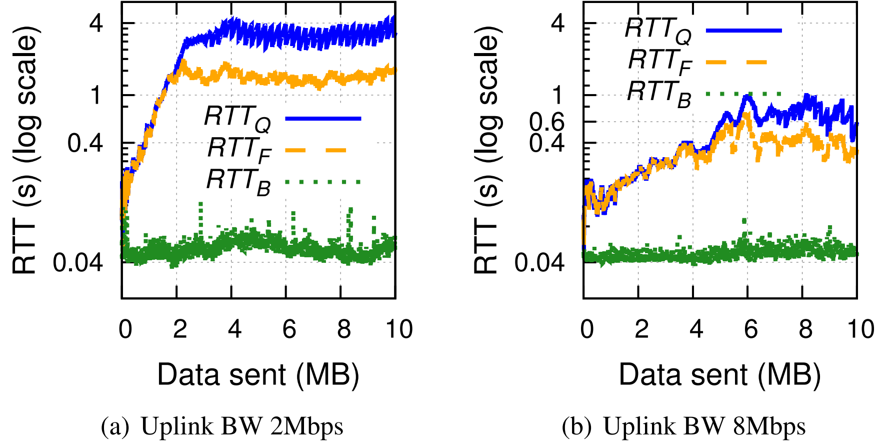
When a mobile device is uploading data, its packets will traverse various buffers in the protocol stack, as illustrated in Figure 3.6: TCP buffers, link-layer buffers (Linux queuing discipline), radio firmware buffers. Each buffer may incur queuing delay. As a result, we may get different RTT values if we conduct measurements at different layers. In this work, we focus on three RTT measurements defined bellow.

- RTT_B consists of only the delay a packet experiencing in the network. It does not include any delay caused by on-device buffer (*B* stands for "base").

- RTT_F includes RTT_B and the delay incurred by the buffer in the radio firmware, which usually resides on the cellular chipset of a mobile device (*F* stands for "firmware").

- RTT_Q includes RTT_F, plus the delay incurred by the queuing discipline (Qdisc), the link-layer buffer in the main memory managed by the OS (*Q* stands for "Qdisc").

Similarly, we can also define RTTs measured at higher layers (TCP, application). Nevertheless, RTT_B, RTT_F, and RTT_Q are our particular interests, because their corresponding network path or buffers are *shared* by multiple applications. As we will show in §3.5, if upload and delay sensitive traffic coexist, the former may severely interfere with the latter due to the shared nature of lower-layer buffers. In contrast, the higher-layer buffers are usually not shared.



(a) Uplink BW 2Mbps (b) Uplink BW 8Mbps

Figure 3.7: Overall latency characterization for a single TCP upload flow under two network conditions.

We now measure RTT_B, RTT_F, and RTT_Q by performing bulk data upload over TCP on Samsung Galaxy S3, using Carrier 1's LTE network. RTT_Q and RTT_F can be directly measured by `tcp_probe` [24] and `tcpdump`, respectively. RTT_B can only be indirectly estimated. For a given upload trace, we keep track of the buffer occupancy and enqueue/dequeue rates of the firmware buffer. We then use them to estimate the firmware queuing delay[3]. RTT_B is then calculated by subtracting RTT_F measured from `tcpdump` trace by the estimated queuing delay. The results of two representative experiments with different uplink bandwidth (2Mbps and 8Mbps, which are 50% and 98% percentiles of upload rates measured from Figure 3.3(b), respectively) are shown in Figure 3.7. Note the Y axes are in log scale.

As shown in both plots of Figure 3.7, at the beginning of the TCP upload, RTT_F increases steadily, and quickly outweighs RTT_B. When the uplink bandwidth is 2Mbps (8Mbps), after 2MB (4MB) of data has been sent out, RTT_F is increased to around 1.3s (330ms), which is much larger than RTT_B maintaining stably at around 50ms. Meanwhile, RTT_Q starts to exceed RTT_F, and becomes twice as large as RTT_F after another 2MB of

[3]The detailed methodology of firmware buffer occupancy estimation is described in §3.6.2. Since the radio firmware we use only reports firmware buffer occupancy of up to 150KB, we validate our methodology when the occupancy is smaller than 150KB and then use it to infer the buffer occupancy at any time in the trace.

data is uploaded. The absolute difference between RTT_Q and RTT_F is as high as 3s and 680ms in Figure 3.7(a) and 3.7(b), respectively.

Overall, we found that for cellular upload, surprisingly, the RTT observed by mobile devices' TCP stack (RTT_Q) can be significantly larger than the RTT perceived by `tcpdump` (RTT_F), which further far exceeds the pure network RTT (RTT_B). Depending on the uplink bandwidth, RTT_Q and RTT_F can be 22x$\sim$100x and 6x$\sim$24x of RTT_B, respectively, during the steady phase of TCP bulk upload. We call such a phenomenon *on-device bufferbloat* since it is caused by excessive queuing delay on the mobile device, as opposed to the network, which is regarded as the main source of excessive queuing for cellular downlink traffic [82]. As we will demonstrate later, on-device bufferbloat has deep implications on, for example, uplink bandwidth estimation, multi-tasking performance, uplink scheduling algorithms, and on-device buffer management.

3.4.2 Root Cause of On-device Queuing

We now explore the root cause of the excessive on-device queuing delay. We begin with an overview of how outgoing TCP packets on the sending path traverse the Linux kernel (also used by Android) and radio chipset. As shown in Figure 3.8, an application invokes the `send()` system call at time ts_A and the data is put into TCP buffer by kernel through TCP sockets at time ts_T. Note ts_T may be later than ts_A if the socket is blocked. In Linux, a packet is stored in a data structure called `skb`. A TCP packet is encapsulated into `skb` (with its TCP header being added) and sent to IP layer in `tcp_transmit_skb()` at time ts_Q. After being processed at the IP layer, the `skb` with TCP/IP header is subsequently injected to the queuing discipline (Qdisc) when `dev_queue_xmit()` is called. When the driver is ready to transmit more data, `dev_hard_start_xmit()` is called by the kernel to dequeue the packet from Qdisc to the driver buffer at time ts_D. Similarly, when the radio firmware of the chipset is ready to receive a packet from the driver, `ndo_start_xmit()` will be called to enqueue the packet to the firmware buffer at time ts_F. The kernel usually

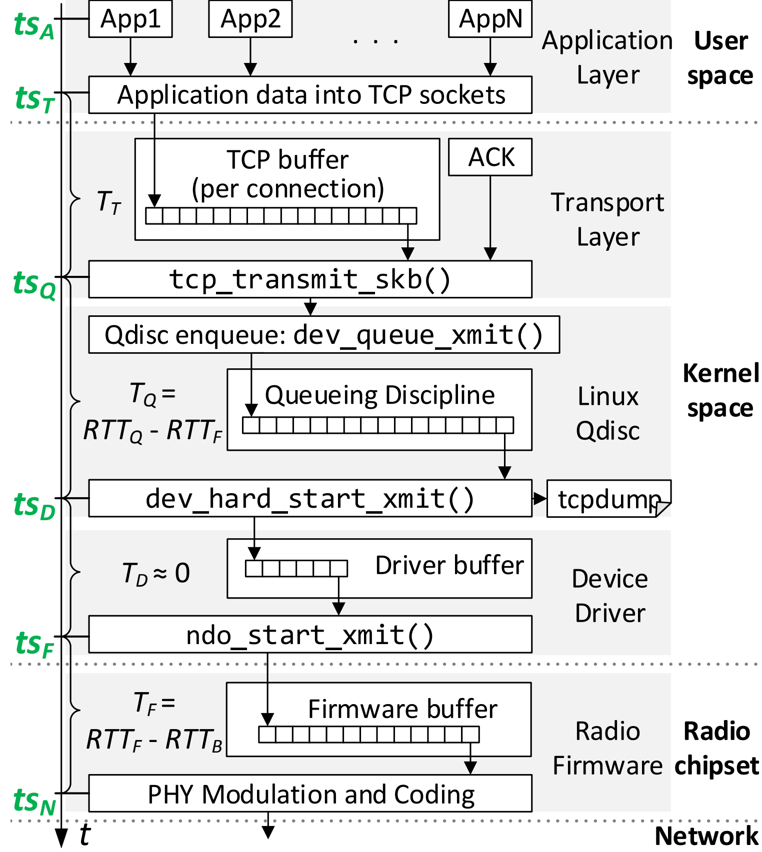


Figure 3.8: Packet processing and transmission on Android devices.

does not have direct control over the logic of radio firmware, which determines when to actually transmit the packet at time ts_N.

As mentioned in §3.4.1, in this study we focus on the queuing delay below the transport layer, as lower-layer buffers are usually shared, causing potential interference across multiple apps. We now describe lower-layer queuing in details.

In-kernel Queuing. To identify where on-device queuing occurs exactly in the kernel, we use Linux kernel debugging tool `jprobe` to log timestamps of the aforementioned function calls. We found that the queuing delay in the queuing discipline (Qdisc), denoted as T_Q, is almost identical to the difference between RTT_Q and RTT_F. Besides, the delay between `tcp_transmit_skb()` and `dev_queue_xmit()`, as well as the driver queuing delay (T_D) are negligible. This indicates that when sending out traffic in cellular networks, packet queuing in Qdisc dominates the on-device queuing delay in the kernel. Also, as an-

other validation, we observe a strong correlation (around 0.86) between the queuing delay and the amount of traffic in Qdisc.

Firmware Queuing. In LTE uplink, the data to be transmitted from applications is processed and queued in the RLC (Radio Link Control) buffer[4], which is physically located in the cellular chipset firmware. The amount of data available for transmission on the UE (*i.e.,* the firmware buffer occupancy) is provided to the eNodeB through control messages called *Buffer Status Reports* (BSR). BSR can report 64 levels of buffer size with each level representing a buffer size range [30]. The highest level of BSR is 150KB or above. Based on BSR from all UEs, the eNodeB uplink scheduler assigns network resources to each UE for uplink transmission in a centralized manner. The eNodeB sends control messages called *Scheduling Grants* to inform a UE of the scheduling decision. A UE is only allowed to transmit on the physical uplink shared channel (PUSCH) with a valid grant.

We observed that the BSR quickly increases to the highest level (150KB+) when there is large LTE upload traffic. Also there exists a strong correlation (around 0.73) between RTT_F and buffer level in BSR. Leveraging the BSR information, we measured that the actual firmware buffer occupancy can reach several hundreds of KBs (using a more accurate algorithm described in §3.6.2), and the firmware queuing delay (T_F) can reach 400ms with 8Mbps uplink. By subtracting the RTT_F by the firmware queuing delay, we can estimate RTT_B, which is constantly low (*e.g.,* around 50ms for Carrier 1), as shown in Figure 3.7.

Overall, the above findings have two important implications. First, cellular uplink scheduling is performed in a centralized manner, different from that in Wi-Fi networks where clients autonomously sense the wireless channel to transmit data and avoid collision in a distributed way. Second, the firmware buffer distinguishes itself from other on-device buffers in that its occupancy plays a role in the cellular control plane which in turn affects eNodeB's scheduling decisions and the achievable uplink throughput.

[4] In the remainder of this chapter, we use the general term "firmware buffer" to refer to the RLC buffer.

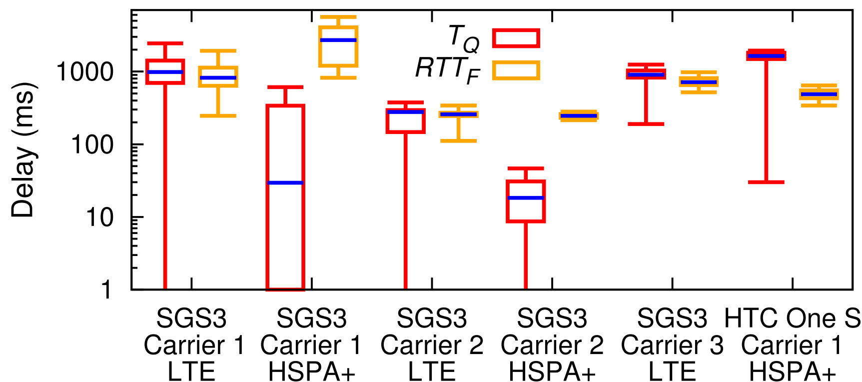


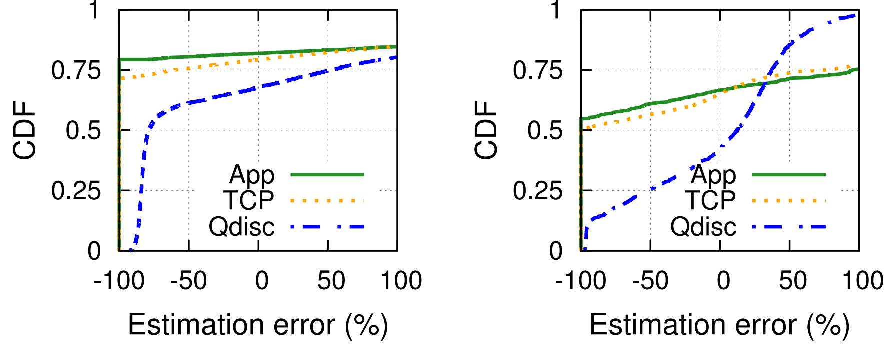
Figure 3.9: On-device queuing delay on diverse devices and cellular carriers.

3.4.3 Prevalence across Carriers & Devices

We show the prevalence of on-device bufferbloat in Figure 3.9 by repeating the upload experiments on various networks using two different devices. For each setting, we report the 5th, 25th, 50th, 75th, and 95th percentiles of T_Q and RTT_F. We observe on-device bufferbloat on all settings in LTE, with median T_Q larger than 200ms. Regarding the HSPA+ network, T_Q is small (around 20ms) for SGS3 using Carrier 2. This is because the TCP sending buffer size is configured to be small by Carrier 2 on this device (we will discuss the impact of TCP buffer size in §3.6.1). Yet across all settings, we found that the RTT_F is much larger than the estimated RTT_B, indicating that excessive firmware queuing happens on all devices and carriers.

3.4.4 Uplink Throughput Measurement

Often applications (*e.g.,* real-time multimedia apps) need to know the instantaneous network throughput. The lower-layer information provided by firmware enables accurate cellular throughput measurement. Recall in §3.4.2 that the UE can only send the amount of data up to the scheduling grant. If a portion of the grant is not used, the firmware uses padding to indicate the unused part. The padding size is also reported by the firmware. Therefore, by subtracting the scheduling grant by the padding size, we can calculate the



(a) Measurement error with 20ms interval (b) Measurement error with 100ms interval

Figure 3.10: Uplink throughput measurement error at different layers.

amount of data sent out from the device, as well as the uplink throughput (the padding is not transmitted).

Since the above approach directly utilizes lower-layer information from the cellular control plane, it gives the ground truth of cellular uplink throughput. An interesting question is, compared to this ground truth, how accurate is the throughput measured at upper layers? We quantify this in Figure 3.10, which plots the measurement error at Qdisc, TCP, and application layer where we use a slide window of 100ms and 20ms to estimate uplink throughput during a bulk upload. The results indicate that the throughput estimation at higher layers are highly inaccurate, with the root mean square being 141%, 136%, and 70% at the application layer, the transport layer, and the Qdisc, respectively, when the estimation interval is 100ms. Reducing the interval further worsens the accuracy. The root cause of such inaccuracy is again the on-device bufferbloat: when a higher layer delivers a potentially large chunk of data into large low-layer buffers, the higher layer thinks the data is sent out but the data will stay in the buffer for a long time. In fact the higher layer has no way to know when the data actually leaves the device. As indicated in Figure 3.10, as the location of measurement moves to higher layers, the overall on-device buffer size increases, leading to worse estimation accuracy.

3.5 Impact of Upload on Mobile Application Performance

This section quantifies the impact of upload on some popular applications' performance: file download, web browsing, video streaming, and VoIP. We compare *user-perceived* application performance in two scenarios: without and with concurrent upload. The experiments in this section were conducted on a Samsung Galaxy S3 phone using Carrier 1's LTE network unless otherwise mentioned. We use a single TCP connection to generate upload traffic in controlled experiments.

3.5.1 Impact of Upload on Bulk Download

When upload and download exist concurrently, upload traffic can affect download traffic in two ways: *in-network* and *on-device*. The former is well-known [139]: upload data shares the same network link with TCP ACK packets of download data, leading to potentially delayed uplink ACK for download. This can cause the server to retransmit download data and reduce the congestion window size, ultimately leading to lower download throughput. On the other hand, the on-device queuing delay triggered by upload can also severely affect download by delaying its ACK packets (shown as t_1 in Figure 3.6(b)), since when download and upload traffic coexist, uplink TCP ACKs share the same queues (*e.g.,* Qdisc and firmware buffers) with uplink data, as detailed in §3.4.2.

We carried out experiments of running a one-minute TCP download flow with and without a concurrent TCP upload flow on different devices and carriers with the setup described in §3.2.1. Table 3.1 quantifies the impact of upload on download in four aspects, using bulk download in absence of upload as the baseline: *(i)* decrease of the average (AVG) download throughput, *(ii)* increase of the relative standard deviation (RSD)[5] of download throughput, *(iii)* increase of AVG RTT, and *(iv)* increase of RSD of RTT. All carriers exhibit performance degradations in various degrees. In particular, large fluctuation of throughput and RTT exists when there is background upload, posing challenges for user-interactive appli-

[5] Relative standard deviation (RSD) = standard deviation / mean.

Setup	Throughput		RTT	
	% AVG decrease	**% RSD increase**	**% AVG increase**	**% RSD increase**
SGS3 C1 LTE	66	253	91	37
SGS3 C1 HSPA+	8	25	7	9
SGS3 C2 LTE	10	36	10	20
SGS3 C3 LTE	80	192	86	42
HTC One S C1 HSPA+	22	260	10	91

Table 3.1: Impact of upload on download performance on different devices, vendors, and networks (C1, C2, and C3 refer to Carrier 1, 2, and 3 respectively).

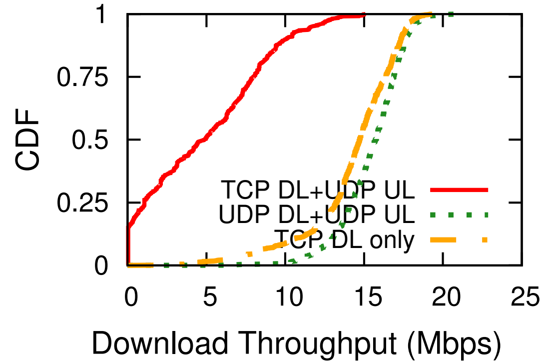


Figure 3.11: Impact of uplink traffic on downlink TCP/UDP throughput.

cations. We also compare the in-network and the on-device impact of upload on download traffic, by computing t_1/t_2 in Figure 3.6(b). The mean and median values of the fractions of t_1 in t_2 are as high as 63% and 74%, indicating *the on-device queuing delay dominates the overall RTT of download traffic.*

Next, we show that when uplink and downlink traffic are both present, the uplink ACK packets being delayed is the dominating cause of degraded download performance. Figure 3.11 plots the download throughput distributions in three scenarios using Carrier 1's LTE network: *(i)* TCP download only, *(ii)* TCP download with concurrent UDP upload, and *(iii)* UDP download with concurrent UDP upload. Figure 3.11 indicates that *(i)* and *(iii)* exhibit similar download performance (scenario *(iii)* is even slightly better because it uses UDP for download) while the throughput in Scenario *(ii)* is much lower. Since the

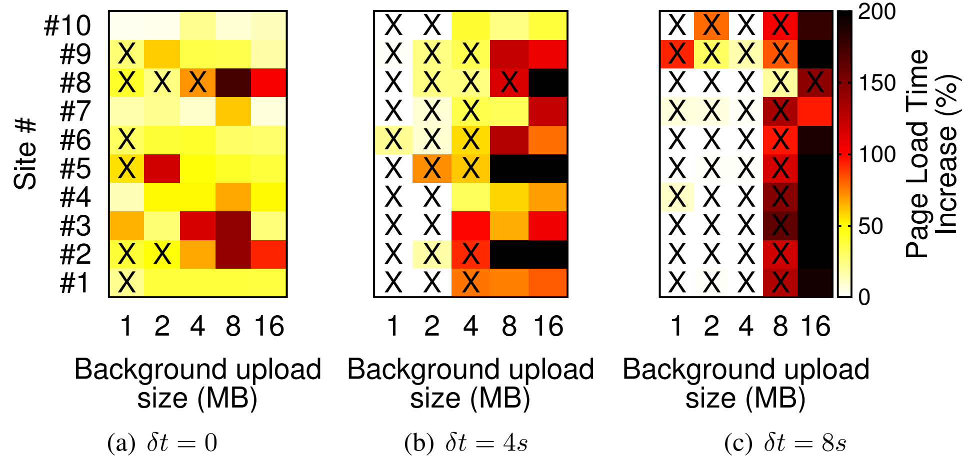
key difference between *(ii)* and *(iii)* is whether the uplink ACK stream exists, the results indicate that *the degraded download performance is almost solely associated with TCP's upstream ACKs*, whereas in the underlying radio layer, uplink and downlink use different channels and can be performed independently. Similar results are observed for upload performance.

3.5.2 Impact on Web Browsing

We next examine the impact of upload traffic on web browsing. We picked ten popular websites from Alexa top sites, and loaded each of them in Google Chrome browser on a Samsung Galaxy S3 phone in two settings: without and with concurrent upload. We repeat the test of each website for 5 times in a row and report the average results. We performed cold-cache loadings for all sites, and measured the page load time (PLT) using QoE Doctor [46, 105].

We found that upload traffic significantly inflates most delay components. For example, the connection setup delay, which usually takes only one round-trip, increases by 64% to 509% due to on-device bufferbloat as the dominating factor. A similar case happens to HTTP requests, which can typically fit into one single TCP packet. HTTP responses that carry downlink data are also affected due to the explanations described in §3.5.1. The response duration inflates by up to 3464%. Overall, the increase of PLT across the 10 websites ranges from 219% to 607%. The results indicate when concurrent upload is in progress, on-device bufferbloat can significantly affect short-lived flows.

Next, we show that *even a medium-sized upload can cause significant degradation of user experience*. Figure 3.12 repeats the above experiments but uses a finite size of upload starting at δt seconds before the web browsing session begins. In each subfigure, a heatmap block (x, y) visualizes the PLT inflation caused by an upload of size x for website y. An "X" mark indicates the upload is completed before the page is fully loaded or even started to load so the measured PLT increase is an under-estimation. We observe two trends. First,



(a) $\delta t = 0$ (b) $\delta t = 4s$ (c) $\delta t = 8s$

Figure 3.12: Impact of upload on PLT. The web browsing session begins δt after upload starts. "X" indicates the upload is completed before the web page is fully loaded.

a larger upload incurs a higher impact on PLT. Second, the PLT impact also becomes higher as δt increases (for blocks without "X"). This is because a larger δt allows more time for the on-device queue to build up, and thus worsens the on-device bufferbloat condition when the web browsing session starts.

3.5.3 Impact on Video Streaming and VoIP

Video Streaming. We randomly chose 10 popular videos of various lengths (from 40 seconds to 6 minutes) from YouTube and played them over LTE on a Samsung Galaxy S3 phone. The playback software is ExoPlayer [6], which uses the standard DASH streaming algorithm. When the signal strength is above -98dBm, the average playback bitrate across 10 videos is 0.93Mbps without any stall when no concurrent traffic is present. With concurrent TCP upload, the average bitrate is reduced by 57% to only 0.39Mbps, with 15.3 stalls (total stall duration 103s) for each video on average. Even when periodical upload is in progress (upload 5MB data every with 5s idle time between consecutive uploads), half of the videos exhibit playback bitrate degradation by up to 19%.

VoIP. We make Skype voice calls from a Samsung Galaxy S3 phone to a desktop in three settings (3 runs each setting): *(i)* Skype call only, *(ii)* Skype call with concurrent TCP

upload, and *(iii)* Skype call with periodical TCP upload of 5MB with 5s idle time between uploads. The experiments were conducted over Carrier 1's LTE network. For each call, we play the same pre-recorded audio (90 seconds) as the baseline and record the audio at the receiver. To quantify the user experience, we use an existing tool [13] to compute the PESQ MOS (Perceptual Evaluation of Speech Quality, Mean Opinion Score) [14] metric. When upload is not present, the average PESQ MOS score is 4.08. With continuous TCP upload, the average PESQ MOS score drops to 1.80. Even for scenario *(iii)*, the average PESQ MOS score is only 1.77.

3.6 QCUT: Solution for On-device Bufferbloat

Given the severity of on-device bufferbloat, we propose our solution called QCUT to mitigate it.

3.6.1 Inadequateness of Existing Solutions

In the literature, numerous solutions have been proposed to mitigate in-network bufferbloat, and some do work with on-device buffers. Table 3.2 lists representative solutions: changing TCP buffer size, changing TCP congestion control (CC), TCP Small Queue (TSQ), Traffic Prioritization (TP), and Active Queue Management (AQM). However, they all have limitations on reducing on-device queuing delay. Changing TCP buffer size and CC are transport layer solutions that adjust TCP behaviors to reduce the delay. However, they do not work with buffers below the transport layer; also they do not provide cross-flow control as each TCP connection has a separate buffer. TSQ, a newly introduced Linux kernel patch, only limits the Qdisc occupancy on a per-connection basis. TP works across flows and improves user experience by prioritizing delay-sensitive traffic. However, it only partially reduces the queuing delay as will be evaluated in §3.7.1. The AQM approaches (*e.g.,* CoDel and PIE) also work across flows. But they do not help reduce the buffer occupancy at the firmware buffer. In §3.7.1, we quantitatively compare all above ap-

Bufferbloat mitigation solution	**Reducing queuing delay**			**Cross-flow control**
	Qdisc	**Driver**	**Firmware**	
Change TCP buffer size	(✓)	(✓)	(✓)	
TCP Congestion Control (CC) [66, 43, 40]	(✓)	(✓)	(✓)	
TCP Small Queue (TSQ) [23]	✓			
Traffic prioritization (TP)	✓			✓
Active Queue Management (AQM) [101, 109]	✓			
Byte Queue Limit (BQL) [3]		✓		
QCUT	✓	✓	✓	✓

Table 3.2: Summary of solutions for reducing queuing delay for upload traffic. "(✓)" means only limited support.

proaches. We also show that jointly applying them may further incur unexpected conflicts, causing additional performance degradation.

We emphasize that *none of the above solutions can be realized at the firmware buffer*, which is usually proprietary hardware making it difficult to incorporate different queue management algorithms. As new wireless technologies and radio chipsets emerge (*e.g.,* 5G and IoT devices), modification to all firmware to solve the on-device queuing is impractical. Also, as shown in §3.4.2, the cellular firmware buffer differs from upper-layer buffers in that it plays a role in the cellular control plane (*i.e.,* the BSR affects uplink scheduling and the LTE uplink throughput). Therefore, even ignoring the implementation issues, naïvely applying existing bufferbloat mitigation solutions on the firmware buffer may lead to unexpected results or performance degradation.

3.6.2 QCUT Design

Motivated by the above, we design and implement a new approach called QCUT to reduce the on-device queuing delay. Here we focus on optimizing cellular uplink but the general concept of QCUT applies to other networks. As illustrated in Figure 3.13, QCUT has three prominent features.

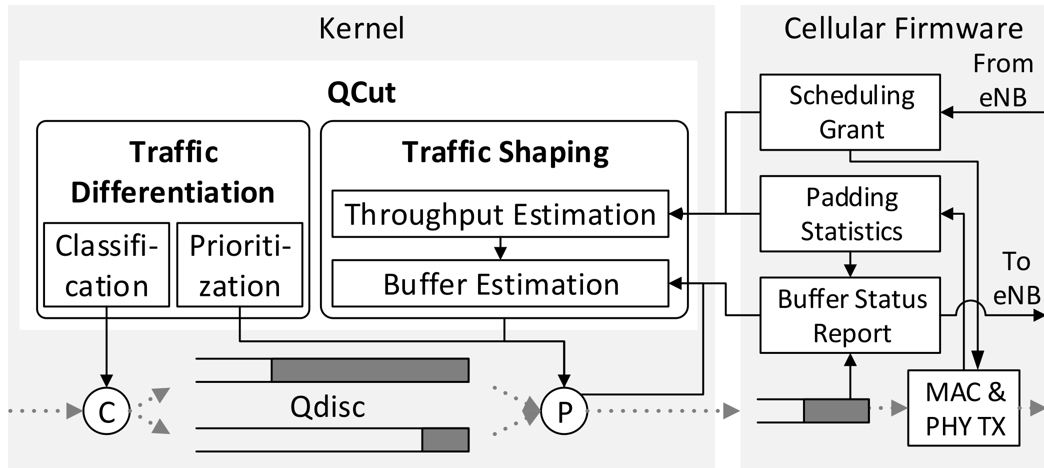


Figure 3.13: The QCUT design.

- Realized as a general OS service, QCUT is independent of firmware implementation. Therefore it can address the on-device queuing problem on any radio firmware, where no modification is needed. QCUT operates in the kernel space and takes as input only information of buffer occupancy and transmission statistics, which is exposed by most cellular radio firmware from Qualcomm and likely other vendors.

- Since directly limiting the firmware buffer occupancy is difficult, QCUT controls the firmware queuing delay *indirectly in the kernel* by controlling how fast packets from Qdisc flow into the firmware buffer. QCUT estimates the radio firmware buffer occupancy and queuing delay to decide the transmission of packets to the firmware dynamically.

- QCUT is flexible on traffic classification and prioritization. By (indirectly) limiting the amount of data in the firmware, packets are queued in the Linux Qdisc, where QCUT can flexibly prioritize packets based on the application requirements. For example, when background upload and interactive traffic co-exist, the latter can be prioritized and transmitted without Qdisc queuing. By contrast, directly realizing fine-grained traffic prioritization in the firmware is impractical and inflexible.

QCUT aims at reducing the on-device queuing delay. When there is no on-device queuing, QCUT does not incur additional delay to RTT_B or other runtime overhead.

As shown in Figure 3.13, QCUT comprises of two components: traffic differentiation

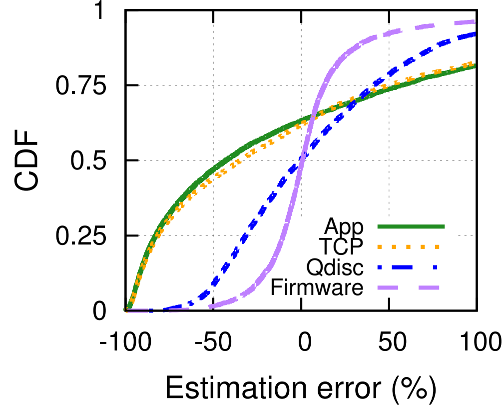


Figure 3.14: Uplink throughput prediction error at different layers with 20ms prediction interval.

and traffic shaping. *Traffic differentiation* classifies packets from applications, and prioritizes certain traffic in the Qdisc (*e.g.,* delay-sensitive traffic) based on applications' requirement. The *traffic shaping* module *(i)* performs accurate throughput prediction, which is then used to *(ii)* estimate the buffer occupancy in the firmware. Based on that, the module *(iii)* controls how fast packets from Qdisc flow into the firmware buffer, in order to limit the firmware buffer occupancy. We describe each component in details below.

3.6.2.1 Achievable Physical Layer Throughput Prediction

Based on recent lower-layer throughput values measured from scheduling grant and padding (§3.4.4), we perform throughput prediction using Exponentially Weighted Moving Average (EWMA) with $\alpha = 0.25$ (empirically chosen). The prediction interval is 20ms. Note that we need to predict the throughput because the lower-layer firmware information is not provided in real time so the throughput measurement is delayed, as we explain shortly. The "firmware" curve in Figure 3.14 plots the prediction error distributions under 20ms prediction interval, in our controlled bulk upload experiments with -95dBm RSRP (8Mbps uplink bandwidth). The ground truth is the lower-layer throughput measured with a delay ($\sim$100ms later). The results indicate that compared to other curves in Figure 3.14 where we perform throughput estimation at higher layers using the same EWMA algorithm, using

lower-layer information for throughput prediction is much more accurate.

3.6.2.2 Buffer Occupancy Estimation

For a wide range of cellular firmware, their buffer occupancy level can be directly read from the buffer status report (BSR). However, a practical issue we found is that, BSR is not reported in real time to allow accurate buffer occupancy estimation. On both Samsung Galaxy S3 and Nexus 5 devices, although BSR is reported to eNodeB every 5ms, there is on average around 100ms delay before this information is reported to the kernel due to various overheads. During this period, the firmware buffer dynamics may fluctuate considerably.

To overcome this issue, we propose to combine the BSR and the predicted throughput to derive accurate firmware buffer occupancy. The basic idea is the following: since we know both the accurate enqueue rate (measured from Qdisc) and dequeue rate (from uplink throughput) of the firmware buffer, we can use them to refine the rough buffer occupancy estimation from delayed BSR. More specifically, let S_0 be the most recently reported BSR generated by the firmware at t_0, which can be obtained from a BSR's timestamp field. Let R_{uplink} be the predicted uplink throughput at t_0. Also we keep track of packets $\{P_i\}$ (i=1,2,...) leaving Qdisc after t_0 by recording their sizes $\{S_i\}$ and timestamps $\{t_i\}$ of leaving Qdisc. Given the above information, the firmware buffer occupancy $B(t_{curr})$ at timestamp t_{curr} can be calculated as follows:

$$B(t_{curr}) = B(t_{n+1}) \tag{3.1}$$

$$B(t_{i+1}) = B(t_i) + S_{i+1} - S_{TX}(i), i \in [0, n] \tag{3.2}$$

$$S_{TX}(i) = min(B(t_i), R_{uplink} \times (t_{i+1} - t_i)), i \in [0, n]; \tag{3.3}$$

where $t_0 < t_1 < .. < t_n \leq t_{n+1} = t_{curr}, S_{n+1} = 0$.

The buffer occupancy is estimated in an iterative manner as shown in Equation (3.2), where S_{i+1} and $S_{TX}(i)$ are the number of bytes enter and leave the firmware buffer since

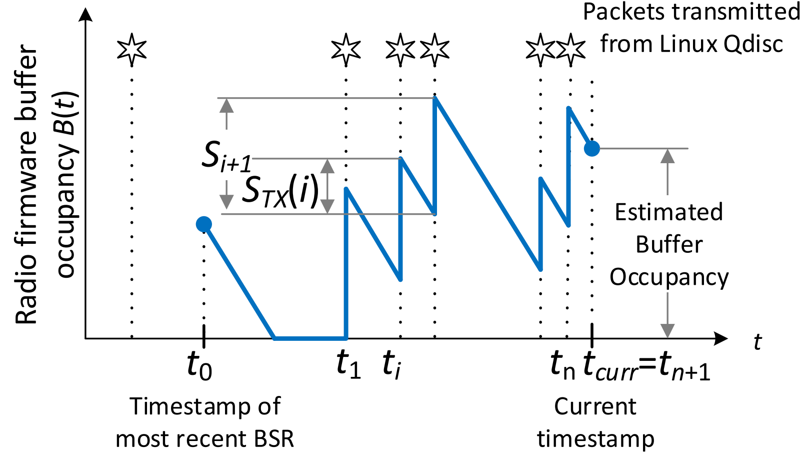
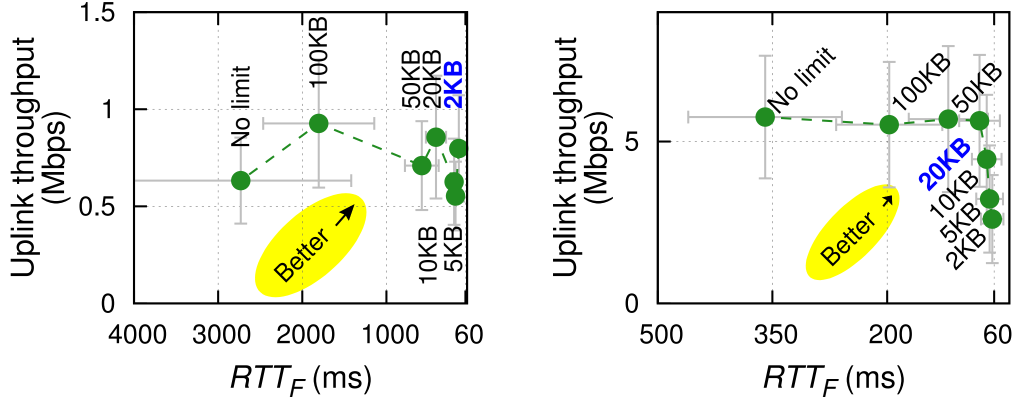


Figure 3.15: Radio firmware buffer occupancy estimation.

t_i, respectively. $S_{TX}(i)$ is computed in Equation (3.3) using the predicted throughput. The process is illustrated in Figure 3.15.

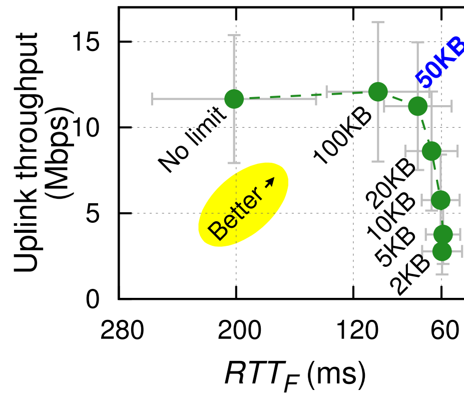
3.6.2.3 Qdisc Dequeue Control

QCUT limits the queuing delay in the radio firmware by throttling the Qdisc in the kernel, *i.e.,* strategically controlling whether a packet should be dequeued from Qdisc into the radio firmware. To realize this, a simple way is to use a fixed threshold of the firmware buffer occupancy, which we refer as QCUT-B (B stands for "bytes"). We evaluated this approach by repeating one-minute TCP uploads five times with different thresholds on a Nexus 5 phone using Carrier 1's LTE network, under different signal strength conditions. As shown in Figure 3.16, different QCUT-B thresholds incur different tradeoffs between throughput and latency (quantified by RTT_F). However, it is difficult to find a threshold that works for all network conditions. The best threshold that achieves low latency without sacrificing the throughput depends on the signal strength: 2KB for -110dBm, 20KB for -98dBm, and 50KB for -85dBm. In particular, a small threshold (*e.g.,* 2KB) works well when the signal strength is low. However, at high signal strength, it causes bandwidth under-utilization. As described in §3.4.2, this is attributed to the very nature of cellular uplink scheduling: the firmware buffer occupancy reported in BSR is used for determining



(a) Signal strength -110dBm

(b) Signal strength -98dBm



(c) Signal strength -85dBm

Figure 3.16: Impact of firmware buffer occupancy threshold of QCUT-B. The best threshold in each plot is in bold blue text.

uplink bandwidth allocation; the base station thus regards a small buffer occupancy as an indicator that the client does not have much data to transmit, thus allocating small uplink bandwidth for the mobile client.

To overcome the above limitation, we propose another scheme called QCUT-D (D stands for "delay"). It instead uses the firmware queuing delay (T_F) as a threshold. The queuing delay is computed from the estimated throughput and the buffer occupancy. If the delay is above the threshold, QCUT-D does not allow a packet to be dequeued to the firmware from Qdisc. Thus, QCUT-D is adaptive to diverse network conditions by dynamically adjusting the firmware buffer occupancy. We empirically found that using 20ms as the delay threshold on LTE networks works reasonably well in diverse network conditions:

it leads to low firmware buffer queuing while incurring very small impact on the uplink throughput, as to be evaluated in §3.7.2. This threshold can also be empirically chosen for other types of networks.

3.6.2.4 Traffic Differentiation

To meet the performance requirement of different applications, QCUT uses the priority queuing in Linux Qdisc for traffic prioritization. For example, the background upload and interactive traffic such as web browsing are put into different queues in Qdisc. As a result, interactive traffic does not experience high queuing delay in Qdisc caused by bulk upload. Also, thanks to the aforementioned traffic shaping module in QCUT, the delay-sensitive traffic also undergoes very low queuing delay in the firmware, thus leading to an overall small on-device queuing delay and thus good user experience. QCUT uses existing traffic classification mechanism on Linux to allow applications and users to flexibly configure priorities for different traffic through the standard `tc` interface.

3.6.3 QCUT Implementation

We implemented QCUT on Android Linux kernel. Our testing devices consist of Samsung Galaxy S3 and Nexus 5 running Android 4.4.4 and 6.0.1 with Qualcomm radio chipset. We expect QCUT to also work with other phones and tablets with cellular firmware from the same vendor. Note that QCUT does not require any special equipment such as QXDM [16].

Traffic shaping and differentiation are implemented as a Linux packet scheduler module in 600 LoC. QCUT keeps track of transmitted packets from Qdisc since the most recent BSR. The traffic shaping module is implemented in the function call `enqueue()` of the Qdisc operation data structure `Qdisc_ops`. In `enqueue()`, the queuing delay in firmware is estimated based on the information from the radio firmware. More specifically, we use the `/dev/diag` interface on the Android phones with Qualcomm radio chipset to extract

the uplink scheduling grant, padding statistics, and BSR from the logs of LTE uplink transport blocks. The online parsing of the logs is implemented in a C++ program in the user space. Each log record has a timestamp of the firmware. The timestamps between kernel and the firmware need to be synchronized. The user-space program sends time request periodically and uses the response, which contains the firmware timestamp, to perform the synchronization.

3.7 Evaluation

We comprehensively assess how a wide range of solutions help mitigate the on-device bufferbloat problem, focusing on existing solutions (§3.7.1) and then QCUT (§3.7.2). For all the following experiments, we conducted on a Samsung Galaxy S3 on Carrier 1's LTE network unless otherwise mentioned. We expect the experimental findings to be general as none of the solutions depends on a specific carrier or vendor.

3.7.1 Existing Solutions

We consider existing bufferbloat-mitigation solutions discussed in §3.6.1. We demonstrate in this section that they can reduce excessive on-device queuing to various degrees. However, they suffer from various limitations, and are all incapable of reducing the firmware buffer occupancy. We conduct bulk upload experiments at two locations with different signal strengths measured by RSRP (Reference Signal Received Power): good signal (RSRP of -69 to -75 dBm) and fair signal (RSRP of -89 to -95), using a Samsung Galaxy S3 on Carrier 1's LTE network.

Changing TCP buffer sizes. The TCP send buffer (`tcp_wmem`) on device imposes a limit on the TCP congestion window (cwnd). As shown in Figure 3.17, under good signal, shrinking the send buffer effectively reduces RTT_Q that is dominated by device-side queuing at Qdisc and firmware. However, the penalty is severely degraded upload throughput, in particular when `tcp_wmem` is smaller than the bandwidth-delay product (BDP). Since

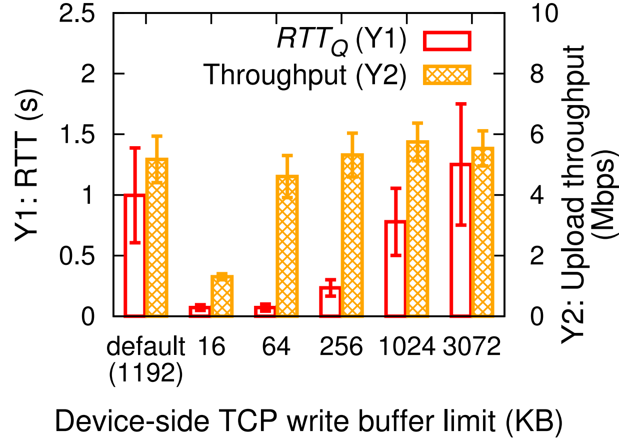


Figure 3.17: Impact of TCP send buffer sizes on upload performance.

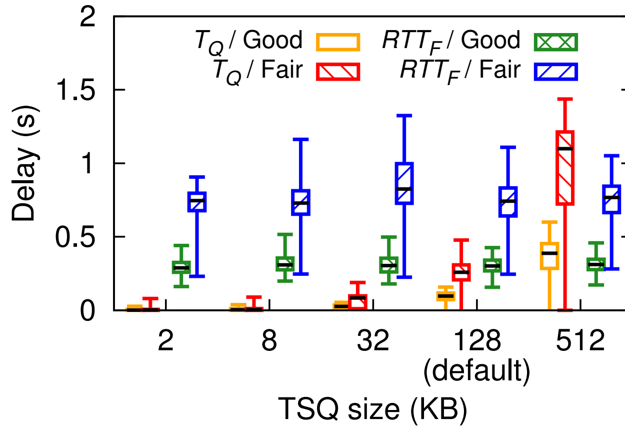


Figure 3.18: Impact of different TCP small queue (TSQ) sizes on upload.

BDP constantly fluctuates in cellular networks [131], a fixed configuration of TCP buffer size does not fit all network conditions.

Changing TCP small queue (TSQ) size. As a newly introduced Linux kernel patch, TSQ [23] limits per-connection data in Qdisc using a fixed threshold. By reducing the threshold, we observe smaller Qdisc queuing delay ($T_Q = RTT_Q - RTT_F$ in Figure 3.8) under both network conditions, as shown in Figure 3.18. Yet Linux's default TSQ threshold is too large to eliminate the Qdisc queuing. However, Figure 3.18 also indicates that TSQ has negligible impact on the firmware queuing delay ($T_F = RTT_F - RTT_B$), because TSQ only controls the bytes in Qdisc. Further, TSQ limits Qdisc occupancy on a *per-connection* basis so the Qdisc occupancy can still be high when concurrent flows exist.

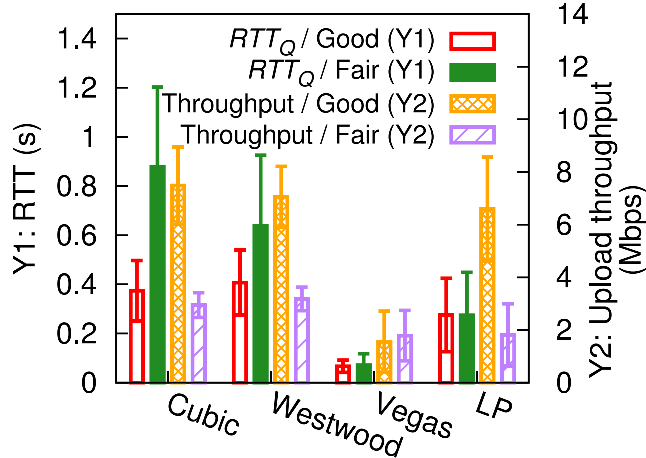


Figure 3.19: Impact of different TCP CC on upload (with TSQ=128KB).

Changing TCP congestion control. Congestion control (CC) affects the aggressiveness of TCP. We consider two representative CC categories: loss-based CC (TCP CUBIC[66] and Westwood[99]) and delay-based CC (TCP Vegas[43] and LP[86]). Generally speaking, lost-based CC, which uses packet loss as congestion indicator, is more aggressive than delay-based CC that treats increased delay as a signal of congestion. We found even with TSQ enabled, loss-based CC incurs severe on-device queuing, measured by RTT_Q, as shown in Figure 3.19. For delay-based CC, regardless of TSQ setting, on-device queuing is almost always negligible. However, such low on-device queuing delays are achieved by sacrificing up to 80% of the throughput.

Active Queue Management is a major in-network solution to reduce queuing delay and network congestion by strategically dropping packets in a queue. We considered two well-known and recently proposed AQM algorithms, CoDel [101] and PIE [109]. Both approaches use a target threshold to control the queuing delay. Under both signal strengths, CoDel effectively keeps the Qdisc queuing delay below the target threshold. However, Since CoDel does not apply to the firmware buffer, it only slightly reduces RTT_F by 10% to 20%, as indicated in Figure 3.20. This is the result of TCP cwnd reduction triggered by packet losses injected by CoDel. The performance of PIE is even worse than CoDel.

Jointly applying multiple strategies. In many case, several mitigation strategies can

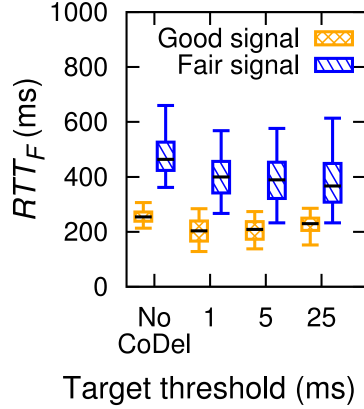
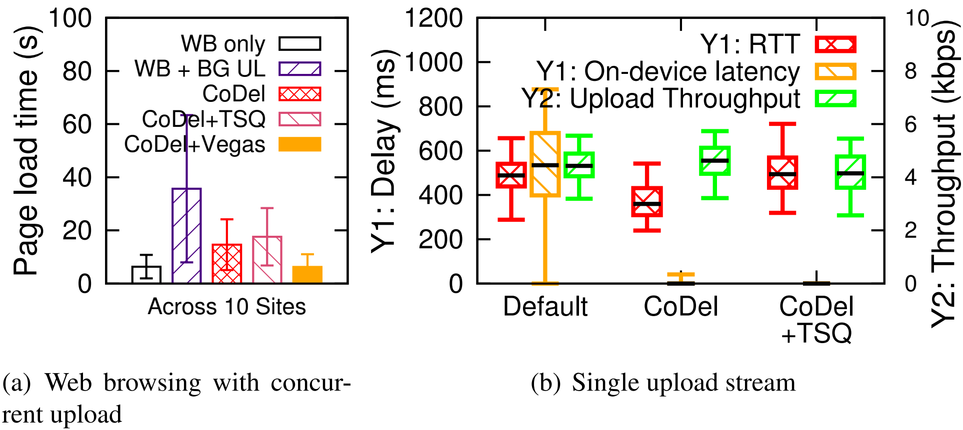


Figure 3.20: Effectiveness of CoDel on reducing latency.



(a) Web browsing with concurrent upload

(b) Single upload stream

Figure 3.21: Effectiveness of jointly applying multiple mitigation strategies on upload.

be jointly applied to better balance various tradeoffs. We study two representative scenarios. In Figure 3.21(a), when concurrent upload is present, jointly applying CoDel and Vegas on upload reduces web browsing PLT to almost the minimum (*i.e.,* close to the PLT without upload). CoDel effectively complements delay-based CC by controlling queuing delay across *all* connections in an aggregated manner.

However, we find that jointly using several approaches may also incur unexpected conflicts, causing performance degradation. For example, when CoDel (with target threshold 5ms) and TSQ (with queue size 4KB) are jointly applied to a single upload flow, RTT_F actually increases by 37% compared to using CoDel alone (figure not shown). This is explained as follows. A small Qdisc achieved by TSQ can reduce the effectiveness of CoDel,

since the small on-device queuing delay allows CoDel to drop very few packets compared to a large queue does. This causes TCP cwnd to increase faster, leading to more noticeable in-network queuing delay. Such an observation is also confirmed in Figure 3.21(a), which shows CoDel+TSQ incurs higher PLT than CoDel does, when upload and web browsing coexist.

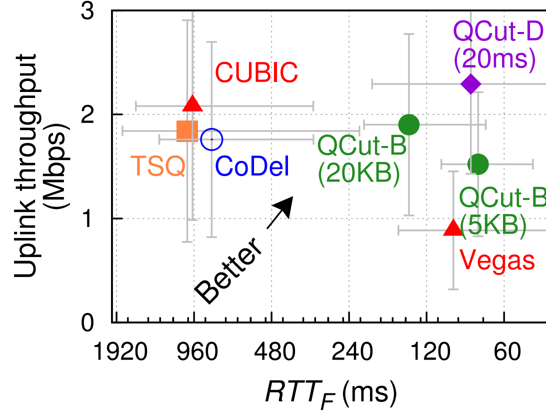
Traffic prioritization. All above solutions focus on reducing on-device queuing for bulk upload. When concurrent upload and download exist, an alternative approach is to prioritize uplink ACK packets over upload data traffic to mitigate the impact of upload on download (§3.5.1). Our experiments indicate that when uplink ACKs are prioritized, their Qdisc queuing delay is reduced significantly from 1363ms to 86ms at -95dBm. However, prioritization can only be realized at Qdisc, causing the uplink ACK stream still to interfere with uplink data at the firmware buffer. As a result, compared to the case where no concurrent upload exists, applying prioritization still increases RTT_F by 112ms. We will demonstrate in §3.7.2 that by combining Qdisc prioritization with firmware queuing delay reduction, QCUT can effectively mitigate on-device bufferbloat.

3.7.2 Evaluation of QCUT

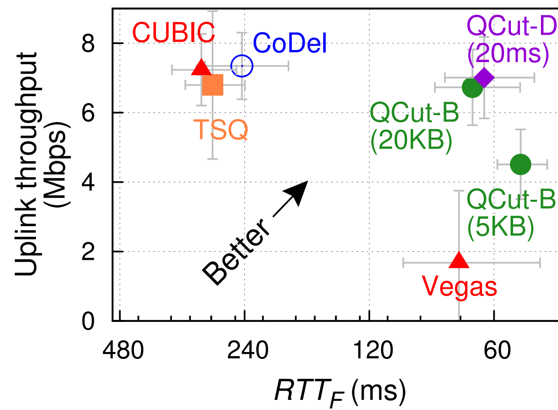
We conduct a thorough evaluation of QCUT to demonstrate that it outperforms existing solutions. First, we show QCUT can significantly reduce RTT_F that mainly consists of the firmware queuing delay. We then conduct a crowd-sourced user study to assess the effectiveness of QCUT under real workload (web browsing and video streaming) when bulk upload is present.

3.7.2.1 Reducing Excessive Firmware Queuing

Using the workload of a single TCP upload, we compare the performance of five schemes: TCP CUBIC, TCP Vegas, TSQ, CoDel, and QCUT. Each experiment thus consists of five back-to-back TCP uploads (one minute each) using Carrier 1's LTE network.



(a) Signal Strength -110dBm



(b) Signal Strength -95dBm

Figure 3.22: Compare TCP upload performance of different schemes.

We repeat the experiment for 10 times at two locations with stable signal strength of -95dBm and -110dBm, respectively. We calculate the throughput every 500ms and measure RTT_F using `tcpdump` traces. For each scheme, we report the average result at each location.

Since the five schemes achieve different tradeoffs between throughput and latency, we visualize the results on a two-dimensional plane in Figure 3.22. The X and Y axes correspond to RTT_F and measured throughput, respectively. A good solution should appear in the upper-right corner of the plane. The results indicate that except for QCUT and TCP Vegas, none of the five solutions is capable of reducing RTT_F because they do not work at the firmware layer. For TCP Vegas, in Figure 3.22(b), it achieves low latency at the cost

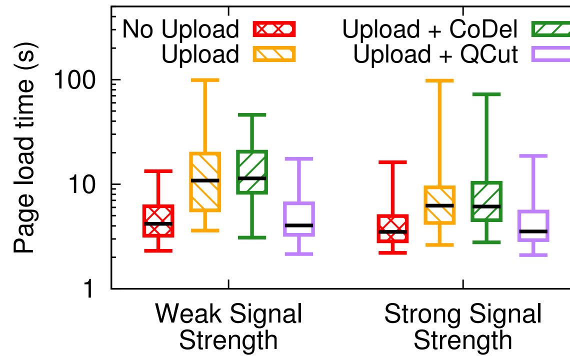
of very low throughput, with the reason explained in §3.7.1. On the other hand, QCUT effectively reduces the firmware queuing with little or small sacrifice of the throughput. Recall in §3.6.2 that we devised two QCUT schemes: QCUT-B and QCUT-D, which use the firmware buffer *occupancy* and *delay* as the threshold to limit the firmware buffer occupancy. We found QCUT-D works reasonably well at both locations since it is adaptive to different throughput, while it is a bit difficult to pick a fixed threshold for QCUT-B for different throughput.

3.7.2.2 Improving Application Performance

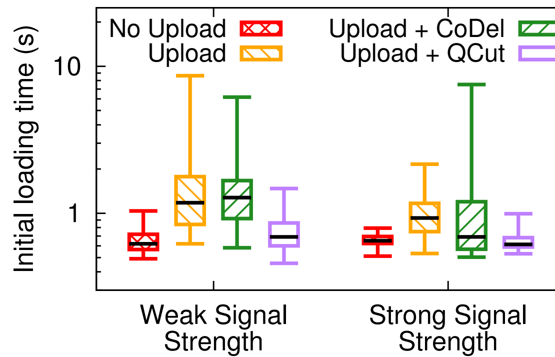
To assess how QCUT improves real applications' performance, we deployed QCUT on five Samsung Galaxy S3 phones used by real users. The phones run crowd-sourced measurements supported by Mobilyzer [106] for a week under diverse network conditions. This user study has been approved by IRB.

We consider two workloads: (1) load five popular webpages, and (2) stream a 2-min YouTube video. For each workload, we run back-to-back measurements under four different settings: *(i)* no background upload, *(ii)* concurrent upload without bufferbloat mitigation, *(iii)* concurrent upload with CoDel on Qdisc, and *(iv)* concurrent upload with QCUT-D. For web browsing, we collect the page load time (PLT) of each webpage; for video streaming, we record initial loading time, playback bitrate and rebuffering events. Note we only triggered these measurements when the phone is idle, so the experiment is not interfered with other user traffic. To mitigate the impact of the varying signal strength within the same experiment that consists of four back-to-back measurements, we discard the entire experiment if the LTE RSRP changes by more than 4dBm. Overall we conducted 1266 and 549 successful experiments for web browsing and video streaming, respectively.

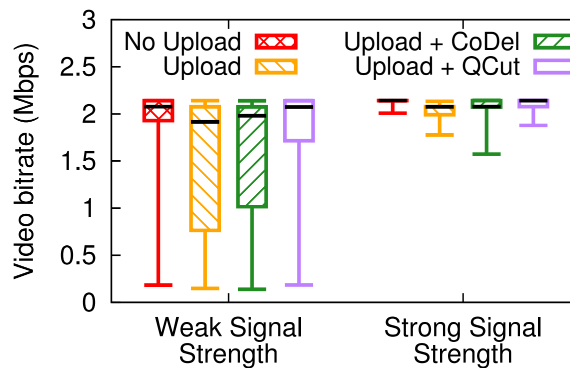
The results are shown in Figure 3.23. In each plot, we show two groups of results corresponding to weak signal strength (LTE RSRP$<$-99dBm) and strong signal strength (LTE RSRP$\geq$-99dBm), respectively. As shown in Figure 3.23(a), due to concurrent up-



(a) Web browsing



(b) Video streaming: initial loading time



(c) Video streaming: bitrate

Figure 3.23: Improvement of application performance brought by QCUT.

load, the median PLT across 5 sites increases by 78% and 159% for strong and weak signal strength, respectively, leading to significantly degraded user QoE. Applying CoDel does not help mitigate the additional ACK delay (of webpage download) incurred by the bulk upload, in particular when the signal strength is weak: the median PLT increases are still as large as 75% and 171% for strong and weak signal strength, respectively, compared to

the no-upload cases. QCUT-D, on the other hand, effectively reduces the PLT to the baseline (*i.e.,* no-upload). For video streaming, we consider two QoE metrics: initial buffering time and playback bitrate, whose results are shown in Figure 3.23(b) and 3.23(c), respectively. Again, QCUT significantly outperforms CoDel on improving the video streaming QoE when concurrent bulk upload is present.

As described in §3.6.2, the effectiveness of QCUT is attributed to two reasons. First, it reduces the firmware buffer occupancy (T_F). But doing that alone is not sufficient because delay sensitive traffic can still be interfered by upload traffic at Qdisc. QCUT addresses this by performing prioritization at Qdisc, resulting in reduced T_Q for delay sensitive traffic.

3.8 Discussion

The firmware of cellular radio chips from major baseband modem vendors, such as Qualcomm [1, 17], is closed-sourced today [26]. Under such scenario, QCUT solves on-device bufferbloat as long as the radio firmware reports the necessary information to the kernel. If radio firmware allows modification, some components in QCUT can be integrated into the firmware to better control the queuing delay in the firmware. Besides traffic differentiation in the kernel, the firmware should also differentiate network traffic with priority queues instead of a single FIFO queue, based on application types and connections indicated by the kernel. Thus, a packet from interactive apps can be prioritized in both Linux Qdisc and firmware queue. Traffic shaping can be implemented in the firmware instead to guarantee that each packet in every queue experiences the low delay before transmitted to the network.

3.9 Summary

We carried out to our knowledge the first comprehensive investigation of cellular upload traffic and its interaction with concurrent traffic. Our extensive measurement using

33-month crowd-sourced data indicates the contribution of upload is large, and the upload speed is high enough to enable applications to upload user-generated traffic. We then comprehensively investigated the on-device bufferbloat problem that incurs severe performance impact on applications. We identified a major source of on-device bufferbloat to be the large firmware buffer, on which existing bufferbloat mitigation solutions are ineffective. We then propose a general and lightweight solution called QCUT, which controls the firmware buffer occupancy from the OS kernel. We demonstrate the effectiveness of QCUT through in-lab experiments and real deployment.

CHAPTER IV

Improving Multipath Architecture for Mobile Networks

In this chapter, we move on from addressing flow-level parallelism to better leveraging interface-level parallelism. We propose a flexible multipath architecture for multipath over mobile networks to address the limitations of MPTCP, the most widely used multipath solution.

4.1 Introduction

Despite existing efforts of understanding and improving multipath transport, there still remain numerous challenges for effectively and efficiently using mobile multipath. To name a few, first, originally designed for data center networking [115], MPTCP may incur unexpected cross-layer interactions when used on mobile devices [54, 68]. Second, as shown later, MPTCP often incurs additional energy overhead, but may not always boost (sometimes even worsen) application performance. Therefore, applications should use multipath only when its benefit outweighs its incurred overhead. Such decision logic is largely missing or done naïvely in practice. Third, protocols such as MPTCP are complex with numerous configurations, and it is unclear how to tune them in an optimal way. Fourth, from a system architectural perspective, MPTCP's "everything in kernel" scheme may not be suitable for mobile multipath to support a rich set of application-specific policies.

In this chapter, we explore the following unanswered question: *How to improve the*

system architecture for mobile multipath? The measurement results from existing work indicate that MPTCP suffers from a few limitations: poor interaction with short/small flows, a lack of infrastructural support for multipath policy, and MPTCP extension often being blocked by middleboxes [54, 68, 104]. We propose a flexible software architecture of mobile multipath called MPFlex that overcomes all the above limitations. MPFlex has several prominent features:

• First, it performs transparent *multiplexing* for application traffic over multipath. Our multiplexing scheme reduces the number of handshakes from many (one per path) to zero, leading to significant improvement of bandwidth utilization for small flows.

• Second, MPFlex *decouples* the high-level scheduling algorithm and the low-level OS protocol implementation. This is realized by maintaining most of MPFlex's logic in the user space, which obtains lower-layer information (*e.g.,* latency and congestion window) from kernel through a unified API. Such a framework dramatically simplifies the development, deployment, and maintenance of multipath features.

• Third, MPFlex has visibility of all traffic on an end host, and thus provides an ideal vantage point for applying user-specified multipath policies.

• Fourth, MPFlex is middlebox-friendly as it does not use any Layer 3 or 4 protocol extensions which may be blocked by ISPs.

Compared to MPTCP, MPFlex reduces single file transfer time by up to 49%, improves bundled short flows' transfer time by up to 63%, and boosts real web page load speed by up to 20%, while incurring negligible overhead. We also demonstrate MPFlex's capability of flexibly plugging-in new features such as buffer-aware scheduling, smart packet reinjection, and per-application policies, which can be implemented in less than 70 lines of user-level code.

4.2 MPFlex: A Flexible Architecture For Mobile Multipath

MPTCP has several limitations: its interplay with short-lived flows can be further improved; it is important but currently difficult to incorporate application policies into MPTCP; the MPTCP extension is often blocked by middleboxes; MPTCP apparently does not work with other protocols such as UDP; MPTCP also incurs unexpected interaction with CDN.

We realize that many of these challenges stem from the origin of MPTCP, which was originally developed for improving the performance and robustness for datacenter networks [115]. Datacenters are "closed" ecosystems where latency is small, long-lived flows are common for intra-datacenter traffic, and administrators have full control over all applications and network elements. The mobile ecosystem, however, is drastically different, making naïvely porting MPTCP to mobile devices suboptimal.

4.2.1 The MPFlex Architecture

Motivated by the above, we attempt to address an important research question: *what is a better system architecture for mobile multipath?* To this end, we design, implement, and evaluate a flexible software architecture of mobile multipath called MPFlex. As illustrated in Figure 4.1, MPFlex is a proxy-based solution. It is transparent to both client applications and servers, with several prominent features.

- MPFlex employs multiplexing to consolidate multiple (potentially short-lived) connections into two long-lived connections: one over WiFi and the other over cellular. The Multiplexed Connections (MC), shown in Figure 4.1, are persistent and are by default shared by all applications. They cover the "last-mile" links that are usually the bottleneck. This design overcomes MPTCP's key limitation when dealing with short-lived flows due to following reasons.

First, in MPFlex, since MCs are pre-established, an application connection (shown in

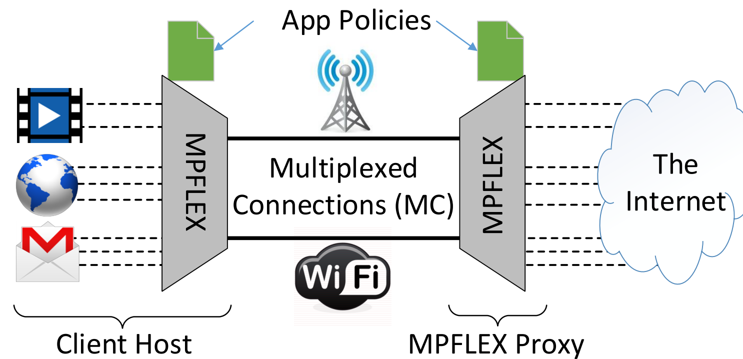


Figure 4.1: The MPFlex architecture.

dashed lines) only needs one handshake over usually the fastest path to inform the proxy to create the corresponding connection with the remote server, instead of handshakes for *every* subflow in MPTCP. This allows all subflows to be usable sooner, thus improving the bandwidth utilization. This handshake can be eliminated by applying the idea of TCP fast open [114], resulting in *zero-RTT handshake over multipath* between the device and the proxy.

The second benefit of multiplexing is, as an MC is persistent and long-lived, it can preserve the congestion window. This avoids the bandwidth probing (*e.g.,* TCP slow start). Note this advantage also exists in single-path, but it is more prominent in multipath by improving *all* subflows. When a long-lived MC has no data to transmit or receive, the radio interface switches to the IDLE state to save energy, while maintaining the TCP state.

Multiplexing has been employed by other application protocols such as SPDY [128], HTTP/2 [42], and QUIC [18]. MPFlex instead performs multiplexing at the transport layer while being transparent to upper-layer protocols. In particular, unlike a SPDY or HTTP/2 proxy that needs to be man-in-the-middle for SSL/TLS sessions (thus breaking the end-to-end security), MPFlex can transparently work with SSL/TLS.

- MPFlex *decouples* the high-level scheduling algorithm and the low-level OS protocol implementation. This is realized by implementing most of MPFlex's logic in the user space (unlike MPTCP's "all-in-kernel" approach), which obtains lower-layer information such as

Multipath Solution	**MPTCP Proxy**	**HTTP/2 Proxy + MPTCP**	**MPFlex**
Multiplexing	No	Yes	Yes
Good Short Flow Performance	No	No	Yes
Protocol Applicability	TCP only	HTTP only	Any protocol
User or Kernel	Mostly kernel	Kernel + user	Mostly user
Transparent To SSL/TLS	Yes	No	Yes
Middlebox-friendly	No	No	Yes
Application Policies	No	No	Yes

Table 4.1: Comparison of three multipath proxy solutions.

latency and congestion window size from kernel through a unified API. This dramatically simplifies the development and deployment of multipath features (§4.2.3).

• MPFlex is a flexible framework. Unlike MPTCP, MPFlex is provided as an OS service and can transparently provide multipath support for non-TCP protocols. An MC can be realized by a wide range of transport protocols such as TCP, reliable UDP, and SCTP [124], enabling continuous transport-layer innovation. Both features are realized through a *pair* of MPFlex modules deployed on both the client and the proxy. They not only perform (de)multiplexing, but also transparently intercept application packets (at the client), and serve as end points of MCs.

• MPFlex has visibility of all client traffic, making it an ideal vantage point for applying user-specified multipath policy based on application usage, performance, cellular data usage, and energy. Realizing such a policy framework by MPTCP itself is difficult unless a centralized traffic manager similar to MPFlex is introduced.

• MPFlex is middlebox-friendly. Since the multiplexing protocol runs *above* the transport layer, it does not require any network-layer or transport-layer extensions such as the MPTCP extension [58] that might not be recognized by today's firewalls and middleboxes. Table 4.1 compares three multipath proxy solutions: MPTCP proxy, MPFlex, and HTTP/2 proxy with MPTCP.

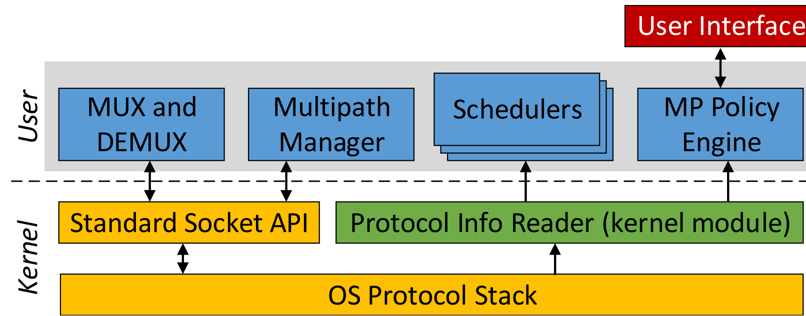


Figure 4.2: Components within an MPFlex endpoint.

4.2.2 MPFlex Design and Implementation

We implement MPFlex as follows. Multiplexing is performed in a way similar to that in SPDY and HTTP/2, but across TCP connections instead of HTTP transactions (an approach similar to [112]). On the client side, uplink TCP data from applications is segmented and encapsulated into *messages*, which are then distributed onto MCs. Each message has a small header containing its application connection ID, length, and message sequence number. Upon the reception of a message, the proxy performs demultiplexing by extracting the data and forwarding it to the remote server based on the connection ID. Downlink traffic is handled similarly but in the reverse direction. TCP SYN, FIN, and RST are also encapsulated into control messages to realize application connection management.

Based on the above basic multiplexing infrastructure, we developed MPFlex by adding three critical new components shown in Figure 4.2: multipath manager, schedulers, and policy engine. We currently implemented two types of MC: TCP and UDP (for multipath UDP). We only describe TCP here due to space constraints.

MPFlex is implemented in C++ with about 5K LoC on Nexus 5 with Android 4.4.4 as the client host and a commodity server as a MPFlex proxy. On the client side, we implemented a lightweight Linux kernel module using `netfilter` hooks to intercept uplink TCP packets and redirect them to the MPFlex userspace program, which manages MCs, application connections, and makes scheduling decision for uplink traffic. The MPFlex

proxy performs similar operations for downlink traffic. Our implementation allows MPFlex to transparently provide multiplexing over multipath, so all applications can immediately benefit from MPFlex without modification.

4.2.2.1 Multipath Manager

Multipath manager provides basic multipath support. At client host, MPFlex configures local routing tables, and sets up two regular TCP connections to the MPFlex proxy over the WiFi and cellular interface, respectively, as MCs. An MC is long-lived, unless its network interface is down. In that case multipath multiplexing falls back to single-path. The MC is reestablished when its interface becomes alive. Extending MPFlex for supporting more than two interfaces is also straightforward.

When an application connection issues a TCP SYN handshake, the client-side MPFlex module intercepts it, and sends a control message, containing the server IP and port, to the proxy-side MPFlex module over one MC selected by the scheduler (described below). The proxy then establishes the connection to the remote server. This differs from MPTCP's connection establishment where *every* path needs to perform its own handshake. A similar situation happens when closing the connection. The handshake message is treated as transport-layer payload, not bound to a particular path. Therefore, if WiFi is congested while LTE is active but less loaded, the handshake is performed over LTE. Also as mentioned before, MPFlex allows the handshake to be piggybacked with the uplink user data (of up to a threshold of n bytes) to achieve 0-RTT handshake over multipath.

4.2.2.2 Schedulers

Schedulers determine how to distribute data across multiple paths (MC in MPFlex). A key architectural design decision is to realize the scheduling logic at user level as much as possible. Toward this goal, we implement a small Linux kernel module that exposes a few in-kernel metrics such as RTT, TCP congestion window size, and TCP bytes-in-

flight to the user space. The kernel module has very simple logic. It does not modify any state in the kernel, and remains unchanged once deployed in a specific kernel. We then build the actual schedulers in user space by utilizing the above kernel information API. Our design dramatically simplifies the scheduler implementation by *decoupling* the high-level scheduling algorithm and the low-level OS protocol implementation. In contrast, in MPTCP, they are tightly coupled, and upgrading MPTCP requires upgrading the entire kernel (tens of MB download). We have replicated two MPTCP schedulers: minimum RTT (MinRTT) and round robin in MPFlex. Other schedulers (including those for non-TCP protocols) can be developed and plugged into MPFlex. To demonstrate the flexibility of MPFlex, we also designed two new schedulers to be described in details in §4.2.3.

4.2.2.3 Policy Engine

Policy engine provides a higher-level abstraction of determining when and how to use multipath according to user-defined policies. In our current implementation, a policy is an ordered list of rules specifying what kind of traffic should use which multipath scheme, such as "multipath with MinRTT scheduler is only used by browsers and YouTube, and single-path is used in all other cases". User-defined policies are applied at a per-process basis. For a given traffic flow, MPFlex finds its corresponding process name using the methodology described in previous work [113]. For uplink traffic, the client-side MPFlex module can execute the policy by itself. For downlink traffic, instead of letting the proxy perform traffic classification, the client directly instructs the proxy on how to apply multipath by attaching a one-byte label to an uplink message. The proxy then applies the policy to the downlink traffic according to the label.

The policy engine can be extended to consider cellular billing (*e.g.,* disable multipath when the monthly data plan has less than 100MB left), energy (*e.g.,* disable multipath when the battery is low), and performance (*e.g.,* for YouTube, use cellular as the secondary path only when WiFi cannot provide 1Mbps throughput). We plan to realize such metrics in our

Plug-in for MPFlex	User-level LoC in C++
MinRTT Scheduler of MPTCP	70
Round-robin Scheduler of MPTCP	15
Buffer-aware Scheduler (§4.2.3, §4.3.4)	30
Smart reinjection (§4.2.3, §4.3.4)	60
Realization of a simple policy (§4.3.3)	20

Table 4.2: Implementation overhead of different MPFlex plug-ins.

future work.

MPFlex can integrate QCUT to address kernel and radio firmware queuing on each network interface. When there is bulk data transfer, packets can be queued in the kernel and firmware buffers, after being transmitted to a specific network interface by the multipath manager in MPFlex. The path selection decision of these packets cannot be changed even the multipath policy of the traffic is modified, leading to delayed policy switching. The multipath manager and policy engine in MPFlex can interact with the traffic differentiation and traffic shaping in QCUT to guarantee low queuing delay on each path and provide responsive switching between different multipath policies, *e.g.,* from multipath to single-path WiFi for VoIP apps.

4.2.3 MPFlex Use Cases

Besides the performance benefit, a major advantage of MPFlex is flexibility. Developers can easily design multipath schedulers or realize custom policies at user-level by using a common API to get the kernel information. Such flexibility is demonstrated in Table 4.2, which summarizes the implementation efforts we made for different plug-and-play components to be discussed and evaluated in §4.3. Here we describe two examples in detail.

4.2.3.1 Buffer-aware Scheduling

Recall that for small flows, MPTCP may perform worse than SPTCP in the following scenario. Suppose WiFi has a much smaller RTT than LTE. When WiFi's cwnd is fully

utilized but LTE has available cwnd space, MPTCP's default scheduler *always* uses LTE regardless of its large latency. The optimal scheduling decision, however, is to buffer data at WiFi subflow's socket buffer unless it is full.

Inspired by this, we modify the MinRTT scheduler to let it consider both the network latency and the local buffering latency. Let $srtt$ be the (smoothed) RTT estimated by TCP. Recall the MinRTT scheduler picks a subflow that (1) has available cwnd space and (2) has the minimum $srtt$. Our modified scheduler, called TxDelay, instead picks a subflow that has the minimum $srtt + Q$ regardless of its cwnd status. Q quantifies how long it takes to drain the sender buffer. It can be estimated by $Q = \frac{B}{cwnd*mss/srtt}$ where B is the TCP sender buffer occupancy and mss is the TCP maximum segment size. It is worth highlighting that thanks to MPFlex's user-level realization, it takes only 30 lines of user level code to implement the TxDelay scheduler.

4.2.3.2 Smart Reinjection

Reinjection is a MPTCP feature allowing the same data to be sent over multiple subflows [73]. It helps reduce receiver side buffering due to out-of-order packets, leading to improved throughput when one or a subset of paths encounter performance degradation such as high loss rate or long latency caused by weak signal strength. MPTCP employs a static and fixed policy for reinjection: reinject packets when a subflow is terminated or its receiver buffer is full. We found for mobile multipath, MPTCP's default reinjection policy is often too conservative. For example, if a packet loss occurs on WiFi, the reinjection does not happen until the packet times out.

We propose to make the reinjection policy dynamic and configurable. For example, the sender can perform proactive reinjection based on different packet loss signals, such as duplicate ACKs and/or the receiver window occupancy (`recvWin`) embedded in the ACK packet. To realize this in MPFlex, the Protocol Info Reader (Figure 4.2) provides information such as `recvWin` and event callback such as TCP timeout and duplicate ACK, which

are utilized by the user-level proxy to make smart reinjection decisions. A large `recvWin` indicates a large number of out-of-order packets are buffered at the receiver side, because packets with smaller sequence numbers (likely being transferred over the path with performance degradation) are not received. Therefore, reinjection of unacknowledged packets when `recvWin` is large helps eliminate the sequence number gap and thus improve the performance. We have implemented the following proof-of-concept reinjection policy. A message[1] transmitted on one path is reinjected to the other path in either of the two conditions: (1) its underlying packet experiences a TCP timeout, or (2) an ACK from the receiver indicates that the receiver buffer occupancy exceeds $\eta\%$ of the total buffer size. Note in case (2) each unique ACK triggers reinjection of at most one unacknowledged message, and η is a threshold determining the reinjection aggressiveness. We empirically set it to 75%.

4.3 Evaluation of MPFlex

We conduct extensive evaluation of MPFlex. For performance, we compare MPFlex with MPTCP v0.89.5 (the latest version of MPTCP available for Android) with default settings. To support MPTCP with unmodified apps, we transparently tunnel all user traffic using Socks5 proxying (using `shadowsocks` [21]). We set up a multipath proxy running MPTCP v0.90 and redirect all the traffic to this MPTCP proxy. The Socks5 protocol [87] only adds a very small header to each packet so its impact on the traffic pattern is negligible. We also verified Socks5 incurs very small runtime overhead. All experiments were conducted using real WiFi and LTE networks on a Nexus 5 phone with Android 4.4.4. We use `tc` to apply bandwidth throttle and to add extra delay on both paths[2] based on recent large-scale measurements of metropolitan LTE [79] and WiFi [123] users. The same configurations were used by another recent MPTCP study [68]. For apples-to-apples com-

[1]Recall in §4.2.2 that a message is the atomic transfer unit in MPFlex. We configure its maximum size to be the TCP MSS.

[2]WiFi: uplink 2020kbps, downlink 7040kbps, RTT 50ms; LTE: uplink 2286kbps, downlink 9185kbps, RTT 70ms.

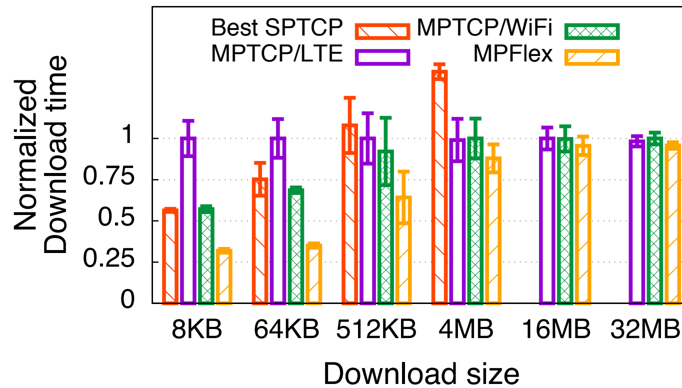


Figure 4.3: Single file download over MPTCP and MPFlex (best SPTCP results shown only for small downloads).

parison, MPFlex and MPTCP employ the same MinRTT scheduling algorithm, the same congestion control (decoupled Cubic *i.e.,* each path runs TCP Cubic independently), and the same proxy server unless otherwise noted. We next describe the evaluation results.

4.3.1 File Download

Download a single file. Figure 4.3 compares single file download time under three schemes: cellular-primary MPTCP, WiFi-primary MPTCP, and MPFlex, for different file sizes. Compared to the best MPTCP scheme, MPFlex reduces the download time by 11% to 49%, because of its simplified handshake procedure that makes better utilization of both paths.

Handling multiple short flows. We wrote a custom benchmark tool that generates small flows sequentially or concurrently. Figure 4.4 plots the overall download time for MPTCP and MPFlex under eight traffic patterns. Compared to downloading a single file, when handling *multiple* short flows, the advantage of MPFlex is more phenomenal with download time reduction ranging from 13% to 63%. As mentioned in §4.2.1, this is attributed to two features brought by MPFlex's strategic multiplexing: simplified handshake (same as the

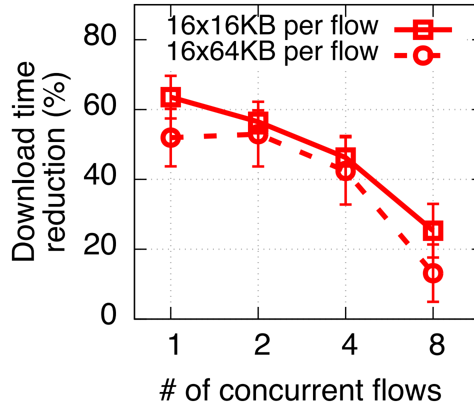


Figure 4.4: Transfer many short flows over MPTCP and MPFlex.

single file download case) and being capable of maintaining the congestion window[3] for multiple connections arriving in a bundle. We also note that the savings reduces a bit when the concurrency becomes higher due to improved bandwidth utilization of concurrent flows.

4.3.2 Web Browsing

How much performance gain can MPFlex offer under realistic applications and traffic patterns? To answer this question, we pick seven diverse websites and load their landing pages automatically with QoE Doctor [46] on Chrome browser on Nexus 5. To overcome frequent content change and server-side load fluctuation for some sites, we use Google Page Replay [10] to take a snapshot of each site, and host it on our replay server. To further make our setup realistic, we measure the RTT from proxy (MPTCP or MPFlex) to the real servers, and set the same RTT for the link between the proxy and our server when replaying each website.

The results are shown in Figure 4.5. Compared to Figure 4.4, the page load time (PLT) reduction is less, mostly due to the additional browser-side overhead (rendering page, parsing JavaScript *etc.*) and inter-object dependencies that often shift the bottleneck from network to local computation [127]. Nevertheless, the improvements are still impressive: com-

[3]Similar to a regular TCP connection, multiplexed connections in MPFlex still conservatively perform slow start after idle period.

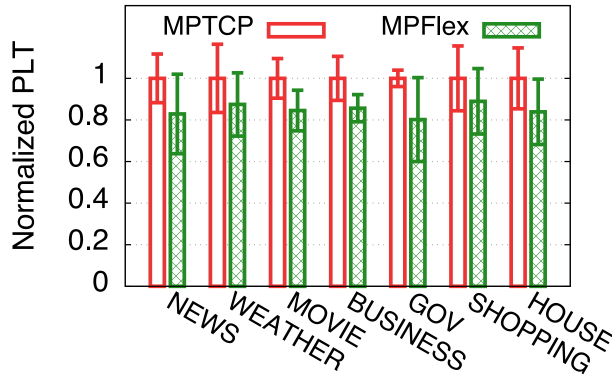


Figure 4.5: Fetch web pages over MPTCP and MPFlex.

pared to MPTCP proxy, MPFlex reduces the PLT by 7% to 20%. We also expect MPFlex will exhibit more advantages on newer mobile devices or tablets where computation is less likely to become the bottleneck.

4.3.3 Applying Multipath Policies

As described in §4.2.2, we have implemented a framework that allows applying different multipath policies at a per-process basis. We demonstrate its effectiveness of saving energy by conducting a case study as follows. We consider four apps: YouTube (playing a 150-second 1080p video), Skype (2-min VoIP call), Google Play (downloading a 50MB app), and Facebook Messenger (sending a message every 30 seconds). We enforce the following policy: use multipath for YouTube and Google Play, and single-path (WiFi) for Skype and Messenger. We compare the radio energy consumption of system-wide MPTCP (applied to all four apps) and MPFlex with the above application-aware multipath policy. As shown in Figure 4.6, MPFlex reduces the radio energy consumption for Skype and Facebook Messenger by 34% and 78%, respectively, while incurring no QoE degradation.

The policy framework can be extended to consider other factors such as billing and battery life. MPFlex can also enforce the policy at a finer granularity. Consider two use cases. (1) Let Chrome browser to use multipath only for `cnn.com`. (2) Disable multipath

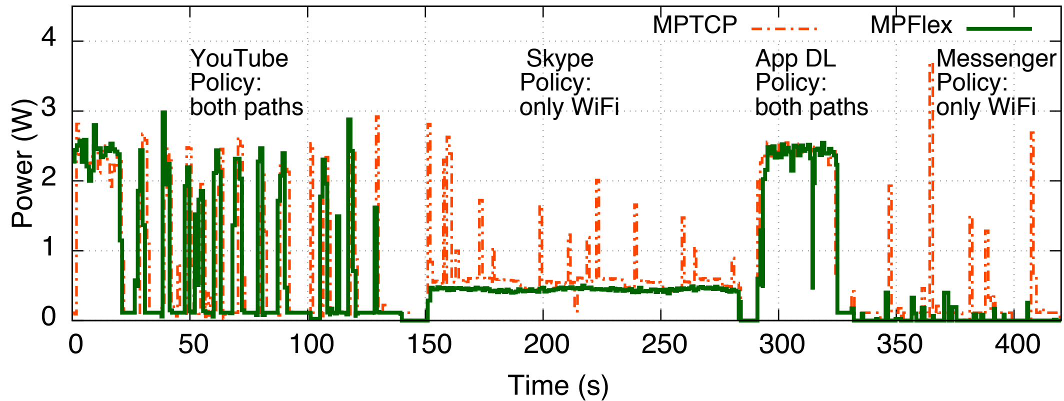


Figure 4.6: Case study: MPTCP applies multipath to all traffic, while MPFlex does that selectively based on user policy.

for all ad traffic. Both use cases can be realized by controlling multipath usage at a per-HTTP-session basis without modifying the apps. First, for an incoming HTTP session, the client-side MPFlex module obtains its associated domain name. This can be realized by examining directly the HTTP request or the TLS/SSL certificate for HTTPS. Second, it consults a local policy database to determine which multipath scheme to use. Third, the MPFlex client informs the proxy of the policy for this HTTP session using a small label attached to the multiplexed message (§4.2.2).

4.3.4 Plugging-in Custom Schedulers

We evaluate the two MPFlex plugins described in §4.2.3.

Buffer-aware Scheduling. We evaluate the performance of our TxDelay scheduler by comparing it with the MinRTT scheduling algorithm. As shown in Figure 4.7, when WiFi RTT is much smaller than LTE RTT, TxDelay significantly outperforms MinRTT by reducing the file download time by 21% to 54%. TxDelay essentially makes multipath performs at least as well as single-path for small files (assuming RTT estimation is accurate). On the other hand, when RTT of both paths are similar, TxDelay exhibits similar performance as MinRTT.

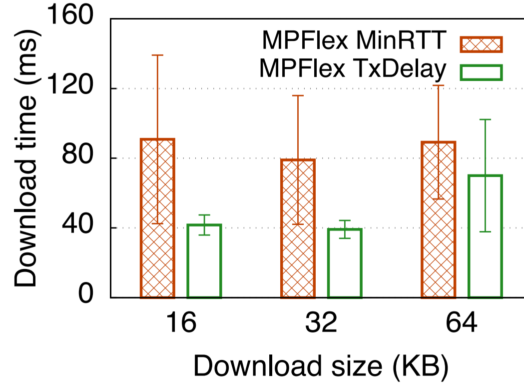


Figure 4.7: Performance of MinRTT vs. TxDelay scheduler when the RTT difference between the two paths is large (20ms vs. 70ms).

Smart Reinjection. We compare our smart reinjection with the default MPTCP and MPFlex without reinjection. The workload is to download a 1MB file hosted at a server near the MPFlex proxy (4ms RTT between proxy and server, as to be justified in §4.3.5). We consider two network conditions: increased LTE latency (WiFi 10Mbps/50ms, LTE 9Mbps/150ms) and increased WiFi latency (WiFi 12Mbps/300ms, LTE 9Mbps/70ms). We use the public WiFi network in our office building. We made two observations. First, during the entire course of our experiment, the default MinRTT scheduler never triggers reinjection, which is done very conservatively as described in §4.2.3. As a result, MPFlex without reinjection performs similarly compared to MPTCP. Second, smart reinjection reduces the overall download time by 10%±5% (over 12 runs) at the cost of reinjecting only about 1.5% of the total bytes, in both network conditions. The aggressiveness of reinjection can further be tuned by adjusting the buffer occupancy threshold.

4.3.5 Impact of Proxy Location

Existing work indicates the location of a CDN server (in our case here, the MPFlex proxy) may affect the network performance in particular for multipath [104]. To quantify this, we deployed the MPFlex proxy at 4 different locations referred to as A, B, C, and D. Their physical distances from our client smartphone are 3, 420, 3300, and 6700 km,

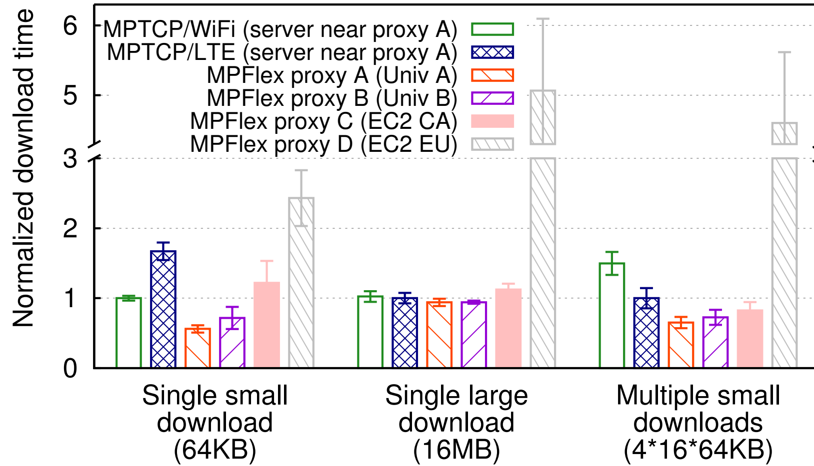


Figure 4.8: Performance impact of the MPFlex proxy location.

respectively. We also measured the minimum WiFi RTT between the mobile client and them to be 27, 44, 81, and 131ms, respectively, and the corresponding cellular RTTs are 49, 66, 100, and 148ms, respectively. We fix the RTT between the proxy and the server to be 4ms, by assuming the server is a nearby CDN node. A recently study measured the median RTT between a mobile carrier gateway and 30 popular content providers' servers to be ∼4ms [112]. We also evaluate a non-proxy configuration by letting the client directly connect to the server near Proxy A, which has the smallest latency from the client, using the default MPTCP.

We consider three workloads shown at the bottom of Figure 4.8. As shown, for small file download(s) whose performance is latency-sensitive, the proxy location matters. Nevertheless, despite being further away, Proxies B and C still achieve better or similar performance compared to the default MPTCP configuration, because the benefits of MPFlex outweigh the penalty of additional latency for B and C. Proxy D exhibits low performance because it is located at a different continent. The transoceanic link shifts the bottleneck from the last mile to the Internet. For a single large download, since bandwidth is more important than latency, Proxy A, B, and C exhibit very similar performance. Also, because the file size is large and no multiplexing is needed for a single application connection, MPFlex

provides little benefit beyond what can be achieved by the vanilla MPTCP.

4.3.6 System Overhead

Despite being realized mostly at user level, MPFlex itself incurs negligible runtime overhead. We monitor CPU usage of a Nexus 5 phone with MPFlex enabled. Compared to the default MPTCP, no noticeable CPU usage increase was observed when downloading large files at high speed (∼30Mbps). The MPFlex protocol overhead, defined as the total message header size divided by the size of all transferred messages, is measured to be less than 1% across 20 Android apps we tested. Multiple instances of MPFlex proxy can be deployed in geographically distributed clouds to achieve scalability.

Since the MCs in MPFlex are long-lived, periodic keep-alive messages may need to be exchanged between a client and the proxy. Their periodicity should be no longer than the minimum NAT/firewall timeout of either path (usually the cellular path). To quantitatively measure the radio energy overhead incurred by MPFlex, we conduct an experiment on a Nexus 5 phone by sending keep-alive messages over the cellular pipes every 10 minutes for 24 hours. We use the hardware energy profiling interface (provided through the `sysfs` file system) of Nexus 5 to measure the energy consumption. We then compare the total energy with the scenario where MPFlex is not running for the same duration of 24 hours. The incurred radio energy is 0.18Wh, only about 2.5% of a typical smartphone's battery capacity. Since most cellular carriers' NAT/firewall timeout is longer than 10 minutes [129], the incurred energy overhead can be further reduced by increasing the keep-alive periodicity.

4.4 Summary

We contribute to mobile multipath research with MPFlex, a flexible software architecture for multipath over mobile networks. We introduce the concept of multiplexing and customized policy in multipath transport to address the limitations of MPTCP. Through experimental evaluation, we demonstrate that MPFlex reduces the download time of short

flows and easily customizes the multipath usage of different mobile applications to save energy while guaranteeing good application QoE.

CHAPTER V

Accelerating Multipath Transport through Balanced Subflow Completion

MPTCP (or any transport-layer multipath scheme) has a complex protocol stack, consisting of several components: subflow management, packet scheduling, congestion control, flow control, *etc.*. The system design, implementation, and evaluation of MPFlex in the previous chapter have shown that MPFlex improves multipath performance by *(i)* multiplexing application data over long-lived subflows, *i.e.,* better subflow management, and *(ii)* adapting multipath usage to application traffic patterns, *i.e.,* customized multipath policy. In this chapter, we focus on optimizing the *schedulers*, which determine how the data is distributed onto the subflows.

5.1 Introduction

MPTCP supports different types of schedulers. For example, the *MinRTT* scheduler attempts to deliver the data as soon as possible by choosing a subflow with the smallest RTT unless its congestion window is full; the *ReMP* scheduler [59] boosts the reliability by duplicating packets over all subflows. There also exist schedulers that consider other dimensions such as energy efficiency [90], path priority [70] and receiver buffer occupancy [85].

Despite these efforts, we found that the multipath scheduler design is far from being optimal. In a pilot experiment conducted in §5.2, we observe that surprisingly, under representative WiFi/LTE network conditions, the MinRTT scheduler inflates the download time for a medium-sized file by up to 33% compared to the optimal scheduling decision derived offline. In real-world networks with fluctuating bandwidth or latency, MinRTT may perform even worse (up to 7.5x download time increase, 49% median increase compared to optimal scheduling) as shown in §5.6. Regarding the root cause, our key insight is that for such a file download, oftentimes the subflows do *not* complete at the same time at the *receiver* side. This inevitably leads to suboptimal performance: if Subflow A completes earlier than Subflow B, one could achieve a shorter overall download time by offloading some of the traffic from Subflow B to A. Therefore, achieving simultaneous subflow completion is a necessary condition for minimizing the data transfer time.

The key contribution of this chapter is the design, implementation, and evaluation of DEMS (**DE**coupled **M**ultipath **S**cheduler[1]), a new multipath packet scheduler aiming at reducing the data chunk download time over multiple paths. A *data chunk* is simply a block of application-defined bytes, which is a very common data transfer workload in a wide range of applications, *e.g.,* fetching an image, JavaScript, MP3 file, or video chunk. The key idea behind DEMS is to *achieve simultaneous subflow completion at the receiver side through strategic packet scheduling over decoupled subflows* in order to minimize the chunk download time. Accomplishing this seemingly straightforward task, however, faces several challenges. First, MPTCP by default treats the user data as a continuous stream without even knowing the chunk boundary, letting alone performing any optimization for it. Second, the scheduler works at the sender side while we need to balance the flow completion time at the receiver side so there exists a "visibility gap" between the sender and receiver. Third, the task is further complicated by the fluctuating network conditions in wireless networks. Our judicious design addresses all above challenges as follows.

[1]Note here "decoupled scheduling" is different from the decoupled congestion control in MPTCP.

- As a first prerequisite for optimizing the chunk download time, DEMS is aware of the boundary of a chunk. Since how the data within a chunk is delivered does not matter (as long as the chunk can be reassembled correctly), DEMS can employ flexible and efficient schemes to split the chunk over subflows. For example, if only two paths are involved, one path can send data from the beginning in the forward direction, and the other path sends data from the end in the backward direction. Doing so simplifies our design and facilitates the performance by decoupling the subflows (§5.3.1).

- We devise a technique to ensure simultaneous subflow completion on the receiver side. To achieve this, at the *sender* side, DEMS strategically introduces a timing offset between the two subflows with different RTTs so that the last packets across all subflows will arrive at the receiver at the same time. The timing offset is dynamically determined based on the network latency and bandwidth dynamics (§5.3.2).

- The fluctuation of bandwidth and latency may still cause some differences in completion times across the subflows. To minimize this negative performance impact, DEMS performs data *reinjection*: if one subflow finishes earlier, it can "help" other subflows by transmitting a small portion of data (typically towards the end of the chunk download) that was already assigned to another subflow. Such data may be redundant (*i.e.,* transferred twice over two subflows) but it helps further reduce the overall download time (§5.3.3). We develop a method that adaptively determines the amount of the redundant data to strike a sweet spot between the performance and the additional data transmission due to reinjection (§5.3.4).

DEMS can work with any data transfer size and/or traffic pattern. The ideal workload on which DEMS yields the highest benefits is downloading application data chunks, which are very common to mobile traffic workloads. A wide range of mobile applications such as web browsing and video streaming involve downloading such data chunks (*e.g.,* an HTTP object or a video segment).

We integrated the DEMS components into a holistic system (§5.4) and implemented it on commodity mobile devices (§5.5). We conducted extensive evaluations over both

emulated and real cellular/WiFi networks. The results indicate that DEMS is robust to diverse network conditions (including challenging multipath environments) and oftentimes brings significant performance boost compared to the state-of-the-art. Below highlights some key results.

• In stable network conditions, compared to MinRTT, DEMS reduces download time by up to 74%, 57% and 21% for 256KB, 1MB and 4MB download, respectively, under different combinations of delay/bandwidth of the paths. DEMS exhibits even better performance compared to ReMP, round-robin, and the best single path approach (§5.6.2, §5.6.5).

• In changing network conditions, DEMS reduces the median download time of 256KB and 1MB files by 12% to 46%, compared to MinRTT. Meanwhile, our adaptive reinjection scheme effectively reduces the redundant bytes by 48% to 86% compared to the naïve reinjection scheme while maintaining similar performance (§5.6.3).

• We conducted field tests at 5 real-world locations such as a parking lot and a grocery store. DEMS reduces the download time by up to 88%, 83% and 77% for 256KB, 1MB and 4MB file, respectively, compared to MinRTT (the median reductions are 33%, 48%, and 42%, respectively). The real-world results are even better than the in-lab emulation results. Compared to wired networks, wireless networks like WiFi and cellular can sometimes exhibit high delay and bandwidth dynamics [78, 82]. Our evaluation demonstrates that compared to existing multipath schedulers, DEMS is able to robustly handle such environments (§5.6.4).

• DEMS reduces the median web page load time (across 10 popular websites) by 6% to 43% (median: 25%) under real network conditions, compared to MinRTT (§5.6.6).

Overall, our findings indicate that *strategically performing decoupled packet scheduling with balanced subflow completion can significantly improve the multipath transport performance*, which translates into better user QoE and improved energy efficiency (due to shortened radio-on time [77]). The remaining sections will focus on the motivation (§5.2), algorithm design (§5.3), system integration (§5.4), implementation (§5.5), and evaluation

(§5.6) of DEMS. We discuss limitations in §5.7.

5.2 Background and Motivation

We reveal the performance inefficiency caused by MPTCP's scheduling algorithm in §5.2.1, which leads to the key principle of DEMS in §5.2.2: ensuring simultaneous subflow completion.

5.2.1 Can We Further Improve MinRTT?

Consider the common task of downloading a data chunk, which can be an image, a Javascript, an audio snippet, or a video chunk, over multipath. Our apparent goal is to minimize the download time. We conduct a pilot experiment to demonstrate (1) the impact of the scheduler on the download time, and (2) the potential room for improving MinRTT.

The experiment was conducted on a laptop with both WiFi and Ethernet connectivity, which emulate WiFi and LTE networks respectively using Linux `tc`. The network characteristics were chosen based on recent large-scale measurements of metropolitan LTE [78] and WiFi [123] users[2]. We use our user-level multipath testbed (to be described in §5.5) to provide the multipath support for the laptop.

We use the above setting to download a medium-sized file of {256KB, 1MB} using the MinRTT scheduler and an optimal scheduling computed offline as follows. We download $p\%$ of the file over WiFi and $1 - p\%$ over cellular. To find the best p that leads to the shortest (*i.e.,* optimal) download time, we perform an "exhaustive search" by conducting many experiments covering the full range of $p \in [0, 100]$. We also vary the latency difference between the two paths by inflating the emulated LTE path, as it is common to have diverse path characteristics in mobile networks [54]. Note that except for the scheduling algorithm, the MinRTT and the optimal schemes share the same configurations (*e.g.,*

[2]WiFi: uplink 2020kbps, downlink 7040kbps, RTT 50ms; LTE: uplink 2286kbps, downlink 9185kbps, RTT 70ms.

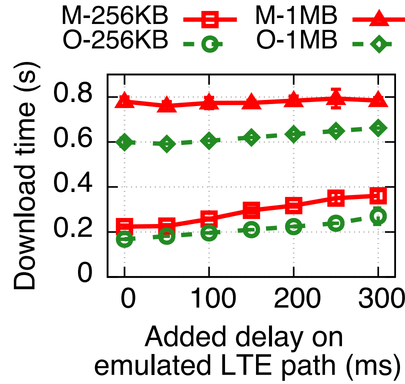


Figure 5.1: Compare chunk download time (M: MinRTT, O: Optimal).

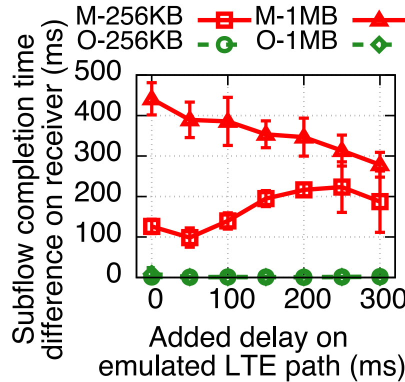


Figure 5.2: Compare subflow completion time on receiver (M: MinRTT, O: Optimal).

the initial congestion window and congestion control algorithm) to ensure apples-to-apples comparison.

The results are shown in Figure 5.1. As shown, the scheduling decision clearly impacts the performance. Surprisingly, compared to the optimal case, MinRTT increases the download time by up to 33%. Note our experiment assumes stable network condition, whereas in real-world networks with fluctuating bandwidth or latency, the gap between MinRTT and the optimal case can be even larger (up to 7.5x download time increase, 49% median increase compared to optimal scheduling) (§5.6). Also note that researchers have proposed other MPTCP schedulers such as deadline-aware scheduler [70], energy-efficient scheduler [90], real-time content scheduler [122], and buffer-blocking-aware scheduler [85].

They usually sacrifice chunk download time performance for other properties such as path priority and energy consumption, so we do not compare with them here.

By examining the results at the packet timing level, we identified a key reason why MPTCP yields suboptimal performance to be that the subflows do not complete at the same time at the receiver side. Figure 5.2 plots the two subflows' completion time difference when using the two schedulers. For MinRTT, the difference ranges from 100ms to 450ms while in the optimal scheme, the last bytes on the two subflows almost always arrive at the receiver simultaneously.

5.2.2 Ensuring Simultaneous Subflow Completion and its Challenges

Having all subflows complete at the same time at the receiver side is a necessary condition for achieving the optimal performance. The reason can be easily shown through *proof by contradiction*: suppose in an optimal scheme, Subflow A finishes earlier than Subflow B; in that case Subflow B can further "offload" some bytes to Subflow A, leading to an even shorter download time. But why cannot the MinRTT scheduler achieve this same-time-completion property? The reasons are multi-fold as explained below.

- When making a scheduling decision, MinRTT only considers the latency of each subflow without taking into consideration the bandwidth. Oftentimes, despite one subflow having a higher RTT than other subflows or even having a full congestion window (so it is temporarily unavailable), its higher bandwidth allows it to drain data from its sender buffer quickly. As a result, choosing the higher-RTT subflow can actually lead to lower end-to-end latency.
- Under practical network conditions (wireless in particular), the RTT and bandwidth are often highly fluctuating, leading to unbalanced subflow completion time. This factor is largely not taken into account by MinRTT.
- MinRTT is stateless in that the scheduling decision of the current packet does not explicitly depend on the previous packets. We will show that by strategically leveraging the information of previously transmitted packets, the scheduling decisions and thus the overall

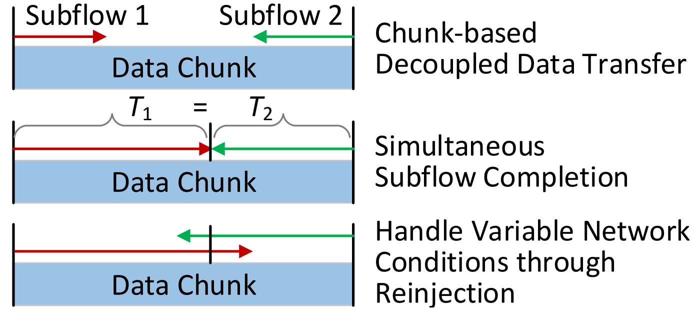


Figure 5.3: Key design decisions of DEMS.

performance can be improved.

5.3 The DEMS Algorithm

We propose DEMS, a new scheduling algorithm for reducing the data chunk download time over multipath. As shown in Figure 5.3, the key design decisions of DEMS include the following. (1) DEMS is aware of the chunk boundary, and it strategically decouples the paths for chunk delivery (§5.3.1). (2) DEMS ensures simultaneous subflow completion at the receiver side (§5.3.2). (3) DEMS allows a path to trade a small amount of redundant data for performance (§5.3.3, §5.3.4). We next elaborate them in this section.

5.3.1 Chunk-based Data Transfer

In DEMS, by default, data is delivered to the application on a *per-chunk* basis. A (data) chunk consists of a block of bytes defined by the application, which can be, for example, an image, a Javascript, an audio snippet, or a video chunk. At the sender side, after the sender app pushes the chunk to the multipath meta buffer, DEMS treats all data in the meta buffer as a chunk by default, thus being fully transparent to applications. At the receiver side, when the chunk is fully received, it is then delivered to the application. We will describe in §5.4 implementation alternatives for making DEMS aware of the chunk boundaries informed by applications through a simple API.

As long as a chunk can be correctly reassembled, bytes within the chunk can be delivered in any order. The data chunk is thus split into different parts that are distributed onto different paths for delivery. For the common scenario involving two paths, we design a "two-way" splitting approach: the two paths transfer the data in opposite directions, one from the beginning and the other from the end; when they "meet" each other, the chunk is fully downloaded. This approach is intuitive and parameterless. Furthermore, it helps improve the multipath performance by *decoupling* the two subflows. In conventional MPTCP, subflows are tightly coupled; a stall (*e.g.,* due to packet loss) in one subflow may slow down other subflows due to their limited and shared meta receive window whose size is difficult to set [104]. DEMS, on the other hand, decouples the two subflows by allowing each subflow to freely and independently transfer the data until the very end when subflows meet and merge. The receive window (16 MB by default) is configured to be larger than a typical chunk size so it will not become a performance bottleneck during a chunk transmission [104]. In rare cases when the application data is larger than the receive window, the data will be split into multiple chunks for transmission.

5.3.2 Simultaneous Subflow Completion

Now let us consider how to simultaneously complete subflows at the receiver side. Recall that the high-level idea is to introduce a timing offset at the sender to compensate the heterogeneous delay across both subflows.

Let us first assume that the one-way delay (OWD) of both subflows can be accurately predicted. Let OWD_1 and OWD_2 be the OWD of Subflow 1 and 2, respectively, where $OWD_2 > OWD_1$. Let t_{s1} and t_{s2} be the time when the last byte is transmitted over Subflow 1 and 2, respectively. Let t_{r1} and t_{r2} be the time when Subflow 1 and 2 receives the last byte, respectively, *i.e.,* the subflow completion time at the receiver. If no packet reordering or loss happens, we have:

$$t_{\text{r1}} - t_{\text{s1}} = OWD_1 + t_{\text{offset}} \tag{5.1}$$

$$t_{r2} - t_{s2} = OWD_2 + t_{offset} \tag{5.2}$$

where t_{offset} is the clock difference between the sender and receiver (handling network condition fluctuation caused by packet losses/reordering will be discussed in §5.3.3 and §5.3.4). Thus, the subflow completion time difference is:

$$t_{r2} - t_{r1} = (t_{s2} - t_{s1}) + (OWD_2 - OWD_1) \tag{5.3}$$

Clearly the receiver-side subflow completion time difference depends on both the sender-side transmission completion time difference denoted as $\Delta t_s = t_{s2} - t_{s1}$ and the forward-path one-way delay difference denoted as $\Delta OWD = OWD_2 - OWD_1$. To ensure simultaneous subflow completion *i.e.,* $t_{r2} = t_{r1}$, we need:

$$t_{s1} - t_{s2} = OWD_2 - OWD_1 = \Delta OWD \tag{5.4}$$

Namely, at sender side, the subflow with a larger OWD must finish transmission ΔOWD earlier than the other subflow.

Recall that DEMS employs the "two-way" data split scheme where the two subflows start from the two ends of the chunk and move towards each other. This process is illustrated in Figure 5.4, which plots the *sender-side* view of the data transfer progresses at both subflows (the X axis is time and Y axis is the sequence number). Assuming the constant bandwidth, the data transfer curves are linear. Subflow 2 stops transmission at t_{s2}. Subflow 1 then spends ΔOWD time to transmit all its data, finishing at t_{s1}. To translate this into the *receiver-side* view is easy: by shifting the two curves towards the right by OWD_1 and OWD_2 respectively, they will meet at $t_{s1} + OWD_1 = t_{s2} + OWD_2$, indicating they complete simultaneously at the receiver side.

Now we describe how to incorporate the above idea into the DEMS scheduler design, which needs to handle two tasks: assigning each packet (with its byte range) to a subflow

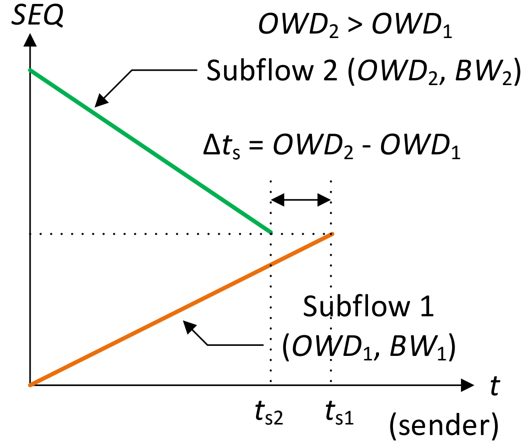


Figure 5.4: Achieve simultaneous subflow completion (sender-side view).

and deciding when to stop a subflow at the sender. The former is trivial as the chunk has already been split in "two-way". So we focus on the latter task. The basic idea is as follows. When the subflow with a larger OWD can transmit a packet over the network, we estimate its arrival time over both subflows, and then choose the subflow with an *earlier* packet arrival time for actual transmission. We next show that this strategy will make the large-OWD subflow stop transmission when Equation (5.4) holds, thus resulting in simultaneous subflow completion shown in Figure 5.4.

Consider a general scenario depicted in Figure 5.5 where Subflow 2 with a larger OWD can now transmit a byte whose sequence number is $i = SEQ_2$. Should this byte be immediately transmitted over Subflow 2 (marked as ❷) or later over Subflow 1 (marked as ❶) so that Subflow 2 can stop now? If we choose Subflow 2, the byte's estimated arrival time at the receiver side is:

$$est(t_{\text{r2}i}) = t_{\text{s2}i} + OWD_{2i} + t_{\text{offset}} \tag{5.5}$$

where $t_{\text{s2}i}$ is the current sender-side timestamp and OWD_{2i} is the current estimation of OWD_2. Now consider choosing Subflow 1 to transmit SEQ_2. Since Subflow 1 is currently working on a byte with a smaller sequence number SEQ_1, SEQ_2 has to be buffered in the meta buffer and gets transmitted at ❶ after all bytes from SEQ_1 to $SEQ_2 - 1$ are

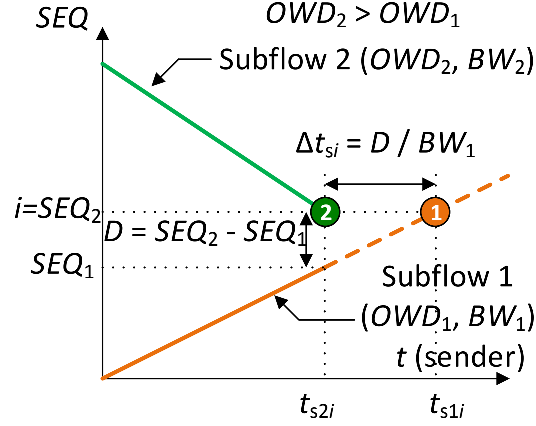


Figure 5.5: Choose a subflow with an earlier estimated data arrival time (sender-side view).

transmitted (over Subflow 1). Therefore, the estimated arrival time of SEQ_2 over Subflow 1 is:

$$est(t_{\mathrm{r}1i}) = t_{\mathrm{s}2i} + \Delta t_{\mathrm{si}} + OWD_{1i} + t_{\mathrm{offset}} \tag{5.6}$$

The buffering delay, Δt_{si}, is computed as:

$$\Delta t_{\mathrm{s}i} = \frac{SEQ_2 - SEQ_1}{BW_{1i}} \tag{5.7}$$

where BW_{1i} is Subflow 1's current bandwidth estimation.

As said, we stop Subflow 2 and choose Subflow 1 when the estimated byte arrival time of the larger-OWD subflow (Subflow 2) is later than the smaller-OWD subflow (Subflow 1) *i.e.*, $est(t_{\mathrm{r}2i}) > est(t_{\mathrm{r}1i})$. In this situation, we have:

$$\Delta t_{\mathrm{si}} = t_{\mathrm{s}1i} - t_{\mathrm{s}2i} < OWD_{2i} - OWD_{1i} \tag{5.8}$$

This is the same criteria as that in Equation (5.4), which guarantees the same receiver-side subflow completion time. Next, plugging Equation (5.7) into (5.8) yields:

$$D = SEQ_2 - SEQ_1 < (OWD_{2i} - OWD_{1i})BW_{1i} \tag{5.9}$$

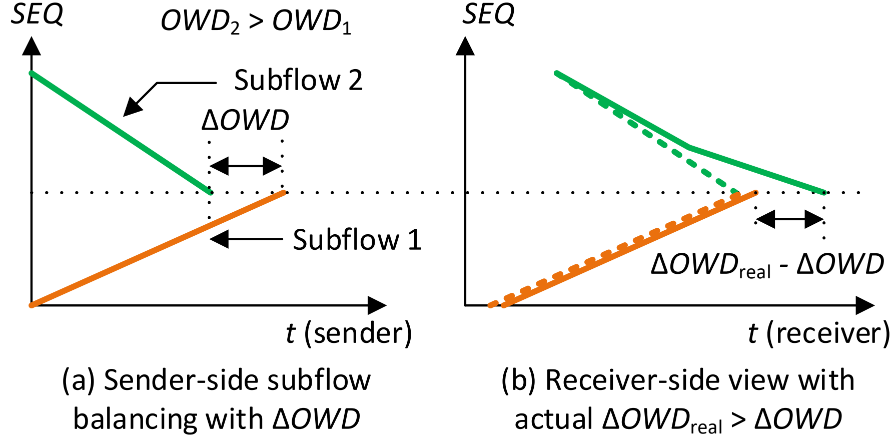


Figure 5.6: Impact of inaccurate ΔOWD estimation.

D corresponds to the number of bytes remaining to be transmitted in the meta buffer (imagine SEQ_1 and SEQ_2 as two "pointers" moving towards each other). We use Equation (5.9) in our system to determine when to stop the transmission for the larger-OWD subflow as the two-way chunk download approaches to its end. Note the small-OWD subflow can always transmit new packets before meeting with the large-OWD subflow, as long as the congestion window allows.

5.3.3 Handling Variable Network Conditions

So far we assume that both subflows' OWD and the smaller-OWD subflow's bandwidth (*i.e.,* those on the right-hand side of Equation (5.9)) can be accurately predicted. Apparently this assumption does not always hold in practice in particular for wireless networks. Figure 5.6 illustrates the impact of inaccurate network condition estimation. At the sender side shown in Figure 5.6(a), DEMS makes scheduling decisions based on its estimated ΔOWD. Now assume Subflow 2's OWD increases over time, causing the sender to underestimate ΔOWD. In consequence, at the receiver side shown in Figure 5.6(b), the actual data reception time over Subflow 2 (in solid line) deviates from the expected time shown in the dashed line. As a result, Subflow 1 completes early than Subflow 2, leading to suboptimal chunk download time. Similar reasoning can be made for ΔOWD overestimation

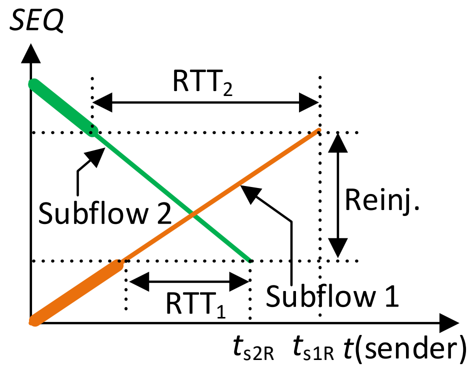


Figure 5.7: A simple reinjection scheme.

and inaccurate bandwidth estimation.

DEMS tolerates unbalanced completion of subflows under variable network conditions. It also takes a key design decision of performing *reinjection*: instead of having a full stop when all bytes of a chunk are transmitted, a subflow may further "overshoot" its portion by sending a small number of bytes that are beyond the meeting point of the two subflows, as illustrated in Figure 5.7. These bytes are redundant because they have already been transmitted over the other subflow. The purpose of reinjection is to trade redundant data for better performance: under uncertain network conditions, if the reinjected (redundant) data arrives earlier than its original copy, the overall download time is reduced.

A key challenge here is to carefully control how many bytes to reinject: reinjecting too little data incurs suboptimal performance, while reinjecting too much causes unnecessary battery drain or data plan usage (for cellular networks). In the extreme case adopted by MPTCP's redundant scheduler (ReMP) that duplicates *every* byte onto the secondary flow, MPTCP falls back to the "best single path" approach.

In DEMS, reinjection only occurs near subflows' meeting point. We first present a simple reinjection approach that gives a reasonable "upper bound" for the number of reinjected bytes. In this approach, after the two subflows meet, they perform reinjection in their corresponding directions until all bytes of the chunk are either acknowledged or reinjected. As shown in Figure 5.7, Subflow 2 keeps reinjecting data until t_{s2R} when it hits the byte that

was transmitted and acknowledged over Subflow 1 (it takes RTT_1 for the ACK to arrive). Similarly, Subflow 1 stops reinjection at t_{s1R} when the remaining data has been acknowledged by Subflow 2. At t_{s1R}, any byte within the chunk has either been acknowledged (shown as thick lines in Figure 5.7, or has a redundant copy in flight (shown as thin lines). By analyzing the "X" shape in Figure 5.7 (assuming the last byte on each subflow is not lost or reordered), we can compute the total number of reinjected bytes to be:

$$Reinj = \frac{BW_1 BW_2}{BW_1 + BW_2}(RTT_1 + RTT_2) \tag{5.10}$$

where BW_1 and BW_2 are the bandwidth of the subflows. Note they are the slopes of the two lines in Figure 5.7.

Equation (5.10) gives a reasonable upper bound of the reinjection overhead, which however is still too high as to be evaluated in §5.6. More importantly, this reinjection approach is not adaptive in that the reinjection behavior remains the same regardless of the network condition fluctuation. Ideally, when the fluctuation is low (high), we should reinject less (more) data given that the subflow completion time balancing technique introduced in §5.3.2 is more (less) reliable.

5.3.4 Adaptive Reinjection

We design a scheme that performs adaptive reinjection while maintaining good performance. Let us first consider a scenario where ΔOWD is underestimated: its real value, denoted as ΔOWD_{REAL}, is ΔOWD (the estimated version) plus δ. We use OWD_1 and OWD_2 to denote the estimated OWD for each subflow, and use $OWD_{1,\text{REAL}}$ and $OWD_{2,\text{REAL}}$ to denote their real values, respectively. Since what really matters is the prediction accuracy of the *difference* between OWDs (*i.e.,* ΔOWD), without loss of generality we assume $OWD_{1,\text{REAL}} = OWD_1$ and $OWD_{2,\text{REAL}} = OWD_2 + \delta$, for the ease of presentation.

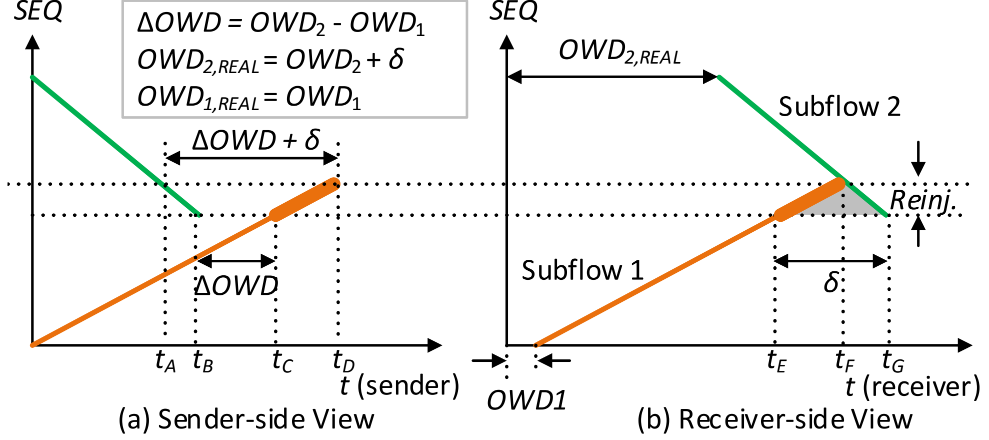


Figure 5.8: The adaptive reinjection scheme. All OWD values are exaggerated in the plot for illustration purpose.

Under the above setting, let us first examine the sender side illustrated in Figure 5.8(a). Based on the completion time balancing technique introduced in §5.3.2, Subflow 1 and 2 would stop at t_C and t_B respectively where $t_C - t_B = \Delta OWD$. However, ΔOWD is underestimated. As a result, if we switch to the receiver's view in Figure 5.8(b), we will see that Subflow 1 completes δ time units before Subflow 2. This is because (assuming there is no clock drift between sender and receiver) $t_E = t_C + OWD_1$ and $t_G = t_B + OWD_{2,\text{REAL}}$, so $t_G - t_E = (OWD_{2,\text{REAL}} - OWD_1) - (t_C - t_B) = \delta$. Now let us consider how to perform reinjection. In Figure 5.8(b), to minimize the download time, Subflow 1 only needs to keep reinjecting data until it meets Subflow 2. The reinjected portion is highlighted in bold. Now we derive the sender-side reinjection policy by shifting the reinjected portion back to Figure 5.8(a). When Subflow 1 reinjects the last byte at t_D, the byte's original transmission time (by Subflow 2) is t_A. As shown, $t_D - t_A = (t_B - t_A) + (t_C - t_B) + (t_D - t_C) = (t_G - t_F) + (t_C - t_B) + (t_F - t_E) = \Delta OWD + \delta$, which is the threshold for stopping reinjection. In other words, when ΔOWD is *underestimated* by δ, the *smaller-OWD* subflow keeps reinjection until the to-be-reinjected byte was transmitted more than $\Delta OWD + \delta$ time units ago. That is (using the notions introduced in §5.3.2):

$$\Delta t_{si} = t_{s1i} - t_{s2i} > OWD_{2i} - OWD_{1i} + \delta \tag{5.11}$$

Let the number of the reinjected bytes be r. By examining the gray triangle in Figure 5.8(b), we have $r/BW_1 + r/BW_2 = \delta$, which leads to $r = \delta BW_1 BW_2/(BW_1 + BW_2)$ where BW_1 and BW_2 are the subflows' bandwidth.

The counterpart scenario of ΔOWD overestimation can be derived in a similar way (proof omitted): when ΔOWD is *overestimated* by δ, the ***larger-OWD*** subflow keeps doing reinjection until the to-be-reinjected byte was transmitted less than $\Delta OWD - \delta$ time units ago, which is:

$$\Delta t_{si} = t_{s1i} - t_{s2i} < OWD_{2i} - OWD_{1i} - \delta \tag{5.12}$$

The reinjection overhead is also $r = \delta BW_1 BW_2/(BW_1 + BW_2)$. In reality we can only estimate (with a certain level of confidence) that ΔOWD_{REAL} falls in the range of $[\Delta OWD - \delta, \Delta OWD + \delta]$. DEMS thus lets both subflows to reinject according to Equation (5.11) and (5.12). The overall reinjection overhead is:

$$Reinj = 2\delta \frac{BW_1 BW_2}{BW_1 + BW_2} \tag{5.13}$$

These redundant bytes also help tolerate the inaccurate bandwidth prediction for the smaller-OWD subflow, as well as help recover from packet losses (within the reinjected byte range) quickly. Compared to (5.10), Equation (5.13) is usually much smaller. It is also adaptive as it is a function of δ that can be configured based on the predictability of the network condition. For example, δ can be set to $k \cdot StdDev(\Delta OWD)$.

5.3.5 Put Everything Together

We now walk through Algorithm 1, which combines chunk-based transfer (§5.3.1), subflow completion time balancing (§5.3.2), and adaptive reinjection (§5.3.4). The scheduling algorithm works at the sender side (the receiver-side logic is trivial). The input consists of the two network paths and a meta buffer that stores the packetized chunk data to be transmitted. The algorithm is invoked whenever either subflow can transmit a packet (*i.e.,* has

Algorithm 1: The DEMS scheduling algorithm

```
   Input: subflow i ∈ {1, 2} that can transmit packet, packets in the meta buffer
          metaBuf[j], j ∈ [0, m].
   Output: packet packet to transmit over subflow i.
 1 packet ← GetNextUnAckedPacketOnThisSubflow(i);
 2 smallOwdNo ← GetSmallOWDSubflowNo();
 3 largeOwdNo ← GetLargeOWDSubflowNo();
 4 ΔOWD ← GetΔOWD();
 5 δ ← GetΔOWDVar();
 6 trans ← false;
 7 if i == smallOwdNo then
 8     if not packet.tx [i] then
 9         trans ← true;
10     else
11         delay ← ts − packet.ts [largeOwdNo];
12         if delay ≤ ΔOWD + δ then
13             trans ← true;
14 else
15     bw ← GetSubflowBW(smallOwdNo);
16     thres ← (ΔOWD − δ) ∗ bw;
17     if GetUntransmittedSize(metaBuf) ≥ thres then
18         trans ← true;
19 if trans then
20     Transmit(packet);
21     packet.ts [i] ← GetCurrTimestamp();
22 else
23     packet ← NULL;
```

some empty congestion window space). It makes a decision of transmitting a new packet, reinjecting a previously transmitted packet, or withholding transmission. Also note it only processes unacknowledged packets.

Line 7–13 handles the subflow with a smaller OWD. Recall that in §5.3.2, the default behavior is to always transmit a new packet over a small-OWD subflow whenever possible. Line 11–13 deals with the reinjection scenario according to Equation (5.11). Line 15–18 processes the subflow with a larger OWD. According to Equation (5.9), we may need to skip the large-OWD subflow to achieve simultaneous subflow completion. Line 16

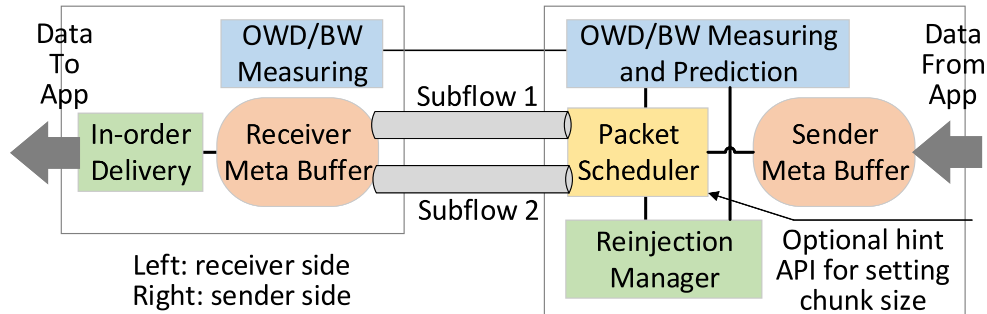


Figure 5.9: System diagram of DEMS.

performs adaptive reinjection over the large-OWD subflow according to Equation (5.12).

5.4 System Design

We now elaborate on how to integrate the DEMS algorithm into a real system. Figure 5.9 plots the system diagram. At the sender side (right), the chunk data coming from the application is stored in the meta buffer, and is then split, scheduled, and transmitted by the packet scheduler. Working with the packet scheduler, the reinjection manager keeps track of packets' transmission states and makes decisions on adaptive reinjection. We also design a module for measuring and predicting network conditions (ΔOWD and bandwidth). The application can also optionally specify the chunk size through an API (to be elaborated next). The receiver side logic is much simpler. It passively receives/acknowledges the data, reassembles it in the receiver-side meta buffer, and delivers the in-order data to the application. Note that data over the (decoupled) subflows are acknowledged separately using the per-subflow ACK numbers. Meanwhile, similar to MPTCP, at the receiver meta buffer, the global sequence number carried by each packet is used to mark which portion within the chunk has been received. While Figure 5.9 illustrates one-way data transfer, our system supports full-duplex data transmission.

Interaction with Applications. DEMS provides applications with an optional hint API (through a socket option) to specify the chunk boundary, allowing DEMS to work on the

chunks sequentially and to minimize the download time for each of them. Alternatively, if no hint is provided, DEMS leverages a heuristic that treats all data in the meta buffer as a single chunk. One issue here is that more data may arrive at the tail of the meta buffer from the app when the transmission is in progress. To handle this, we modify the packet fetch function (Line 1 in Algorithm 1) to let one subflow always fetch unacknowledged packets from the head of the meta buffer and the other subflow fetch from the tail of the meta buffer. The meta buffer is realized using a circular queue to allow efficient space reuse. In this way, DEMS is fully transparent to both the client-side and server-side applications. When there are multiple connections transmitting data simultaneously, DEMS can use a separate meta buffer for each connection and take bytes from each meta buffer in a round robin manner so that each connection is given a fair share of the network bandwidth. Other scheduling strategies such as flow prioritization can also be realized in DEMS.

ΔOWD **Measurement and Prediction.** OWD measurement requires cooperation between the sender and the receiver. The sender records the timestamp of each outgoing packet; the receiver records the reception time and sends it and the sequence number back to the sender through an encapsulated control message (§5.5). OWD is then estimated at the sender using exponential weighted moving average (EWMA) with α empirically chosen to be 0.25. Note that estimated OWD contains the sender-receiver clock drift, which is nevertheless cancelled out when a ΔOWD sample is calculated between two network paths. Since DEMS requires ΔOWD between two paths instead of the actual OWD on each path to make multipath scheduling decision as mentioned in §5.3, DEMS is not affected by the the sender-receiver clock drift. Also, in wireless networks (cellular in particular), network latency is often correlated with the number of bytes-in-flight (BIF) [63, 82, 138]. We thus bin OWD samples for each subflow separately using measured BIF with a bin size of 10KB. The samples in each bin are processed separately using EWMA to facilitate a more accurate OWD estimation for each subflow at different BIF levels.

We set δ, the parameter controlling adaptive reinjection (§5.3.4), to be the variance of

ΔOWD. For each network path, when the sender receives an OWD sample, a delay difference sample is calculated between this OWD sample and the estimation of OWD based on the corresponding BIF of this path. Similar to OWD estimation, the sender uses EWMA with α empirically chosen to be 0.25 over the delay difference samples to estimate the variance of OWD on the corresponding network path. The variance of ΔOWD is then estimated by the average of the variance of OWD of two network paths over time. The subflow bandwidth is also estimated using EWMA over measured samples. Note DEMS only needs the bandwidth estimation for the subflow with a smaller OWD (see Equation (5.9)). DEMS can also incorporate more sophisticated prediction methods (*e.g.,* those taking special consideration of network uncertainties [75]) to improve the prediction accuracy.

Congestion Control and Packet Losses. DEMS is compatible to any congestion control (CC) algorithm such as decoupled CC, LIA [132] and OLIA [84]. In our experiments we use decoupled CC that mobile multipath typically uses [68, 104]. Also, DEMS is robust to packet losses. Since the network condition prediction is performed in an online manner, delay or bandwidth fluctuation caused by (real or spurious) packet losses can be quickly picked up and reflected in the scheduling decision changes. We demonstrate in §5.6.4 that DEMS works well under diverse real-world scenarios including those with poor network conditions.

5.5 Implementation

DEMS can be directly integrated into MPTCP as a new scheduler. However, from the perspective of conducting real-world evaluations, MPTCP has two issues: first, most of today's commercial Internet servers do not yet support MPTCP, making testing real workload difficult; second, MPTCP uses special TCP options that are often blocked by commercial cellular middleboxes [104].

To facilitate real-world test, we implemented a multipath TCP proxy infrastructure in C/C++. Between the proxy and the client host, multipath is realized as multiple con-

ventional TCP connections each corresponding to a subflow established over a network path. For uplink traffic, at the client side, an application TCP connection's data is transparently split over the subflows using a custom light-weight encapsulation protocol (as opposed to using special TCP options); the data is then merged at the proxy and delivered to the server over conventional single-path TCP to ensure server transparency (assuming the client-proxy paths are the bottleneck). The downlink traffic is handled symmetrically. We have replicated MPTCP's three schedulers: MinRTT, round-robin, and ReMP by precisely following their algorithms. Compared to MPTCP, our proxy-based approach offers similar performance (based on our lab test[3]) while providing server transparency and middlebox friendliness.

We then implemented DEMS on our multipath proxy infrastructure. Most of the scheduling logic is implemented in the user space. Some low-level functionalities such as OWD/bandwidth measuring and prediction are implemented in the kernel through a light kernel module. Overall DEMS is lightweight: our implementation (not including the base proxy system) consists of 970 LoC (450 LoC for the scheduler and 520 LoC for network condition measurement/prediction). DEMS is generally compatible with Linux-based systems.

5.6 Evaluation

We systematically evaluate DEMS under various settings including different network setups (emulated vs. real networks), different network conditions (stable vs. fluctuating), different workload (raw chunk download vs. real application workload), different clients (smartphone vs. laptop), *etc.*

[3]In the test, the server is co-located with the proxy to ensure MPTCP and our approach traverses the same network paths.

5.6.1 Experimental Setup and Methodology

We study three variants of DEMS: DEMS-B, DEMS-S, and DEMS-F. They all employ chunk-based data transfer, decouple the subflows in the "two-way" manner, and attempt to ensure simultaneous subflow completion (§5.3.1 and §5.3.2). Their differences are the reinjection policy. DEMS-B ("Basic") does not perform reinjection; DEMS-S ("Simple reinjection") employs the naïve reinjection strategy described in §5.3.3; DEMS-F ("Full") performs the adaptive reinjection described in §5.3.4.

Our evaluation focuses on file download (upload can be handled by DEMS symmetrically). We set up our multipath proxy (developed in §5.5) on a commodity server with 4-core 3.6GHz CPU, 16GB memory, and 64-bit Ubuntu 16.04. The meta buffers at both the proxy and the receiver side are configured to be sufficiently large to avoid performance degradation due to limited buffer size. For apples-to-apples comparison, all schedulers use the same decoupled TCP congestion control (TCP CUBIC) and same subflow-level TCP send/receiver buffer sizes (8MB by default). Unless otherwise noted, the application server (hosting a file or web contents) is near the proxy, and uses only single-path. The RTT between the proxy and the app server is configured to be 4ms, which is the median RTT between a mobile ISP gateway (where our proxy can be deployed) and 30 popular content providers' servers based on a recent measurement study [112]. The client-proxy paths covering the "last-mile" wireless hops are thus the bottleneck.

We evaluated DEMS over both emulated and real multipath environments of WiFi and cellular. For emulation, we use Linux `tc` to throttle the bandwidth and to add extra delay on the client-proxy paths. By default, we use the network condition profiles from large-scale measurements of metropolitan LTE [78] and WiFi [123] users (same numbers as those used in §5.2.1). To emulate in-network buffering, we keep a 50ms bottleneck buffer for WiFi and a 500ms bottleneck buffer for LTE (set based on [82]) using `tc`.

We use two devices for evaluation: a laptop and a smartphone. The laptop is an HP EliteBook 840 with 1.90GHz dual-core CPU and 8GB memory, running Linux 3.18. When

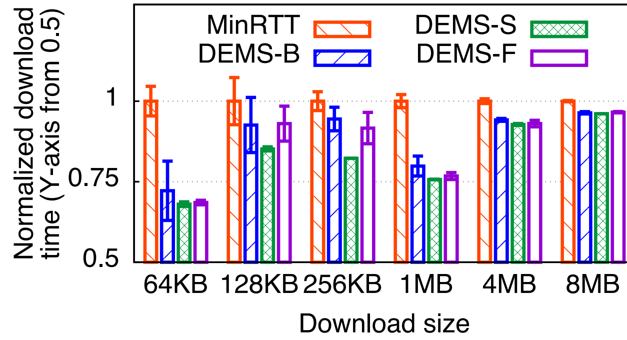
doing emulations over the laptop, we use Ethernet to emulate the LTE network, and use WiFi as it is. The smartphone is a Nexus 6P running Android 7.0 on Linux 3.10. We use its WiFi and cellular interfaces for experiments.

Recall in §5.4 that an application can interact with DEMS either using or not using a hint API. We adopt the non-API mode so DEMS is fully transparent to our applications. Finally, note that all evaluations were conducted on our multipath proxy infrastructure instead of MPTCP (recall in §5.5 that we replicated all MPTCP's schedulers on our infrastructure).

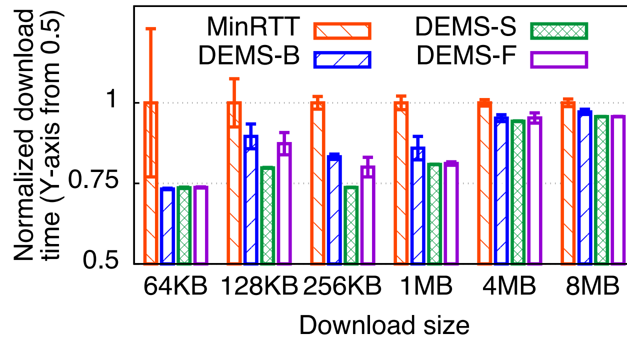
5.6.2 Stable Network Conditions

We first study the performance of DEMS under stable network conditions using emulation. The workload is file download. For each test, we repeat the download for 10 times and report the average value. We consider different delay differences and bandwidth of the two paths. Unless otherwise noted, we use MinRTT, the default MPTCP scheduler, as the comparison baseline.

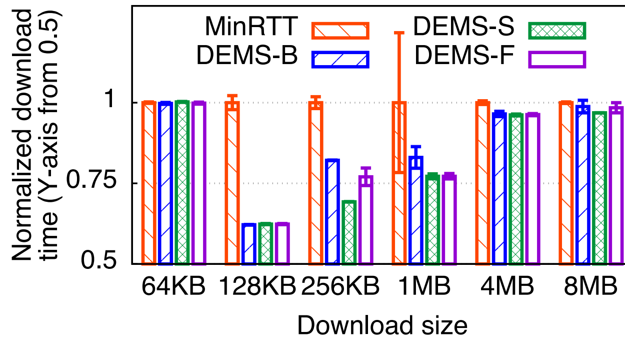
Different Delay Differences. Figure 5.10 compares the download time across the three DEMS variants and MinRTT for different file sizes. We consider three RTT combinations to cover different subflow delay differences. All three DEMS variants outperform MinRTT. Among the three variants, DEMS-S achieves the highest download time reduction (15% to 38% compared to MinRTT) for medium-sized files (128KB to 1MB) because of its aggressive reinjection behavior. The performance gain brought by DEMS-F decreases slightly in most cases. However, Figure 5.11, which compares the size of reinjected data incurred by DEMS-S and DEMS-F, clearly indicates that DEMS-F strikes a much better balance between the performance and the reinjection overhead: compared to DEMS-S, DEMS-F reduces the reinjected data size by 23% to 100%. Regarding DEMS-B, it slightly falls behind DEMS-F by up to 8%. The difference is small because the network conditions are stable here.



(a) WiFi RTT 50ms, LTE RTT 70ms



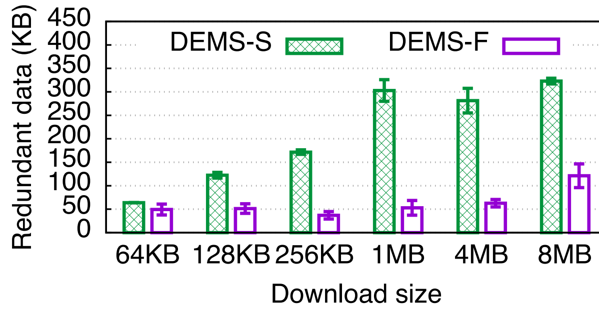
(b) WiFi RTT 50ms, LTE RTT 270ms



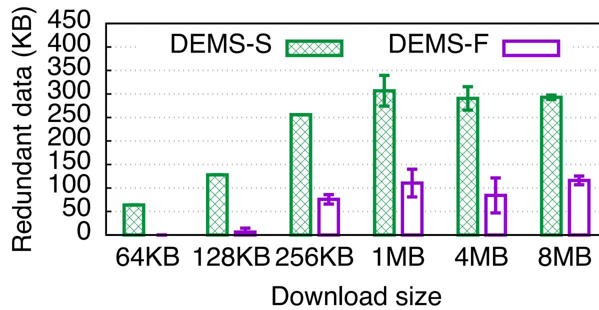
(c) WiFi RTT 50ms, LTE RTT 470ms

Figure 5.10: Compare performance between DEMS and MinRTT on downloading files with different sizes (laptop, emulation). The WiFi and LTE bandwidth are 7040kbps and 9185kbps respectively.

Figure 5.10 indicates that DEMS brings more download time reduction as the two paths' RTT difference becomes larger. This is because a larger RTT difference usually indicates a larger ΔOWD that leads to more room for DEMS to balance the subflow completion time. Figure 5.10 also shows that DEMS's benefits are maximized when the file



(a) WiFi 50ms, LTE 70ms



(b) WiFi 50ms, LTE 270ms

Figure 5.11: Compare redundant data between DEMS-S and DEMS-F (laptop, emulation).

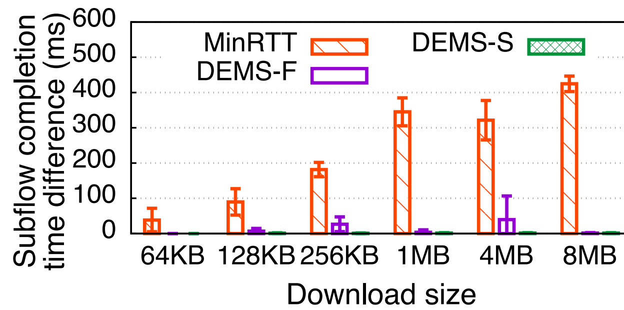


Figure 5.12: Compare subflow completion time difference (laptop, emulation, WiFi RTT 50ms, LTE RTT 270ms).

size is small or medium. For large files such as 8MB, all four schemes exhibit similar performance. The reason, as mentioned in §5.7, is that the subflow completion time difference is dwarfed by the long download duration. Nevertheless, we do expect that for future faster networks such as 5G network, DEMS will benefit large transfers. For the 64KB file download in Figure 5.10(c), all four schemes have the same performance because in this case the best strategy is to use the best single path.

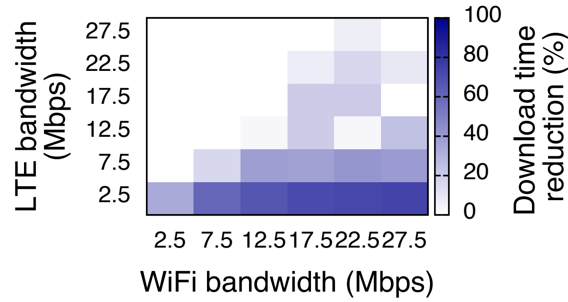
Figure 5.12 plots the subflow completion time difference (measured at the receiver) between the two paths when three schedulers are used. It confirms that our algorithm in §5.3.2 effectively achieves simultaneous subflow completion, which leads to shorter chunk download time compared to MinRTT.

Different Bandwidth Combinations. To understand the impact of the paths' bandwidth on DEMS, We compare the download time of DEMS-F and MinRTT under 6×6=36 bandwidth combinations of WiFi and emulated LTE networks, for three file sizes (256KB, 1MB, and 4MB). The bandwidth of each path varies from 2.5Mbps to 27.5Mbps. The heat maps in Figure 5.13 visualize the download time reduction brought by DEMS-F compared to MinRTT. We observe two trends. First, as the WiFi bandwidth increases, the overall download time decreases. Therefore the optimizable portion of DEMS (*i.e.,* the two subflows' completion time difference) becomes more prominent, leading to more perceived performance enhancement. Second, increasing the LTE bandwidth also shortens the download time. However, it also reduces the in-network queuing delay and henceforth reduces ΔOWD. Therefore when fixing the WiFi bandwidth and increasing the LTE bandwidth, the download time reduction incurred by DEMS is not prominent. Overall, DEMS-F achieves up to 74%, 57%, and 21% of download time reduction for 256KB, 1MB and 4MB download, respectively, compared to MinRTT.

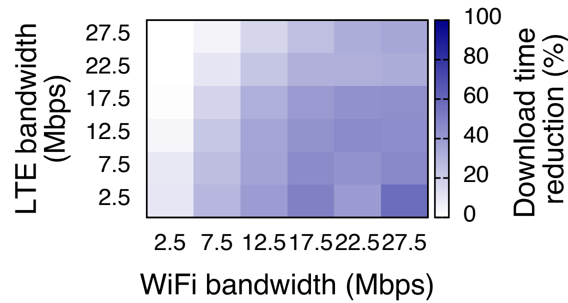
5.6.3 Varying Network Conditions

We next evaluate how DEMS performs under changing network conditions. We consider fluctuating latency and bandwidth separately.

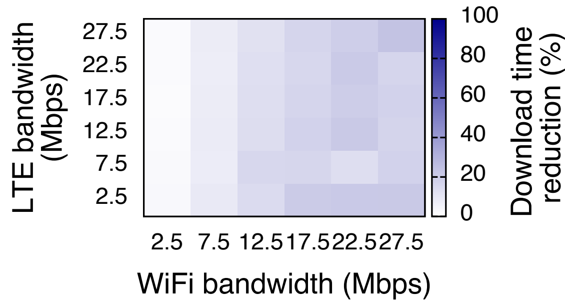
Varying Latency. We conduct experiments at a location on our campus where the RTTs of both WiFi and LTE experience high variance (the *stddev* of RTT is up to 40% and 50% of the mean for WiFi and LTE, respectively) while the bandwidth is sufficiently high. We then throttle the bandwidth to 7040kbps and 9185kbps for WiFi and cellular respectively before



(a) 256KB download



(b) 1MB download



(c) 4MB download

Figure 5.13: Download time reduction brought by DEMS-F compared to MinRTT under 36 bandwidth combinations (laptop, emulation, WiFi RTT 50ms, LTE RTT 70ms).

launching the file download experiments (256KB and 1MB, each repeating 10 times). The results are the following. DEMS-F and DEMS-S achieve similar download time that is 14% to 27% lower compared to MinRTT, respectively. However the reinjected data incurred by DEMS-F is much smaller – only 15% to 53% compared to DEMS-S. Also DEMS-B's download time is 12% worse compared to DEMS-F due to the reinjection performed by DEMS-F, which is particularly useful when it is difficult to accurately predict the network

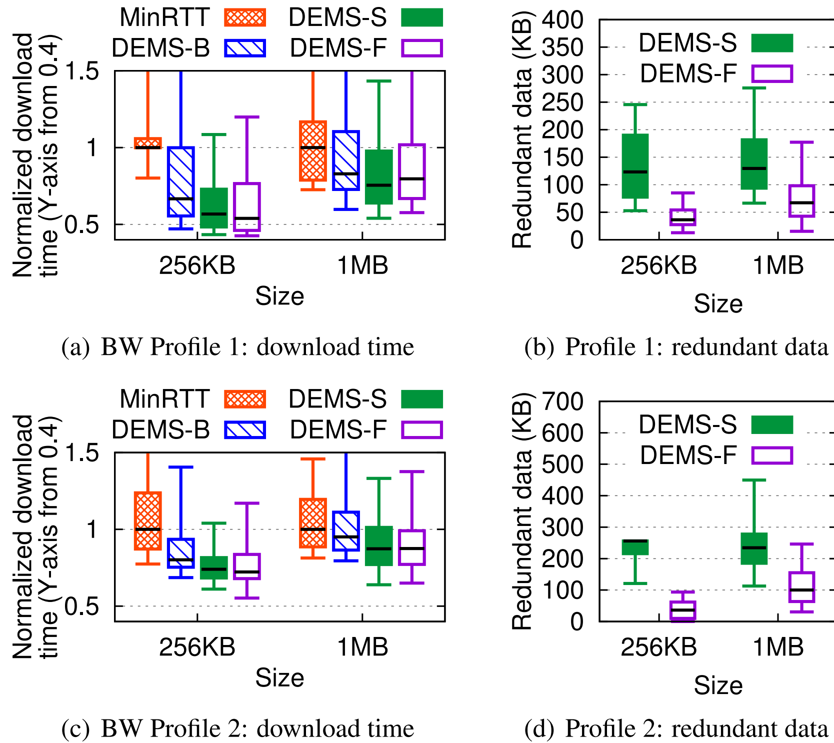


Figure 5.14: Compare different scheduling algorithms under varying network conditions (laptop, trace-driven emulation).

condition.

Varying Bandwidth. We take a "record and replay" approach to realistically emulate the varying bandwidth. We collect 5-minute bandwidth traces of both paths from two campus locations. The bandwidth at both locations is highly variable, with their *stddev* being up to 54% and 92% of the mean bandwidth for WiFi and LTE, respectively. The first location has overall lower bandwidth compared to the second. We then replay (emulate) the two bandwidth traces in our lab while maintaining stable baseline RTT. The results are plotted in Figure 5.14. Compared to MinRTT, DEMS-F reduces the median download time by 12% to 46% for the two bandwidth profiles, as shown in subplot (a) and (c). DEMS-S only marginally outperforms DEMS-F, at the cost of reinjecting much more data (1.9x to 7.1x) as shown in subplot (b) and (d). Also both DEMS-F and DEMS-S yield better results than DEMS-B, again because reinjection helps absorb the uncertainty of the varying network conditions.

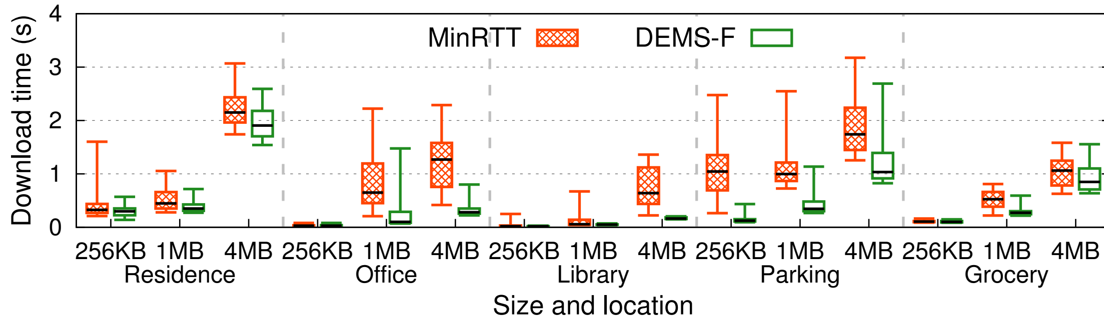


Figure 5.15: Compare performance of different scheduling algorithms (smartphone, real WiFi/LTE at five locations).

5.6.4 Field Test under Real-World Settings

We now perform field test to assess DEMS under real-world settings. We went to five locations: residential apartment, office, campus library, parking lot, and grocery store to conduct experiments of downloading files of different size (256KB, 1MB, and 4MB). For each file size, we used the Nexus 6P smartphone to repeatedly perform file download using DEMS-F and MinRTT back to back for 5 minutes over multipath (cellular network provided by a large U.S. carrier and commercial WiFi). The results are shown in Figure 5.15, which indicates that the network conditions at the five locations are indeed very diverse. Overall we obtained encouraging results: compared to MinRTT, DEMS-F reduces the download time by up to 88%, 83%, and 77% for 256KB, 1MB and 4MB download, respectively; the median download time reduction is 33%, 48%, and 42%, respectively. The download time variance is also reduced by DEMS-F.

The above improvements are attributed to the different path characteristics between WiFi and cellular networks that are not handled well by MPTCP. Compared to wired networks, wireless networks like WiFi and cellular can sometimes exhibit high delay and bandwidth dynamics [78, 82] that pose challenges to multipath schedulers. To illustrate this, Figure 5.16 shows the downlink throughput of each subflow calculated every 200ms when downloading 4MB files. As shown, at many locations, WiFi and LTE networks ex-

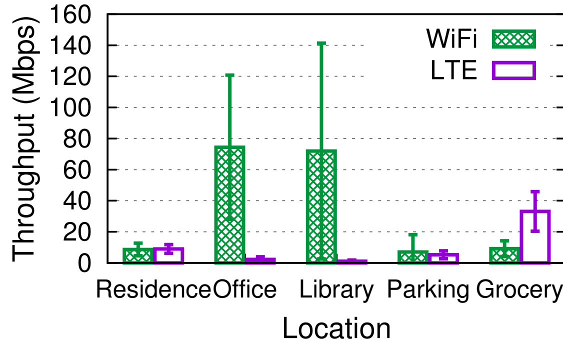


Figure 5.16: Downlink throughput of real WiFi and LTE at five locations.

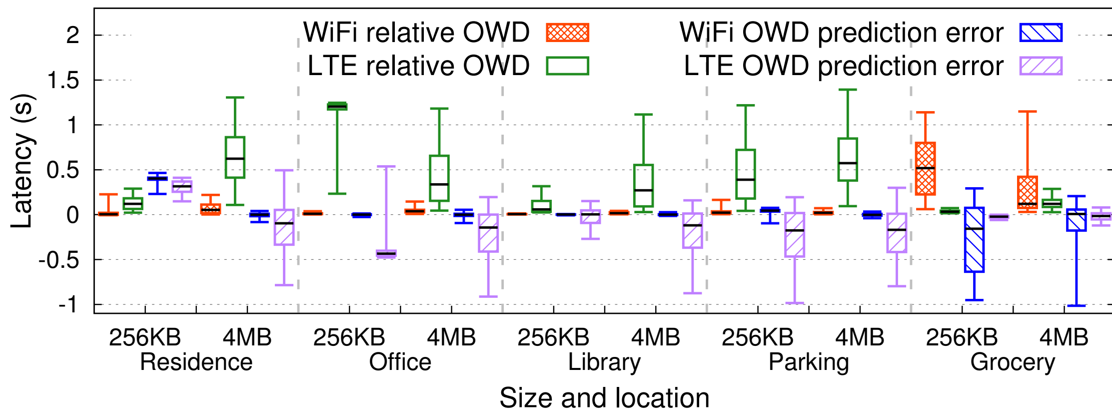
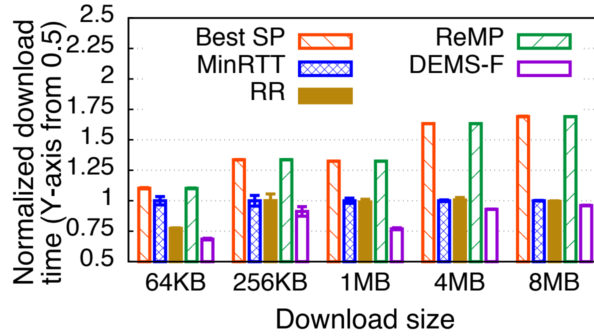
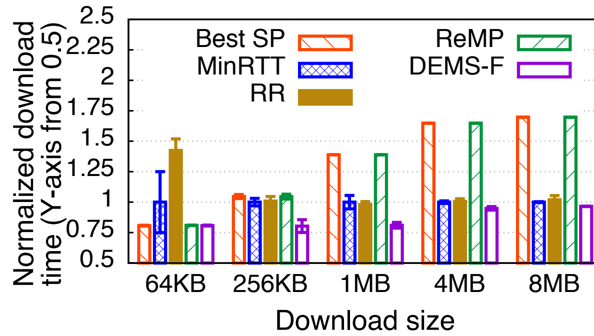


Figure 5.17: Relative OWD and prediction error of real WiFi and LTE networks at five locations.

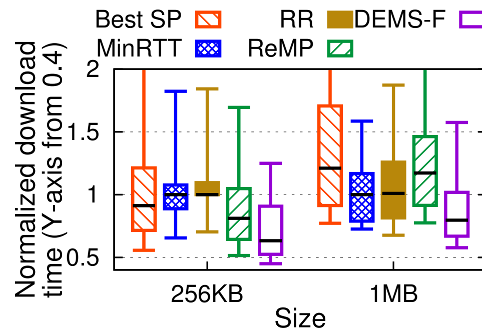
hibit different and highly variable bandwidth that can affect their OWD and the OWD prediction accuracy. As shown in Figure 5.17, oftentimes the relative OWD of WiFi and LTE are indeed highly variable, and larger file downloads have even higher variabilities. The fluctuation of bandwidth and delay makes it hard to accurately predict OWD. For example, when downloading 4MB files at the *Residence* location, the relative OWD of LTE ranges from 100ms to 1.3s, causing its prediction error to reach up to 800ms. DEMS does not fully rely on such predictions and instead employs adaptive reinjection described in §5.3.4 to more robustly handle the performance variability of many wireless network settings.



(a) Stable condition: WiFi 50ms, LTE 70ms



(b) Stable condition: WiFi 50ms, LTE 270ms



(c) Replay Bandwidth Profile 1

Figure 5.18: Compare performance among DEMS-F and four other schedulers (laptop, emulation).

5.6.5 Compare with Other Schedulers

Besides studying MinRTT, We further compare DEMS-F with three other scheduling approaches: round-robin, ReMP [59], and using the best single path to download a file. Figure 5.18(a) and (b) compare the file download time using the five schemes for two emulated stable network conditions. Figure 5.18(c) conducts a similar comparison under

varying network condition (replaying Bandwidth Profile 1 collected in §5.6.3).

We observe that DEMS-F outperforms all other four schedulers. Besides this, in most cases, MinRTT achieves similar performance compared to round-robin (RR). This is likely explained by two reasons. First, in our implementation, the round-robin selection starts with the low-RTT path; starting from the high-RTT path will inflate the download time for small files. Second, since we are performing bulk data transfer, it is unlikely that *both* paths have empty congestion window space at the same time. So during most of the time both MinRTT and RR have only one choice for path selection. Figure 5.18 also indicates the apparent limitation of the best single-path approach: compared to using multipath, using only one path significantly inflates the download time due to insufficient bandwidth. Regarding ReMP, it blindly duplicates *every* byte onto all subflows, making it essentially fall back to the "online" version of the best single-path approach. While ReMP and DEMS-F both transmit redundant data, DEMS-F's reinjection strategy is much more adaptive and strategic. For a 4MB file transfer, DEMS-F only transmits 2% of the redundant bytes sent by ReMP.

5.6.6 Web Browsing Performance

All experiments so far use file (raw data chunk) download as the workload. We now investigate how DEMS helps improve the QoE of web browsing, one of the most popular applications on mobile devices. A typical web page may consist of tens of objects, and downloading them faster would improve the QoE. We conduct web page loading experiments using off-the-shelf Chrome browser (53.0.2785.124) on our Nexus 6P smartphone. We picked 10 popular websites and use their mobile-version landing pages as the target web pages. The 10 websites cover diverse categories including news, education, travel, shopping, and government. We use the page load time (PLT), which is programmatically measured by the Chrome debugging interface, as the QoE metric. To make the experiments reproducible, we use Google Page Replay [10] to take a snapshot of each landing page and

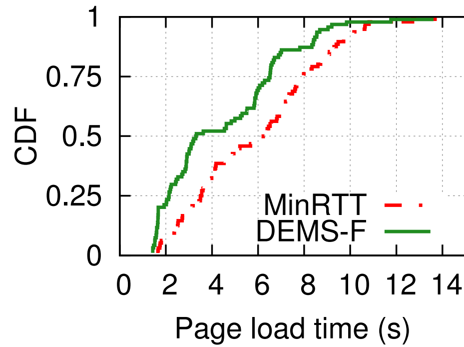
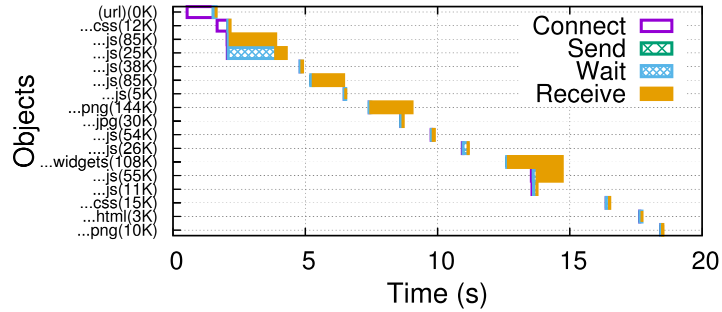


Figure 5.19: Compare web page load time when using DEMS-F and MinRTT (smartphone, real WiFi/LTE).

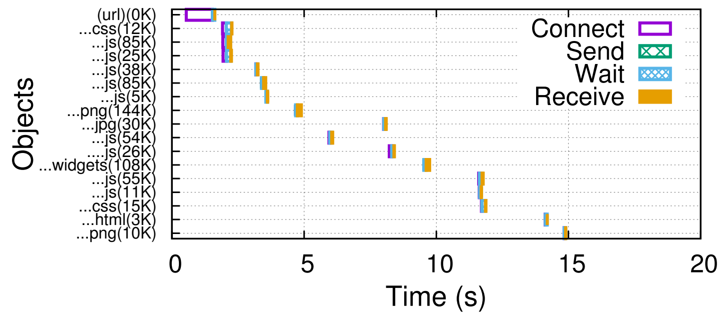
host it on our web server that is near the multipath proxy (§5.6.1). We compare the PLT of the 10 landing pages loaded by DEMS-F and by MinRTT under real-world network condition. The experiments were performed at a location on our campus where the RTTs and bandwidths of both WiFi and LTE experience high variance, similar to the network condition described in §5.6.3. We load each web page for 10 times and report the median PLT. As shown in Figure 5.19, compared to MinRTT, DEMS brings the per-page median PLT reduction of 6% to 43% across the 10 pages (median: 25%). The results indicate that DEMS can significantly improve the web QoE under real-world settings.

To understand why DEMS helps reduce the PLT, we take a closer took at the web object transmission pattern. Figure 5.20 exemplifies two waterfall diagrams for the same webpage when MinRTT and DEMS-F are used as the multipath scheduler. As shown, DEMS-F effectively reduces the reception duration (*i.e.,* download time) for many objects. Since many of them are on the critical path for web page loading, the overall PLT is effectively reduced. Also note that one key difference between file download and web page loading is that for the latter, computation and network activities are interleaved. Therefore, although the entire web page may be large, the web server has to intermittently feed data to the transport layer. This naturally forms many small/medium-sized chunks that can be well optimized by DEMS.

Recall in §5.4 that an application can interact with DEMS either using or not using a



(a) MinRTT



(b) DEMS-F

Figure 5.20: Example waterfall diagrams for MinRTT and DEMS-F (some objects are omitted for better illustration).

hint API. Since it is challenging to modify the Chrome browser source code, we adopt the non-API mode in our experiments. In reality, since an HTTP(S) persistent connection may carry multiple web objects back to back, we thus envision that using the hint API to inform DEMS of the boundary between objects can provide additional benefits of minimizing the delivery time of each object. We leave this as future work[4].

5.6.7 System Overhead

We compared CPU utilization across the three schedulers: DEMS-F, MinRTT, and ReMP by running them on the smartphone. The workload is to upload a large file as fast as possible, with the aggregated bandwidth being around 50Mbps. Compared to the other two simpler schedulers, the additional CPU overhead incurred by DEMS-F is unnoticeable. For download traffic, since the mobile device acts as a receiver, the overhead is even more

[4]In HTTP/2, objects might be multiplexed together. To handle this, multiplexed objects can be treated as one chunk.

negligible.

5.7 Discussions

Integration with MPTCP. While we implement DEMS on our own multipath TCP proxy infrastructure, DEMS can also be directly integrated into MPTCP. Since MPTCP provides a modular scheduler framework [108], the core logic of DEMS can be implemented as a new scheduler module. To facilitate OWD measurement (§5.4), the receiver side can leverage MPTCP options in TCP headers to carry necessary information used for estimating OWD at the sender side. The network prediction functionality can be implemented as a general API in MPTCP for all schedulers to use.

Applicability. DEMS can work with diverse traffic patterns and transfer sizes despite the ideal usage scenario being chunked data transfer, a very common workload as mentioned in §5.1. As DEMS shows the most promise for small-to-medium-size downloads, it can benefit a wide range of today's mobile applications (*e.g.,* video streaming) that involve downloading chunks from a few hundreds KBs to a few MBs. For very large files (*e.g.,* tens of MBs), when the network bandwidth is limited (a few Mbps), the subflow completion time difference is dwarfed by the long download duration, leading to overall small relative improvement brought by DEMS. Nevertheless, we do expect for future faster networks such as 5G with bandwidth of hundreds of Mbps, DEMS will significantly benefit large transfers.

Limitation. We discuss some limitations of DEMS. First, our current design focuses on two subflows. This was driven by realistically the most common mobile multipath use cases today. Nevertheless, the core concepts of DEMS (*e.g.,* chunk-based transfer, simultaneous flow completion, and adaptive reinjection) are also applicable to more than two paths, though involving more complexities. For example, instead of using the "two-way"

splitting, each subflow can transmit packets in an interleaved (but still decoupled) manner. To achieve simultaneous subflow completion, a subflow can decide to stop transmission if the remaining data in the meta buffer can be delivered earlier by some other subflows with smaller OWD (some coordination algorithms need to be developed). In this way, at the sender side, the subflow with the largest and the smallest OWD will be the first and the last finishing subflow, respectively. This can compensate the delay differences among the subflows, leading to simultaneous subflow completion at the receiver side. We leave fleshing out the details on this to future work.

Second, the current DEMS design focuses primarily on improving download time, a critical factor impacting user QoE. In the future, we plan to explore additional dimensions like energy and limiting data usage on a particular subflow. This could possibly be done by adopting concepts from other special-purpose schedulers such as energy-aware [90] and subflow-priority-aware [70] schedulers.

5.8 Summary

Compared to single-path, multipath transport brings more complexities due to not only more involved paths but also their sophisticated interactions. Through judicious algorithm design, system integration, and extensive evaluation, we demonstrate that by strategically scheduling the packets, we can improve the multipath performance significantly (*e.g.,* median download time reduction of 33%–48% for fetching files and median loading time reduction of 6%–43% for fetching web pages under real-world settings compared to MinRTT). In our future work, we plan to extend DEMS in several aspects (§5.7).

CHAPTER VI

A First Look at Android Wear Networking Performance under Mobility

Chapter III, IV, and V address performance problems and improve application QoE with the flow-level and interface-level parallelism on a single device, *i.e.,* a smartphone. In this chapter, we embrace the device-level parallelism by providing a first look at the wearable network management under mobility, where the wearable device needs proper coordination with the smartphone to access the Internet.

6.1 Introduction

Smart wearable devices have become an important member of the mobile computer family. According to IDC, by volume, smartwatches account for the largest part of smart wearables and are expected to reach a total value of $17.8 billion dollars in 2020 [28].

In the literature, several efforts have been made towards understanding and improving the OS execution performance [92, 93], power management [95], graphics and display [100], storage [76], and user interface [47, 134] of wearable OSes. In this chapter, we investigate an important yet under-explored component: *the wearable networking stack.* We conduct to our knowledge the first investigation of the networking performance of Android Wear under mobility.

Wearable networking is important. Take smartwatches as an example. One may argue they only incur light traffic such as push notifications. This is indeed true for the *current* smartwatch ecosystem where traffic flows are largely small, short, and bursty [95]. However, we envision that future wearable apps will involve much heavier network activities fueled by new hardware, OS support, and applications. For example, the latest speaker/LTE-capable watches such as Samsung Gear S3 Frontier allow users to directly make hands-free VoIP calls; the latest Android Wear 2.0 allows standalone apps on watches; also, many emerging applications incur heavy network traffic such as continuous computer vision on smart glasses [50, 65], remote camera preview, and multipath collaboration between phone and watch [97].

Wearable networking is also different from, for example, smartphone networking that has been well studied in the past decade. First, wearables often do not directly access the Internet; instead, it uses its paired smartphone as a "gateway" over Bluetooth (BT), which, if not carefully designed, may incur additional performance degradation. Such a gateway mode accounts for 84% of the daytime usage period as measured by a recent user study [95]. Second, due to BT's short range, network handover frequently occurs on a wearable: when it moves away from the phone, the BT session will be torn down and the wearable has to use standalone WiFi or LTE to communicate with the external world.

In this chapter, we conduct controlled experiments using a commodity Android Wear smartwatch and Android Wear applications. Our key finding is that, surprisingly, there exist serious performance issues under mobility regarding aforementioned aspects that distinguish wearable networking from smartphone networking. Due to passive network switching and insufficient protocol support for handovers, the BT-WiFi handover that frequently occurs on wearables may last more than 40 seconds, leading to significant degradation of application performance.

We find that the above performance inefficiencies are caused by poor design of the networking stack of Android Wear OS due to a lack of thorough understanding of the

cross-layer interactions. To summarize, our study makes two contributions.

• Discovery of network performance issues of Android Wear OS under mobility based on thorough in-lab measurements using commodity devices and identification of their root cause and impact on application performance.

• Design improvement for networking subsystems of future wearable OSes. We propose WearMan, a wearable network manager to perform preemptive handovers using changes of BT signal strength as an indicator that tells in advance when a handover needs to be performed.

6.2 Poor Handover Performance

We explore the performance of Android Wear real-time application when the watch moves away from or closer to the smartphone, *i.e.,* handovers between Bluetooth (BT) and WiFi. We highlight the issues of current Android Wear networking framework on handling mobility.

6.2.1 Impact of Handover on Real-time Apps

Real-time applications on wearables provide rich functionalities. The latest smartwatches such as Samsung Gear S3 Frontier enables hands-free VoIP calls for users to directly make phone calls from the smartwatch. There are a few Android Wear apps that currently support real-time content delivery. To name a few, the Wear Camera Remote app [38] streams image previews from the phone camera to the smartwatch in real time; the tinyCam Monitor PRO app [37] allows the user to monitor real-time video from the security camera on the smartwatch; there are also music and radio streaming apps [33, 35] for Android Wear. All these applications require continuous network connectivity on the wearable to deliver satisfying application QoE, even if the user moves around with the wearable.

While WiFi networks may be often available for the smartwatch to use for data transfers, Android Wear by default prefers BT for network transmission because the energy consumption of BT is much smaller than that of WiFi. As shown in a previous measurement study [95], the power consumption of WiFi is more than three and five times the power consumption of BT for actively receiving and transmitting network data, respectively, on a state-of-the-art smartwatch, LG Watch Urbane. Since real-time apps usually continuously transmit and receive network data and may not require high bandwidth, using BT when available can significantly reduce energy consumption on the smartwatch. It is important for Android Wear to properly handle network switching between WiFi and BT to achieve both energy saving and good performance.

Due to BT's short range, when a wearable moves away from the phone, network handovers are very likely to occur from BT to standalone WiFi or cellular network to provide continuous Internet access. When a wearable is connected to WiFi and moves to the range of smartphone BT, the wearable can choose to switch the active network to BT and turn off WiFi to reduce network energy consumption. Compared to smartphone handovers between cellular network and WiFi, smartwatch handovers happen more frequently as the user moves and require careful handling as the wearable can lose BT connectivity in short time under mobility.

6.2.1.1 Experimental Setup

We set up a testbed shown in Figure 6.1 to understand the networking performance of Android Wear. The network access for the smartwatch includes communicating locally with the phone over BT, accessing the Internet with direct WiFi, as well as surfing the Internet via the smartphone as the gateway. Our experiments are performed on a cutting-edge smartwatch: Huawei Watch with Android Wear 2.0. The phone we use is a Nexus 6P with Android 7.0. The server we use is an HP desktop running Ubuntu 16.04 with Linux kernel 4.4.0.

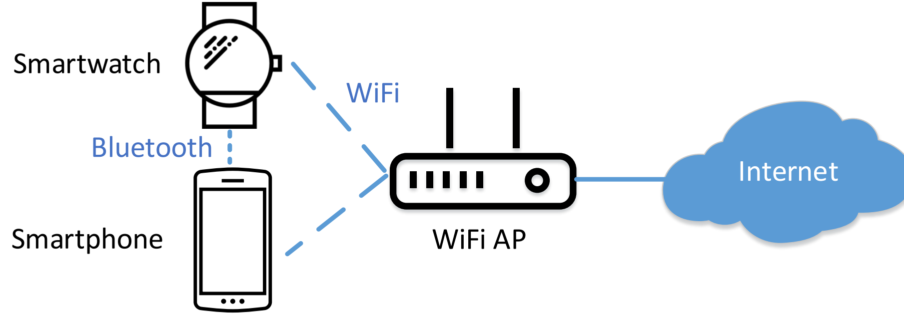


Figure 6.1: Wearable measurement testbed.

To understand the impact of network handovers on Android Wear 2.0, we study the application performance and QoE metrics of the tinyCam security camera app [37] during handovers. Since most smartwatches have both BT and WiFi, we focus on understanding the handovers between BT and WiFi. We paired a Huawei Watch with Android Wear 2.0 to a Nexus 6P phone with Android 7.0 over BT. Both the smartwatch and smartphone are configured to connect to the same WiFi network. To stream security cam videos, we connect a Samsung SNH-V6414BN SmartCam [36] to the same WiFi network and make it accessible to the tinyCam app on the Huawei watch through either BT or WiFi.

We define two QoE metrics to understand the performance of tinyCam app: (1) *frame one-way delay (OWD)*, which refers to the time to transmit a video frame from the security camera to the smartwatch and display it in the tinyCam app, and (2) *network throughput*, which is the actual data transmission rate to the smartwatch. To measure the frame OWD, we use the same Nexus 6P phone to display constantly increasing timestamps to the Smart-Cam, and use an iPhone 7 Plus to record the screens of Nexus 6P and Huawei Watch with 30 fps. We record the screen of Nexus 6P for a few second and retrieve the timestamp, t_i, of each displayed frame i with a timestamp in the recorded video. We also record the screen of Huawei Watch during handovers and manually parse the recorded video to get the timestamp, t'_i, for each received frame with the same displayed timestamp on the smartwatch. Thus, $t'_i - t_i$ is the frame OWD delay. We calculate the BT and WiFi throughput from BT snoop logger and `tcpdump` traces, respectively. We keep both BT and WiFi enabled

in the settings on the smartwatch and let the Android Wear OS use the default network management policy to choose the network.

Besides tinyCam app, we also write our own Android Wear app called RTApp to understand how application logic affects the performance during handovers. Our app imitates the traffic pattern of the tinyCam app, *i.e.,* downloading a data chunk of 3KB every 160ms from our server using TCP socket, generating 150kbps downlink traffic. Each data chunk has a sequence number. For each new TCP connection initiated by the app, the app first sends a request with the sequence number which the app expects to receive next, and the server starts sending data chunks from the corresponding sequence number. Whenever the active network switches, *e.g.,* from BT to WiFi, the app establishes a new TCP connection to the server and resumes chunk downloading. A new TCP connection is initiated only after a previous connection encounters data transfer error and the new network is ready to transmit data. We install this app on the same smartwatch as above and deploy the server in the same local network so that the RTT between the smartwatch and the server is below 5ms.

6.2.1.2 BT to WiFi Handover Performance

Figure 6.2 shows the QoE metrics of tinyCam app over time when the smartwatch is moving out of the BT coverage. Contrary to our expectation of reasonable handover performance, the application QoE of tinyCam app is severely degraded during the BT-to-WiFi handover. At around $t = 6$s, the app stops receiving security video from the SmartCam, and the BT throughput drops to zero. The video frame transmission resumes over WiFi after around 72.5s, with high frame OWD at the beginning. We repeat this experiment under the same settings for five times and the average handover delay during which the app cannot receive any video is 70.0s. This shows that tinyCam app experiences poor performance during BT-to-WiFi handovers.

We also look at how RTApp performs under the same handover setting. The average

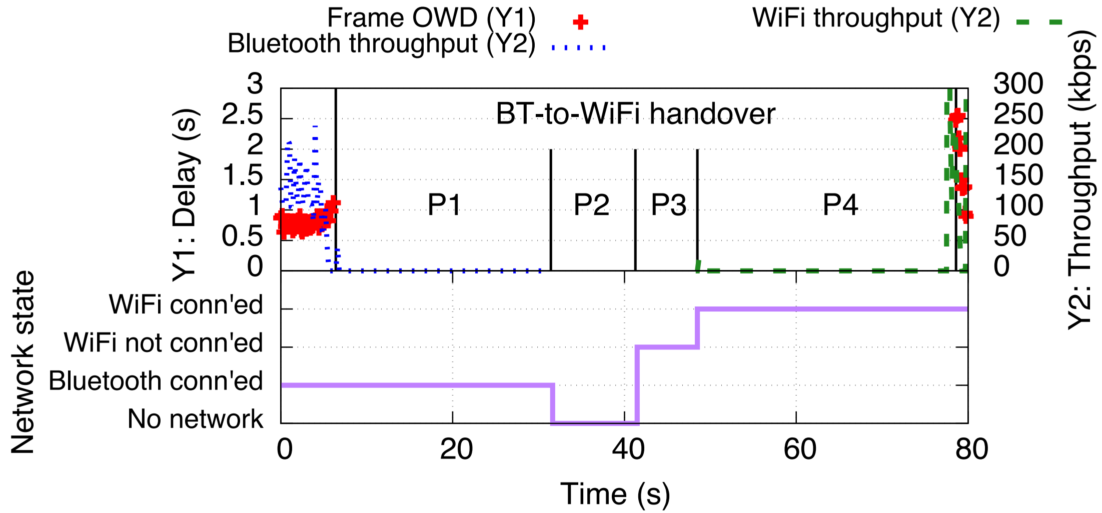


Figure 6.2: Impact of BT-WiFi handover on the QoE of tinyCam security camera app on Huawei Watch.

handover delay of 5 repeating experiments is 43.6s, during which RTApp cannot receive any data chunk from the server side. This clearly shows that the current network switching management in Android Wear 2.0 during handovers is far from satisfaction.

6.2.1.3 WiFi to BT Handover Performance

We next look at the performance of tinyCam and RTApp when the smartwatch is moving towards the coverage of smartphone BT network. In the beginning, the smartwatch is located in a place where BT is out of range and only WiFi network is available. We let the app first use WiFi network to download application data. After the app continuously downloads data, we move the smartwatch closer to the smartphone so that the smartwatch initiates BT connection. We repeat each app for five times. During this process, the average handover delays, *i.e.,* network interruption time when no data can be transferred, of tinyCam and RTApp are 17.7s and 0.98s, respectively, which are much smaller compared to the handover delay of BT-to-WiFi handovers. We understand the root cause of handover delays in the next section.

6.2.2 Root Cause Analysis

To understand the root cause, we write another Android Wear app to capture additional network information from the `ConnectivityManager` in the background on the smartwatch. For every 200ms, the app asks the `ConnectivityManager` to return a list of currently tracked networks on the smartwatch. For each monitored network, the app logs whether the network is possible to use by applications, *i.e.,* available or not, and whether the interface provides actual network connectivity, *i.e.,* connected or not, through Android `NetworkInfo` APIs. For example, the standalone WiFi on the smartwatch can be possible to use, but there is no actual network connectivity during the connecting state. We thus use this information from Android Wear OS to understand its behavior of network switching.

6.2.2.1 Networking Switching Delay from Android Wear OS

We break down the overall interruption time of BT-to-WiFi handover into 4 periods, as shown in Figure 6.2 as an example: (P1) BT connected but data cannot be actually transmitted, (P2) No network available, (P3) WiFi available but not connected, *i.e.,* WiFi connecting period, and (P4) WiFi connected but no application data transmission. Under the default network management policy, WiFi is not available when BT is connected, even if the smartwatch is under the coverage of both BT and WiFi. In this case, when the smartwatch moves away from BT coverage, the Android Wear OS needs to first determine when the BT cannot transmit application data (P1), then enable WiFi (P2) and connect to an available AP (P3). While trying to save energy by passively turning on WiFi when necessary, this procedure, however, rather incurs an extended period of no network connectivity. As shown in Figure 6.3, across five repeated runs, the average time of P1, P2, and P3 for tinyCam (RTApp) are 12.9s (21.1s), 15.5s (11.9s), and 8.3s (4.9s), during which the app data cannot be transmitted over the network, accounting for 52% (87%) of the overall handover delay. When we keep WiFi always connected during the handover, the average time of P1 is 13.3s across another five runs of the tinyCam app, still large enough to degrade

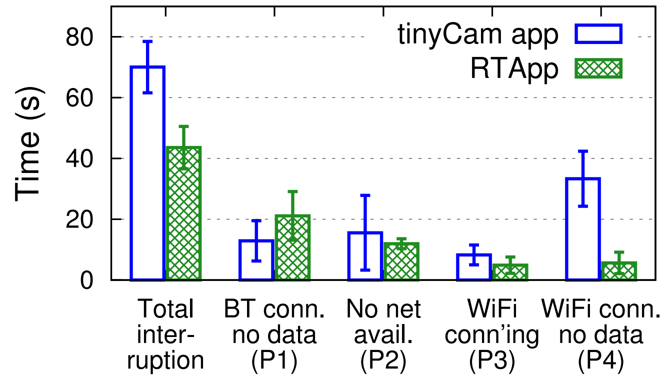


Figure 6.3: Breakdown of handover delay.

application QoE, while P2 and P3 are minimized. The results show that Android Wear OS needs better network management during BT-to-WiFi handovers to prevent application QoE degradation.

6.2.2.2 Insufficient Protocol Support for Applications

We next look at how an application uses the network provided by Android Wear OS. For BT-to-WiFi handovers, under the default network management policy, even after WiFi gains network connectivity (confirmed by looking at `tcpdump` trace), the tinyCam app still takes around 33.3s on average to use WiFi for data transmission (P4). Surprisingly, if we keep WiFi always connected on the smartwatch during handovers, we observe an even larger period of P4, 57.0s on average, and overall similar handover delay, 70.2s on average. In Figure 6.3, compared to tinyCam, RTApp only takes 5.6s on average to start chunk downloading after WiFi is connected (P4). The large time difference of P4 between two apps contributes to the discrepancy of total interruption time.

For WiFi-to-BT handovers, similarly, we find that network disruption (17.7s) experienced by tinyCam app happens after the BT connection can transmit data, whereas RTApp barely experiences data interruption (0.98s). This shows that the application's specific logic on resuming data transmission after handovers also affect the performance. Since Android Wear does not provide protocol or API support, such as MPTCP, on seamlessly migrating

data transfers between networks, the Android Wear application require the implementation of their customized data transfer migration at the application layer. It is hard for the app developers to handle such scenarios properly and achieve the satisfying handover performance.

6.3 WearMan: Improving Handover Performance

The handover results in §6.2 suggest that considerable improvements need to be made to improve WiFi-BT handovers on wearables.

6.3.1 Solution Overview

As discussed in §6.2, ill-timed network switching and insufficient protocol support lead to undesirable handover performance to Android Wear applications. To solve this problem, first, the OS should account for the network switching delay and choose the right timing to switch the active network to avoid any network interruption. For BT-to-WiFi handover, this means that the OS needs to start network switching to WiFi much earlier before the smartwatch is out of BT range. For WiFi-to-BT handovers, the BT connection must only be used as the active network when it provides sufficient bandwidth. Second, the OS should provide a default abstract API to transfer data from networks without requiring the apps to handle network switching and ask for a particular network interface. If necessary, an app only needs to choose an active network to use if other aspects are considered, such as high bandwidth requirement.

To understand the right timing for handovers between BT and WiFi, we first measure the BT download performance of a smartwatch at different relative distances to the paired smartphone. We use the same devices as mention in §6.2. For each location, we first measure the average BT received signal strength from the smartphone, *i.e.,* RSSI, on the smartwatch using BT discovery for 30 seconds. We then keep the smartwatch at the same location and download a large file for 1 minute from our server in the same local network

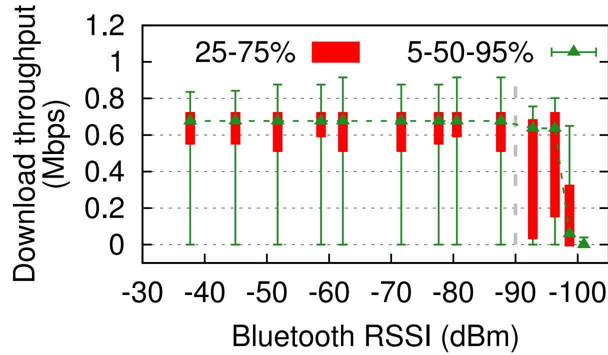


Figure 6.4: BT download throughput under different signal strength. The zero throughput of 5th percentile for BT RSSI above -90dBm is due to BT sniff mode.

as the smartphone. We calculate the BT download throughput every 200ms from the BT snoop logger trace captured on the smartwatch.

Figure 6.4 shows the relationship between the BT download performance and RSSI on the Huawei Watch. The BT download throughput has the same distribution when the RSSI is above -90dBm. When the RSSI is below -90dBm and the smartwatch is far away from the smartphone, the BT download throughput has high variance, and the smartwatch may experience intermittent network connectivity. This shows that we can use BT signal strength to indicate BT performance and choose the right timing for network switching.

We propose a proactive handover approach, WearMan (**Wear**able Network **Man**ager), by performing preemptive handovers using the changes of BT signal strength as an indicator that tells in advance when the OS should performance a handover. Our approach continuously monitors BT RSSI over time and decides the active network to use:

- When BT is the active network to transmit application data, if BT RSSI is below a threshold $RSSI_{th1}$, WearMan prepares WiFi for the app and starts a new TCP connection over WiFi when the WiFi network can transmit application data.

- When WiFi is the active network to transmit application data, if BT RSSI is above a threshold $RSSI_{th2}$, WearMan prepares BT for the app and starts a new TCP connection over BT when it is ready to migrate application data transfer from WiFi to BT.

Before the new TCP connection can transmit data, previous TCP connection continues to transmit data so that the app still has network connectivity.

There are two approaches of how WearMan prepares a particular network for the app. The first approach is on-demand network enabling, by only enabling the alternative network besides the active network when necessary and disabling a network when not used. This reduces energy consumption of network interfaces but incurs additional waiting time for the alternative network to be ready, which can be very long for WiFi network (§6.2). The second approach is always-active networking, by always keeping both networks ready to transmit data. This helps to dramatically reduce the time to switch to another network but potentially increases energy usage. For real-time applications, the second approach is usually better because small network interruption time is more valuable for this type of applications.

6.3.2 Implementation

Since there is no method to modify Android Wear OS or transparently intercept application network traffic on Android Wear to our knowledge, we prototype WearMan in RTApp by providing the proactive handover approach to download data chunks over time. In Android Wear, the OS does not provide any API to retrieve BT RSSI when a BT device is connected. To get the BT RSSI, we leverage `BluetoothAdapter` in Android Wear to trigger BT discovery and get one received signal strength sample from each discovery. We periodically trigger BT discovery on the smartwatch over time to continuously retrieve BT RSSI. We envision that Android Wear will add APIs to get BT RSSI for BT connected devices in the future to help develop a better network management system on the smartwatch.

However, we find that such approach of retrieving RSSI has a negative impact on both WiFi and BT throughput. To find a strategy of triggering BT discovery over time to achieve a good tradeoff between RSSI reporting delay and throughput degradation, we explore two types of BT discovery strategies with different parameter settings: (1) start BT discovery

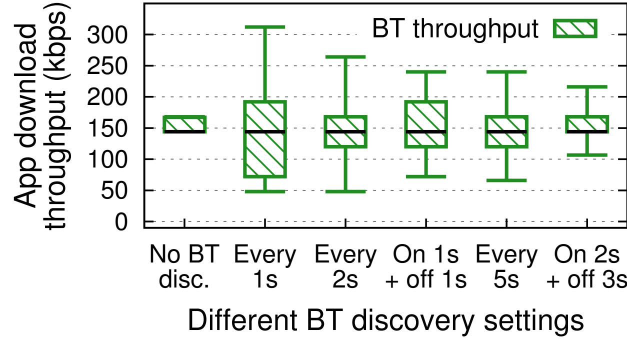


Figure 6.5: Download throughput of RTApp when using BT under different BT discovery settings.

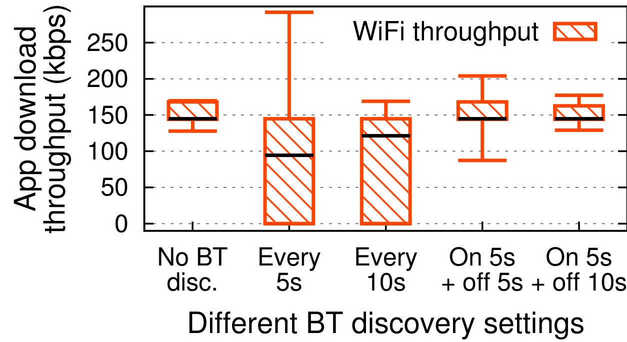


Figure 6.6: Download throughput of RTApp when using WiFi under different BT discovery settings.

every x seconds, and (2) start BT discovery, wait for y_1 seconds (BT discovery on), stop BT discovery, and wait for y_2 seconds (BT discovery off). We measure the app download throughput of RTApp for 2 minutes with different strategies and parameters under both WiFi and BT. As shown in Figure 6.5, for BT download, triggering BT discovery every 2s achieves a good tradeoff. Figure 6.6 shows that, for WiFi download, a different strategy, starting and canceling BT discovery every 5s, achieves a good tradeoff. We thus use these two different strategies in WearMan and RTApp according to the type of network that transmits data.

As discussed in §6.3.1, we use -90dBm for the threshold $RSSI_{th1}$ which determines the handover from BT to WiFi. We use a different value, -85dBm, for the threshold $RSSI_{th2}$ which determines the handover from WiFi to BT, to avoid frequent network handovers

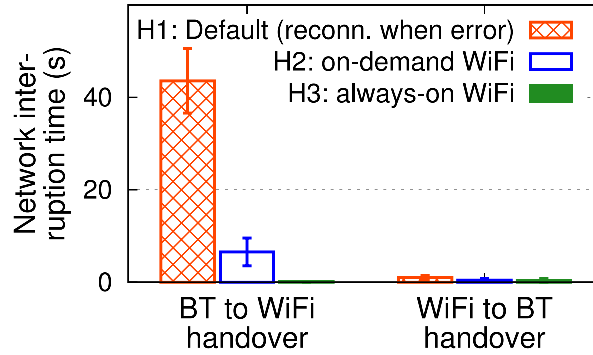


Figure 6.7: Comparison of network interruption time during handover under different handover strategies.

around such signal strength.

While we build our prototype of WearMan in RTApp, WearMan can also be implemented in Android Wear OS and incorporated with existing network protocols to provide seamless handovers. We can further apply multipath TCP (MPTCP) to wearables, using MPTCP's *backup mode* that allows seamless fallback from a primary path to a backup path. MPTCP provides an additional benefit of keeping the application view consistent before and after a handover.

6.3.3 Evaluation of WearMan

We use RTApp to evaluate the performance of WearMan in the same settings described in §6.2. The baseline (H1) we compare WearMan with is the default handover handling strategy in Android Wear. In this strategy, the app only reconnects to the server when it encounters R/W error to the TCP socket. We also consider two network management strategies in WearMan discussed in §6.3.1, and focus on how to enable WiFi network as it severely affects BT-to-WiFi handover performance: (1) H2: enabling WiFi only when necessary, (2) H3: always keeping WiFi enabled when the app is transmitting data on either network.

For BT-to-WiFi handovers, As shown in Figure 6.7, both H2 and H3 significantly reduces the average network interruption time, by 85% and 100% respectively, compared

to H1. However, we still see network interruption time of 6.5s on average for H2, where the WiFi network is enabled on demand. This is because even if WearMan proactively starts enabling WiFi when it observes poor BT RSSI, the app may still get disconnected when the user moves out of the BT range before WiFi network is ready to transfer data. Compared to H2, when making WiFi always enabled (H3), the WiFi connection can be established immediately after the BT RSSI is smaller than the threshold $RSSI_{th1}$, resulting in near-zero network interruption time. For WiFi-to-BT handovers, we observe similar performance across all schemes, as WiFi is transmitting data before BT network becomes ready to migrate data transfer.

6.4 Summary

We find that BT-WiFi handovers that occur on wearables last more than 40 seconds on current Android Wear OS, due to passive network switching and insufficient protocol support for handovers. To address this problem, We design and prototype WearMan, a wearable network manager to perform preemptive handovers. WearMan significantly reduces handover delay by improving network selection mechanism and providing seamless handover experience.

CHAPTER VII

Related Work

7.1 Improving TCP Performance over Cellular Networks

Measuring cellular performance. Several prior efforts focus on characterizing cellular performance at various aspects. Studies [56, 104, 118, 121] collect data from deployed user trials to understand smartphone performance at different layers. To name a few, Sommers *et al.* leveraged `speedtest` data to compare cellular versus Wi-Fi performance for metro area mobile connections [123]. Huang *et al.* examined LTE bandwidth utilization and its interaction with TCP [78]. Shafiq *et al.* conducted a study of cellular network performance during crowded events [119]. Liu *et al.* measured performance of several TCP variants on 3G EvDO networks [96]. Rosen *et al.* studied the impact of RRC state timers on network and application performance [116, 117]. Jia *et al.* performed a systematic characterization and problem diagnosis of commercially deployed VoLTE (Voice over LTE) [80]. None of the above studies deeply examined cellular upload traffic that is becoming increasingly popular.

Improving transport protocols. Over the past 30 years, researchers have produced a large body work on improving TCP. We already mentioned many TCP congestion control algorithms in §3.6.1, such as [40, 43, 44, 57, 60, 66, 83, 86, 88, 94, 99]. Some recent proposals such as TCP-RRE [89], Sprout [131], Verus [138] and TCP ex Machina [130] leverage throughput forecasts or machine learning to find the optimal data transmission strategy.

All these approaches face the problem of balancing between throughput and latency, which is a key factor to be considered when selecting the desired CC given the application requirement. Other solutions like RSFC [136] and DRWA [82] uses receive buffer to limit the queuing impact. However, transport-layer solutions do not provide cross-flow control, which may not fully eliminate interference between flows. Compared to these transport-layer solutions, QCUT uses accurate throughput estimation based on the information from the firmware and explicitly reduces on-device queuing in cellular networks.

Understanding excessive queuing delay ("bufferbloat"). The bufferbloat problem is known in both wired and wireless networks. Gettys *et al.* presented an anecdotal study [61] on large queuing delay of interactive traffic. The study focuses on the scenario of concurrent bulk data transfers in cable and DSL networks. Using real network traces, Allman argued that although bufferbloat can happen, the problem happens more in residential than non-residential networks and the magnitude of the problem is modest [39]. However, the issue is indeed severe in cellular networks that usually employ deep buffers, as shown in a study conducted by Jiang *et al.*, who explored the bufferbloat problem of downlink traffic in 3G and LTE networks [82]. Recent work by Xu *et al.* indicates that some newer smartphones seem to buffer packets in the kernel when UDP packets are transmitted continuously [137]. However, they did not study TCP or consider how application is affected. In contrast, we carry out comprehensive measurements to quantitatively understand *(i)* on-device bufferbloat caused by TCP *upload* traffic and its impact on applications, *(ii)* the interaction between upload and other traffic patterns, *(iii)* the interplay between TCP and lower layer queues, and *(iv)* the effectiveness of a wide range of mitigation strategies at different layers.

Mitigating bufferbloat. Besides those evaluated in §3.7, there exist other proposals for reducing excessive queuing delay. Dynamic Receive Window Adjustment (DRWA) [82] is a receiver-side solution to reduce the queuing delay by adjusting the TCP receive window. Originally it is deployed on mobile devices to reduce the latency for downlink traffic. For

the uplink case, DRWA needs to be deployed at server side that serves both cellular and non-cellular clients, thus posing deployment challenges. Byte Queue Limits (BQL) [3] is another proposal that puts a cap on the amount of data waiting in the device driver queue. It does not apply to Qdisc that contributes the majority of the on-device latency. Moreover, BQL needs driver support.

7.2 Improving Multipath Transport

Characterization of Mobile Multipath. Several prior efforts focus on characterizing the multipath performance over mobile networks. Chen *et al.* [48] studied the file download performance using MPTCP over 3G/4G and WiFi. Deng *et al.* [54] compared the performance between single-path and multipath in the mobile context. Han *et al.* [68] investigated the interaction between MPTCP and web protocols such as HTTP/1.1 and SPDY. Nika *et al.* [103] characterized the energy efficiency and performance of radio bundling (*i.e.,* multipath) in outdoor environments. Some other recent studies examined MPTCP performance [51] and its impact on applications [104] through crowd-sourced measurements.

Multipath Schedulers have been experimentally shown to affect download performance [103, 108, 110]. Several multipath scheduling algorithms have been proposed for different application scenarios. ECF makes scheduling decision using both congestion window size and RTT to avoid undesirable idle transmission periods and achieve higher aggregate throughput [91]. Compared to ECF, DEMS relies on chunk-based data transfer to decouple subflows and employs adaptive reinjection to combat variable network conditions. MPRTP focuses on real-time content delivery over multipath by adapting to the changing path characteristics [122]. DAPS aims at reducing receiver's buffer blocking time over multiple wireless networks [85]. ReMP duplicates the same packet onto all paths to reduce latency and to improve reliability [59]. eMPTCP takes energy consumption into consider-

ation when making scheduling decisions [90]. Another energy-aware MPTCP scheduling algorithm was proposed in [111]. Compared to the above work, DEMS aims at reducing the download time for data chunks through a set of novel techniques.

Applications of Mobile Multipath. Previous studies have leveraged the MPTCP API extension to support enhanced socket options [71], fine-grained control on transport behavior [72], and multipath over UDP [41]. Some other studies investigated how to better use multipath for different applications, such as video streaming [49, 52, 70, 133], web browsing [69], smooth handover [53], and traffic sharing across users [102]. In contrast, DEMS is a general-purpose multipath scheduler that can benefit a wide range of applications.

7.3 Wearable Networking

Liu *et al.* analyzed the execution of Android Wear OS and identified several inefficiencies [92, 93]. Liu *et al.* conducted a user study to understand smartwatch usage in the wild [95]. Chauhan *et al.* [45] characterized smartwatch apps. Hester *et al.* [74] designed an multi-application wearable platform. There also exist studies on other aspects of wearable systems including display [100], storage [76], user interface [47, 134], energy [67], and security [98, 126]. Researchers have also designed new sensing applications [107, 120] using wearables. In contrast, our work focuses on the networking aspect of commercial wearable OS.

CHAPTER VIII

Conclusion and Future Work

Despite seemingly bringing benefit to mobile applications at first glance, the flow-level, interface-level, and device-level parallelism incur rather complex interactions with applications due to the diverse traffic patterns and QoE metrics of these applications, potentially leading to severely degraded application QoE. My dissertation addresses this challenge, demonstrating that with a better understanding of the complex interaction between the flow-level, interface-level, device-level parallelism and applications, the networking stack on mobile systems can efficiently use diverse network resources to improve application QoE without modifying the applications.

Specifically, to address the interference incurred by flow-level parallelism, we conduct a comprehensive characterization of cellular upload traffic and investigate the on-device bufferbloat problem frequently incurred by upload traffic accessing diverse types of cellular networks on mobile devices. Our findings suggest that this problem leads to significant performance degradation on real mobile applications, *e.g.,* 66% of download throughput degradation and more than doubling of page load times. We further propose a system called QCUT to control the firmware buffer occupancy from the OS kernel to mitigate this problem. We demonstrate the effectiveness of QCUT through in-lab experiments and real deployment.

To better leverage the interface-level parallelism, we propose a flexible software archi-

tecture for mobile multipath called MPFlex, which strategically employs multiplexing to improve multipath performance (by up to 63% for short-lived flows). MPFlex decouples the high-level scheduling algorithm and the low-level OS protocol implementation, and enables developers to flexibly plug-in new multipath features. MPFlex also provides an ideal vantage point for flexibly realizing user-specified multipath policies and is friendly to middleboxes. Based on this flexible multipath architecture, we propose DEMS, a novel multipath scheduler aiming at reducing the data chunk download time. DEMS is robust to diverse network conditions and brings significant performance boost compared to the default MinRTT scheduler (*e.g.,* median download time reduction of 33%–48% for fetching files and median loading time reduction of 6%–43% for fetching web pages), and even more benefits compared to other state-of-the-art schedulers.

To embrace the device-level parallelism, we explore the wearable networking system where the interaction between the wearables and the smartphone is essential. We investigate the networking performance under mobility on a popular smartwatch OS, Android Wear, to understand multi-device networking under device-level parallelism. We observe that BT-WiFi handovers on Android Wear smartwatch can last more than 40 seconds, due to passive network switching and insufficient protocol support. Motivated by the observations and insights, we further propose WearMan, a wearable network manager to switch networks proactively and provide seamless handover experience under mobility.

In summary, my dissertation indicates the importance of all of the following:

- **Understanding the application requirements, network traffic patterns, and the interaction between the application and networking stack thoroughly in various settings**;
- **Designing practical and deployable network systems to solve problems and considering important factors, *e.g.,* highly changing network conditions in mobile networks**;

- **Carefully combining the application-transparent design and the application hint API depending on the targeted scenarios**.

8.1 Future Work

In the course of my research, I have learned that the in-depth understanding of the complex interaction between mobile applications and different levels of parallelism in mobile networking helps design the application-transparent network stack and improve application QoE. In the near future, using this guideline, I am interested in extending my research works by designing flexible networking systems for new types of applications and devices.

8.1.1 New Types of Applications: Beyond DEMS and Multipath over TCP

Different mobile apps exhibit various traffic patterns, from video playback to real-time streaming. While DEMS targets at reducing data chunk download time over multiple paths, some apps such as on-demand video streaming download multiple chunks at the same time and may require priority among multiple downloads. In video streaming such as HTTP adaptive streaming (HAS), the video player usually downloads multiple small video chunks of a few seconds in a burst [135]. To improve QoE of this type of apps, the multipath scheduler can prioritize video chunks that are going to be played first. DEMS can be extended to take such hints and prioritize the usage of different paths for different chunks. DEMS can also incorporate multipath policies to enable the tradeoff between the performance and energy consumption.

Besides applications that use HTTP and TCP for network transfers, an increasing number of applications and protocols use UDP as the underlying network transport today, such as video conferencing, VPN, RTP, and QUIC. These applications and protocols have various requirements. Video conferencing applications, *e.g.,* Skype [22], Google Hangouts [8], and Zoom [29], which use UDP, need both high bandwidth and low latency. VPN over UDP also benefits from high bandwidth as well. RTP is designed to deliver video and audio con-

tent with low latency. The latest UDP-based protocol QUIC targets at improved bandwidth usage and no head-of-line blocking for web browsing. All these usage scenarios can benefit from using multipath. However, applications and protocols using UDP do not benefit from MPTCP, as the design of MPTCP couples multipath and TCP together. MPTCP is designed and implemented to support multipath over TCP, which provides reliable data transfer with higher bandwidth, a network transport that may be different from what is necessary for the above usage scenarios. Supporting multipath over UDP is a promising direction to further improve QoE of apps that use UDP-based protocols.

There remain three main challenges in designing a multipath UDP transport: *(i)* how to make multipath scheduling decision among UDP sessions over multiple paths to satisfy different application requirements, *(ii)* how to react to packet loss and proactively prevent packet loss on a single path, since some apps like video conferencing may not necessarily need retransmission of video data as to prioritize the transmission of the latest video frames, and *(iii)* how to make multipath UDP transport be aware of the application adaptation logic, such as video bitrate adaptation.

8.1.2 New Types of Devices: from Smartwatch to Internet of Things (IoT) and Autonomous Vehicles (AVs)

The future belongs to connected devices. Besides wearable devices, there are two types of connected devices that have gained much attention in recent years.

The first is the Internet of Things (IoT). According to the research from International Data Corporation (IDC), the worldwide IoT market will grow from 655.8 billion dollars in 2014 to 1.7 trillion dollars in 2020 [7]. IoT devices, such as voice controller, smart bulb, smart thermostat, and smart lock, have enabled smart remote control through a centralized control device like a smartphone, making life much easier. Third-party developers can build applications to realize advanced control on a number of these smart devices. Existing IoT platforms, *e.g.,* Apple HomeKit [2], Samsung SmartThings [20], and Google Brillo

and Weave [9], have already gained much popularity among consumers today.

The second is Autonomous Vehicles (AVs). AVs have witnessed rapidly increasing interest from academic research [55, 81], government [5], and tech companies [15, 25, 27]. One key component in AVs is the network connectivity. AVs receive navigation information, updated map tiles and information of surrounding environment when necessary to help self-driving for example. AVs can also interact with the smart transportation system and other AVs by using the network access, *e.g.,* receiving traffic signals, acquiring traffic congestion information, and sending emergency messages or calls.

All types of connected devices including the smartphone, wearables, IoT, and AVs require network connectivity for different applications to function. While most of these devices have direct network access by connecting to existing WiFi or cellular network infrastructure, these devices can share their network resources to have Internet access together. For example, wearables and IoT devices can gain Internet access from the paired smartphone over Bluetooth to save energy when they are close to each other. AVs with cellular connectivity can act as WiFi APs for the user with a smartphone to access the Internet in the vehicle. Multiple devices thus from a **Personal Mesh Network** (PMN) to share network resources, which I believe is one of the future research directions for connected devices.

In PMN, at least one of the devices connects to the Internet and acts as a gateway to share Internet access to all of the devices. The device which acts as a gateway may change over time as the user moves from one location to another. PMN also enables device-to-device communication among multiple personal devices. To support such multi-device networking scheme, we need an integrated underlying networking stack that is general to diverse types of devices. To start with, we need an in-depth understanding of the requirement of IoT and AV applications and how these applications interact with the networks. Based on the insights, we plan to design a flexible networking stack for PMN that supports dynamic device-to-device network connection establishment and flexible network management.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] 2016 LTE Baseband Winners Announced: Intel, HiSilicon, MediaTek, Samsung LSI and Spreadtrum Gain Share reports Strategy Analytics. http://www.walb.com/story/35671441/2016-lte-baseband-winners-announced-intel-hisilicon-mediatek-samsung-lsi-and-spreadtrum-gain-share-reports-strategy-analytics.

[2] Apple HomeKit. https://www.apple.com/ios/home/.

[3] Byte Queue Limit. https://lwn.net/Articles/454390/.

[4] Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 20162021 White Paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[5] Connected and Automated Vehicles and New Technology White Paper, Michigan Department of Transportation. http://www.michigan.gov/documents/mdot/2040_SLRP_CAV_NewTech_readyforweb_40816_521014_7.pdf.

[6] ExoPlayer. http://developer.android.com/guide/topics/media/exoplayer.html.

[7] Explosive Internet of Things Spending to Reach $1.7 Trillion in 2020, According to IDC. http://www.businesswire.com/news/home/20150602005329/en/Explosive-Internet-Things-Spending-Reach-1.7-Trillion.

[8] Google Hangouts. https://hangouts.google.com/.

[9] Google takes on IoT with Brillo and Weave. http://internetofthingsagenda.techtarget.com/feature/Google-takes-on-IoT-with-Brillo-and-Weave.

[10] Google Web Page Replay Tool. https://github.com/chromium/web-page-replay.

[11] Introduction to LTE Advanced. http://www.androidauthority.com/lte-advanced-176714/.

[12] iOS: Multipath TCP Support in iOS 7. https://support.apple.com/en-us/HT201373.

[13] OPTICOM, PESQ - perceptual evaluation of speech quality. http://www.opticom.de/technology/pesq.php.

[14] P.862: Perceptual evaluation of speech quality (PESQ). https://www.itu.int/rec/T-REC-P.862-200102-I/en.

[15] Pittsburgh Offers Driving Lessons For Uber's Autonomous Cars. http://www.npr.org/sections/alltechconsidered/2017/04/03/522099560/pittsburgh-offers-driving-lessons-for-ubers-autonomous-cars.

[16] Qualcomm eXtensible Diagnostic Monitor. https://goo.gl/LODgRY.

[17] Qualcomm Still Standing Tall in Baseband Despite Intel, Says Wells Fargo. http://www.barrons.com/articles/qualcomm-still-standing-tall-in-baseband-despite-intel-says-wells-fargo-1490959969.

[18] QUIC Protocol. https://www.chromium.org/quic.

[19] Samsung Download Booster. http://www.samsung.com/uk/support/skp/faq/1061358.

[20] Samsung SmartThings. https://www.smartthings.com/.

[21] Shadowsocks Socks5 Proxy. https://shadowsocks.org.

[22] Skype: Free calls to friends and family. https://www.skype.com/en/.

[23] TCP Small Queues. https://lwn.net/Articles/507065.

[24] tcp_probe. http://www.linuxfoundation.org/collaborate/workgroups/networking/tcpprobe/.

[25] Tesla Autopilot. https://www.tesla.com/autopilot.

[26] The secret second operating system that could make every mobile phone insecure. https://www.extremetech.com/computing/170874-the-secret-second-operating-system-that-could-make-every-mobile-phone-insecure.

[27] Waymo. https://waymo.com/.

[28] Worldwide Smartwatch Market Will See Modest Growth in 2016 Before Swelling to 50 Million Units in 2020, According to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS41736916.

[29] Zoom: Video Conferencing, Web Conferencing, Webinars, Screen Sharing. https://zoom.us/.

[30] 3GPP TS 36.321: Medium Access Control (MAC) protocol specification (V10.3.0), 2011.

[31] In Korean, Multipath TCP is pronounced GIGA Path. http://blog.multipath-tcp.org/blog/html/2015/07/24/korea.html, 2015.

[32] Android ConnectivityManager. https://developer.android.com/reference/android/net/ConnectivityManager.html, 2017.

[33] myTuner Radio App - Free FM Radio Station Tuner. https://play.google.com/store/apps/details?id=com.appgeneration.itunerfree&hl=en, 2017.

[34] Network Access and Syncing on Android Wear 2.0. https://developer.android.com/training/wearables/data-layer/network-access.html, 2017.

[35] Pandora Music. https://play.google.com/store/apps/details?id=com.pandora.android&hl=en, 2017.

[36] Samsung SmartCam HD Plus 1080p Full HD Wi-Fi Camera. http://www.samsung.com/us/smart-home/security/cameras/smartcam-hd-plus-1080p-full-hd-wi-fi-camera-snh-v6414bn/, 2017.

[37] tinyCam Monitor PRO. https://play.google.com/store/apps/details?id=com.alexvas.dvr.pro&hl=en, 2017.

[38] Wear Camera Remote For Android Wear. https://play.google.com/store/apps/details?id=net.dheera.wearcamera&hl=en, 2017.

[39] M. Allman. Comments on Bufferbloat. *ACM SIGCOMM Computer Communication Review*, 43(1):30–37, 2013.

[40] A. Baiocchi, A. P. Castellani, and F. Vacirca. YeAH-TCP: Yet Another Highspeed TCP. In *Proc. PFLDnet*, volume 7, pages 37–42, 2007.

[41] M. Bednarek, G. Barrenetxea, M. Kühlewind, and B. Trammell. Multipath Bonding at Layer 3. In *ANRW*, pages 7–12, 2016.

[42] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, 2015.

[43] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *Selected Areas in Communications, IEEE Journal on*, 13(8):1465–1480, 1995.

[44] C. Caini and R. Firrincieli. TCP Hybla: a TCP Enhancement for Heterogeneous Networks. *International Journal of Satellite Communications and Networking*, 22(5):547–566, 2004.

[45] J. Chauhan, S. Seneviratne, M. A. Kaafar, A. Mahanti, and A. Seneviratne. Characterization of Early Smartwatch Apps. In *Pervasive Computing and Communication Workshops (PerCom Workshops), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.

[46] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau. QoE Doctor: Diagnosing Mobile App QoE with Automated UI Control and Cross-layer Analysis. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 151–164. ACM, 2014.

[47] X. Chen, T. Grossman, D. J. Wigdor, and G. Fitzmaurice. Duet: Exploring Joint Interactions on a Smart Phone and a Smart Watch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 159–168. ACM, 2014.

[48] Y.-C. Chen, Y.-s. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley. A Measurement-based Study of MultiPath TCP Performance over Wireless Networks. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 455–468. ACM, 2013.

[49] Y.-C. Chen, D. Towsley, and R. Khalili. MSPlayer: Multi-Source and multi-Path LeverAged YoutubER. In *CoNEXT*, 2014.

[50] Z. Chen, L. Jiang, W. Hu, K. Ha, B. Amos, P. Pillai, A. Hauptmann, and M. Satyanarayanan. Early Implementation Experience with Wearable Cognitive Assistance Applications. In *Proceedings of the 2015 workshop on Wearable Systems and Applications*, pages 33–38. ACM, 2015.

[51] Q. D. Coninck, M. Baerts, B. Hesmans, and O. Bonaventure. A First Analysis of Multipath TCP on Smartphones. In *17th International Passive and Active Measurements Conference*, volume 17. Springer, March-April 2016.

[52] X. Corbillon, R. Aparicio-Pardo, N. Kuhn, G. Texier, and G. Simon. Cross-layer Scheduler for Video Streaming over MPTCP. In *Proceedings of the 7th International Conference on Multimedia Systems*, page 7. ACM, 2016.

[53] A. Croitoru, D. Niculescu, and C. Raiciu. Towards WiFi Mobility without Fast Handover. In *NSDI*, pages 219–234, 2015.

[54] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. WiFi, LTE, or Both? Measuring Multi-Homed Wireless Internet Performance. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 181–194. ACM, 2014.

[55] Y. J. J. Ding Zhao, Yaohui Guo. TrafficNet: An Open Naturalistic Driving Scenario Library. In *Proceedings of the IEEE 20th International Conference on Intelligent Transportation System Conference (ITSC'17)*, Yokohama, Japan, October 2017.

[56] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in Smartphone Usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 179–194. ACM, 2010.

[57] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, 2003.

[58] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, 2013.

[59] A. Frommgen, T. Erbshäußer, A. Buchmann, T. Zimmermann, and K. Wehrle. ReMP TCP: Low Latency Multipath TCP. In *Communications (ICC), 2016 IEEE International Conference on*, pages 1–7. IEEE, 2016.

[60] C. P. Fu and S. C. Liew. TCP Veno: TCP Enhancement for Transmission over Wireless Access Networks. *Selected Areas in Communications, IEEE Journal on*, 21(2):216–228, 2003.

[61] J. Gettys. Bufferbloat: Dark Buffers in the Internet. *IEEE Internet Computing*, 15(3):96, 2011.

[62] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating Mobile Applications through Flip-Flop Replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 137–150. ACM, 2015.

[63] Y. Guo, F. Qian, Q. A. Chen, Z. M. Mao, and S. Sen. Understanding On-device Bufferbloat for Cellular Upload. In *Proceedings of the 2016 ACM on Internet Measurement Conference*, pages 303–317. ACM, 2016.

[64] Y. E. Guo, A. Nikravesh, Z. M. Mao, F. Qian, and S. Sen. Accelerating Multipath Transport through Balanced Subflow Completion. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. ACM, 2017.

[65] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards Wearable Cognitive Assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.

[66] S. Ha, I. Rhee, and L. Xu. CUBIC: a New TCP-friendly High-speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[67] M. Ham, I. Dae, and C. Choi. LPD: Low Power Display Mechanism for Mobile and Wearable Devices. In *USENIX Annual Technical Conference*, pages 587–598, 2015.

[68] B. Han, F. Qian, S. Hao, and L. Ji. An Anatomy of Mobile Web Performance over Multipath TCP. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 5. ACM, 2015.

[69] B. Han, F. Qian, and L. Ji. When Should We Surf the Mobile Web Using Both Wifi and Cellular? In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 7–12. ACM, 2016.

[70] B. Han, F. Qian, L. Ji, V. Gopalakrishnan, and N. Bedminster. MP-DASH: Adaptive Video Streaming Over Preference-Aware Multipath. In *CoNEXT*, pages 129–143, 2016.

[71] B. Hesmans and O. Bonaventure. An Enhanced Socket API for Multipath TCP. In *Proceedings of the 2016 Applied Networking Research Workshop*, pages 1–6, 2016.

[72] B. Hesmans, G. Detal, S. Barre, R. Bauduin, and O. Bonaventure. SMAPP: Towards Smart Multipath TCP-enabled APPlications. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 28. ACM, 2015.

[73] B. Hesmans, H. Tran-Viet, R. Sadre, and O. Bonaventure. A First Look at Real Multipath TCP Traffic. In *International Workshop on Traffic Monitoring and Analysis*, pages 233–246. Springer, 2015.

[74] J. D. Hester, T. Peters, T. Yun, R. A. Peterson, J. Skinner, B. Golla, K. Storer, S. Hearndon, K. Freeman, S. Lord, et al. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. In *SenSys*, pages 216–229, 2016.

[75] B. D. Higgins, K. Lee, J. Flinn, T. J. Giuli, B. Noble, and C. Peplin. The Future is Cloudy: Reflecting Prediction Error in Mobile Applications. In *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on*, pages 20–29. IEEE, 2014.

[76] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale. WearDrive: Fast and Energy-Efficient Storage for Wearables. In *USENIX Annual Technical Conference*, pages 613–625, 2015.

[77] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2012.

[78] 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 *ACM SIGCOMM Computer Communication Review*, volume 43, pages 363–374. ACM, 2013.

[79] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. *ACM SIGCOMM Computer Communication Review*, 44(4):187–198, 2015.

[80] Y. J. Jia, Q. A. Chen, Z. M. Mao, J. Hui, K. Sontinei, A. Yoon, S. Kwong, and K. Lau. Performance Characterization and Call Reliability Problem Diagnosis for Voice over LTE. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 452–463. ACM, 2015.

[81] Y. J. Jia, D. Zhao, Q. A. Chen, and Z. M. Mao. Towards Secure and Safe Appified Automated Vehicles. In *Proceedings of the 28th IEEE Intelligent Vehicle Symposium (IV'17)*, Redondo Beach, US, June 2017.

[82] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling Bufferbloat in 3G/4G Networks. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 329–342. ACM, 2012.

[83] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. *ACM SIGCOMM computer communication Review*, 33(2):83–91, 2003.

[84] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec. MPTCP is not Pareto-Optimal: Performance Issues and a Possible Solution. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2012.

[85] N. Kuhn, E. Lochin, A. Mifdaoui, G. Sarwar, O. Mehani, and R. Boreli. DAPS: Intelligent Delay-aware Packet Scheduling for Multipath Transport. In *Communications (ICC), 2014 IEEE International Conference on*, pages 1222–1227. IEEE, 2014.

[86] A. Kuzmanovic and E. W. Knightly. TCP-LP: A Distributed Algorithm for Low Priority Data Transfer. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1691–1701. IEEE, 2003.

[87] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928, 1996.

[88] D. Leith and R. Shorten. H-TCP: TCP for High-speed and Long-distance Networks. In *Proceedings of PFLDnet*, 2004.

[89] W. K. Leong, Y. Xu, B. Leong, and Z. Wang. Mitigating Egregious ACK Delays in Cellular Data Networks by Eliminating TCP ACK Clocking. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–10. IEEE, 2013.

[90] Y.-s. Lim, Y.-C. Chen, E. M. Nahum, D. Towsley, R. J. Gibbens, and E. Cecchet. Design, Implementation and Evaluation of Energy-Aware Multi-Path TCP. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 30. ACM, 2015.

[91] Y.-s. Lim, E. M. Nahum, D. Towsley, and R. J. Gibbens. ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths. In *Proceedings of the 2017 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, pages 33–34. ACM, 2017.

[92] R. Liu, L. Jiang, N. Jiang, and F. X. Lin. Anatomizing System Activities on Interactive Wearable Devices. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 18. ACM, 2015.

[93] R. Liu and F. X. Lin. Understanding the Characteristics of Android Wear OS. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–164. ACM, 2016.

[94] S. Liu, T. Başar, and R. Srikant. TCP-Illinois: A Loss- and Delay-based Congestion Control Algorithm for High-speed Networks. *Performance Evaluation*, 65(6):417–440, 2008.

[95] X. Liu, T. Chen, F. Qian, Z. Guo, F. X. Lin, X. Wang, and K. Chen. Characterizing Smartwatch Usage in the Wild. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 385–398. ACM, 2017.

[96] X. Liu, A. Sridharan, S. Machiraju, M. Seshadri, and H. Zang. Experiences in a 3G Network: Interplay between the Wireless Channel and Applications. In *Proceedings of the 14th ACM international conference on Mobile computing and networking*, pages 211–222. ACM, 2008.

[97] X. Liu, Y. Yao, and F. Qian. Rethink Phone-Wearable Collaboration From the Networking Perspective. In *Proceedings of the 2017 Workshop on Wearable Systems and Applications*, pages 47–52. ACM, 2017.

[98] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang. When Good Becomes Evil: Keystroke Inference with Smartwatch. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1273–1285. ACM, 2015.

[99] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297. ACM, 2001.

[100] H. Miao and F. X. Lin. Tell Your Graphics Stack That the Display is Circular. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*, pages 57–62. ACM, 2016.

[101] K. Nichols and V. Jacobson. Controlling Queue Delay. *Communications of the ACM*, 55(7):42–50, 2012.

[102] C. Nicutar, D. Niculescu, and C. Raiciu. Using Cooperation for Low Power Low Latency Cellular Connectivity. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 337–348. ACM, 2014.

[103] A. Nika, Y. Zhu, N. Ding, A. Jindal, Y. C. Hu, X. Zhou, B. Y. Zhao, and H. Zheng. Energy and Performance of Smartphone Radio Bundling in Outdoor Environments. In *Proceedings of the 24th International Conference on World Wide Web*, pages 809–819. International World Wide Web Conferences Steering Committee, 2015.

[104] A. Nikravesh, Y. Guo, F. Qian, Z. M. Mao, and S. Sen. An In-depth Understanding of Multipath TCP on Mobile Devices: Measurement and System Design. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 189–201. ACM, 2016.

[105] A. Nikravesh, D. K. Hong, Q. A. Chen, H. V. Madhyastha, and Z. M. Mao. QoE Inference Without Application Control. In *Internet-QoE@ SIGCOMM*, pages 19–24, 2016.

[106] 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*, pages 389–404. ACM, 2015.

[107] S. Nirjon, J. Gummeson, D. Gelb, and K.-H. Kim. Typingring: A Wearable Ring Platform for Text Input. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 227–239. ACM, 2015.

[108] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure. Experimental Evaluation of Multipath TCP Schedulers. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*, pages 27–32. ACM, 2014.

[109] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem. In *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*, pages 148–155. IEEE, 2013.

[110] B. Partov and D. J. Leith. Experimental Evaluation of Multi-path Schedulers for LTE/Wifi Devices. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, pages 41–48. ACM, 2016.

[111] Q. Peng, M. Chen, A. Walid, and S. Low. Energy Efficient Multipath TCP for Mobile Devices. In *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing*, pages 257–266. ACM, 2014.

[112] F. Qian, V. Gopalakrishnan, E. Halepovic, S. Sen, and O. Spatscheck. TM3: Flexible Transport-layer Multi-pipe Multiplexing Middlebox Without Head-of-line Blocking. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 3. ACM, 2015.

[113] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 321–334. ACM, 2011.

[114] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 21. ACM, 2011.

[115] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 266–277. ACM, 2011.

[116] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 177–188. ACM, 2014.

[117] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Understanding RRC State Dynamics through Client Measurements with Mobilyzer. In *Proceedings of the 6th annual workshop on Wireless of the students, by the students, for the students*, pages 17–20. ACM, 2014.

[118] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian, and S. Sen. Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild. In *Proceedings of the 2015 ACM conference on internet measurement conference*, pages 339–345. ACM, 2015.

[119] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, S. Venkataraman, and J. Wang. A First Look at Cellular Network Performance during Crowded Events. In *ACM SIGMETRICS Performance Evaluation Review*, volume 41, pages 17–28. ACM, 2013.

[120] S. Shen, H. Wang, and R. Roy Choudhury. I am a Smartwatch and I can Track my User's Arm. In *Proceedings of the 14th annual international conference on Mobile systems, applications, and services*, pages 85–96. ACM, 2016.

[121] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. LiveLab: Measuring Wireless Networks and Smartphone Users in the Field. *ACM SIGMETRICS Performance Evaluation Review*, 38(3):15–20, 2011.

[122] V. Singh, S. Ahsan, and J. Ott. MPRTP: Multipath Considerations for Real-time Media. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 190–201. ACM, 2013.

[123] J. Sommers and P. Barford. Cell vs. WiFi: On the Performance of Metro Area Mobile Connections. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 301–314. ACM, 2012.

[124] R. Stewart. Stream Control Transmission Protocol. RFC 4960, 2007.

[125] S. Sundaresan, W. De Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè. Broadband Internet Performance: A View from the Gateway. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 134–145. ACM, 2011.

[126] H. Wang, T. T.-T. Lai, and R. Roy Choudhury. Mole: Motion Leaks through Smartwatch Sensors. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 155–166. ACM, 2015.

[127] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *NSDI*, pages 473–485, 2013.

[128] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *NSDI*, pages 387–399, 2014.

[129] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 374–385. ACM, 2011.

[130] K. Winstein and H. Balakrishnan. TCP ex Machina: Computer-generated Congestion Control. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 123–134. ACM, 2013.

[131] K. Winstein, A. Sivaraman, H. Balakrishnan, et al. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, pages 459–471, 2013.

[132] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, volume 11, page 8, 2011.

[133] J. Wu, C. Yuen, B. Cheng, M. Wang, and J.-L. Chen. Streaming High-quality Mobile Video with Multipath TCP in Heterogeneous Wireless Networks. *IEEE Transactions on Mobile Computing*, 15(9):2345–2361, 2016.

[134] J. Xu, Q. Cao, A. Prakash, A. Balasubramanian, and D. E. Porter. UIWear: Easily Adapting User Interfaces for Wearable Devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. ACM, 2017.

[135] S. Xu, Y. J. Jia, Z. M. Mao, and S. Sen. Dissecting HAS VOD Services for Cellular: Performance, Root Causes and Best Practices. In *Proceedings of the 2017 ACM on Internet Measurement Conference*. ACM, 2017.

[136] Y. Xu, W. K. Leong, B. Leong, and A. Razeen. Dynamic Regulation of Mobile 3G/HSPA Uplink Buffer with Receiver-side Flow Control. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–10. IEEE, 2012.

[137] Y. Xu, Z. Wang, W. K. Leong, and B. Leong. An End-to-End Measurement Study of Modern Cellular Data Networks. In *PAM*, pages 34–45. Springer, 2014.

[138] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg. Adaptive Congestion Control for Unpredictable Cellular Networks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 509–522. ACM, 2015.

[139] L. Zhang, S. Shenker, and D. D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-way Traffic. *ACM SIGCOMM Computer Communication Review*, 21(4):133–147, 1991.