Improving Mobile Internet Performance with Cross-Device Network Transport and Cross-Layer Application Adaptation

by

Xiao Zhu

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
2022

Xiao Zhu

shawnzhu@umich.edu

ORCID iD: 0000-0002-0300-7676

To my family.

ACKNOWLEDGEMENTS

Five years ago, a lucky guy got a ship boarding pass. He soon realized it was not a cruise and the sea had a bad temper. He started to get scared, but he didn't give up. Today, he finally made it to the other end because he was never alone in this adventure.

I'm deeply indebted to my advisor, Zhuoqing Morley Mao, who provided constant support during my Ph.D. study. As an expert in many computer science fields, Morley helped me realize how lessons learned elsewhere can be cleverly applied to specific problems I was working on. I also could not remember how many times I was impressed by her brief yet to-the-point comments, which kept refreshing my mind and guided me in the right direction. It was always a pleasure working with her on each and every one of our challenging and rewarding research projects.

My dissertation committee member and long-time collaborator, Feng Qian, has been a mentor and friend. We worked together on most of the projects in this dissertation and many other exciting projects during my Ph.D. If I've learned anything from him, it's that research discussions and presentations can be made the most understandable regardless of the audience's technical expertise.

I would like to thank my other committee members, Mosharaf Chowdhury and Hessam Mahdavifar, for their insightful comments and constructive suggestions. This dissertation won't be complete without their valuable input.

During my Ph.D., I was fortunate to have not one but two internship experiences. My internship mentor, Subhabrata (Shubho) Sen at AT&T Labs–Research, opened my eyes to the video streaming industry. Shubho was very enthusiastic about science and technologies and always inspired me to examine things behind abstractions. I'm also truly honored to keep collaborating with him on our live streaming research project as part of this dissertation after my internship at AT&T. My Uber internship mentors, Yihua Ethan Guo and Rajesh Mahindra, engaged me in solving real networking problems for ride-sharing apps. This experience further strengthened my experimental and data analysis skills.

I would also like to thank Kira Barton, Dawn Tilbury, James Moyne, Yassine Qamsane, Ilya Kovalenko, and Mu Zhang. We worked together on many creative ideas to improve cyber-physical system performance and security. I learned a lot from them in how networked computers interact with physical processes and how mathematical models can be applied to solve systems problems.

I'm grateful to my friends and colleagues at RobustNet and CSE: Yihua Ethan Guo, Qi Alfred Chen, Ashkan Nikravesh, Mehrdad Moradi, Sanae Rosen, Yunhan Jia, Shichang Xu, Yuru Shao, David Ke Hong, Yikai Lin, Chao Kong, Jeremy Erickson, Jie You, Shengtuo Hu, Yulong Cao, Jiachen Sun, Xumiao Zhang, Can Carlak, Won Park, Eric Newberry, Jiwon Joung, Qingzhao Zhang, Ruiyang Zhu, Shuowei Jin, Wenyuan Ma, Junpeng Guo, Ze Zhang, Di Jin, Kaiyu Yang, Boyu Tian, Yibo Pi, Chun-Yu Chen, Duc Bui, *etc.* I will miss the tough and fun times we spent together at BBB. My thanks also go to my roommate, Xin Zan, and my other friends at Ann Arbor, in the States, and on the planet. I am lucky to have them by my side.

Our CSE staff, Ashley Andreae, Karen Liska, Stephen Reger, Zachary Champion, Steve Crang, *etc.*, have made everything in the department smooth and administrative things less concerned for me during my Ph.D. study.

Above all, I would like to thank my parents for their unconditional support and love. Throughout all these years, they sacrificed too much for me. Their positive life attitude motivates me to keep climbing.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table

ABSTRACT

The mobile Internet is becoming increasingly complex with a wide diversity of end systems (including wearables and automobiles), the co-presence of multiple devices with collaboration potential, and the growth of user-generated application traffic fueled by improved mobile sensors and wireless access. As the Internet evolves along with these trends, the increased complexity of different components and protocol layers makes it more challenging to achieve high network utilization and meet the diverse QoE requirements for mobile applications. As a result, despite the richness of various network resources, the performance of today's mobile applications still falls behind expectations. To address this challenge, in this dissertation, I demonstrate that with a better understanding of the various components and different protocol layers of the increasingly complex mobile Internet, we can identify unique performance problems and leverage such knowledge to develop network transport protocols with cross-device awareness and application adaptation strategies with cross-layer considerations for better mobile app performance.

Specifically, to understand the interaction between multiple mobile devices and its impact on end-to-end performance, we conduct an empirical study on wearable networking, where wearables oftentimes rely on their paired smartphones for Internet access. Based on our measurement findings, we develop cross-device network transport management solutions for improving wearable networking performance. To better support multi-device collaboration in a more general setting, we develop MPBond, a distributed multipath transport system for efficient network-level collaboration among personal mobile devices, with cross-device connection management and packet scheduling. To explore opportunities on the mobile application design, we build Livelyzer, a generalized measurement tool for characterizing commercial live video streaming upstream ingest performance under mobile networks. Based on our measurements, we identify deficiencies in broadcasting app rate adaptation, and propose network-aware adaptation strategies to improve the same. We also investigate another emerging mobile application, vehicular sensing for autonomous driving, where we build Harbor, a cross-layer system architecture for collaborative vehicular sensor data sharing, with adaptive usage of V2V and V2I network resources.

CHAPTER I

Introduction

The Internet has witnessed dramatic changes over the past decade. The worldwide usage of mobile devices exceeded desktop computers for the first time. Thousands of mobile apps are being added to the Play Store every day. Various smart devices have been equipped with mobile OSes and wireless network interfaces. Mobile network traffic has grown by 17-fold from 2012 to 2017 and is expected to further increase by 7-fold by 2022 [23].

We observed three major trends as the mobile Internet keeps evolving:

- **Increased diversity of end systems.** Diverse end systems such as wearable devices and connected vehicles are gaining popularity as fueled by new hardware, OS support, and applications. This introduces new networking and application paradigms. For example, instead of directly accessing the Internet with TCP/IP, a wearable device such as a smartwatch usually speaks Bluetooth and uses its paired smartphone as a “gateway”. Connected vehicles usually use V2X communications, where each vehicle can exchange data through either vehicle-to-vehicle (V2V) or vehicle-to-infrastructure (V2I) communications.

- **Co-presence of multiple local devices.** It is increasingly common that a user possesses multiple mobile devices. Smart mobile devices are equipped with diverse network interfaces such as cellular, Wi-Fi, and Bluetooth, making them capable of communicating with remote Internet servers and other local devices. This growing trend of multi-device ownership and multi-interface provision creates abundant wireless resources and enhanced network connectivity for multi-device collaboration. Mobile devices from different users can also collaborate, *e.g.,* connected vehicles in the same area can share each other's sensor data through V2V or/and V2I communications.

- **Growing upload traffic generated by users.** The rise of user-generated content makes mobile apps more interactive and prevalent. For example, personalized live streaming services such as Youtube and Facebook Live have gained ubiquitous access thanks to the universal coverage of cellular networks and the integration of HD cameras in commodity mobile devices. Connected and autonomous vehicles generate high-volume data through local sensors and transfer them to more powerful edge servers for obstacle detection. These types of applications usually impose a higher latency requirement to ensure interactivity.

However, as the mobile Internet evolves, the increased complexity of different components makes it more challenging to achieve high network utilization and meet the diverse QoE requirements for mobile applications. End system diversity brings more challenges in managing the networking stack for different use cases and wireless technologies. The presence of multiple mobile devices creates opportunities to transfer data cooperatively. Still, it also requires architectural support and strategic scheduling to fully utilize the WWAN and WLAN resources on different devices. Interactive applications such as live video streaming and analytics usually have a complex pipeline that includes performance-impacting components at different layers such as the codec and network transport, making it challenging to satisfy the QoE requirement that desires both high throughput and low latency. As a result, despite the richness of diverse network resources, the performance of today's mobile applications still falls behind expectations. These mobile applications include both the same versions of common Internet applications running on mobile devices and new applications enabled by mobile computing.

This dissertation is dedicated to addressing these challenges. The overall goal is to improve network transport protocol design and application adaptation along with these trends. Sitting on the top of the Internet stack, the application and transport layers play crucial roles in determining the end-to-end performance of mobile applications. Traditionally, the transport layer was designed end-to-end for a single client-server pair over a single network path. However, the aforementioned trends brought new components to this end-to-end path, including multiple wireless links and multiple mobile devices in a collaboration group, allowing opportunities to extend existing network transport design with cross-device awareness. The application layer can also benefit from information from and control at the lower layers (*e.g.,* transport, network, and MAC layers), especially in wireless networks.

My dissertation demonstrates that: **With a better understanding of the various components and different protocol layers of the increasingly complex mobile Internet, we**

Table 1.1: Summary of dissertation work.

Problem scope	Project
Characterizing and improving wearable network transport management	Understanding the Networking Performance of Wear OS
Designing distributed multipath transport for multiple mobile devices	MPBond: Efficient Network-level Collaboration among Personal Mobile Devices
Characterizing and improving mobile live video upload rate adaptation	Livelyzer: Analyzing the First-Mile Ingest Performance of Live Video Streaming
Designing collaborative sensing architecture for vehicular applications	Harbor: Hybrid Architecture for Collaborative Vehicular Sensing

can identify unique performance problems and leverage such knowledge to develop network transport protocols with cross-device awareness and application adaptation strategies with cross-layer considerations for better mobile app performance. As summarized in Table 1.1, this dissertation explores this problem along with four use cases.

1.1 Characterizing and Improving Wearable Network Transport Management

We explore the networking performance on wearable systems that do not always directly access the Internet with end-to-end TCP/IP and often rely on a paired smartphone as the gateway. We first identify the limitations of existing networking stacks on wearable systems by conducting the first in-depth investigation of the networking performance of Wear OS, one of the most popular OSes for smartwatches and potentially other wearable systems. Our measurement study reveals that the existing Wear OS suffers from serious performance issues regarding key aspects that distinguish wearable networking from smartphone networking. To mitigate the identified performance impairment, we design, implement, and evaluate several readily deployable transport management solutions and demonstrate that wearable networking performance can be improved with a better understanding of the end system diversity and heterogeneous wireless links.

1.2 Designing Distributed Multipath Transport for Multiple Mobile Devices

Part 1 sheds some light on how a smartphone can play an essential role in wearable networking and the complex interactions when there are multiple mobile devices and wireless links in the end-to-end path. In part 2, we stand on the opposite side and explore the feasibility of leveraging wearables' network interfaces to bring benefits to smartphones, which further leads to a more general question: Can we leverage the network interfaces of multiple nearby mobile devices to improve networking performance? To this end, we propose MPBond, an efficient system allowing multiple personal mobile devices to collaboratively fetch content from the Internet. Inspired by the success of multipath TCP (MPTCP), MPBond applies the concept of distributed multipath transport, where multiple subflows can traverse different devices. We develop device/connection management schemes, a buffering strategy, a packet scheduling algorithm, and a policy framework tailored to MPBond 's architecture to efficiently utilize the heterogeneous network resources. We evaluate MPBond using real-world mobile devices, networks, and applications and demonstrate that a cross-device transport protocol considering the interaction of multiple mobile devices and heterogeneous wireless links improves the mobile application performance.

1.3 Characterizing and Improving Mobile Live Video Upload Rate Adaptation

While parts 1 and 2 improve the networking stack design on mobile systems without modifying the applications, in part 3, we look at the problem from a different perspective and try to optimize the application design to adapt to varying network conditions to improve mobile application performance. We focus on the emerging live streaming application, which is both bandwidth-intensive and latency-sensitive. We aim to understand how commercial live streaming broadcast and distribution platforms such as Youtube and Facebook Live perform over mobile networks. Specifically, we look at the upstream ingest path from the broadcasting app to the video server, which is responsible for capturing the video content with a camera, encoding it, and transmitting it over cellular or Wi-Fi uplinks. Delivering high-quality video over mobile uplinks in real time is challenging, and there exists little related research. To this end, we develop Livelyzer, a tool to analyze the first-mile ingest path of commercial live streaming, and provide best-practice suggestions to developers. Our study demonstrates that existing live video upload applications incur poor coordination between the application decisions and network conditions, and schemes that better adapt real-time encoding rates to network bandwidths can improve QoE.

1.4 Designing Collaborative Sensing Architecture for Vehicular Applications

In part 4, we further study the mobile application design by looking at another emerging live video analytics application: collaborative vehicular perception. Connected and autonomous vehicles digest real-time sensor data to understand the physical world. Nearby vehicles can share each other's sensor data for collaborative sensing to form a complete view with a higher resolution. We design Harbor, a hybrid system architecture that leverages both the direct communication between vehicles (V2V) and direct communication between vehicles and the remote server (V2I) to better utilize the available network and compute resources. We develop methods for dynamically establishing of different V2V and V2I channels to better adapt to heterogeneous network resources and an algorithm that efficiently relays sensor data considering the available V2V and V2I resources. This part demonstrates that a flexible and adaptive sensor data sharing application architecture considering the underlying wireless networking protocols improves collaboration perception performance for autonomous driving.

1.5 Thesis Organization

The rest of the dissertation is organized as follows. We first provide background knowledge and summarize related work in Chapter II. Chapter III presents our experimental study on the networking performance of Wear OS, where we characterize wearable network transport management and propose design improvements. In Chapter IV, we propose MPBond, a distributed multiple transport system we develop to realize efficient network-level collaboration among personal mobile devices. Chapter V describes our study on live streaming ingest performance, where we characterize and improve the broadcasting app rate adaptation over mobile uplinks. In Chapter VI, we develop Harbor, a hybrid system architecture for collaborative vehicular sensing. We conclude this dissertation in Chapter VII.

CHAPTER II

Background and Related Work

This chapter introduces the background of these emerging mobile systems and applications.

2.1 Wearable Networking

Wearable networking is *important*. Take smartwatches as an example. One may argue they only incur light traffic, such as push notifications. This might be true for the *current* smartwatch ecosystem where traffic flows are mainly small, short, and bursty [103]. However, we envision that future wearable apps will be more network-intensive by incurring much heavier network activities fueled by new hardware, OS support, and applications. For example, recently debuted speaker/LTE-capable watches such as LG Watch Urbane 2nd Edition allow users to directly make hands-free VoIP calls; the latest Wear OS 2.x allows standalone apps on wearables; also, many emerging wearable applications incur heavy network traffic such as continuous computer vision on smart glasses [73, 56], remote camera preview [15], real-time screen projection [10], and network-level collabora-

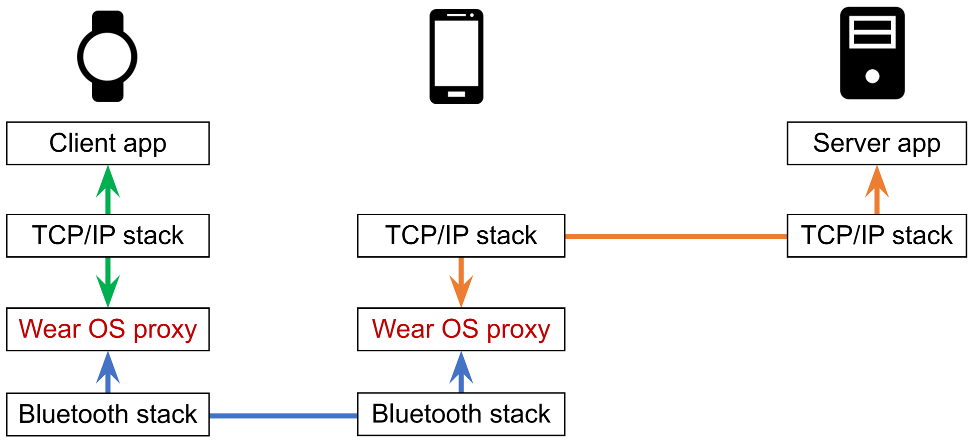


Figure 2.1: The wearable networking workflow of Wear OS.

tion between phone and watch [105].

Wearable networking is also *different* from smartphone networking that has been well-studied in the past decade. First, most of the time, a wearable transfers data with Bluetooth (BT), whose many characteristics are different from WiFi and cellular that dominate the smartphone interface usage. BT has a complex protocol stack and various working modes. Second, a wearable often does not directly access the Internet; instead, it uses its paired smartphone as a "gateway", which, if not carefully designed, may incur additional performance degradation. Specifically, since BT by default does not speak TCP/IP, the wearable OS typically introduces a pair of proxies on the smartphone and the wearable to bridge TCP/IP and BT, as shown in Figure 2.1. The phone-side proxy maintains TCP connections to remote servers on behalf of the wearable. The wearable-side proxy also maintains local TCP connections with client apps. This way, everything becomes transparent to the client and server applications. Last but not least, due to BT's short range, network handovers frequently occur on a wearable: when it moves away from the phone, the BT connectivity will be torn down, and the wearable has to use standalone WiFi or LTE to communicate with the external world.

2.2 Multipath Transport

Multipath transport is a promising technique that simultaneously leverages multiple network paths to accelerate data transfers. In mobile networks, most existing research focuses on the joint use of WiFi and cellular on a single mobile device [112, 72, 113], to improve the QoE of mobile apps such as video streaming and web browsing.

MPTCP [131], the *de facto* multipath solution, brings a shim layer between the socket interface and multiple underlying TCP subflows. Operating at the transport layer, it requires no modifications to both applications and networks. The MPTCP sender distributes data onto multiple TCP subflows; the receiver reassembles the data into the original byte stream and transparently delivers it to the application layer.

As the core component of a multipath transport system, a packet scheduler distributes data onto potentially heterogeneous network paths. MinRTT [118] is the default scheduler of MPTCP, which selects the path with available space in the congestion window and the minimum network RTT. There are also studies on innovating the scheduling algorithm design to improve MPTCP performance [69, 97, 135].

2.3 Live Video Streaming

As shown in Figure 2.2, the live video streaming pipeline consists of the ingest path and the distribution path. The ingest path is responsible for capturing and encoding a video

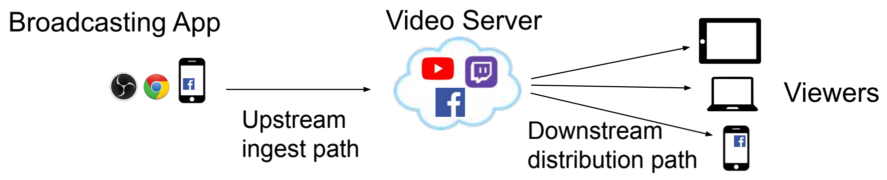


Figure 2.2: Live video streaming end-to-end workflow.

from a broadcasting app on the broadcaster device, transmitting it over the network uplink, and transcoding the received video stream at the video server into a number of different adaptive bitrate or ABR tracks. The distribution path delivers these different versions of the video to different viewers.

The performance of the first-mile ingest path is critical to the end-to-end QoE because the video delivered on the first mile imposes an upper limit on the quality of ABR tracks created from it. However, this all-important first mile is largely under-explored, with existing research primarily focusing on the downstream distribution path.

One important research problem for the ingest path is broadcasting app rate adaptation, where it needs to determine the encoding rate in real time. The encoding rate choice impacts the quality and timeliness of the stream received by the video server, especially under time-varying mobile uplink network conditions, which eventually affects the end viewer QoE.

2.4 Connected and Autonomous Vehicles

Autonomous vehicles rely on various on-board 3D vision sensors to understand the physical world consisting of road segments, pedestrians, cyclists, other cars, *etc.*, to make correct driving decisions. For example, LiDAR (Light Detection and Ranging) [34, 39] is a major on-board vision sensor, which fires laser lights at different angles and measures how long it takes for the lights to return to the sensor after reflection from objects, based on which it calculates the distance of these objects and generates 3D point clouds to represent the surroundings. Different software modules will further process the collected point cloud data and make appropriate driving decisions.

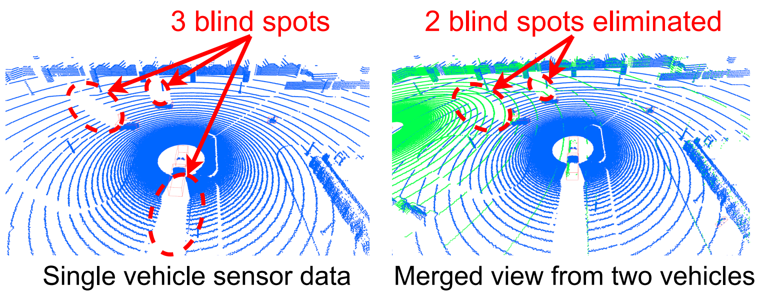


Figure 2.3: A LiDAR point cloud example showing the benefits of collaborative sensing by merging a nearby vehicle's data (green) to a single vehicle (blue).

However, a single vehicle can only sense a limited range, and its view may be further restricted due to occlusion. One promising solution to overcome these limitations is collaborative sensing, where multiple vehicles share sensor data using wireless networks, given that today's vehicles are "connected cars": they are increasingly equipped with WiFi and cellular interfaces [3, 7, 1]. Collaborative sensing would thus benefit autonomous driving and Advanced Driving Assistance Systems (ADAS) by merging sensor data from different vehicles to form a complete view. Video analytics tasks (*e.g.,* drivable space detection [67] and object detection [151, 92]) can then be performed based on the merged 3D data, whose results can finally be shared among all the cars through network communications. Figure 2.3 shows a concrete example where the point cloud data are generated from a state-of-the-art autonomous driving simulator [31]. As shown, in a single vehicle's point cloud data (marked as blue), there are three blind spots caused by occlusion. After merging this point cloud with another one from a nearby vehicle (marked as green), two of the three blind spots can be eliminated.

2.5 Related Work

We summarize the related work in several categories below.

Smartwatch systems, energy, and applications. There are a few recent studies on understanding the smartwatch systems, energy, traffic patterns, and applications. Liu *et al.* [100] examined the execution efficiency of Android Wear OS and identified some inefficiencies and their root causes. Liu *et al.* [103] performed an in-depth characterization of usage patterns, energy consumption, and network traffic of smartwatch usage from real users in the wild. Kolamunna *et al.* [86] studied the user behavior and application traffic characteristics for SIM-enabled wearables. Chauhan *et al.* [50] characterized smartwatch apps. There also exist studies on other aspects of wearable systems including display [107], storage [78], user interface [152, 55], and security [147, 106]. Our work [165] thoroughly looks into the networking stack and its performance on smartwatches under diverse scenarios, which is not well understood, and proposes new network transport management schemes for wearables.

Multi-device network-level collaboration. Previous studies have demonstrated the benefits of leveraging multiple mobile devices to boost network performance. Kibbutz [111] is a scheme that leverages tethering and MPTCP. PRISM [85] is a system that uses inverse multiplexing of TCP flows to support collaboration. COMBINE [46] is a framework that uses HTTP byte-range requests split over multiple devices for bandwidth aggregation. There are also solutions working in the application layer targeting a specific type of applications [83, 134]. Our work [168] instead is light-weight and offers several advantages, including better performance as boosted by its judiciously designed scheduler, buffer management, and connection split schemes, application transparency, and more flexibility. Also, none of the above work has been applied to wearables. Besides network-level collaborations, there also exist systems [115, 44, 116] that share other I/O resources among multiple mobile devices.

Characterizing live video streaming. Recent studies examined both the technical aspects [146, 137, 136, 104] and human factors [142, 74] of live video streaming. There exist studies using service-specific APIs to study Periscope and Facebook Mobile [137, 136, 146]. LIME [104] studied 360-degree live streaming from an open-source broadcasting software to Facebook and Youtube. Compared to these studies, our work [167] differs in three significant ways: First, we focus on the upstream ingest path of the end-to-end live streaming pipeline. Second, their methodologies are based on specific service features compared to our more general measurement approach. Third, we suggest and evaluate design best practices for live video upload on the ingest path.

Collaborative vehicular sensing. Various efforts have been made on cooperative vehicular perception. The use of sensor data from multiple sources for vehicular perception has been shown to be beneficial [117]. Existing data sharing systems either transfer sensor data between vehicles [128, 52, 90] or directly send each vehicle's data to an edge or cloud server [162]. Our work advocates the use of both direct communication between multiple vehicles (V2V) and direct communication between each vehicle and the server (V2I) to better adapt to different network connectivity and bandwidth conditions for each vehicle.

We develop solutions for the dynamic establishment of V2V and V2I data channels in a holistic network system to adapt to heterogeneous V2X network resources, as well as algorithms that efficiently assign relay vehicles considering the wireless network and physical properties.

CHAPTER III

Understanding the Networking Performance of Wear OS

This chapter takes a first look at the wearable network stack, especially its transport management aspects. Through carefully designed controlled experiments conducted in a cross-device, cross-protocol, and cross-layer manner, we identify serious performance issues of Wear OS, which are unique to wearable networking. We pinpoint their root causes and quantify their impacts on network performance and application QoE. We further propose practical suggestions to improve wearable networking performance with cross-device awareness in transport management.

3.1 Introduction

Smart wearable devices are becoming increasingly popular. Take smartwatches, arguably the most important type of smart wearables, as an example. According to a market research report published recently [20], the global market value of smartwatches was estimated to be $10.2 billion in 2017 and will experience an annual growth rate of 22.3% from 2018 to 2023.

In the literature, several efforts have been made towards understanding and improving the OS execution performance [100, 99], power management [103], graphics and display [107], storage [78], and user interface [152, 55] of wearable OSes. In this chapter, we investigate an important yet underexplored component: *the wearable networking stack.* We conduct to our knowledge a first in-depth investigation of the networking performance of Wear OS, one of the most popular OSes for wearables. Wear OS is a version of Google's Android OS tailored to small-screen wearable devices. Used by a wide range of smartwatches and potentially other wearables, Wear OS is expected to account for 41.8% of the market share of smartwatch OSes in 2020 [18].

Understanding the networking performance of commercial wearables is challenging, as it involves multiple devices, networks, and protocols, which incur complex interactions. The proprietary nature of Wear OS makes it even harder to gain deep visibility into the wearable networking stack. Note that unlike Android for handheld devices, Wear OS is not open-source.

To address these challenges, we first build a wearable networking testbed consisting of commodity Wear OS based smartwatches, off-the-shelf smartphones, commercial wearable apps, as well as a series of tools we developed for instrumenting the system and collecting various types of data. We then leverage the testbed to conduct controlled experiments in a cross-device, cross-protocol, and cross-layer manner. Through judiciously designed experiments, we demystify the Wear OS networking stack and quantify how it affects the wearable networking performance. Our key findings consist of the following.

• When acting as a gateway proxy for a wearable, the phone dramatically inflates the end-to-end (server to wearable) latency to 30+ seconds due to its incurred "bufferbloat". We then break down the end-to-end latency into various components, and identify the root cause to be the phone-side TCP receive buffer, whose configuration does not take into account the path asymmetry between the wearable-phone path and the phone-server path (§3.3).

• Wearables are equipped with multiple network interfaces such as BT and WiFi. When multiple networks are available, the Wear OS's default interface selection policy strictly prefers one interface (*e.g.,* BT) over others (*e.g.,* WiFi). However, we find that such a strategy oftentimes leads to suboptimal tradeoffs between performance and energy consumption. In addition, we explore the feasibility of performing multipath transport (simultaneously using WiFi and BT) on wearables, and identify potential obstacles such as the interference between BT and 2.4GHz WiFi (§3.4).

• BT's short communication range makes handovers occur frequently on wearables. Due to insufficient protocol support and poor cross-layer coordination, a BT-WiFi handover may last for more than 60 seconds, leading to significant disruption of the wearable application performance. By looking into each phase of a handover, we find that both the OS and user application are responsible for such unacceptably long handover delays (§3.5).

The above performance inefficiencies are caused by the poorly designed networking stack of Wear OS. Our identified issues appear on all 8 wearables of heterogeneous vendors and Wear OS versions (including the latest version as of December 2018) as well as a variety of paired phones as tested by us using synthetic and real apps. To mitigate the identified performance impairment, we design, implement, and evaluate several readily deployable transport management solutions including the following. (1) We develop a simple yet effective flow control scheme that mitigates the phone-side bufferbloat problem, achieving up to 78x latency reduction with less than 3% of the throughput decrease (§3.3.3), (2) We design and implement to our knowledge a first multipath transport framework for wearable devices that enables adaptive interface selection, multi-network bandwidth aggregation (§3.4.2), and smooth handovers between IP and non-IP networks (§3.5.4). For example, our improved handover scheme reduces the BT-to-WiFi handover delay from more than 28 seconds to less than 0.6 seconds with negligible energy overhead incurred.

3.2 Background and Methodology

The wearable networking is unique in several aspects, making analyzing its performance and resource consumption challenging.

• Instead of accessing the Internet directly, a wearable typically leverages a paired mobile device such as a smartphone as a gateway.

• Compared to performing pure TCP/IP networking on a regular host, wearable networking involves both BT and TCP/IP. In particular, since BT by default does not speak TCP/IP, the wearable OS typically introduces a pair of proxies on the smartphone and the wearable to bridge TCP/IP and BT. For the phone-side proxy, it maintains TCP connections to remote servers on behalf of the wearable. It strips off TCP/IP (BT) headers for downlink (uplink) traffic, and encapsulate the application data into BT (TCP/IP) packets. A reverse operation is performed at the wearable-side proxy, which also maintains local TCP connections with client apps.

• The BT protocol stack itself is complex. It consists of higher-layer protocols realized in the *host* (software) and lower-layer functions implemented in the *controller* that resides on the BT chip. The host and controller are bridged by the Host-Controller Interface (HCI).

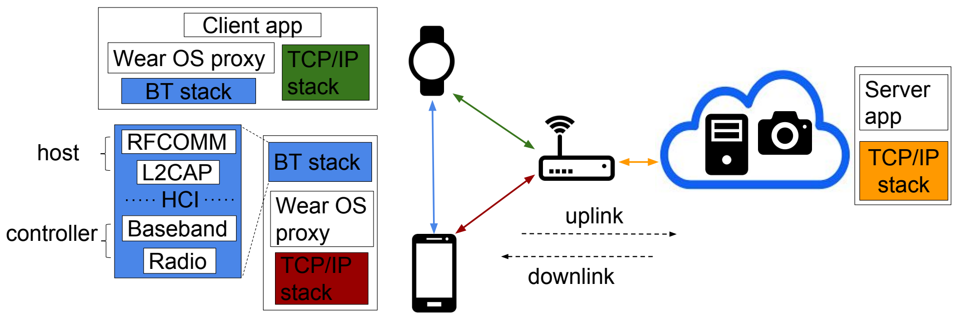


Figure 3.1: The measurement testbed (middle) and the protocol stacks of the wearable, phone, and server (left and right).

The BT performance can thus be affected by multiple factors at different layers as well as its interplay with TCP/IP and the aforementioned proxying mechanism.

• Wearable OS developers usually keep their implementation proprietary. Unlike Android for handheld, Wear OS is not open-source.

To address the above challenges, our high-level approach is to develop a holistic testbed and a suite of measurement tools that comprehensively examine not only each of the aforementioned components, but also the cross-device, cross-protocol, and cross-layer interplay on real wearables over real wearable apps' workload. We next describe our testbed and measurement toolkit design.

3.2.1 Wearable Networking Testbed

We set up a testbed shown in Figure 3.1 to cover common usage scenarios for a wearable to communicate with the external world. They include communicating locally with the phone over BT, accessing the Internet directly with WiFi/LTE, as well as surfing the Internet via the smartphone as the gateway (called the *CPROXY* mode in Wear OS). Our

Table 3.1: Mobile devices used in our experiments.

Smartwatch Model	Wear OS Version	Paired Smartphone	Smartphone Android OS
LG Urbane	1.5	Nexus 5	6.0.1
LG Urbane	2.15	Nexus 5X	7.1.1
LG Urbane 2nd Edition	2.0	Samsung Galaxy S5	5.1.1
LG Urbane 2nd Edition	2.20	Pixel 2	9.0.0
Huawei Watch	2.0	Nexus 6P	7.0.0
Huawei Watch 2	2.9	Nexus 5X	7.1.1
Asus ZenWatch 3	2.0	Nexus 5	6.0.1
LG G Watch R	2.0	Nexus 5	6.0.1

testbed consists of 8 state-of-the-art smartwatches listed in Table 3.1. All of them support BT and WiFi while some higher-end watches such as LG Urbane 2nd Edition and Huawei Watch 2 support LTE as well. The OSes we study include the latest release (Wear OS 2.20 released in December 2018) as well as the older Android Wear OSes 2.x and 1.x. Our measurement findings apply to all OSes unless otherwise mentioned. The Internet server we use is equipped with a quad-core 3.6GHz CPU and 16GB memory, running Ubuntu 16.04. We run on the testbed the workload generated by our measurement tools (described shortly) and real apps that perform bulk data transfer, constant bitrate transfer, and real-time streaming. We also employ a Samsung SNH-V6414BN SmartCam to stream real-time video to smartphones and smartwatches.

3.2.2 The Wearable Network Measurement Tools

Given a lack of tools for measuring and analyzing wearable network performance especially over BT, we also develop a suite of tools to fill this gap. They consist of software programs for both active and passive measurements. We will use them to conduct carefully crafted black-box testing without requiring the OS source code. This is to our knowledge the most comprehensive toolkit for wearable networking performance analysis and diagnosis.

For active measurements, we develop a custom server application running on the server and a custom client app running on the wearable. Supporting all aforementioned communication paradigms, the client and server apps can exchange data using two traffic patterns: bulk data transfer and constant bitrate over the uplink (from the wearable), downlink (to the wearable), or both. Our application also allows automatic reconnection upon network failure for testing the handover support, an important feature needed for wearables due to their short BT range (§3.5).

For passive measurements, we collect both WiFi and BT traces on multiple entities (phone/wearable/server). The BT trace is captured at both the host-controller interface (HCI, using `btsnoop` log) and the OS (using `tcpdump`), and contains both the data packets and the BT control messages. In addition to the network traces, we collect the network state and signal strength information to understand their impact on network performance. We also develop a tool that can instrument different components of the packet transmission/reception pipeline in the OS kernel to identify the performance bottleneck for the end-to-end data delivery (§3.3.2).

Compared to prior measurement studies, our measurement and instrumentation techniques are comprehensive in that they cover multiple entities (wearable, phone, server), protocols (BT, WiFi), and instrumented layers (HCI, OS, TCP, App). Note this may not be the case for many prior works. For instance, some previous studies on smartwatches [103, 157] collect BT traces only at HCI, incurring various limitations such as inaccurate goodput measurement (due to lower-layer padding) and not being able to separate individual application streams from the multiplexed traffic captured at HCI. Some other methods [81, 140, 71] extract the RTT from only one side, and are therefore incapable of inferring the end-to-end RTT when a wearable–server connection is split by a phone when the *CPROXY* mode is used. Table 3.2 lists all types of data collected by our toolkit. The runtime CPU overhead of collecting those types of data is less than 3% on our wearables.

The collected data will be analyzed offline. Given a lack of tools to decode BT messages, we follow the BT specification [21] to build a tool that can parse the BT traffic to extract both the user payload and control messages. In addition, this offline tool can perform various types of correlation analysis on different data sources, including cross-technology (e.g., WiFi vs. BT), cross-device (e.g., wearable vs. phone), and cross-layer (e.g., app performance vs. BT radio state) correlation analysis. Our toolkit is written in about 3,000 LoC using C++, Java, and Python. We have open-sourced the entire toolkit on GitHub [25].

Leveraging the above measurement infrastructure, we next answer the following important research questions.

- How does the smartphone gateway impact the performance?
- How does the network selection policy affect the tradeoff between performance and energy consumption?
- What is the performance when a network handover occurs?

Table 3.2: Data collected by our measurement toolkit.

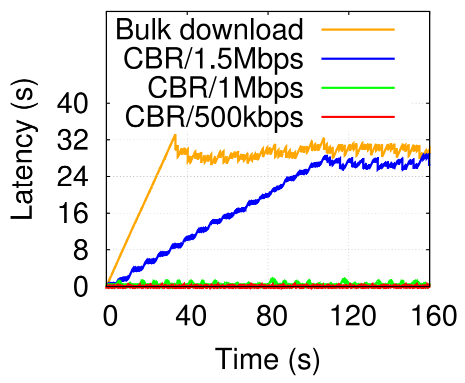
Category	Data Item	Method	Source
Watch-side	BT HCI trace	Callback	`btsnoop` log
	BT and WiFi packet trace		`tcpdump`
	BT RSSI	Poll (0.2s)	Wear OS API
	BT and WiFi network state		
Phone-side	BT HCI trace	Callback	`btsnoop` log
	TCP/IP packet trace		`tcpdump`
	kernel packet transmission events		Kernel log
	BT packet trace		Android log
Server-side	packet trace	Callback	`tcpdump`

3.3 Impact of Smartphone Proxying

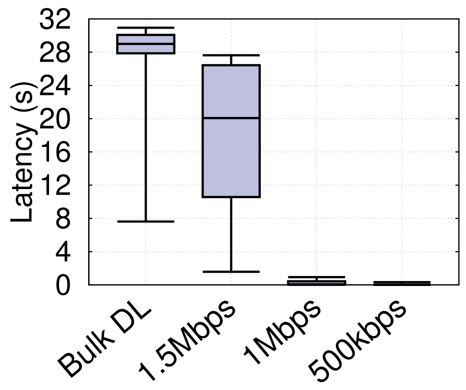
As mentioned earlier, a paired smartphone gateway plays a critical role in wearable networking. In this section, we study the performance impact of the *CPROXY*. Recall that typically residing on a paired phone, the *CPROXY* splits an end-to-end client-server connection into a server-phone TCP connection and a phone-wearable BT RFCOMM connection, while being transparent to both the wearable-side and server-side apps. Because of the two heterogeneous links, the *CPROXY* needs multiple buffers at various layers, such as the receive buffer in the TCP/IP stack, the app-layer buffer, and the transmission buffers in the BT RFCOMM stack. These buffers, along with other existing in-network and on-device buffers, can potentially cause “bufferbloat” that inflates the end-to-end delay. This is particularly undesired for real-time traffic with low latency requirements.

3.3.1 Substantial Bufferbloat in *CPROXY*

We begin with characterizing the overall end-to-end latency under the *CPROXY* mode. Specifically, we measure the one-way delay (OWD) from the server to the wearable, an



((a)) E2E delay over time.



((b)) E2E delay distribution.

Figure 3.2: E2E delay of bulk transfer and CBR traffic (LG Urbane paired with Nexus 5X, normal RSSI).

important performance metric for real-time applications. To emulate the real-time traffic, we use the CBR traffic with three rates as the workload: 1.5Mbps, 1Mbps, and 500kbps. For comparison, we also measure the OWD of bulk data download without a rate limit.

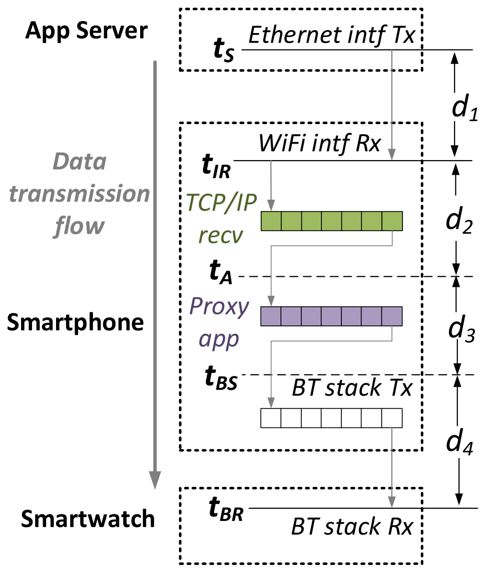
For each sending rate, the server sends data to the wearable for at least 160s. The OWD from the server to the wearable is an important performance metric for real-time applications. It is continuously measured as the difference between the transmission and the reception time of each byte. Before each experiment, we connect the wearable through a USB cable to the server, and use a custom program we developed to synchronize their clocks.

Figure 3.2 shows the OWD of CBR traffic and the bulk download from the server to an LG Urbane Watch paired with a Nexus 5X over one representative measurement. For CBR traffic whose data rate is much lower than the BT bandwidth, we still observe fluctuating OWD over time, with the standard deviation being 324ms (99ms) for 1Mbps

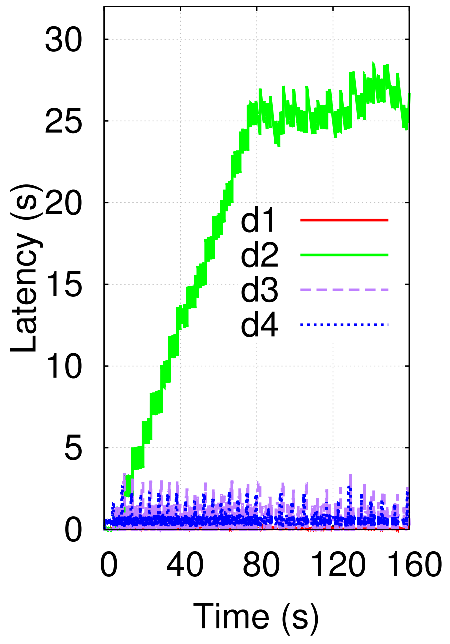
(500kbps). When the CBR rate becomes higher, *e.g.,* at 1.5Mbps, the OWD inflates to an unacceptably high level, with the median delay being 20.1s (up to 28.6s). For bulk download, its median OWD further increases to 29.0s. We also observe high delays on other combinations of watches and phones we have. Recall from §6.1 that many wearable apps incur high-bitrate real-time traffic, such as real-time camera streaming, HD VoIP, and real-time screen projection [10]. The high OWD will incur unacceptable QoE for such apps.

3.3.2 Identifying the Root Cause

We now seek to understand the root cause of the high OWD under the *CPROXY* mode. The multiple buffers scattered in the end-to-end data transmission pipeline present a challenge towards our analysis. We thus dissect the end-to-end (E2E) delay by instrumenting at multiple entities and layers. Specifically, we use our toolkit (Table 3.2, §3.2) to collect BT and TCP/IP traces at several locations, and then perform offline analysis to obtain for each byte various timestamps as illustrated in Figure 3.3. (1) t_S: from the `tcpdump` trace captured on the server when the data is being transmitted out; (2) t_{IR} from the `tcpdump` trace captured on the smartphone when the data is received in the smartphone OS kernel; (3) t_A: from the kernel log captured on the smartphone when the data is copied to the proxy app's userspace (by instrumenting `tcp_input.c`); (4) t_{BS}: from the Android log captured on the smartphone when the proxy app sends the data to the BT stack (by instrumenting `BluetoothSocket.java`); (5) t_{BR}: from the `tcpdump` trace on the wearable when the data is delivered to wearable OS. The end-to-end latency can thus be broken down into four parts: the transmission delay from server to phone ($d_1 = t_{IR} - t_S$), the buffering



((a)) E2E delay breakdown methodology.



((b)) Breakdown of E2E delay for CBR traffic at 1.5Mbps.

Figure 3.3: E2E delay breakdown of CBR traffic in *CPROXY* mode (LG Urbane with Nexus 5X, normal BT RSSI).

time in the TCP/IP stack on the phone ($d_2 = t_A - t_{IR}$), the buffering time in the proxy app buffer on the phone ($d_3 = t_{BS} - t_A$), and the delay of BT transfer from the phone to the wearable ($d_4 = t_{BR} - t_{BS}$). Note that d_2 is dominated by the delay incurred by the TCP receive buffer on the smartphone. The IP queueing delay at the qdisc is confirmed to be very small. Also, we separate d_2 and d_3, both residing on the smartphone, due to the difference between their associated buffers: the TCP buffer incurring d_2 is maintained at a per-connection basis, whereas the proxy app buffer incurring d_3 is shared by all wearable app streams, and is therefore more likely to cause potential cross-traffic interference.

Measurement Results. Figure 3.3(b) shows the OWD breakdown for CBR traffic at

Table 3.3: Impact of TCP receive buffer size on the severity of *CPROXY* bufferbloat on different phones.

	Nexus 5X	SGS5	Nexus 5
`tcp_rmem_max`	8,291,456	4,525,824	2,097,152
`rmem_max`	8,388,608	2,097,152	2,097,152
d_2: TCP/IP recv (s)	26.1 $\sim$ 28.6	4.0 $\sim$ 5.5	4.1 $\sim$ 5.7
Total E2E OWD (s)	27.9 $\sim$ 30.1	5.7 $\sim$ 6.7	5.9 $\sim$ 7.0

1.5Mbps for an LG Urbane watch paired with a Nexus 5X, over a representative experiment. We observe that the buffering delay in the TCP/IP stack (d_2) accounts for almost the entire OWD. Recall that d_2 is dominated by the delay incurred by the TCP receive buffer (recvBuf). We thus explicitly confirm how the recvBuf size affects the OWD on three smartphones in Table 3.3. The *effective* recvBuf size is determined by the minimum value of two configurable OS parameters `rmem_max` and `tcp_rmem_max` (both are in bytes). As shown, a phone with a smaller recvBuf indeed experiences a smaller d_2 as well as a lower overall E2E OWD. However, setting the recvBuf to be too small will throttle the TCP congestion window and hence the throughput – a tradeoff that is difficult to balance.

While the bufferbloat problem has been well studied in different contexts such as broadband wired network [141], cellular download [81], and cellular upload [71], we highlight two differences that make bufferbloat in the *CPROXY* mode a unique problem. First, due to the highly asymmetric bandwidth of the BT/BLE link and the WiFi/cellular link, the *CPROXY*-side bufferbloat will always occur when the WiFi/cellular link throughput becomes higher than $\sim$1.1Mbps. The above breakdown analysis indicates that the TCP recvBuf configuration does not take into account such bandwidth asymmetry. Second, the lower-layer BT state machine also affects the severity of the bufferbloat. In particular, its Sniff Mode slows down the BT data transmission and thus causes the proxy-side buffer to

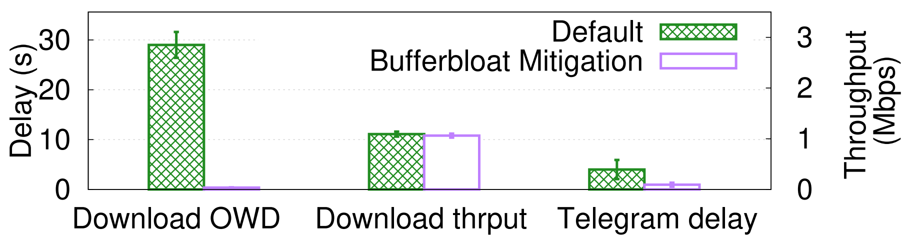


Figure 3.4: *CPROXY* bufferbloat mitigation for bulk download and messaging with competing traffic (LG Urbane, Nexus 5X).

further build up. This is confirmed in Figure 3.3(b) where d_4 exhibits periodical spikes, whose occurrences well match those of entering the Sniff Mode.

3.3.3 Mitigating the *CPROXY* Bufferbloat

We now consider how to mitigate the *CPROXY* bufferbloat. In the literature, numerous bufferbloat mitigation solutions have been proposed, but we found it is difficult to directly apply them in our context due to various practical or fundamental issues. For example, blindly reducing the TCP recvBuf may throttle the congestion window and thus the throughput [71]; delay-based TCP congestion control [49, 101] is not aware of the BT protocol stack between the phone and the wearable; various Active Queue Management (AQM) techniques [110, 121] only regulate the qdisc buffering, and may need substantial modifications for tackling the *CPROXY*-side bufferbloat.

Developing a full-fledged bufferbloat mitigation solution for wearable networking with heterogeneous links is beyond the scope of this study. Here, we propose a simple, practical, yet effective solution to demonstrate the need for coordinating the heterogeneous links as well as the substantial performance improvement. Note that other (better) solutions may exist.

In our scheme, the phone maintains a virtual queue (shared by all apps) whose size increases as bytes arrive from the remote server and decreases upon the reception of BT ACKs. Based on the virtual queue size, our scheme dynamically throttles the connection between the phone and the server (if needed) to bound the actual buffering delay. Specifically, we maintain two thresholds, an upper bound Q_{UB} and a lower bound Q_{LB}. The throttling is enabled when the buffer level exceeds Q_{UB}, and is disabled when the buffer level drops below Q_{LB}. Q_{UB} is set to $BW \times (1 - \varepsilon)T$ where BW is the current estimation of the BT link bandwidth, T is the upper bound of the tolerable queueing delay (configurable based on the app's QoE requirement), and ε controls the aggressiveness of our scheme. Q_{LB} is set to $BW \times (1 - 2\varepsilon)T$ so that both thresholds are proportional to the BT link bandwidth. We empirically use T=1s, ε=0.3, and set the throttling rate to $\frac{BW}{2}$. Note that BW may vary over time.

Evaluation. We implement the above scheme using our toolkit (§3.2) for performance monitoring and Linux `tc` for bandwidth throttling. We then conduct controlled experiments to evaluate its effectiveness. We consider two workloads: TCP bulk download and receiving short messages delivered by the Telegram messaging app [12] when there is an on-going concurrent transfer. The latter scenario may happen when, for example, a user receives a message when a media player is performing audio or video streaming in the background. We repeat both experiments 10 times under a normal network condition (-60 dBm BT RSSI) on an LG Urbane smartwatch paired with a Nexus 5X phone. Figure 3.4 measures the OWD and throughput for the bulk download, as well as the per-message delivery time for Telegram messaging. As shown, for bulk download, our scheme substantially reduces the packet OWD by 78 times with less than 3% of throughput reduction. Our scheme also reduces the Telegram message delivery delay by 76%.

3.4 Performance & Energy Impact of Network Selection

Today's wearables are usually equipped with multiple network interfaces. For example, most smartwatches have WiFi and BT/BLE, and advanced editions even have the cellular interface [87]. Typically, the Wear OS employs a *static* interface selection policy: all 7 smartwatches except Huawei Watch 2 use BT (through the *CPROXY*) when both BT and WiFi networks are available. At first glance, this simple policy is energy-wise beneficial as BT is known to be more power-efficient than WiFi. Interestingly, Huawei Watch 2, which uses a custom Wear OS, actually prefers WiFi over BT, leading to potentially high energy consumption. In this section, we quantitatively analyze how the network selection policy affects the important tradeoff between performance and energy consumption, using real-world workload on COTS smartwatches.

3.4.1 Impact of Single-path Interface Selection

We first study the *single-path* interface selection, *i.e.,* using only one interface at any given time. We consider four real-world workloads: (a) downloading a wearable app of 16MB from the Google Play Store, (b) streaming a 2-min YouTube video to a watch, (c) delivering a short message by Telegram, and (d) streaming from an IP-camera in real time for 150s using the TinyCam app [14]. For these diverse workloads, we employ the app download time, the video throughput, the message delivery delay, and the real-time data streaming rate as the QoE metrics, respectively. We calculate the energy consumption

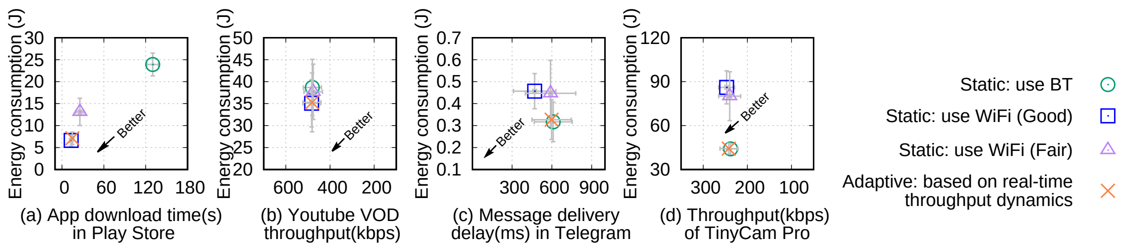


Figure 3.5: QoE-energy tradeoffs across four real workloads using different interface selection policies (LG Urbane Watch, normal BT RSSI, Good/Fair WiFi network condition).

using the energy model developed by [103]. Regarding the network selection policy, we consider the following four options: (1) always using BT, assuming a good network condition (-50 dBm RSSI), (2) always using WiFi, assuming a good network condition (10Mbps BW, 10ms RTT), (3) always using WiFi, assuming a fair network condition (5Mbps BW, 20ms RTT), and (4) an approach that dynamically switches between BT and WiFi as to be detailed in §3.4.2.

For each combination of the workload and network selection policy, we repeat the experiment 10 times. We show the results in Figure 3.5 to illustrate the tradeoff between QoE and energy consumption. Each plot in Figure 3.5 corresponds to a workload; each plot has four clusters corresponding to the four interface selection policies described above. Ideally, we prefer a cluster to be located in the bottom-left corner with a good QoE (the X Axis) while incurring a low energy overhead (the Y Axis). Our key observation from Figure 3.5 is that, depending on the app workload, the preferred interface selection policy differs. For (a) and (b), given their large data sizes, WiFi offers both lower energy consumption and a better QoE due to its higher throughput and higher energy efficiency (*i.e.*, joule per byte) compared to BT. In contrast, for (c), WiFi only marginally reduces the message delivery latency while incurring considerably higher energy consumption compared to BT. This is because the small message size and WiFi's high base power consumption lead to a higher joule per byte compared to BT. For (d), the workload consists of CBR traffic that BT can already sustain. This makes BT more energy-efficient than WiFi, which has a higher base power consumption and bandwidth under-utilization.

The energy results in Figure 3.5 only consider the energy consumed on the wearable. In addition, using BT incurs additional energy footprint on the paired smartphone that acts as a proxy forwarding traffic between the wearable and the server. The smartphone needs to utilize both its WiFi interface (with the server) and BT interface (with the wearable). To quantify such an energy overhead, we focus on the "static: use BT" scenario in Figure 3.5, and apply the smartphone WiFi [53] and BT [68] power models to calculate the overall smartphone-side radio energy consumption to be 121.984 J, 125.76 J, 0.524 J, and 157.2 J, respectively, for the four workloads. This non-trivial energy overhead on the smartphone makes it more complex to make interface selection decisions for the wearable.

The above results indicate that the static interface selection policy, which strictly prefers one interface over another as employed by almost all of today's smartwatches, does not always provide a preferred tradeoff between performance and energy consumption. The results suggest the need for a more adaptive interface selection policy. In §3.4.2, we will describe such an example corresponding to the "Adaptive" cluster in Figure 3.5.

3.4.2 Multipath Performance on Wearables

Multipath transport, which simultaneously uses multiple network interfaces, is becoming popular on smartphones, as fueled by standardized solutions such as MPTCP [8]. Despite a lack of prior work, we do believe that multipath transport can also benefit wearable networks in two aspects: (1) enhancing the throughput by aggregating bandwidth, and (2) facilitating seamless handover or fast interface switch. We examine the first aspect now and address the second one later.

We consider a common usage scenario involving a WiFi path and a BT path. In the wearable context, we do *not* expect multipath to be always used due to energy constraints. Instead, a wearable can *adaptively* enable multipath (*e.g.,* enhancing BT using WiFi) to meet user-specified deadlines or to prevent stalls for multimedia streaming [75]. Note that maintaining active WiFi connectivity incurs negligible power consumption due to WiFi's deep power-saving mode [88, 45, 126].

A Multipath Framework for Wearables. The Wear OS by default does not support multipath transport. Also, it is difficult to directly use MPTCP because BT does not speak TCP/IP by default. We thus make a methodological contribution of adding the multipath transport feature (over WiFi and BT) to the Wear OS. Specifically, we first leverage *ConnectivityManager* in the Wear OS to keep WiFi active when BT is also on. We then use the Linux socket API and Bluetooth API to build a custom multipath framework. In our framework, each path is a standalone TCP connection. The WiFi path is established directly between the wearable and the server[1], and the other path is wearable–*CPROXY*–server where the wearable–*CPROXY* segment is over BT. On the sender side, the original data stream is split into data chunks that are distributed onto the paths. We add to each chunk a custom header containing the metadata such as the size and global sequence number of the chunk. The receiver side then uses the metadata to reassemble the received chunks into

[1]In our experiments, to make our multipath framework fully transparent to the original server, we actually run the server-side code of our framework on an in-cloud proxy. The proxy–server path is verified not to be the performance bottleneck.

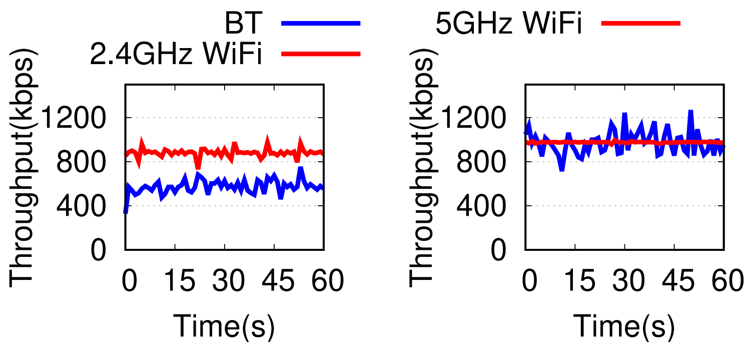


Figure 3.6: BT-WiFi multipath under 2.4/5 GHz WiFi (Nexus 5, normal BT/WiFi RSSI).

the original data stream. To provide application transparency, we use `netfilter` [13] to transparently intercept application TCP connections on the wearable side. We also implement three off-the-shelf scheduler algorithms that determine how to distribute the traffic onto the paths: MPTCP's default minRTT scheduler [119], a round-robin scheduler, and a redundant scheduler. The first two schedulers help improve the throughput by aggregating the bandwidth of all paths; the third scheduler helps reduce the latency by sending duplicate data to all paths. Our system consists of around 10K lines of Java and C/C++ code. It is also open-sourced on GitHub [24].

Energy Overhead. We measure our multipath framework's energy overhead using a Monsoon power monitor [19]. Compared to the base power level of an LG Urbane Watch with the screen being turned off, our framework incurs only 0.6% of additional device-level power consumption. Some use cases such as fast interface switch further require our framework to keep the WiFi interface turned on and maintain a long-lived TCP subflow. We find that doing so incurs a device-level energy overhead of 6.2% based on an 8-hour measurement, using a 4-minute keep-alive timer as suggested by the RFC [48].

Performance Aggregation Results on Wearables. Leveraging our wearable multipath framework, we conduct experiments on an LG Urbane paired with a Nexus 5X to assess the multipath performance over WiFi and BT. Other watch and phone pairs yield qualitatively similar performance. We focus on two types of improvements brought by multipath: the latency reduction when the redundant scheduler is used, and the bandwidth aggregation when the minRTT scheduler is used. For the latency reduction, we observe positive results. For example, using the redundant scheduler helps reduce the average RTT by 29% for CBR traffic at 500kbps (WiFi: 10Mbps BW, 10ms RTT; BT: -50 dBm RSSI). However, we find that the bandwidth aggregation results are much worse than our expectation. Ideally, for long-lived data transfers, the aggregated throughput achieved by multipath should be the sum of all paths' data rate. In reality, we observe that the bandwidth gain from multipath is far less than that. For example, when we throttle the WiFi and BT path's bandwidth to both 1Mbps, the multipath bandwidth gain compared to a single path is only 7%.

We realize the reason for the above unexpected results are multifold and cross-layer. For example, at the transport layer, we need a better scheduler that takes into account the heterogeneity between WiFi and BT. Very importantly, we also discover another key reason rooted deeply at the PHY layer: all our smartwatches support only 2.4 GHz WiFi that operates at the same frequency band of BT. The WiFi and BT thus cause interference when simultaneously transmitting data. This is confirmed by the following experiment: a Nexus 5 smartphone performs bulk data transfers over BT and WiFi at the same time (the phone supports both 2.4 GHz and 5 GHz WiFi), with the WiFi throughput being capped at 1Mbps. The left (right) plot in Figure 6.4 shows the BT and 2.4 GHz (5 GHz) WiFi throughput measured on the phone. As shown, compared to 5 GHz WiFi, when 2.4 GHz

WiFi is used, the BT and WiFi throughput drops by 47% and 7%, respectively. Overall, our findings suggest the need for introducing 5 GHz WiFi on COTS wearables for reducing the WiFi-BT interference, in order to facilitate multipath transport over BT and WiFi.

Fast Interface Switch on Wearables. Recall from the beginning of this subsection that another important use case of multipath transport is to support fast interface switch, which seamlessly and transparently migrates a TCP connection from one path to another path without requiring re-establishing the connection. We utilize this feature to develop an adaptive interface selection policy corresponding to the "adaptive" cluster in Figure 3.5. Specifically, assuming an on-going download (the upload case is similar), our scheme uses BT over the *CPROXY* mode by default. Meanwhile, it monitors the number of bytes buffered at the *CPROXY* by tracking the incoming and outgoing bytes' to/from the *CPROXY*. When the buffer occupancy level exceeds B bytes for T_1 seconds (we empirically choose B=10KB and T_1=500ms), we switch to WiFi as BT does not drain the buffer fast enough. The switch from WiFi back to BT is triggered by a low WiFi throughput (we use $\leq$500kbps) for T_2 seconds (we use T_2=5 seconds).

We implement this adaptive interface selection strategy using our developed wearable multipath framework. The experimental results in Figure 3.5 suggest that it outperforms the static policies over all four workloads. In addition, we will apply multipath to improve the BT-WiFi handover performance in §3.5.

3.5 BT-WiFi Handover Performance

In previous sections, we consider the scenario where both the wearable and its paired phone are stationary. In reality, either device can be mobile. Consider a typical mobility scenario where a user wearing a smartwatch walks away from her paired smartphone placed on a table. As the user walks away, the wearable will lose its BT connectivity. In this case, ideally the wearable needs a seamless handover from BT to WiFi, an important feature that is missing on today's wearables as we will reveal in this section.

3.5.1 Wearable Handovers are Common

Although the theoretical BT range can be up to 100m [21], in real-world scenarios the range is much shorter due to attenuations incurred by obstructions or vendors' intentional reduction of the radio power for saving energy. For example, on the Samsung Galaxy Gear, the effective BT range is less than 2 meters based on our measurement. Due to such a short range, network handovers are likely to occur very frequently when a wearable moves away from the paired smartphone. To understand how often handovers occur "in the wild", we conduct an IRB-approved user study involving 10 voluntary users each wearing an LG Urbane Watch. The 10 participants consist of 4 students, 3 faculty members, and 3 staff members in a large U.S. university. 5 of them are female. We develop a data collector that infers handover events by monitoring the network interfaces' states (the method will be described in §3.5.2). The user study lasted for two months in 2017. During the daytime (9 AM – 9 PM) across all users, the median handover frequency is once every 1.6 hours. For some users, handovers can happen as frequently as every 7 minutes. The results suggest the need for properly handling handovers to provide smooth network switches.

3.5.2 Poor Wearable Handover Performance

Motivated by the user study, we quantify the handover performance on state-of-the-art wearables through controlled experiments.

Monitoring the Network State. A prerequisite for measuring handovers is to monitor the network state change. We capture the state of each network interface from the Wear OS's *ConnectivityManager* in the background. The state information includes whether the network interface is up, *i.e., available* or not, and whether the interface provides actual network connectivity, *i.e., connected* or not. For example, when a smartwatch is associating with the WiFi AP, its WiFi is available but not yet connected.

Experimental Setup. Our experiment focuses on understanding handovers from BT to WiFi (handovers from WiFi back to BT can be studied using similar methods). We keep both BT and WiFi enabled on the wearable (so both interfaces are available) and let the Wear OS use the default network management policy. We use two wearable apps to generate the traffic workload. The first is a simple app developed by us (conveniently called RTApp). It represents a typical wearable app developer's *best-effort* user-space implementation of the handover logic, which requires the synergy between both the app and the OS. Our app emulates the same traffic pattern as the tinyCam app (to be detailed soon), *i.e.,* downloading a data chunk of 3KB every 160ms from our server, generating 150kbps downlink traffic over TCP. When a handover occurs, the old interface (BT) will lose its connectivity and shortly after that, the connectivity will appear on the new interface (WiFi). At this time (detected through polling), our RTApp will establish a TCP connection over the new interface and resume the data transfer.

The second app we test for handover is the tinyCam security camera app [14]. It

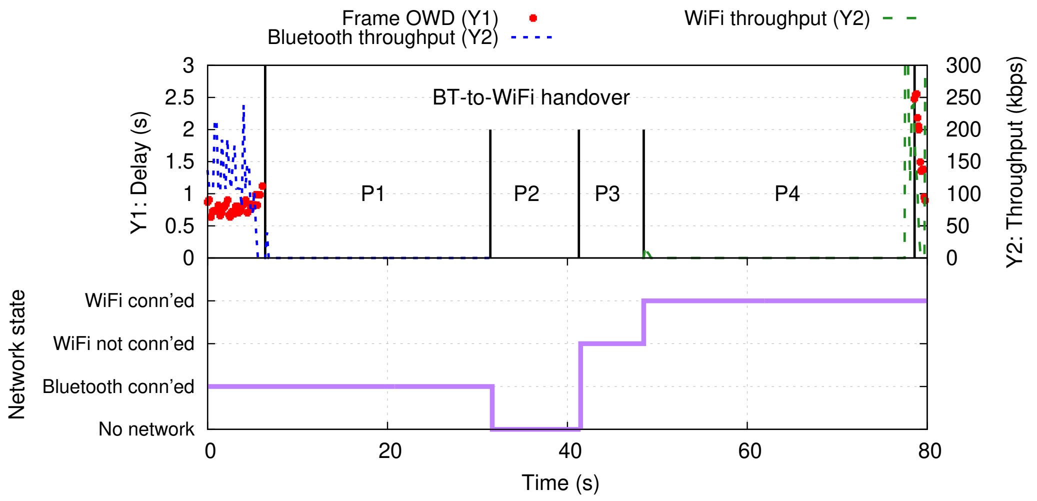


Figure 3.7: Impact of BT-WiFi handover on QoE of tinyCam app (Huawei Watch, normal BT/WiFi RSSI).

is a popular, professionally designed commercial app that requires continuous network connectivity to stream real-time video captured from an IP camera to a wearable. We perform a black-box test for this app to reveal its handover performance. We define two QoE metrics for the tinyCam app: (1) the *frame one-way delay (OWD)*, which is the time to transmit a video frame from the security camera to the watch (including the encoding and rendering time)[2], and (2) the *downlink throughput* on the watch.

Measurement Results. The top plot in Figure 3.7 shows the QoE metrics of the tinyCam app during a typical BT-WiFi handover: the frame OWD, the BT throughput, and the WiFi throughput. As shown, the app QoE severely degrades during the handover. At around $t = 6$s, the app stops receiving the video data from the SmartCam and the BT throughput drops to zero. The video transmission resumes over WiFi at around 72.5s,

[2]To measure the frame OWD, we use a phone to display continuously increasing timestamps from a stopwatch app as the input stream to the SmartCam. The tinyCam app then shows the captured timestamp on the watch. The frame OWD can thus be calculated by comparing the timestamps when the same stopwatch frame appears on the phone and watch, whose timestamps are synchronized beforehand.

with high frame OWD observed at the beginning. We repeat this experiment for 10 times and the average "blackout" period during which the app does not receive any video data is surprisingly 70.0s. We then run the experiment under the same setting for our RTApp, whose average handover delay is measured to be 38.6s across 10 runs (we will explain the difference shortly). We further conduct the experiment on three different smartwatches and observe high handover delays on all of them as shown in Table 3.4. The results show that handovers are poorly handled by the Wear OS and/or the wearable app.

3.5.3 Root Cause of the High Handover Delay

To understand the root cause of the high handover delay, we break it down into four phases based on the captured network state information, as shown in the bottom plot in Figure 3.7: (P1) BT is still connected but the data cannot be actually transmitted due to poor signal strength, (P2) no network is available, (P3) the WiFi AP association period, *i.e.,* WiFi is available but not connected, and (P4) WiFi is connected but there is no application data transmission. The methodology for the breakdown analysis is as follows. For each network, our measurement tool (Table 3.2, §3.2) logs whether the network is ready to use by applications, *i.e.,* available or not, and whether the interface provides actual network connectivity, *i.e.,* connected or not, through Wear OS APIs. We then group both networks' logged states as shown in Figure 3.7.

Our analysis reveals two sources of delay that contribute to the overall handover latency: the delay from the Wear OS (P1, P2, and P3), as well as the delay incurred by the wearable app (P4). We next detail both types.

Delay from the Wear OS. Under the default network management policy of Wear OS based wearables, when BT is connected, WiFi is not available (*i.e.,* its interface is turned off by the OS) even if the device is under the coverage of both BT and WiFi. In this case, when the wearable moves away from the BT coverage, the Wear OS needs to: wait until the BT connectivity is completely lost as its RSSI drops below a threshold (P1), turn on the WiFi interface (P2), and then perform an AP association (P3). The whole process incurs a long period of time. Across the 10 runs on an LG Urbane watch, the average duration of P1, P2 and P3 are 12.9s, 15.5s and 8.3s, respectively, with their total duration accounting for 52% of the overall handover delay.

Delay Incurred by the Wearable App. We next investigate the wearable app's behavior during a handover event. In the tinyCam app case, even after WiFi gains its connectivity (after P3), the app still takes around 33.3s on average before the actual data transfer resumes over WiFi (P4). In contrast, our RTApp only takes 5.6s on average to resume the data transfer. Such a disparity of the P4 duration causes the two apps' vastly different handover duration shown in Table 3.4. In other words, although the handover completes after P3 from the OS's perspective, it takes additional time for the app to actually resume the data transfer (P4). The variation of P4 is very likely attributed to the app logic. Unfortunately, since Wear OS does not provide an API for seamlessly migrating data transfers between IP-based and non-IP networks, wearable apps need to implement their own data migration logic at the app layer. Doing so is tedious and challenging for average app developers.

Table 3.4: BT-to-WiFi handover delay on 3 smartwatches for tinyCam app and RTApp (normal BT/WiFi RSSI).

	LG Urbane	LG Urbane 2nd	HUAWEI Watch
tinyCam	43.1 $\pm$ 5.7 s	52.9 $\pm$ 8.2 s	70.2 $\pm$ 9.7 s
RTApp	28.3 $\pm$ 2.6 s	14.3 $\pm$ 1.3 s	38.6 $\pm$ 5.3 s

3.5.4 Reducing the Handover Delay

We now design and implement a solution that reduces the handover delay. Our basic idea consists of the following. First, an important reason for Wear OS's bad handover performance is its *reactive* nature, *i.e.,* the WiFi connectivity is not established until the BT connectivity is fully torn down. Our scheme instead predicts a BT-WiFi handover by monitoring the BT channel quality. When the quality drops below a threshold (but BT is still usable), we *proactively* establish the WiFi connectivity and perform a handover to WiFi (assuming the WiFi channel quality is acceptable). Second, we leverage the multipath framework introduced in §3.4.2 to provide application transparency. Before a BT-WiFi handover, once the WiFi connectivity is established, the OS adds a new WiFi subflow to the corresponding TCP connection, and schedules future data to the WiFi subflow. No modification is needed at the user application, which always sees the same TCP connection. Third, our scheme further leverages *reinjection* to facilitate seamless data migration. Specifically, when it decides to perform a BT-WiFi handover, it also sends all unacknowledged (*i.e.,* "in-flight") data on the BT path, which may experience long delays due to its weak channel quality, to the WiFi path. In multipath transport, this is called packet reinjection, which trades a small number of redundant bytes for better performance (smoother handover in our case).

We implement the above design points and integrate them into our wearable multipath framework (§3.4.2). We use the BT RSSI as the BT channel quality metric [143], and empirically set its threshold for initiating a handover to -66dBm. A future research direction here is to further leverage the wearable's motion sensors or acoustic ranging [122] to precisely track the wearable's relative position to the phone in order to facilitate more accurate handover prediction. Using a similar approach, we also implement the handover mechanism from WiFi back to BT. We take two approaches to prevent oscillations between BT and WiFi. First, we use the Kalman filter to smooth the RSSI samples [51]. Second, after a BT-WiFi handover, we require the wearable to stay on WiFi for at least 5 seconds (a configurable parameter) unless the WiFi connectivity is lost.

Evaluation. We evaluate how our scheme helps accelerate the BT-WiFi handover process. The experimental setup is as follows. A user puts her Nexus 5X smartphone in a typical conference room (5m by 6m) and moves out of the room at a normal walking speed with a paired LG Urbane smartwatch worn on her wrist. As she walks out of the smartphone's BT coverage, the watch will experience a BT-WiFi handover. We use the two apps introduced in §3.5.2 as the workload: the tinyCam app that streams video contents from our camera in real-time, and our RTApp program that performs CBR streaming with improved application handover logic. The handover delay is measured using the same approach for generating Figure 3.7. We compare three handover schemes in Figure 3.8: "default" is the reactive handover approach used by Wear OS; "on-demand WiFi" corresponds to our proposed scheme where the WiFi connectivity and the multipath subflow are established in an on-demand fashion based on BT channel quality prediction; "always-on WiFi" also refers to our scheme, but we always maintain the WiFi connectivity and pre-establish the WiFi subflow to further speed up the handover. We repeat each experiment 10

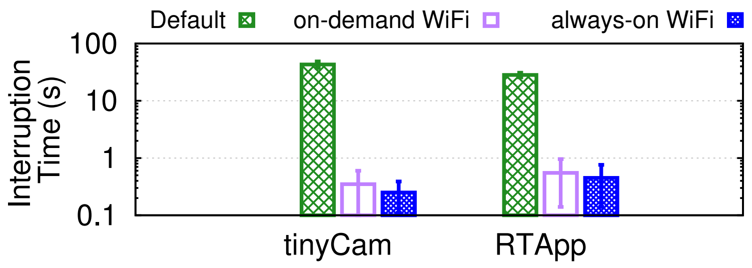


Figure 3.8: Reducing the BT-to-WiFi handover delay for tinyCam/RTApp on an LG Urbane paired with Nexus 5X.

times to overcome the randomness incurred by the user's walking paths. As shown in Figure 3.8, our scheme is extremely effective: it reduces the handover delay from more than 28s to less than 0.6s (123x and 51x reduction for tinyCam and RTApp, respectively). Note the Y axis is in log scale. For the "always-on WiFi" variation, the improvement is even higher (172x and 63x respectively) since the pre-established WiFi subflow allows immediate data transfers, but the cost is a slightly higher radio energy consumption (measured to be 6.2% as described in §3.4.2) compared to the "on-demand WiFi" variation.

3.6 Summary

This chapter focuses on characterizing and improving the transport management of wearable networks. Wearable end systems' recent debut made the mobile Internet more diverse, bringing new challenges and opportunities for cross-device network transport as a wearable often relies on a paired mobile phone for Internet access. Experiments on existing and modified wearable network stacks show that the default Wear OS has network performance issues, and adding cross-device awareness to network transport management can improve wearable network performance and application QoE.

CHAPTER IV

MPBond: Efficient Network-level Collaboration among Personal Mobile Devices

This chapter explores how an existing multipath transport protocol, MPTCP, can be extended to support multiple mobile devices by having cross-device awareness and carefully handling the interactions among heterogeneous wireless links facing different mobile devices. Specifically, we develop MPBond, a distributed multipath transport system that can be used for efficient network-level collaboration among personal mobile devices such as smartphones and smartwatches.

4.1 Introduction

It is increasingly common that a user possesses multiple mobile devices. For example, despite being a full-fledged computer, a smartwatch naturally needs to pair with a smartphone; business people oftentimes carry two phones, one for work and the other for personal tasks [9, 2]; tablets bear large screens and reasonable portability, making them good companions of smartphones.

From the networking perspective, smart mobile devices are equipped with diverse network interfaces such as cellular, WiFi, and Bluetooth (BT), making them capable of communicating with remote Internet servers as well as other local devices. We make a key observation that despite such a mature wireless *hardware* support, the potential of the devices' network interfaces that can operate collaboratively is far from being fully exploited. In this chapter, we bridge this critical gap by bringing networking *software* innovations to the smart mobile device ecosystem. Specifically, we develop MPBond, a holistic system allowing multiple personal mobile devices to collaboratively fetch content from the Internet. MPBond enables a wide range of use cases that today's mobile/wearable OSes do not support or provide optimal performance for:

• A smartwatch can assist its paired smartphone with downloading data over cellular (many COTS smartwatches today have direct cellular access). This leads to a much higher throughput compared to using a single device.

• WiFi networks offered by public places such as hotels often impose per-interface rate limit. Such a limit can be naturally overcome by multi-device collaboration since each participating device has its own WiFi interface.

• Two smartphones can share each other's LTE bandwidth. In other words, their cellular interfaces are "combined" by MPBond and can be used by apps as a single virtual interface.

• Wearables can be placed at a spot with good signal and act as WiFi/LTE "range extenders". When running low on battery, a smartphone can offload power-hungry LTE access to a smartwatch paired over an energy-efficient BT link.

By closely examining the above use cases, we notice that all of them can be realized under the *multipath transport* scheme, where user data can be distributed over multiple subflows (paths).

Unlike traditional multipath paradigms such as MPTCP [130], MPBond needs to support *distributed* multipath where subflows traverse different devices. Specifically, MPBond involves one *primary* device, where the client app runs, and multiple *helper* devices, which boost the primary's network performance. Without loss of generality, for the first use case above, the traffic from the primary is intercepted by the MPBond service, which distributes part of the traffic to the helpers over local wireless links (called *pipes*), and transmits the remaining over the primary's cellular interface. The helpers then forward the traffic to the remote server through their own cellular interfaces. The reverse (downlink) direction works in a similar way: the server or an MPBond-capable proxy distributes the content to the primary and helpers. The primary merges all the received parts and delivers the content to the client app.

The above scheme provided by MPBond appears to be intuitive. However, we face numerous challenges when designing and implementing the system. How to properly manage heterogeneous devices and local wireless links? How to strategically leverage the helper devices to improve the network performance? How to design a robust multipath scheduler that considers both remote paths and local pipes, with the latter being unique in MPBond? How to expose appropriate interfaces to users and applications? How to make the whole MPBond system transparent to client and server applications? We next highlight our key design aspects.

• As a distributed multipath transport framework, MPBond allows a subflow to traverse a helper, and enables helpers to exchange data with the primary device over pipes. We develop a scheme to flexibly manage the pipes using different wireless technologies such as WiFi and BT. To support distributed multipath and pipes, we extend MPTCP's control plane protocol to coordinate the primary and helpers (§4.3.1).

• MPBond splits any subflow into two TCP (sub)flows, one between the primary and the helper, and the other between the helper and the server. TCP splitting benefits end-to-end TCP sessions that span heterogeneous networks as the case of MPBond (the Internet and the pipes). More importantly, doing so allows buffers to be set up between the split flows, which effectively mitigate the negative performance impact incurred by the fluctuating network condition on either network. Although TCP splitting is not new [156, 79], we take this concept a step further by applying it to helper devices in the context of mobile multipath transport (§4.3.2).

• We develop a Pipe-Aware Multipath Scheduler (PAMS) that strategically distributes traffic onto multiple subflows. Tailored to MPBond, PAMS consists of three key components: (1) a subflow latency estimation module that considers pipes, helper-side buffering, and heterogeneous networks; (2) an algorithm that makes judicious scheduling decisions to ensure low delivery latency for each packet; and (3) a smart reinjection scheme that deals with fluctuating network conditions and possible failures over pipes (§4.3.3).

• MPBond allows users to flexibly specify various policies such as granting per-app usage permission, limiting per-device resource consumption, and prioritizing traffic (§4.3.4).

We implement MPBond on commodity mobile devices including Android smartphones and smartwatches. We showcase that most of MPBond logic can be implemented in the user space while maintaining full application transparency and good performance

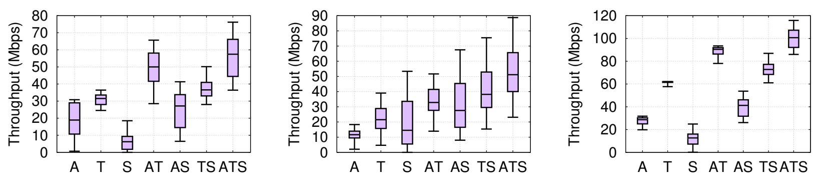
(§6.4). We then systematically evaluate MPBond over real mobile networks. Our key evaluation results consist of the following (§6.5).

• Compared to kibbutz [111] and COMBINE [46], two state-of-the-art systems, MPBond reduces the energy consumption by 10%-57% under a wide range of network conditions with various workloads (file download, video streaming).

• Under varying and in-the-wild network conditions, MPBond reduces the file download time by 13%-35% compared to kibbutz and COMBINE. The reduced download time also translates to lower energy consumption.

• We show the need of three collaborative mobile devices to deliver good QoE for bandwidth-hungry 360-degree video streaming. We also demonstrate the effectiveness of MPBond's dual mode.

Overall, MPBond is an efficient and practical system that innovates network-level collaboration among personal mobile devices through applying the concept of distributed multipath. Compared to other cross-device data sharing schemes [111, 46, 134], MPBond offers several advantages including better performance as boosted by the PAMS scheduler, application transparency, and more flexibility (§4.2.4). Also none of the above studies has considered or experimented using wearable devices. Our contributions made in this work consist of novel use cases, the MPBond design/implementation, and comprehensive evaluation in real-world settings. Note that MPBond is open-source on GitHub [29].



((a)) Office ((b)) Residence ((c)) Grocery Store

Figure 4.1: Throughput distributions of different devices (carriers) and their combinations at 3 locations. (A: LG Urbane Watch 2 with AT&T; T: Pixel 2 smartphone with T-Mobile; S: Samsung Galaxy S9 phone with Sprint)

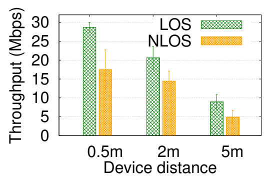


Figure 4.2: WLAN throughput from LG Urbane Watch 2 to Pixel 2 smartphone under different settings.

4.2 Motivation

4.2.1 Incentives to Carry Multiple Devices

It is increasingly prevalent that users possess more than one mobile device [16]. People oftentimes carry two smartphones due to various reasons. For example, one phone is used for work and the other is used for personal tasks – such a physical separation minimizes the likelihood of business data being leaked or compromised [9, 2]. Another important reason for carrying two phones with different carriers is that the carriers have complementary coverage [93] so that the user can switch between the devices to enjoy better performance. This is in particular popular in countries such as India where prepaid plans are prevalent [11]. People may also carry their old phones as portable WiFi hotspots [22], or have a second phone with a roaming-friendly sim card [6]. Other reasons for having two phones include mitigating battery anxiety, preventing theft, providing extra storage, and backing-up sensitive data locally [4]. Note that people do not explicitly buy two new phones; instead they typically use their old phones as a second device – around 46% of

Americans upgrade their smartphone every two years or less [5].

Compared to carrying two phones, an even more popular trend is to wear a smartwatch while carrying a smartphone. In particular, many smartwatches such as Apple Watch Series 4 and Samsung Gear Frontier have built-in sim cards, allowing them to access cellular data networks as a typical smartphone does. In addition, there are many other common combinations of dual or triple mobile devices with Internet access capabilities, such as smartphone+tablet and smartphone+laptop+smartwatch. Despite their prevalence, the potential of the devices' network interfaces that are concurrently operational is far from being fully exploited.

4.2.2 Benefits of Multi-device Collaboration

A *network-level* collaboration among multiple devices can significantly improve the network performance. The basic idea is straightforward: when the last-mile wireless hop is the bottleneck (which is typically the case), having multiple devices download data simultaneously can effectively improve the overall WWAN-side (wireless wide-area network, *i.e.,* the Internet) throughput. Meanwhile, the content received by individual devices is merged over WLAN (wireless local-area network) and delivered to the application (§4.2.4).

We now experimentally demonstrate the benefit of WWAN-side throughput aggregation, by measuring the performance of concurrent multi-device data transfers, in particular when wearable devices are involved – few prior studies have investigated that to our knowledge. We consider three COTS mobile devices with each using a different cellular carrier, as detailed in Figure 4.1. We place them side-by-side (0.2 meters apart), and let them perform concurrent bulk download from a nearby server over their own LTE networks for 1 minute. On each device, we sample TCP throughput every 200ms. The experiment was conducted in three locations: a university office, a residential apartment, and a grocery store. Figure 4.1 plots the throughput distributions of different devices and their combinations. As we do not consider the WLAN-side merging step in this measurement, the overall throughput achieved by multiple devices is the sum of that for each individual device (*e.g.,* “AS” corresponds to the total throughput of A and S).

Figure 4.1 indicates that the three carriers exhibit different performance at the three locations, with median throughput ranging from 6.2Mbps to 61.8Mbps. Assuming WLAN-side merging is not the bottleneck, leveraging two interfaces improves the overall throughput by 15.8% to 474.2%, and simultaneously using three devices boosts the throughput by 63.1% to 695.4%, compared to using only a single device (interface). The aggregated throughput can effectively support bandwidth-hungry applications such as UHD video streaming, mobile VR [91], and mobile holographic communication [124]. We also notice that there is no device whose network performance constantly outperforms the other two in all three locations. Therefore, one can also dynamically choose the best device based on its live network condition in order to satisfy the app’s minimum QoE requirement – this is supported by MPBond.

4.2.3 Networking Capability of Wearables

The experiment in Figure 4.1 involves an LTE-capable smartwatch. Despite its decent throughput, it may still raise concerns about its networking capability compared to full-fledged smartphones and tablets. We therefore experimentally compare the LTE through-

Table 4.1: Advantages of MPBond compared to existing systems designed for multi-device network-level collaboration.

Collaboration Scheme	System Layer	WLAN Layer	Application Transparency	Scheduling Consideration	Server-side Deployability	Mobile-side Deployability
MPBond	L4	**L4**	**Yes**	**WWAN + WLAN**	**standardized proxy**	**mostly userspace, $>$ 2 mobile**
kibbutz [111]		L3	Yes	bottleneck	standardized proxy	kernel, up to two mobile
PRISM [85]		L3	Yes	bottleneck	server + new proxy	kernel, not available for mobile
COMBINE [46]	L5	L5	No	bottleneck	HTTP byte-range	userspace, $>$ 2 mobile
MicroCast [83]		L5	No, video only	bottleneck	HTTP byte-range	userspace, $>$ 2 mobile
Cool-Tether [134]		L3	No, web only	bottleneck	new proxy + byte-range	userspace, $>$ 2 mobile

put of two devices: an LG Urbane 2 smartwatch and a Nexus 6P smartphone, both released in the same year. To ensure fair comparisons, we repeatedly insert the same sim card (AT&T) into the two devices, and conduct 10 back-to-back TCP bulk download experiments on them over LTE at the same location, to understand the impact of the device capability on performance. The watch indeed achieves a statistically lower throughput compared to the phone: the smartwatch's median throughput (29.9Mbps) is only 53.3% of the phone's median throughput (56.1Mbps), likely due to the watch's small form factor that limits the antenna's size and tx/rx power. However, the absolute throughput values (median: 29.9Mbps, 90-th percentile: 31.4Mbps) indicate that commodity smartwatches' network interfaces can still contribute considerably to collaborative content delivery in particular when other devices' WWAN-side performance is poor. For example, Figure 4.1(a) and (c) show that the smartwatch (A) yields a higher throughput than the Sprint phone (S). Another potential concern regarding wearables is energy, which we will assess in §4.5.8.

4.2.4 Do Existing Network-level Collaboration Schemes Suffice?

We summarize existing network-level collaboration schemes in Table 4.1. They suffer from several limitations as described below.

A Lack of Flexibility. A desired network-level collaboration must be flexible to support different types of applications and require minimal changes to the mobile and network infrastructure at the same time. From the application's perspective, a tethered device in kibbutz [111] performs as a simple layer-3 router, making it difficult to flexibly support various enhancement and policies at layer-4/5 (§4.3.4). In addition, the tethering subsystem is usually tightly coupled with the OS/kernel, and is therefore difficult to be modified or extended. In Android, tethering has many practical limitations. For example, (1) tethering to more than one device is not supported, therefore, kibbutz supports at most two devices; (2) only one tethering connection (either WiFi or Bluetooth, but not both) can be established, hindering smooth handovers; (3) many carriers and devices lock the tethering feature or only provide limited data plan for tethering based hotspot. PRISM [85] relies on modified kernel TCP stack for both the sender and receiver and a custom PRISM proxy in the network, which incur significant deployment overhead. And its WLAN architecture relies on WiFi Ad-hoc mode, which is not available for Android and iOS smartphones and smartwatches. Other schemes including COMBINE [46], MicroCast [83], and CoolTether [134] require modification to the apps at layer-5 and the server must support HTTP byte-range requests for the network collaboration. Some of them are designed solely for a particular type of application traffic.

Suboptimal Performance due to Fluctuating Network Conditions. In kibbutz [111], the tethering approach, an end-to-end path consists of two segments: WLAN and WWAN. Figure 4.1 shows that the WWAN-side (LTE) throughput is indeed fluctuating. We next experimentally study the WLAN-side performance. Figure 4.2 shows the WLAN throughput when fetching data from an LG Urbane Watch 2 to a Pixel 2 phone with their physical distance varying under line-of-sight (LoS) and non-line-of-sight (NLoS) settings[1]. WLAN throughput varies significantly and is oftentimes *lower* than the WWAN throughput depicted in Figure 4.1. In other words, due to their heterogeneous link characteristics and complex interactions with the environment, WWAN and WLAN may exhibit highly different performance and *either* can become the bottleneck, in a very dynamic manner. The default tethering mechanism, however, often poorly deals with such heterogeneity and dynamics due to its simple layer-2/3 forwarding. For example, in tethering, the effective data rate is always the minimum of the WWAN and WLAN bandwidth; this can be improved by properly buffering data at the device that decouples the WWAN and WLAN. We will revisit this problem when describing MPBond's solution (§4.3.2). Besides, when making the scheduling decision, existing schemes only consider the performance of the bottleneck link between the WWAN and WLAN at a specific time, causing suboptimal workload distribution and hence the multipath performance.

Excessive Energy Consumption is one of the consequences of the suboptimal network performance. This is in particular an issue for wearable devices with small battery capacities. Even if the network condition is stable, in the tethering approach the congestion control is end-to-end, so the WWAN (WLAN) would be throttled to the WLAN (WWAN) bandwidth for data download (upload) when the WLAN (WWAN) is the bottleneck, causing prolonged WWAN (WLAN) radio-on time leading to increased energy consumption. Solutions that work at the application layer introduce idle network period between consecutive HTTP byte-range requests, lowering the energy efficiency.

[1]The distance between two devices can be large, *e.g.*,, when one device is charging or with another family member.

4.3 MPBond Design

MPBond enables a user to jointly leverage her mobile devices to access the Internet in an application-transparent manner. As shown in Figure 4.3, MPBond involves two types of devices: one *primary* device and one or more *helper* devices (referred to as "primary" and "helper(s)" for brevity). We discuss the scenario of multiple primaries in §4.3.4.1[2]. The client application, such as a file downloader or a video player, only runs on the primary. TCP traffic from the app is transparently intercepted by the MPBond service and scheduled to transmit either over primary's own interface or through helpers with forwarding, i.e., different *subflows* shown in Figure 4.3. The reverse direction works in a similar way by distributing the traffic over multiple subflows from the MPBond-capable remote server and merging the content on the primary. To be fully transparent to Internet servers, the system can introduce an MPBond-capable proxy which hides MPBond from remote servers by establishing single-path connections with them. In the remainder of this chapter, unless otherwise noted, the term "server" refers to either an MPBond-capable remote server or an MPBond proxy. Also, deployed as an OS service on the primary and helpers, MPBond is transparent to client-side apps as well.

We next describe how we address key design challenges of MPBond: How to properly manage subflows (§4.3.1)? How to overcome the limitations of the state of the art as described in §4.2.4 (§4.3.2)? How to intelligently distribute the traffic onto multiple paths while accounting for the heterogeneity between pipes and HS-Paths (§4.3.3)? How to properly interface MPBond with upper layers while considering various user-specified

[2]In this work, we assume the primary and all helpers are mutually trusted – the same assumption that other collaboration schemes make. Standard security primitives such as encryption and authentication can be applied to pipes to prevent attacks such as session hijacking and eavesdropping.

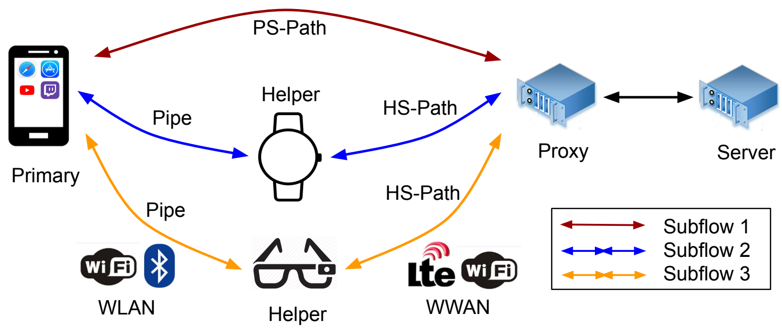


Figure 4.3: System Architecture of MPBond.

policies (§4.3.4)?

4.3.1 Subflow Management

The high-level concept of MPBond subflows is similar to that of MPTCP, except that (1) the subflows traverse different mobile devices, and (2) the primary and helpers need to perform local data exchanges to merge the received parts. We call the data channels between the primary and helpers *pipes*. We also denote the network paths between helpers and the server *HS-Paths* (Helper-server Paths), and the network path between the primary and the server (without a helper) the *PS-Path* (Primary-server Path). An end-to-end subflow therefore traverses through either a PS-Path, or an HS-Path and a pipe.

MPBond supports multiple concurrent pipes using different radio technologies such as WiFi and Bluetooth. The pipe is established by connecting the helper through WLAN to the primary which acts as a WiFi AP, or pairing the helper to the primary through Bluetooth. The scheduler dynamically selects a pipe by considering factors including performance, reliability, and energy efficiency, or simultaneously using multiple pipes to increase the data rate. We will revisit this feature in §4.3.4. Similar to MPTCP's subflows, the pipes can be flexibly torn down or established, and they are loosely coupled with a user TCP connection, allowing seamless migration among pipes without interrupting the data transfer.

The overall handshake procedure in MPBond to establish a user TCP connection with subflows between the primary and the server leveraging helpers follows that in MPTCP, with additional control messages over pipes to coordinate with the helpers. Specifically, for the subflow involving an HS-Path and a pipe, the primary sends an INIT_MP_JOIN (INIT_MP) message with the necessary client and server information to the helper, allowing it to establish the second (first) subflow through an MP_JOIN (MP_CAPABLE) message. When the subflow is established, an MP_JOIN_OK or MP_OK message is returned to the primary as an acknowledgement.

Error Handling. MPBond should be robust to a wide range of errors. Compared to MPTCP, MPBond needs to further deal with pipes' failures. For example, on a subflow traversing through a helper, a failure of either its HS-path or its pipe will cause the subflow to be torn down and all its pending (unacknowledged) data to be reinjected to another subflow (§4.3.3.4). This ensures that no application data is lost due to either a WWAN or WLAN link failure.

4.3.2 Buffer Management and Helper-side Connection Split

MPBond maintains buffers at both end points (the primary and the server) to absorb network fluctuations and to accommodate the subflows' heterogeneous characteristics. Besides having these buffers, we make an important design decision of setting up buffers on helpers. Specifically, MPBond splits any subflow into two TCP (sub)flows, one between the primary and the helper, and the other between the helper and the server. The two flows thus cover the pipe and the HS-Path, respectively. Although TCP splitting is not a new idea [156, 79], we take this concept a step further by strategically applying it to helper devices, in particular wearable devices, in the context of mobile multipath transport.

Recall from §4.2.4 that when a helper is involved, the WWAN and WLAN exhibit vastly different link characteristics. TCP splitting can effectively improve the performance in such a scenario by shortening the TCP control loop [123]. More importantly, it allows buffers to be set up between the two flows. Such buffers effectively mitigate the negative performance impact caused by the bottleneck shift on a subflow. To illustrate this, consider a simple example where the pipe bandwidth increases due to the helper device being moved closer to the primary (Figure 4.2), causing the pipe's throughput (Th_{pipe}) becomes higher than that of the HS-Path (Th_{HS}). If there is a buffer at the helper, the buffered data can be transmitted at Th_{pipe} (instead of at Th_{HS} when there is no buffer), leading to a shorter data transfer time.

4.3.3 Pipe-aware Multipath Scheduler

As a critical component of a multipath transport system, a scheduler determines how to distribute the traffic onto multiple paths. There are several studies that improve the scheduler design in wireless settings [93, 112, 72, 97]. However, directly applying them to MPBond is difficult. First, most existing mobile multipath schedulers only deal with two paths (WiFi and cellular), and many solutions such as [72] are inherently difficult to scale to more than two paths, which MPBond needs to handle. Second, none of prior studies considers the pipes, which are unique in MPBond.

We design a Pipe-aware Multipath Scheduler (PAMS) for MPBond. It differs from existing mobile multipath schedulers in two aspects: PAMS is capable of scheduling an arbitrary number of subflows, and it takes MPBond's TCP splitting and helper-side buffering (§4.3.2) into consideration when performing scheduling.

PAMS can be used by a wide range of applications such as file transfer, video-on-demand (VoD), web browsing, and cloud synchronization. All these applications involve transferring *data chunks* such as a file, a video chunk, an image, and a web page, which need to be delivered as fast as possible. Besides such chunked transfers, another type of traffic pattern is *real-time data streaming* such as live video streaming and low-latency gaming. Generally speaking, the benefits of multipath transport on these applications require switching the scheduling algorithm to a latency-favoring one such as [69] and [93], and blindly applying multipath to them may incur QoE penalty [112]. Designing a full-fledged scheduler tailored to the MPBond architecture for such latency-sensitive traffic is beyond the scope of this study. Nevertheless, we provide easy-to-use interfaces (§4.3.4) for users to specify policies such as letting delay-sensitive traffic use single-path and giving it higher priority than other traffic so as to prevent potential latency inflation.

4.3.3.1 MinRTT Considered Harmful.

We first demonstrate the performance issue of directly applying the default minRTT scheduler. The primary establishes two subflows to a nearby server, one directly and the other through a helper. The downlink bandwidth of the PS-Path, the HS-Path, and the pipe are configured to be 8Mbps, 10Mbps, and 5Mbps, respectively. The primary downloads from the server a 10MB file using MPBond configured with minRTT. Figure 4.4 shows

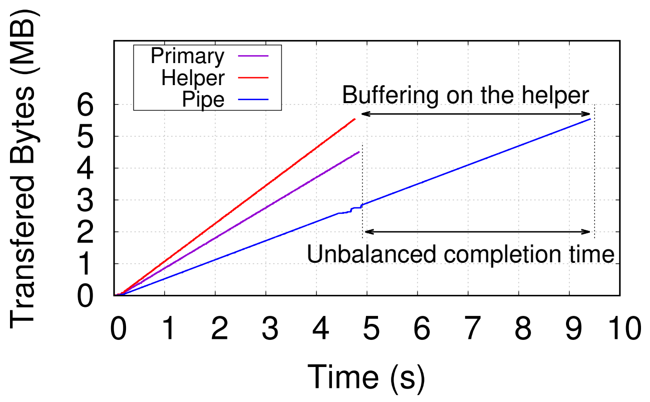


Figure 4.4: Performance of MPBond configured with the minRTT scheduler.

the download progress. Recall that MPBond maintains a buffer at the helper. As shown, on the positive side, due to TCP splitting and helper-side buffering, the bandwidth of all three paths is fully utilized. On the negative side, under the default minRTT scheduler, the two subflows cannot complete at the same time: the helper subflow finishes about 4.5 seconds later than the direct subflow. Note that in multipath transport, *simultaneous subflow completion* is a necessary condition for achieving the optimal download time [72]. This is because in the case where one subflow finishes earlier than the other subflow, the fast subflow can always "assist" the slow one, leading to an even reduced data transfer time.

The unbalanced subflow completion in Figure 4.4 is attributed to the fact that the scheduler, which runs at the server, only monitors the PS-Path and the HS-Path, and is unaware of TCP splitting mechanism and the downstream pipe. In other words, minRTT only tries to balance the completion time of the PS/HS-Path instead of the two end-to-end subflows. In this particular experiment, since the pipe bandwidth is lower than the HS-Path bandwidth, downlink data will be buffered at the helper and drained slowly over the pipe, leading to highly unbalanced subflow completion time.

A possible way of achieving simultaneous subflow completion is to modify the subflow availability condition: a helper subflow is considered to be available when the congestion window (CWND) of *both* the HS-Path and the pipe have available space (minRTT only considers the former). We implement this modification and find that it indeed almost achieves simultaneous subflow completion. However, by requiring an available CWND space for the pipe, this approach loses the capability of buffering at the helper, a key feature that MPBond should provide (§4.3.2). Therefore, the key challenge that PAMS should address is to *enable buffering at the helper while achieving simultaneous subflow completion.*

4.3.3.2 Deriving the Pipe-aware Delay (PAD)

We now describe the PAMS algorithm. We focus on the scheduler residing on the server for downlink traffic. We first derive the end-to-end (E2E) packet delay: at a given time T, if a packet is scheduled over a given subflow, how long does it take for the packet to arrive at the receiver (the primary)? Let B_s and B_p be the number of bytes buffered at the server and the helper, respectively, at T (they include both the TCP send buffer and the userspace buffer maintained by MPBond); let Th_{ps}, Th_{hs}, and Th_p be the measured downlink throughput of the PS-Path, the HS-Path, and the pipe, respectively; let OWD_{ps}, OWD_{hs}, and OWD_p be the one-way delay of the corresponding path. Given the above notions, the E2E delay for a direct subflow is $OWD_{ps} + \frac{B_s}{Th_{ps}}$, including both the propagation delay and the queuing/transmission delay. A subflow with a helper involves two buffers. It takes $T_1 = \frac{B_s}{Th_{hs}}$ to drain the server-side buffer. After T_1, the helper-side buffer level changes from B_p to $B'_p = \max\{0, B_p - Th_pT_1 + B_s\}$, which needs $T_2 = \frac{B'_p}{Th_p}$ to deplete. Therefore the overall E2E delay is $T_1 + OWD_{hs} + T_2 + OWD_p$. Plugging T_1 and T_2 into the above, we can derive the Pipe-Aware Delay (PAD) as:

$$\begin{cases} OWD_{ps} + \frac{B_s}{Th_{ps}}, & \text{if } i = 1 \\ OWD_{hs} + \frac{B_s + B_p}{Th_p} + OWD_p, & \text{if } i > 1, \frac{B_p}{B_s} + 1 > \frac{Th_p}{Th_{hs}} \\ OWD_{hs} + \frac{B_s}{Th_{hs}} + OWD_p, & \text{if } i > 1, \frac{B_p}{B_s} + 1 \leq \frac{Th_p}{Th_{hs}} \end{cases}$$

where i is the index of the subflow (i=1 corresponds to the direct subflow). Th_{ps}, Th_{hs}, and Th_p can be estimated as an exponential weighted moving average of the ratio between CWND and RTT of the corresponding path. In practice, as directly measuring OWD is difficult, we approximate it using $\frac{RTT}{2}$. The second and third case in the above formula deals with $B'_p > 0$ and $B'_p = 0$, the two conditions considered by the max function when calculating B'_p.

4.3.3.3 The PAMS Algorithm

PAD gives us an estimation of the E2E packet delay of a subflow. Now we consider how to use it to make scheduling decisions. A possible approach is to modify minRTT into "minPAD", *i.e.,* select the subflow with the minimum PAD as long as the subflow is idle (*i.e.,* the PS-Path or HS-Path has empty CWND space). Although this approach outperforms minRTT, it still tries to occupy all the HS-Path CWND space, thus may schedule more data than the subflow's actual capacity. We next show that it can be further improved through strategically *deferring* the scheduling to make more judicious scheduling

Algorithm 1 The Pipe-Aware Multipath Scheduler (PAMS).

```
Input: $S$ = A set of $N$ subflows. The algorithm executes when at least one subflow is idle, i.e., its PS-Path
       or HS-Path has empty space in CWND.
Output: Packet to transmit on a subflow $[packet, subflow]$.
1  $packet \leftarrow GetNextPacket()$
2  $Th[1..N] \leftarrow GetSubflowThroughput()$
3  $PAD[1..N] \leftarrow GetPipeAwareDelay()$
4  $Idle \leftarrow GetIdleSubflows()$
5  $Busy \leftarrow GetNonIdleSubflows()$
6  $target \leftarrow GetIdleSubflowWithMinPAD()$
7  $Diff \leftarrow 0$
8  for each subflow i in Busy do
9      if PAD[i] < PAD[target] then
10         $Diff += (PAD[target] - PAD[i]) \times Th[i]$
11 if Diff > GetUntransmittedSize() then
12     return NULL
13 else
14     return [packet, target]
```

decisions.

Let us consider two cases that require different scheduling strategies. First, when the server has a *large* amount of remaining data in the meta buffer[3] to send, it is important to improve the overall bandwidth utilization (*i.e.,* throughput) by keeping all the subflows busy. In this case, PAMS applies minPAD: as long as there are any idle subflows, the one with the minimum PAD will be immediately selected and made busy. The second case is that when there is only a *small* amount of remaining data, ensuring low-latency delivery and simultaneous subflow completion time is more important than maximizing the throughput. In this case, even when there is an idle subflow, PAMS may skip it (*i.e.,* deferring the scheduling) when there are non-idle subflows that can shorten the delivery latency.

[3]In multipath transport, the (sender-side) meta buffer stores data passed from the application but is not yet scheduled. The meta buffer is different from the per-subflow buffer (B_s and B_p), which contains data that has already been scheduled to a subflow.

Following the above idea, we develop the PAMS algorithm listed in Algorithm 1. As shown, *Idle* and *Busy* are the set of idle and non-idle subflows, respectively, and *target* is the idle subflow with the minimum PAD. Line 8 to 14 is the core part of the algorithm. It determines if it is possible to deliver all to-be-scheduled bytes in the meta buffer (or an application-defined data chunk, see §4.3.4) over currently busy subflows before the *target* subflow completes. For a given busy subflow i, its current buffered data will be drained in $PAD[i]$ time units, so the time budget allowing it to deliver additional unscheduled data before the *target* subflow completes is $PAD[target] - PAD[i]$. The throughput of the subflow i, $Th[i]$, is the minimum of the HS-Path throughput and pipe throughput when $i > 1$. The total number of unscheduled bytes that can be delivered by all busy subflows before the completion of the *target* subflow is therefore calculated as $Diff$ (Line 10). If such bytes are more than the total number of unscheduled bytes, we defer scheduling the current packet, allowing it to be later scheduled over a currently busy subflow (Line 12). Doing so will shorten its delivery time and facilitate simultaneous subflow completion. Otherwise, we immediately schedule the packet over the *target* subflow to ensure high bandwidth utilization (Line 14). Note that Algorithm 1 is for the downlink traffic, and the scheduler for uplink traffic is developed in a conceptually similar manner.

4.3.3.4 Data Reinjection

In multipath transport, reinjection is a mechanism where data that has already been scheduled over one subflow (A) is "reinjected" into another subflow B. This may occur when, for example, A experiences unexpected performance drop or failure, or B's capacity suddenly increases. MPTCP employs a conservative and fixed reinjection policy where packets are reinjected only when their associated subflow is terminated or the receiver buffer is full. In MPBond, the involvement of multiple devices, heterogeneous networks, and helper-side buffering makes the network performance potentially more dynamic and fluctuating, thus necessitating more judicious reinjection decisions.

We next describe MPBond's reinjection scheme by detailing when, who, and how to perform reinjection. Specifically, a reinjection is triggered when there are no unscheduled bytes and $\frac{\text{maxPAD}-\text{minPAD}}{\text{minPAD}} > \eta$, where minPAD and maxPAD are the minimum and maximum PAD values across all subflows, respectively, and η is a parameter. The rationale is as follows. Ideally, all subflows' PAD should be similar, as PAMS implicitly controls the buffer levels (B_p and B_s) of the subflows to facilitate simultaneous subflow completion (§4.3.3). When some subflows' PAD becomes too large or too small, it implies severe fluctuations of their network performance. It is therefore the right time to launch a reinjection for promptly rebalancing the subflows. Regarding η, it determines the aggressiveness of the reinjection: reducing η incurs more frequent reinjections at the cost of increased bandwidth utilization. We empirically set η to 20%.

When a reinjection is triggered, MPBond moves up to r unacknowledged bytes with the highest sequence numbers from the subflow with maxPAD back to the meta buffer[4]. We calculate r as $(\text{maxPAD} - \text{minPAD}) \times B$ where B is the effective throughput of the subflow with maxPAD. Intuitively, r is determined in such a way that a slow subflow can catch up with the fastest subflow in terms of PAD. These r "recalled" bytes are scheduled again by PAMS.

[4]Data residing in the user-space buffer can be directly moved; data that stays in the kernel-level buffers will become redundant.

4.3.4 User/App Interfaces and Policy Engine

MPBond provides 2 types of interfaces to users and app developers, respectively. First, it has a built-in console on the primary. This allows users to pair/unpair with helpers, manage the pipes, grant apps permission to use MPBond, monitor the devices' runtime status, and configure various policies (see below). In addition, MPBond exposes APIs allowing 3rd-party apps to programmatically use its service. The APIs include device/pipe management, status query of devices/pipes, callback functions of important events such as a change of the pipe configuration, and marking the boundaries of application data chunks[5]. Note that using such APIs is optional: MPBond is fully transparent to apps; the APIs just provide more fine-grained manipulation and detailed monitoring of MPBond. For example, based on its data rate, an app can dynamically switch between pipes with different bandwidth (*e.g.,* WiFi vs. Bluetooth), to reduce the energy footprint while meeting the QoE requirement.

MPBond allows users to flexibly specify various policies. Our current prototype supports the following policies. (1) *per-app whitelist.* MPBond takes a "whitelist" approach: users need to explicitly grant permissions to apps and specify a (super)set of devices/pipes that the app can access through MPBond. Typically this is a one-time effort, provides good flexibility, and boosts security. (2) *Resource Usage.* MPBond allows disabling a device (either helper or primary) when its battery level drops below a threshold or its monthly cellular data usage reaches a pre-defined cap. (3) *Prioritization.* Users can configure rules

[5]By default, the *GetUntransmittedSize()* function in Algorithm 1 returns the total size of the to-be-sent data in the meta buffer. Developers can optionally define application-layer data chunks to make *GetUntransmittedSize()* return the remaining bytes of the current data chunk. This will expedite the delivery of each data chunk as opposed to all data in the meta buffer as a whole.

that prioritize certain applications.

4.3.4.1 Dual Mode in MPBond

MPBond allows a device to have dual roles of both a primary and helper, and multiple primary devices may co-exist. We call this the *dual mode*, which enables the collaborating devices to better utilize their collective bandwidth in particular when the primary devices generate traffic at different time. Consider the following use case. Two close friends are watching different DASH videos, but each one's individual device may not provide sufficient bandwidth for its own video. To overcome this limitation, the two devices can be paired up using the dual mode: each device acts as the primary by fetching its own video, and meanwhile also as the helper by delivering the content for the other device. Since the two devices usually do not fetch video chunks simultaneously, the video chunks requested by each device can be downloaded over the two subflows with small probabilities of competing for the bandwidth with video chunks requested by the other device. This leads to improved QoE for both users.

In our current prototype, we realize the dual mode by running multiple independent instances of MPBond, as either a primary or a helper, on the same device. A limitation of this approach is that each MPBond instance independently makes scheduling decisions, which may be suboptimal due to a lack of global view of the network condition and traffic patterns. This issue can be addressed by introducing a lightweight "global manager" that coordinates all MPBond instances [123, 47]. We leave this as future work.

4.4 Implementation

We implement MPBond on commodity Android smartphones and Wear OS smartwatches. To support real-world evaluations with commercial Internet servers that may not support MPTCP and middleboxes that may block it, we implement a multipath TCP proxy in C/C++, following the methodology in [72]. Our implementation of MPBond consists of 8K lines of C/C++ and Java code excluding the base proxy system and is accessible on GitHub [29].

On the primary, most of the logic lies in a userspace MPBond service. It establishes the PS-Path (pipe) connections with the MPBond proxy (helpers). To support unmodified applications, we built a lightweight kernel module using *netfilter* hooks that intercept and redirect client application traffic to the MPBond service. WiFi pipes are implemented as long-lived TCP connections between the primary and helpers. We also implement Bluetooth pipes by leveraging the Android BluetoothSocket APIs to establish RFCOMM connections. The MPBond helper module is implemented in the userspace for the ease of deployment. It establishes HS-Path (pipe) connections with the proxy (primary). For each subflow, we use a circular queue to buffer packets in the userspace. These buffers work with the in-kernel send/recv buffers of the HS-Path and pipes together to achieve the performance and energy benefits.

A pipe is a long-lived data channel over which multiple user TCP connections are multiplexed. To do this, we add a tiny header before the application payload containing the TCP connection ID, message length, and sequence number to identify individual TCP connections. PAMS is implemented as a userspace scheduling module plugged into the MPBond proxy. The PS-Path and HS-Path information is obtained at the server by leveraging Linux *getsockopt* API. Pipe throughput is measured on the primary and sent to the helper through an encapsulated control message. We implement a flexible interface for a helper or the primary to determine when and which pipe's information to send to the proxy. In §4.5.2 we demonstrate how this flexibility can be helpful instead of fixing the feedback mechanism. Currently we use an out-of-band UDP channel to carry pipe-specific information over the return path of HS-Path/PS-Path for the sake of prototyping. In the future we will replace it with TCP options that can be integrated to the HS-Path/PS-Path ACKs. User-defined policies are enforced at a per-process basis. The MPBond services on the primary looks up the process name of a given flow by following the methodology in [125].

4.5 Evaluation

We extensively evaluate MPBond under various network and device settings using synthetic and real apps to show the benefit of network-level collaboration. We examine the effectiveness of key design choices of MPBond through micro-benchmarks. We quantitatively compare MPBond with kibbutz [111] and COMBINE [46], the two major state-of-the-art solutions in Table 4.1, on network performance, energy consumption and app QoE using commodity smartphones and smartwatches over real LTE and WiFi networks.

4.5.1 Experimental Setup and Methodology

Our proxy supporting both MPBond and our implementation of kibbutz [111], which employs tethering-based MPTCP, runs on a commodity Ubuntu 16.04 server with 4-core 3.6GHz CPU and 16GB memory. The proxy uses the decoupled CUBIC as the congestion control algorithm (*i.e.,* each path runs TCP CUBIC independently). The server hosting files and video contents is in close proximity to the proxy, and the path between them has very high network bandwidth, not being the bottleneck of the end-to-end paths. For COMBINE [46], as no proxy is required, multiple mobile devices send HTTP byte-range requests directly to the server to fetch the chunks of different ranges in the same object. The requests are scheduled by a work-queue algorithm that sequentially downloads chunks on each path and returns them to the primary device. For COMBINE, we use a default chunk size of 256KB. For small file download (*e.g.,* 512KB) we also try two smaller chunk sizes (128KB and 64KB) and report the best performance. By default, MPBond uses PAMS as the multipath scheduler.

Our mobile devices include a Pixel 2 phone, a Nexus 6P phone, and an LG Urbane 2 smartwatch. We perform evaluation of MPBond using both emulated and real network conditions. To emulate certain network conditions, we use Linux `tc` to throttle the bandwidth on real WWAN and WLAN, while capturing the latency dynamics from commercial wireless networks. We also conduct experiments using real LTE networks at different places. To understand the impact on battery, we use full-fledged energy models [54, 103] to estimate the energy consumption incurred by network transfers.

4.5.2 Microbenchmarks

We start with examining the key design choices of MPBond. We focus on a two-device setting where a Pixel 2 with T-Mobile acts as the primary and a Nexus 6P with AT&T acts as the helper.

Benefit of Helper-side Connection Split. One key design aspect of MPBond is to

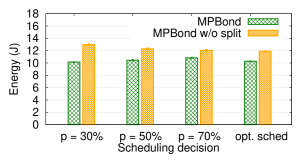


Figure 4.5: Energy benefits of split under stable network conditions.

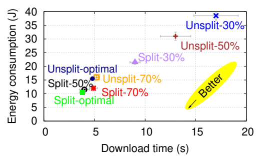


Figure 4.6: Performance and energy benefits of split under changing network conditions.

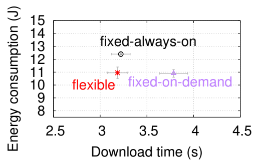


Figure 4.7: Performance and energy consumption for different feedback mechanisms.

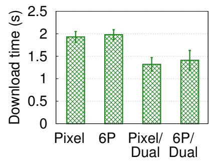


Figure 4.8: Dual Mode reduces download time.

decouple the HS-Path and the pipe with a buffer on the helper (§4.3.2), to absorb network fluctuations and accommodate heterogeneous subflow characteristics. To understand its impact on energy and performance, we compare MPBond with kibbutz, which does not incorporate this design, using various fixed scheduling ratio on subflows – transmitting $p\%$ of 4MB file over the PS-Path and $1 - p\%$ over the HS-Path and the pipe. We also derive the optimal ratio offline from an exhaustive searching of p.

We start with stable network condition where the bandwidth of both PS-Path and pipe are 5Mbps, and the bandwidth of HS-Path is 10Mbps. Figure 4.5 shows that MPBond reduces energy consumption by 10%-22% compared to kibbutz using fixed scheduling ratio, while achieving almost the same download time. This is because helper-side connection split allows the transmission on HS-Path to finish much earlier than that on the pipe, reducing the LTE radio-on time.

We then study the performance and energy impact under a changing network condition. We start from the stable profile described above, and after 2s, drop the bandwidth of HS-Path to 1Mbps. Figure 4.6 shows the download time and energy consumption of downloading a 4MB file. Both the download time and energy consumption are reduced when helper-side connection split is in effect. The improvement is much higher when more data is scheduled to the HS-Path and the pipe at the time of sudden bandwidth drop. This indeed confirms that the buffer between the split flows absorbs the fluctuation of network condition.

Benefit of Flexible Feedback. MPBond allows pipe information to be shared over the multiple HS-Paths and/or the PS-Path (§6.4). A simple yet effective policy of sharing such information for the 2-device case is to send the pipe's information over both the HS-Path and the PS-Path, when there's data transfer on the corresponding path. We call this policy "flexible" and compare it with sending the pipe feedback over the HS-Path only, with different timings: (1) when there's data transfer on either HS-Path or pipe ("fixed-always"), and (2) when there is data transfer on HS-Path only ("fixed-on-demand"). We run an experiment of downloading a 4MB file. Initially, PS-Path=5Mbps, pipe=5Mbps, and HS-Path=10Mbps. After 2s, pipe bandwidth increases to 10Mbps. As Figure 4.7 shows, "fixed-always" inflates the energy consumption since it keeps the helper's radio active by sending information feedback even if there is no data transfer on the HS-Path. "Fixed-on-demand" mitigates the issue by sending the feedback only when there is data transfer on the HS-Path. However, it still incurs performance degradation as the pipe information is not up to date. Instead, the "flexible" policy keeps sending feedback over the *PS*-Path when there is no transfer on the *HS*-Path, keeping the pipe information updated without waking up the helper's radio, thus improving both performance and energy consumption.

Estimating Pipe Buffering. PAMS estimates the pipe buffering time of a packet based on the buffered data on the helper and the pipe throughput ($\frac{B_p}{Th_p}$) (§4.3.3.2), instead of directly measuring the packet buffering delay incurred by the helper, *i.e.,* the time between when a packet arrives at the helper and when it comes out, which may not be up-to-date.

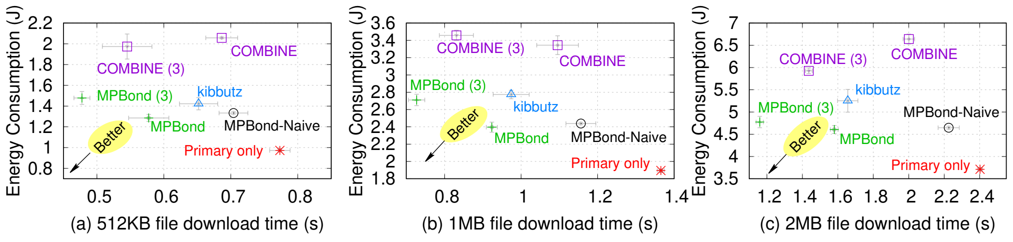


Figure 4.9: Bulk download performance under stable network condition (PS-Path: 8Mbps, HS-Path: 10Mbps, pipe: 5Mbps): Single device (Pixel2), MPBond/COMBINE w/ 2 devices (Pixel2+Nexus6P), MPBond/COMBINE w/ 3 devices (Pixel2+Nexus6P+LG2), and kibbutz (Pixel2+Nexus6P).

To demonstrate the advantage of such approach, we conduct file downloads of 4MB under a stable network condition: the PS-Path and pipe bandwidth are 5Mbps, and the HS-Path bandwidth is 10Mbps. With the estimation based on the buffered data on the helper and the pipe throughput, the file downloads take 3.6s on average, compared to 4.4s on average using direct buffering delay measurements. The suboptimal performance of the latter approach is due to the fact that the scheduler always receives the stale buffering delay measurements which are inaccurate, thus making the scheduling decisions suboptimal.

Reinjection Under Changing Network Condition. Reinjection in MPBond helps to reduce the download time under changing network conditions (§4.3.3.4). To examine the effectiveness, we download a 4MB file under a changing network condition. At the beginning of transfer, the PS-Path and pipe bandwidth are 5Mbps, the HS-Path bandwidth is 10Mbps. After 2s, pipe bandwidth drops to 1Mbps. When reinjection is enabled, the download time is 4.8s, 49% of the download time without reinjection (9.7s). The improvement is attributed to the data on the slower pipe being reinjected to the PS-Path so that the pipe transmission can catch up.

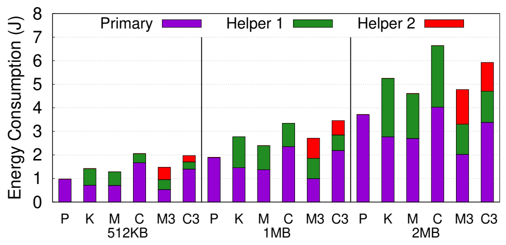
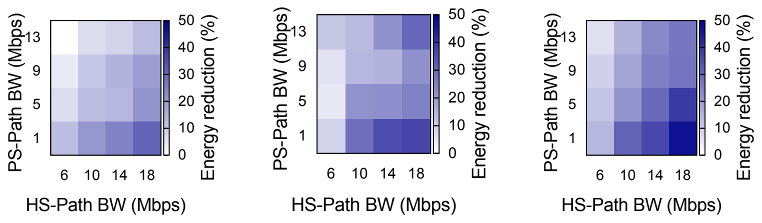


Figure 4.10: Energy breakdown of different schemes under stable network condition (PS-Path: 8Mbps, HS-Path: 10Mbps, pipe: 5Mbps): Primary only (P), kibbutz (K), MPBond (M), COMBINE (C), MPBond w/ 3 devices (M3), COMBONE w/ 3 devices (C3).



((a)) 512KB download. ((b)) 1MB download. ((c)) 2MB download.

Figure 4.11: Energy consumption reduction: MPBond compared to kibbutz.

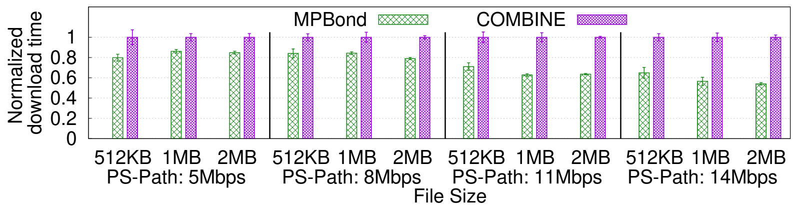


Figure 4.12: Performance of MPBond v.s. COMBINE under different BW combinations: PS-Path: {5, 8, 11, 14} Mbps, HS-Path: 10Mbps, pipe: 5Mbps.

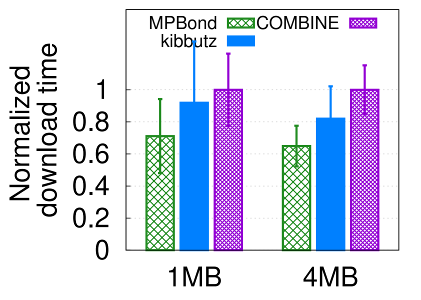
4.5.3 Stable Network Conditions

In this section, we evaluate the performance and energy efficiency of MPBond under stable network conditions. The workload is downloading files with sizes ranging from 512KB to 2MB. We vary the number of mobile devices from 1 to 3 and measure the download time and energy consumption. We also study MPBond-Naive, another variant of MPBond where the default minRTT scheduler instead of PAMS is used. For each test, we repeat the download 20 times and report the mean value and the standard deviation.

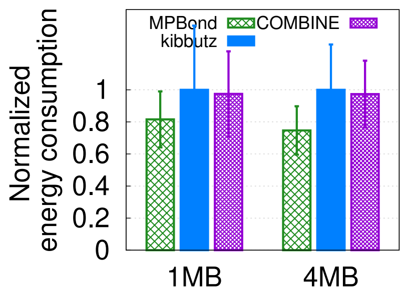
Figure 4.9 shows the results under a common network bandwidth setting. Each plot in Figure 4.9 has 7 clusters corresponding to different schemes with different number of devices. A cluster that is closer to the bottom left has a lower energy consumption and a shorter download time. Note that when calculating the energy consumption, we consider all the mobile devices involved. Compared to kibbutz, MPBond reduces the download time (energy consumption) by 5%-11% (10%-14%), when there are 2 devices. When the number of devices becomes 3, compared to kibbutz which cannot utilize the extra device due to its architectural limitation, MPBond improves the download time by 25%-30% while maintaining a similar total energy consumption. Compared to COMBINE, MPBond brings even higher improvements in terms of both download time and energy consumption. With two (three) devices, MPBond improves the download time by 15%-21% (12%-26%), and reduces the energy consumption by 28%-38% (22%-25%). While MPBond-Naive has a similar energy consumption compared to MPBond, it sacrifices the performance due to suboptimal scheduling decisions that lead to imbalanced subflow completion (§4.3.3). These improvements are attributed to multiple design choices of MPBond including the system and pipe realization at Layer 4, the helper-side connection split and buffer, as well as the carefully designed multipath scheduler.

To better understand the impact of using more devices, we further break down the total energy consumption for different schemes in Figure 4.10. MPBond-Naive is omitted here for brevity. As shown, using more devices does increase the total energy consumption, and COMBINE even increases the energy of the primary due to its poor scheduler design that does not distribute the workload in an efficient manner. MPBond instead reduces the energy on the primary when more devices are used, while keeping a reasonably higher total energy consumption. Compared to kibbutz, the energy improvement of MPBond goes mostly to the helper device thanks to its buffering strategy that reduces radio-on times.

To more systematically understand the benefits of MPBond against kibbutz and COMBINE, we further carry out experiments under more bandwidth combinations. We focus on the 2-device case where we use Pixel 2 as the primary and Nexus 6P as the helper, with the pipe bandwidth limited at 5Mbps. We first examine the energy improvement of MPBond over kibbutz, with different PS-Path and HS-Path bandwidths. Figure 4.13 plots the energy saving results. With higher HS-Path bandwidth and lower PS-Path bandwidth, MPBond's energy benefit is maximized: for 1Mbps PS-Path and 18Mbps HS-Path, energy consumption is reduced by 31%, 37% and 47% for 512KB, 1MB, and 2MB download, respectively, while the download time of MPBond is slightly better than kibbutz. The energy savings mainly come from the effectiveness of helper-side buffering that reduces the radio-on time of the faster link under heterogeneous WWAN and WLAN links. We then compare MPBond with COMBINE by changing the PS-Path bandwidth. Figure 4.12 plots the download time for both schemes. As shown, MPBond reduces the file download time by 14%-46%, leading to energy savings of 24%-57%. Overall, as the heterogeneity between pipe and PS-Path increases, the improvement brought by MPBond becomes larger.



((a)) Download time under varying network condition.



((b)) Energy consumption under varying network condition.

Figure 4.13: Energy consumption reduction: MPBond compared to kibbutz.

4.5.4 Varying Network Conditions

We next evaluate how MPBond performs under changing network conditions. We focus on the 2-device case where Pixel 2 is the primary and Nexus 6P is the helper. We first replay the real WWAN and WLAN bandwidth profiles we collected in §6.2. Figure 4.13(a) shows the download time of different schemes. Compared to kibbutz (COMBINE), MPBond reduces the download time by 21%-23% (29%-35%). The corresponding energy consumption reduction is 18%-25% (16%-23%), as shown in Figure 4.13(b). This again shows that leveraging helper-side connection split, buffer management, and the judiciously designed PAMS scheduler helps MPBond to achieve high network utilization under fluctuating network conditions.

In-the-wild Experiments. To further understand the benefits of MPBond, we conduct field test in real world settings. We focus on comparing MPBond with kibbutz whose performance is closer to MPBond. Specifically, we conduct experiments at two locations by performing 1-min download for each scheme back-to-back at each place and repeat it for

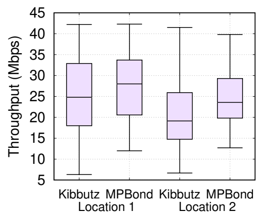


Figure 4.14: Results of in-the-wild experiments.

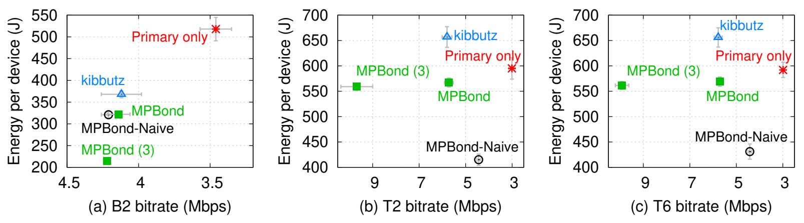


Figure 4.15: Video streaming QoE & energy. (PS-Path: 5Mbps, HS-Path: 10Mbps, pipe: 5Mbps): Single device (Pixel2), MPBond (Pixel2+Nexus6P), MPBond w/ 3 devices (Pixel2+Nexus6P+LG2), and kibbutz (Pixel2+Nexus6P).

10 times. We measure the instantaneous throughput every 100ms. Figure 4.14 shows the throughput distribution of MPBond and kibbutz. At the first location, MPBond improves the median throughput by 13% compared to kibbutz. At the second one, the improvement is 23%. MPBond also greatly reduces the low throughput periods, with a 90% improvement of 5th percentile throughput over kibbutz in both locations, due to its buffer management and helper-side connection split that exploit the capacity of the fluctuating WWAN and WLAN links as much as possible (§4.3.2). The energy per byte is improved by 19% and 24% at the two places respectively (not shown in the figure).

4.5.5 Video Streaming Performance

All experiments so far use bulk file download as the workload. We now examine how MPBond helps improve the QoE and energy efficiency of video streaming, one of the applications that dominate mobile network traffic. We stream adaptive bitrate (ABR) videos using Exoplayer [26] to study the impact of different schemes on video bitrate. We use three video settings: Big Buck Bunny with 2-second segment duration (B2), Tears of Steel with 2-second segment duration (T2), and Tears of Steel with 6-second segment duration (T6). Big Buck Bunny has 20 bitrates ranging from 46kbps to 4.2Mbps, while Tears of Steel has 9 bitrates ranging from 253kbps to 10Mbps. The total video duration for them are 596s and 734s, respectively.

We focus on the comparison between MPBond and kibbutz, both of which don't require modification to the video streaming application. Figure 4.15 shows the video bitrate and per device energy consumption. With two devices, MPBond reduces the energy consumption by 13%-14% compared to kibbutz, while achieving similar video bitrate. When the number of devices become three, MPBond improves the video bitrate by 118% compare to kibbutz for two (T2 and T6) of the three settings. The rest one (B2) doesn't show much improvement since using two devices can already reach the highest bitrate. Nevertheless, the per device energy consumption in B2 is reduced because of the help of the third device. We further make two observations in T2 and T6: (1) MPBond-Naive achieves even lower energy consumption compared to MPBond and kibbutz, but with the cost of a lower video bitrate. (2) While MPBond helps improve the bitrate as the number of devices increases, the per device energy consumption doesn't get reduced like bulk download does, because a higher bitrate corresponds to a larger video segment: this is a classic tradeoff between QoE and data usage in ABR streaming.

360-degree Video Streaming. 360 degree video streaming has a much higher bandwidth requirement compared to regular video streaming and is an ideal use case for MPBond. For our experiment, the video bitrate is fixed at 64 Mbps. Since the mobile devices we have do not support the decoding of such high definition, we instead employ a video player emulator to download the video content without rendering it. We employ video stall ratio (stall time divided by video length) as the QoE metric and vary the number of device from 1 to 3 to study how much improvement MPBond brings. For the single primary device the stall ratio is as high as 145%, while using a helper helps reduce it to 27%. It's further reduced to 3% when three devices are used – this clearly shows that in reality the fluctuating LTE oftentimes does not always meet the high bandwidth requirement of 360-degree videos and further motivates the need of MPBond to support more than 2 devices.

4.5.6 Leveraging the Dual Mode

We now evaluate the benefits of dual mode by involving two users, each carrying a smartphone (Pixel 2/Nexus 6P) with LTE connectivity (T-Mobile/AT&T). Both of their LTE are capped to 5Mbps. The pipe is unthrottled. To examine how much improvement MPBond's dual mode brings, the two users start the following workload at the same time: sequential 1MB chunks are requested on each smartphone, with the inter-chunk time being a random number between 1 and 5 seconds, emulating the video streaming traffic. Figure 4.8 compares the chunk download time in dual mode of MPBond and the download time when each of them download independently without MPBond. As shown, the download time is improved by 32% and 29% for Pixel 2 and Nexus 6P, respectively.

4.5.7 Indoor Applicability

Above experiments focus on MPBond's main use case – outdoor cellular networking. Now we consider indoor environments where WiFi infrastructures most likely exist. When there's a WiFi infrastructure, MPTCP over WiFi and LTE can be easily applied for bandwidth aggregation. To understand how MPBond compares to it, we conduct 4MB file download experiments at two different indoor locations to study the performance and cellular data usage of two schemes: (1) MPBond on a primary and a helper where the PS-Path and the HS-Path are over LTE, pipe is over WiFi, (2) MPTCP over WiFi and LTE on the primary. The two locations are with different WiFi network conditions: location A has a high average WiFi signal strength of -51dBm while location B receives weaker WiFi signals (-68dBm on average). Across 10 back-to-back runs, the average file download times of scheme 1 and 2 at location A are 1.5s and 1.0s, respectively. At location B, the corresponding download times for scheme 1 and 2 are 1.2s and 1.6s, respectively. The cellular data usage of scheme 2 at location A and B are 1.6MB and 3.3MB, respectively. The results show that (1) MPBond always has a higher cellular data (metered) usage (4MB) compared to scheme 2, (2) depending on the WiFi network condition, MPBond may either outperform (*e.g.,* at location B) or underperform (*e.g.,* at location A) scheme 2. In indoor environments with good WiFi networks such as location A, an MPBond user can choose to fall back to scheme 2, the regular multipath over WiFi and LTE, *e.g.,* by leveraging context information [129]. We leave developing a full-fledged context-aware framework for automatic switching between scheme 1 and 2 as future work.

4.5.8 System Overhead and Energy Concerns

We measure the CPU utilization on the MPBond primary as well as the helper when running the same workload as kibbutz: downloading a large file from remote server. We repeat the experiment for 10 times and the average extra CPU utilization compared to kibbutz is no more than 4% for both the primary and the helper. We also answer the question left in §4.2.3 to examine the battery drain of a wearable when acting as a helper.

We stream a 15-min video and examine the battery drain of a fully charged LG Urbane 2 smartwatch. We repeat it for 10 times and observed no more than 7% average battery drop: this shows the feasibility of our solution.

The previous energy measurement results in the evaluation section are based on power models instead of hardware tools. To understand the measurement errors (*i.e.,* model inaccuracies), we now use a commercial power monitor [19] to measure the real energy consumption. We employ a Samsung Galaxy S5 smartphone that can to be hooked by the power monitor. We use it as the helper and a Pixel 2 as the primary. Our workload is downloading a 4MB file under the following network setting: PS-Path: 8Mbps, HS-Path: 10Mbps, pipe: 5Mbps. The power monitor measures the helper-side energy consumption with both MPBond and COMBINE. We focus on energy measurement on the helper due to the limited number of power monitors we have and a helper is usually less powerful and more energy-constrained (*e.g.,* a wearable). We repeat the experiment 10 times and the helper on average consumes 2.3J and 3.4J energy for MPBond and COMBINE, respectively. Compared to the absolute energy numbers derived from models (1.9J and 2.6J), the error can be as high as 24%. However, the difference between model-based (27%) and power monitor-based (32%) relative energy reductions (MPBond over COMBINE) is as low as 5%.

4.6 Summary

This chapter provides cross-device connection management and packet scheduling support for network-level collaboration among personal mobile devices. By developing and evaluating MPBond, a distributed mobile multipath transport system, we show that network transport can be designed in a cross-device manner to efficiently aggregate heterogeneous wireless network resources on multiple mobile devices.

CHAPTER V

Analyzing the First-Mile Ingest Performance of Live Video Streaming

While Chapters §III and §IV improve the transport protocols on mobile systems without modifying the applications. Starting from this chapter, we look at the problem from a different perspective and optimize the application-layer design to adapt to varying network conditions in order to improve mobile application performance. This chapter focuses on the emerging live streaming application, which is both bandwidth-intensive and latency-sensitive. We demonstrate that existing live video upload applications incur poor coordination between the application decisions and network conditions, and schemes that better adapt real-time encoding rates to network bandwidths can improve QoE.

5.1 Introduction

Live video streaming traffic has grown significantly, fueled by improvements in camera technologies, computing power, and wireless resources. The rise of services such as

Facebook and Youtube creates global platforms to disseminate user-generated content. According to a recent industry report [80], live video will account for more than 15% of the Internet video traffic by 2022.

An end-to-end (E2E) live streaming pipeline consists of the ingest and distribution paths shown in Figure 5.1. On the upstream ingest path, the video is captured in real time by a camera, then fed into a *Broadcasting App* that compresses the video and transmits it to a remote *Video Server* owned by some streaming service over a network connection, typically cellular or Wi-Fi. On receiving the ingest stream, the video server transcodes it into a number of different ABR tracks (referred to as ABR track ladder), each corresponding to a different encoding quality level and bitrate. Each track consists of several video segments (usually 2–10 seconds each). Viewers watching the live stream request a mixture of segments from the video server using adaptive bitrate (ABR) streaming [138] over the downstream distribution path.

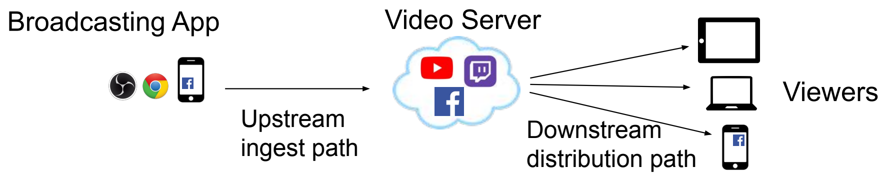


Figure 5.1: Live video streaming end-to-end workflow.

Existing studies have focused largely on the last-mile distribution path from the video server to the viewers. There has been little exploration of the first-mile ingest path from the broadcasting app to the video server. However, this first mile is critical to the E2E performance of the pipeline. The quality of the video delivered on this first mile to the video server imposes an upper limit on the quality of the ABR tracks created from it, and therefore on the quality of experience (QoE) of the viewers of the live stream. In addition to delivering a good quality video stream to the video server, the first mile also needs to provide the content with low latency. Any latency on the first mile impacts the overall E2E latency for the end viewers (see §5.2.1). Improving the ingest performance would therefore benefit the QoE of all the downstream viewers. However, achieving this goal is also challenging due to the usually more dynamic and limited wireless uplink resources (*e.g.,* cellular uplinks) and the complexity of the ingest path.

In this study, we examine the all-important first-mile ingest path in commercial live streaming platforms to understand their performances and designs. Such insights can assist developers in identifying deficiencies and creating designs with improved performance and network providers to better understand and manage the associated traffic [155, 154, 70]. Our goal is to analyze a wide range of commercial live video broadcasting apps and streaming services from an objective third-party point of view, in a controlled, repeatable, and fine-grained manner (§5.2.2). This task is made challenging by the complex E2E pipeline, the proprietary closed-source software components, and the wide diversity of designs across different live streaming systems. The live nature of the content introduces further challenges in conducting measurements (see §6.2.3).

In view of these challenges, we develop a generalized black-box measurement methodology and tool, Livelyzer, for analyzing the performance of the upstream ingest path for commercial live streaming systems. Livelyzer enables third parties to conduct active measurements to profile the performance under various network conditions in a repeatable and controlled manner, thereby gaining insights into the corresponding design. The design of Livelyzer and its capabilities are detailed in §5.3.

We use Livelyzer to study a wide range of broadcasting apps such as third-party, browser-based, and mobile-based broadcasting apps streaming to commercial services including Facebook, Youtube, and Twitch, using different video contents and network conditions. In total, we study seven (broadcasting app, streaming service) combinations. Our key findings are:

• Different broadcasting apps have very different encoding rate control designs/configurations. Many of them use Constant Bit Rate (CBR) encoding, leading to inefficient use of bits on the upstream ingest path (§5.4.1).

• Different broadcasting apps behave very differently when the network conditions change. Our evaluations show that while all the broadcasting apps we study exhibit adaptation to changing uplink network conditions, they differ widely in the specific adaptation behavior and resulting ingest performance. Further, our results suggest that the existing adaptation strategies have limitations and sometimes lead to poor performance. For example, the Open Broadcaster Software (OBS), by default, drops frames in an inefficient way to cope with network condition degradation, leading to poor video quality (§5.5.1.1). Although OBS recently introduced a "dynamic bitrate mode" for encoding rate adjustment, we find that the scheme can largely under-utilize the newly available network resources due to how it adapts the encoding bitrate when the network bandwidth increases (§5.5.1.2). Browser-based (§5.5.2) and mobile-based (§5.5.3) broadcasting apps also exhibit performance issues.

• We leverage Livelyzer to conduct a what-if analysis of the rate adaptation logic usage, in order to understand the impact of different rate adaptation schemes on the live video ingest performance (§5.6). We show how even a relatively straightforward adaptation strategy inspired by the findings of Livelyzer can help improve the performance.

• The video server design also has implications for QoE. For example, we find that different services choose different segment durations, making the ingest delay variable across different broadcast settings (§5.4.2).

5.2 Background and Motivation

5.2.1 First Mile in Live Video Streaming

As mentioned in §6.1, an E2E live streaming pipeline consists of the ingest and distribution paths (Figure 5.1). The E2E QoE of live streaming is fundamentally constrained by a single video stream delivered over the first-mile ingest path. First, the quality of the video delivered on this path to the video server imposes an upper limit on the quality of the ABR tracks created from it, and therefore on the QoE of the viewers of the live stream. Second, a player can only download a video segment after the corresponding video content is uploaded to the video server, imposing a latency dependency.

The broadcasting app, a critical component on the ingest path, usually comes in three different forms [160]:

(1) Third-party broadcasting app: There exists standalone software that captures and transmits videos to commercial live streaming services. For example, Open Broadcaster Software (OBS) [139] is a popular broadcasting app that supports live streaming to many commercial services, which highly recommend the use of it [61, 161, 144]. The widely used OBS software supports RTMP (Real Time Messaging Protocol [43]), which is also one of the main protocols that commercial video services use.

(2) Browser-based broadcasting app: Many services such as Facebook and Youtube provide GUIs to open cameras to capture and stream real-time content from their web pages [60, 159].

(3) Mobile-based broadcasting app: Instead of using the browser, smartphone users may prefer the service's mobile app, which also includes GUIs to open the camera and stream videos [59, 158].

Many live broadcasts are originated from mobile devices and transmitted to remote servers using available connections such as Wi-Fi or cellular. To understand how well today's mobile uplinks support the needs of live video ingest, we measure the uplink bandwidth relative to the live ingest bandwidth requirements of commercial broadcasting apps. Specifically, we conduct uplink throughput measurements by uploading a large file over multiple LTE networks, covering various scenarios involving different movement patterns, signal strengths, and locations. Our measurements indicate that the uplink bandwidth exhibits significant variability and can be lower than the sustained bandwidth requirements of commercial broadcasting apps ($\sim$1–4Mbps for many live streaming systems). As an illustration, for each of the ten traces that we collected, the 5^{th} percentile of the bandwidth values was less than 2.5Mbps. For four traces, the median observed uplink bandwidth was less than 2.1Mbps. While 5G is expected to further improve the bandwidth, the technology is not yet widely deployed, and has its own challenges (*e.g.,* directivity and sensitivity to blockage for mmWave [114]). Therefore, it is important for applications that need to use uplink cellular connections to be designed appropriately.

5.2.2 Design Goals

A sound measurement system for ingest path analysis should be able to meet the following requirements.

G1 Enable interested entities such as a testing service or a network operator (who usually do not have access to the detailed design of, or the source code for the software) to conduct third-party measurements of the performance of a live streaming system.

G2 Be generally applicable to different live video broadcast and distribution platforms instead of targeting a specific setup.

G3 Enable controlled and repeatable experiments.

G4 Be capable of measuring performance dynamics at a fine timescale and enable a tester to holistically reason about the design of the live ingest pipeline.

5.2.3 Limitation of Existing Analysis Approaches

Existing live streaming analysis tools have several limitations. *First, they conduct limited ingest analysis.* [104] focuses on the distribution path, *e.g.,* instead of directly measuring the video quality (§5.3.4), it measures the end-viewer perceived video resolution, which characterizes the downlink ABR performance instead of the ingest performance. As §5.5 will show, the same resolution (ABR track) can have very different quality due to the quality difference of the video delivered on the ingest path. [136, 137] measure the overall E2E QoE instead of for the ingest path, making it difficult to distinguish performance issues on the ingest and distribution paths. They measure different QoE metrics separately under different settings, making it hard to correlate one metric with another. Besides, they mainly focus on the overall session-level QoE instead of its fine-grained dynamics over time. [146] focuses on measuring the delay aspect of the QoE. Therefore, they are not able to meet **G4**.

Second, existing approaches are either broadcasting app-specific or streaming service-specific and hard to generalize. [104] only focuses on streaming from OBS, which provides a user interface to stream local video files, making it hard to generalize to different commercial broadcast platforms such as browser-based and mobile-based broadcasting apps. [136, 137] and [146] rely on service-specific APIs to measure the QoE. As a result, they fail to achieve the aforementioned **G2**.

*Third, they do not meet the **G3** requirement for controllable and repeatable measurements.* [137] and [146] watch online live streams broadcast by other people, having no control of the video source and network conditions. [136] shoots videos playing on a laptop screen as the source, but it suffers from distortion and lighting issues. Although [104] can guarantee the same input video source, it is hard for it to control the network conditions under which its data are collected due to its "in-the-wild" nature.

5.2.4 Challenges

Achieving the goals mentioned in §5.2.2 is not straightforward. In addition to the live nature of the video content, the live streaming pipeline is complex and heterogeneous. This creates the following challenges that Livelyzer needs to address:

Complex pipeline. Live video ingest involves a complex pipeline. A broadcasting app captures a video from a camera, encodes it using a codec with an encoding rate control[1] scheme, and transmits a sequence of video frames to the remote server over some network connection. To cope with time-varying network conditions, a broadcasting app may dynamically adapt its upstream transmission. There can be different ways of adjusting the amount of data to send to the remote server, *e.g.,* dropping frames or reducing the encoding bitrate. The video server transcodes uploaded video frames into ABR segments. Each of the above components plays an important role and can impact the E2E QoE. This complexity makes it challenging to identify, exercise, and understand the key pieces that impact the QoE.

Heterogeneous design. Different services can have very different designs. Even for the same service, the broadcaster can choose different broadcasting apps of different designs, such as the service's web page in a browser, its mobile app, or a broadcasting app from third parties such as OBS. Unlike the distribution path where HTTP Adaptive Streaming (HAS) is the predominant approach, there is no single *de facto* delivery solution on the ingest path. In fact, there exists a wide range of ingest solutions (*e.g.,* RTMP [43], WebRTC [58], FTL [76], DASH-IF Live Media Ingest [66], *etc.*) with varying levels of publicly available specifications. This makes achieving G2 hard for the ingest path.

Proprietary nature of systems. Live streaming systems usually run proprietary closed-source software. A third party typically does not have visibility into the source code of broadcasting apps and video servers. The uplink network traffic is also usually encrypted, *e.g.,* broadcasting apps may use RTMPS [89] or WebRTC with DTLS [132]. In addition, the increasing use of SSL pinning in mobile applications [136] renders MITM proxy-based approaches increasingly unusable.

[1]In video coding, rate control means what an encoder does to determine how many bits to spend for each frame to reach a target bitrate or quality level for the video.

Live nature of content. Unlike video on demand (VoD), where the same content can be replayed across multiple experimental runs, live streaming contents are generated in real time. This makes the task of repeating experiments using the same source (**G3**) difficult.

Need for suitable performance metrics. An end user watches a video that flows over both the ingest and distribution paths. While there are well-defined QoE metrics (*e.g.,* quality, stall ratio, startup delay) for the distribution path, there are no well-defined or widely accepted performance metrics for the ingest path. In §5.3.4 we shall define such metrics of interest.

5.3 The Livelyzer Measurement System

We build Livelyzer, a holistic measurement system that comprehensively examines the ingest performance of different broadcasting apps streaming to various services. As shown in Figure 5.2, Livelyzer interacts with the live streaming pipeline by generating video source contents and uplink traffic control rules (TCR), monitoring the upstream network packets (NP), and collecting ABR track and manifest information. Livelyzer consists of components running on a test device and an analysis server. The test device hosts different broadcasting apps and part of our software that annotates source videos (§5.3.3), injects them to broadcasting apps (§5.3.2), automates measurement tasks, and sends local measurement data to our analysis server after measurement sessions. The analysis server runs the rest of our software, which downloads the top ABR track (TAT) and video manifests, and later analyzes the ingest performance offline (§5.3.4).

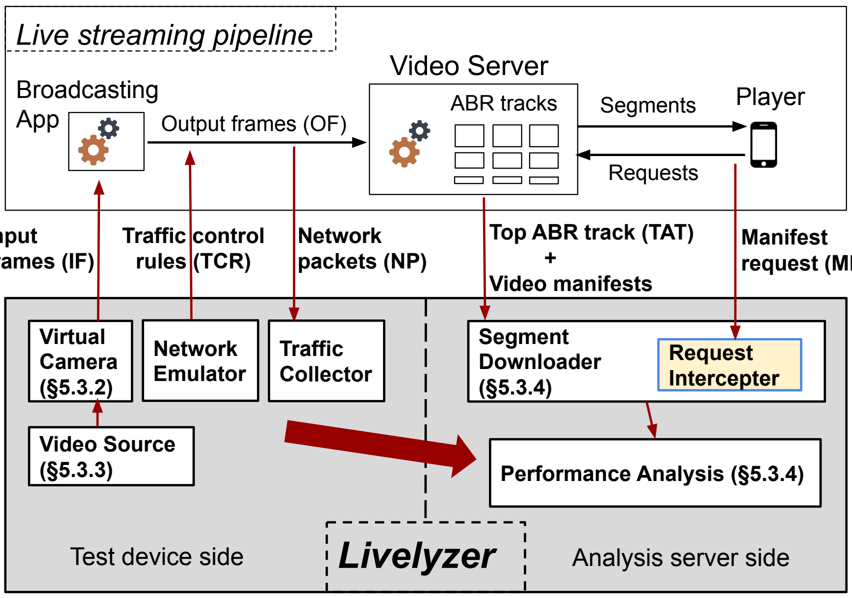


Figure 5.2: The system architecture of Livelyzer.

5.3.1 Black-box Testing

Ideally, we would like to have visibility of every internal point on the ingest path, *e.g.*, the encoder output frames of the broadcasting app (OF in Figure 5.2) and the application data in the network upstream. However, as mentioned in §6.2, commercial streaming services and broadcasting apps are usually closed systems. Hence, it is hard to access either the ingest endpoint of the video server to gain visibility into the uploaded frames or the encoder of the broadcasting app to examine the compressed frames.

Given the lack of such internal visibility, we adopt a black-box analysis approach. Specifically, we control both the input (*i.e.*, the video content) and the ingest components (*e.g.*, broadcasting app, streaming service, network condition, *etc.*), and observe the output (*e.g.*, quality of TAT). By changing the input or/and the ingest settings (*e.g.*, through issuing different TCR), we can observe how the output would be affected, and reason about the ingest components.

Our method can test a specific broadcasting app streaming to a particular service under various network conditions in a controlled manner. Specifically, we use a network traffic control tool like Linux `tc` to replay different network conditions based on real-world network traces we collect. Livelyzer runs a traffic collector module on the test device to collect packets on the ingest path. Livelyzer also runs a segment downloader that collects video server output for performance analysis (§5.3.4). For scalable testing, the broadcast and playback processes are automated (*e.g.,* through Android UI automation and Selenium [133] for browser automation).

5.3.2 Virtual Video Capture Function

As mentioned in §6.2, we need to do measurements in a repeatable way. This translates to two requirements: *First*, we need to be able to feed the same video content to different broadcasting apps. *Second*, we need to be able to provide the same video content to the same broadcasting app across different runs. The first requirement comes from our need to compare different broadcasting apps and streaming services fairly. The second requirement is because, for the same (broadcasting app, streaming service) setting, we may want to vary a factor (*e.g.,* network condition) over different runs and keep other factors including the video source the same to examine the sole impact of this specific factor on the ingest performance.

One way to address this is to capture the same scene using a real camera. This approach has several issues: First, it is difficult to provide the same physical scene multiple times in the real world where time and space cannot be reverted. Second, even if we can provide a "repeatable" physical scene like [136] that plays a pre-recorded video on another screen, this still makes it hard to keep the captured video the same due to lighting-related dynamics. Furthermore, the video captured by the camera can contain frames that are a composite of multiple consecutive source frames in the pre-recorded video. This can be caused by the exposure time of the camera capture process lining up with the display times of those frames. Such composites can cause harmful interactions with other components in Livelyzer, such as source annotation (§5.3.3).

Some broadcasting apps such as OBS support local file input, but not every broadcasting app supports this mode. For example, browser-based broadcasting apps only support camera or screen sharing mode, mobile-based broadcasting apps such as the Facebook and Instagram apps only support camera mode. Screen sharing also introduces non-deterministic distortions in the screen recording process [153], making the video captured by a broadcasting app (*i.e.,* the recorded screen) just an approximation of the source video.

To provide a universal interface to commercial broadcasting apps for capturing repeatable video contents, we create a virtual camera in Livelyzer. It takes a local video file as input and behaves like a normal camera device from the perspective of a broadcasting app. The video source can be fed into the virtual camera at different frame rates. The virtual video capture function extracts the sequence of frames from an input video file, and redirects them to the virtual camera. The broadcasting app then captures input frames (IF in Figure 5.2) by sampling these raw frames based on the broadcasting app's frame rate setting. In Linux, a virtual camera can be realized using `/dev/videoX`. We leverage `v4l2loopback` [145] to create such a new virtual video device. To capture video with it, we use `FFmpeg` [63] to specify a local video file as the input and the virtual video device as the output.

5.3.3 Crafting Video Source Files

Given the virtual camera, we still need to prepare the input video content. To measure both frame loss and quality of delivered video on the ingest path, we need to associate each frame in the received video with its corresponding frame in the source.

Achieving frame alignment in live video ingest is challenging because the frames in the source and received videos are not naturally aligned for several reasons. First, different broadcasting apps could capture videos at different frame rates. Second, during transmission to the remote video server, a broadcasting app may drop frames to adapt to varying network conditions. Third, depending on when it joins the live event, a player will not necessarily start playing a video from the beginning of a broadcast. Therefore, the first frame being played may be different from the first frame captured by the broadcasting app. Also, depending on the extent of time synchronization between when the camera is turned on and when the broadcaster starts streaming to the remote video server, even the first frame captured (to be encoded) by the broadcasting app may be different from the first frame that is seen by the recording camera.

One approach for achieving frame alignment would be to compare every frame in the received video and every frame in the source based on content similarity. However, the challenge with this approach is that in the case of a static or a repeating scene, multiple frames in the source would be visually similar to each other. In this case, a frame in the received video may be mapped to multiple frames in the source, making unambiguous frame alignment hard.

To achieve frame alignment, we overlay a unique signature to every source frame and leverage computer vision techniques to detect the signatures from frames in the received video to match each of them to a source frame. The signature should be robust to compression artifacts (*i.e.,* be still recognizable from low bitrate streams created under poor network conditions). We leverage the Quick Response (QR) code [150] to create this specific signature since it is more robust than the approach of overlaying a sequence of digits (*e.g.,* a frame number) due to the QR code's use of Reed–Solomon error correction [149]. We empirically verified this when we were researching suitable signature technologies in the early phase of our study. In addition, we pad the source video with dummy frames before and after the original video content to make sure that the original video content gets captured and eventually played regardless of the level of time synchronization among different components.

5.3.4 Analyzing Ingest Performance

As mentioned in §6.2, we need to define performance metrics on the ingest path that impact end-user QoE. The metrics we consider are video quality, effective frame rate, ingest freshness, and ingest smoothness (to be detailed shortly). To facilitate ingest performance analysis, we need to measure the quality and timing of the stream delivered to the video server. However, extracting video frames in such a stream on the ingest path is challenging (§6.2.3). Instead, we use the transcoded TAT to approximate the above stream, easily accessible using standard HTTP APIs from the distribution path. TAT is also more closely associated with end viewers' experience as players ultimately need to fetch video segments from ABR tracks, and TAT represents the best quality encoding any user can receive. Livelyzer runs a segment downloader on its analysis server to fetch ABR tracks. We parse the corresponding live video manifest for the session to obtain the address information for the TAT, which is passed to our segment downloader for downloading the TAT segments. During the live stream, the manifest is periodically updated over time as new segments are generated at the server. Our segment downloader fetches these updated versions at regular intervals to extract the information required for computing the various performance metrics described next.

Video quality. We examine the quality of video segments in the top ABR track (TAT) created by the video server. TAT represents an upper bound on the video quality experienced by any user served by the video server. The video quality of the TAT depends on the quality of the content received by the server on the ingest path, which is the cumulative end result of all ingest activities (including broadcasting app encoding and adaptation, network performance, *etc.*).

For the specific video quality, we adopt Video Multimethod Assessment Fusion (VMAF) [102, 98, 95], which is a recently proposed perceptual quality metric and has been shown to perform much better than traditional video quality metrics that do not accurately capture human perception, such as Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) [148]. VMAF is a full-reference model allowing us to measure the perceptual quality of a distorted video by comparing it with regard to a pristine quality reference of the same content. VMAF was originally designed for evaluating compression artifacts where the reference and distorted videos have the same frame rate. In contrast, in the live video ingest use case, the source video and TAT can have different frame rates (see §5.3.3). Therefore, we first resample the TAT to match its frame rate with the source video using FFmpeg [63]. We then use the frame alignment step (§5.3.3) to align the first frame in the TAT (t1) with its corresponding frame in the source (s1). Then we calculate VMAF using the sequence of source frames starting from s1, as the reference sequence, and the sequence of TAT frames starting from t1, as the distorted sequence. Since we are particularly interested in the quality of the content consumed by mobile users, we use the VMAF phone model (designed for small screens [96]) for VMAF calculation. Since we are interested in measuring video quality at a fine granularity over the video session, we use the arithmetic mean of the VMAF values of all the frames in a segment as the segment's VMAF value, and analyze the distribution of per-segment VMAF values [127, 42] across the session. Note that while we use VMAF for the reasons stated above, Livelyzer can easily accommodate other video quality metrics, *e.g.,* PSNR and SSIM.

Effective frame rate. Broadcasting apps may capture frames at a different frame rate (FPS) and drop frames when the network bandwidth becomes insufficient. Besides, networks may drop frames, and video servers may reduce frame rates as well. To quantify the impact of frame loss, we define effective frame rate (effective FPS, or eFPS), the number of *distinct* frames in each second in TAT. eFPS equals FPS when there are no duplicate frames (*i.e.,* every frame is distinct). However, according to our observations (§5.5.1.1), video servers may duplicate frames to maintain a constant FPS in ABR tracks when the frame rate on the ingest path is variable. Therefore, to compute eFPS, we consider only distinct frames by identifying and removing duplicates using our frame annotation and alignment methods (§5.3.3).

Ingest freshness. We are also interested in understanding the latency impact of the ingest path, which affects the E2E broadcaster-to-viewer (B2V) delay (§5.2.1) – a measure of how much a viewer is behind the live event. To characterize ingest freshness, we define the *ingest delay* for each ABR segment as the time elapsed from when its first frame is generated at the source to when all the ABR track ladder variants of that segment become available at the video server for players to download. This segment-level delay is also more related to end users' experience compared to the frame-level delay — a segment[2] becomes available for players to download only after all the frames in the segment arrive at the server and the different ABR track variants for that segment have been created. The ingest delay for a segment is the sum of the times spent on broadcasting app encoding, network transmission, and server transcoding. The ingest delay is part of the E2E B2V delay. Therefore, a longer ingest delay will lead to a longer E2E B2V delay, everything else remaining the same. For each new updated version of the live manifest, we extract the time t_s that it was updated at the server. We mark the arrival time as t_s for all ABR segments that first appear in the latest version of the manifest and were absent in earlier versions. The generation time (t_b) of the first frame of each segment is recorded at the virtual camera. The ingest delay, or broadcasting-app-to-server (B2S) delay of a segment, can thus be calculated as $t_s - t_b$.

Ingest smoothness. Once a viewer joins a live event, in order to play a live stream smoothly without stalls, the player needs to get subsequent ABR segments from the video server in a timely fashion. This requires the ABR segments to be created and made available for downstream players in a timely manner. However, during a live stream, if the ingest delay increases significantly (*e.g.,* because the uplink bandwidth become very low for some time), the arrival of the video frames at the video server and subsequent creation

[2]In this chapter, we use segment to refer to the smallest unit of data that can be requested by an end viewer, *e.g.,* an ABR segment or a CMAF [65] chunk.

of the corresponding ABR segments will be delayed, increasing the chance that a player may not receive some segments in a timely fashion and therefore experience stalls. While due to live streaming freshness considerations, a player cannot stay far behind the live edge, for many common use cases, the player can still start several seconds behind the live edge (we define it as *offset*) and so can tolerate some variability in the availability time of the segments. Here, to measure the effect of this variability on user experience, we assume a player with an X seconds offset behind the live edge (we empirically set $X = 10$ in our experiments), and measure the stall behavior due to segment availability time variability. We define the stall ratio to be the aggregate stall duration as the percentage of the total live video session duration.

5.4 Using Livelyzer for Live Video Encoding Analysis

We next use Livelyzer to characterize the video encoding design on the live streaming ingest path, spanning broadcasting apps' encoding and video servers' transcoding, both important QoE-impacting components in the E2E live streaming pipeline (§5.2.1). We first consider high network bandwidths scenarios to ensure that the observed video outputs of the two components reflect their inherent application logic and are not caused by any network bandwidth limits. We shall later use this behavior as a baseline when we explore more bandwidth-constrained situations in §5.5.

We use three different representative video contents as our broadcast sources: (1) City – a city view with a lot of detail, (2) Concert – a live show that involves significant movement, (3) Talking – a talking person with an almost static background. The videos are obtained from Youtube in 1080p and 30fps. We select a 5-minute long sample of each

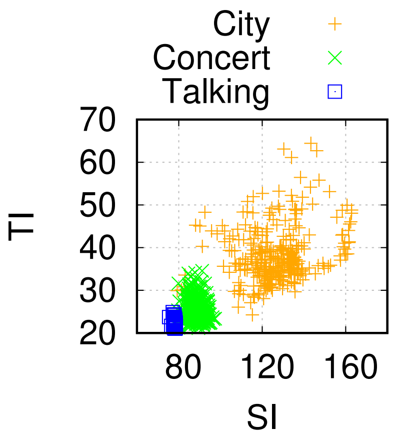


Figure 5.3: Content complexity measured by spatial information (SI) and temporal information (TI).

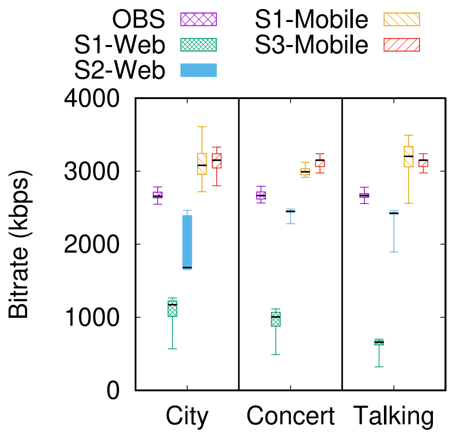


Figure 5.4: Encoding bitrate measured with different videos.

for this study. We stream the 720p variant of the content (obtained by downscaling the 1080p reference to 720p) as input to broadcasting apps, in line with industry recommendations [62], and use the original 1080p version as the pristine quality reference when computing VMAF (§5.3.4). Figure 5.3 depicts the spatial and temporal information [41], commonly used to characterize scene complexity, for these videos. Each data point in Figure 5.3 represents an individual segment from the corresponding video.

The three commercial services we examine are denoted as *S1*, *S2*, and *S3* in the rest of the chapter. We use the terms Web and Mobile to refer to the browser-based and mobile-based broadcasting apps, respectively, for each service.

5.4.1 Encoding Design of Broadcasting Apps

We first study the encoding bitrates of videos created by different broadcasting apps covering common broadcast use cases (§5.2.1). As mentioned earlier, experiments in this section are conducted under high-bandwidth network conditions, so the encoder should not be constrained by any bandwidth concerns[3]. This scenario represents the best-case performance of commercial broadcasting apps - we shall use it as a baseline when studying the rate adaptation performance of these systems under real-world network conditions when the upstream network bandwidth is not plentiful and is time-varying (§5.5).

Figure 5.4 shows the encoding bitrate distributions for different contents encoded by different broadcasting apps. We measure these values from the uplink network traffic by computing the data sending rates over time. Since we cannot extract the precise Group of Pictures (GOP) structure used by the different commercial broadcasting apps (*e.g.,* due to traffic encryption), we use 6 seconds as the interval to compute each bitrate sample. For OBS, we only present its encoding bitrate distribution when streaming to *S1* as the results for streaming to *S2* and *S3* are very similar.

We make two main observations. First, different broadcasting apps have very different outputs with different encoding bitrate distributions even for the same content, suggesting different encoding settings, *e.g.,* mobile-based broadcasting apps tend to use consistently higher bitrates than OBS and browser-based broadcasting apps.

Second, different broadcasting apps have very different encoding rate control behaviors. The OBS encoding bitrate is tightly concentrated around 2.7Mbps across all the content, suggesting the use of a CBR encoding independent of the content type. *S1-Web*, however, produces encodings with average bitrates and bitrate spreads that differ for different content – this is more consistent with VBR-like encoding behaviors. Other broadcasting apps, such as the mobile-based broadcasting apps, exhibit modest bitrate variations: the

[3]We leverage our source video padding (§5.3.3) to ensure that the bitrate has already “ramped up” when the actual video content comes.

bitrates are within 18% and 8% of the corresponding mean values for *S1* and *S3*'s mobile-based broadcasting apps, respectively. They exhibit similar bitrate spreads for encoding the three different video contents.

Above, we observe that many broadcasting apps we study use CBR. Given that VBR has advantages over CBR, such as being able to achieve higher video quality with the same average bitrate or provide the same quality with a lower bitrate encoding to better accommodate to bandwidth constraints [127], one potential research direction is to explore using VBR encoding in broadcasting apps.

5.4.2 Server ABR Transcoding Design

We now study how different services transcode a received live video stream into ABR tracks. We focus on the top ABR track (TAT), which represents the highest-quality stream that end viewers could enjoy. Any quality impairment observed in this track is entirely caused by the ingest component. We measure the bitrate, duration, frame rate, and ingest delay of each segment in this track.

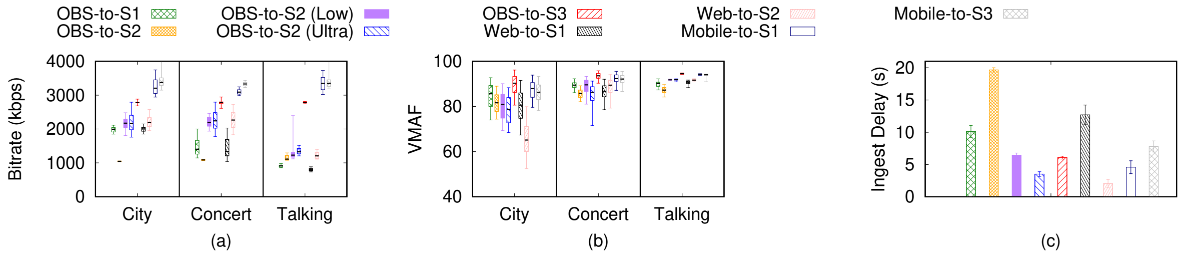


Figure 5.5: (a) Bitrate distribution of TAT. (b) Quality of TAT. (c) Broadcasting-app-to-server delay of segments in TAT.

Figure 5.5(a) shows the bitrate distribution of the TAT created by different services' remote video servers. For OBS streaming to *S2*, we include three modes that *S2* provides for RTMP ingest: default normal mode, low latency mode (denoted as Low), and ultra-low latency mode (marked as Ultra). As shown, different video servers also have very different encoding (transcoding) bitrate distributions for encoding the same content, similar to different broadcasting apps do (§5.4.1). However, here the content dependency compared to broadcasting apps' encodings appears to be higher overall: many video servers (*e.g., S1*'s video server receiving streams from OBS and *S2*'s video server receiving streams from *S2-Web*) use fewer bits to encode less complex content such as the "Talking" video. Also, we observe that the "dependence" between broadcasting apps' encoding bitrates and the ABR transcoding bitrates differ among services. For example, OBS encoding is very CBR-like (§5.4.1) while the corresponding *S1* video server creates content-dependent VBR encodings. However, for OBS streaming to *S3*, the bitrate distributions of OBS encoding and that of *S3* server transcoding are very similar (both centered around 2.7Mbps).

Table 5.1 shows the duration and frame rate of segments created by different servers. As shown, the segment duration can be very different depending on the service and broadcast platform. Later we shall see how the segment duration affects the ingest delay.

Table 5.1: Comparison of server ABR transcoding design.

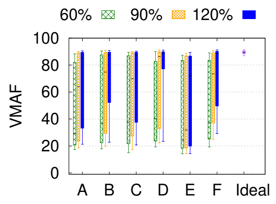
Streaming service	Broadcasting app	Mode	Segment duration	Frame rate
S1	OBS	N/A	2s	30 fps
	S1-Web	N/A	2s	30 fps
	S1-Mobile	N/A	2s	24 fps
S2	OBS	normal	5s	30 fps
		low latency	2s	30 fps
		ultra-low latency	1s	20-30 fps
	S2-Web	N/A	1s	20-30 fps
S3	OBS	N/A	2s	30 fps
	S3-Mobile	N/A	5s	30 fps

5.4.3 QoE Impact

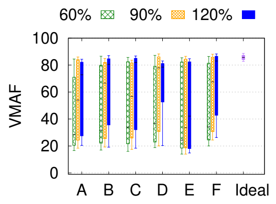
We next examine how different encoding rate control schemes in broadcasting apps and video servers affect the end viewer QoE.

Figure 5.5(b) shows the video quality of TAT created by each video server. Usually, a change of 6 or more VMAF points would be noticeable to a viewer. We observe that even under unconstrained uplink network conditions, the video quality is not always high. The "Talking" video has a much higher quality than the other two videos, likely a result of its relatively lower content complexity. Overall, settings with both high encoding bitrates at the broadcasting app and server sides (*e.g., S1-Mobile* and *S3-Mobile*) also have a high video quality.

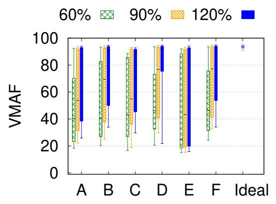
Figure 5.5(c) shows the ingest delay (§5.3.4). Overall we can see a correlation between the delay and the segment duration shown in Table 5.1: the larger the segment duration, the higher the ingest delay even though the broadcasting app is not necessarily doing segmented delivery to the server. The reason for the correlation is: (1) The ingest delay of a segment covers the time between when its first frame is generated at the source and when its last frame is uploaded to the ingest server, which depends on the segment's duration, (2) transcoding a larger segment usually takes longer than transcoding a shorter one. We also observed differences across services. For instance, both *S2* (normal mode) and *S3-Mobile* use 5s as the segment duration, while *S3-Mobile* has a much shorter ingest delay than *S2*.



((a)) Stream to *S1*



((b)) Stream to *S2*



((c)) Stream to *S3*

Figure 5.6: Video quality of OBS streaming to different services: each network trace (A-F) is scaled to 60%/90%/120% of the baseline video encoding bitrate

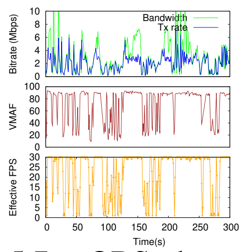
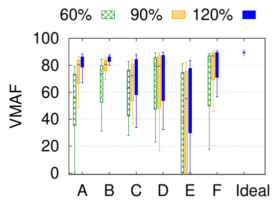
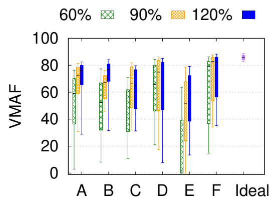


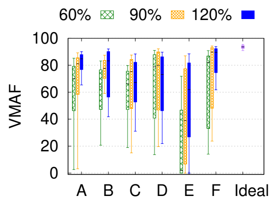
Figure 5.7: OBS drops frames to adapt to changing network conditions.



((a)) Stream to *S1*



((b)) Stream to *S2*



((c)) Stream to *S3*

Figure 5.8: Video quality of OBS dynamic bitrate mode streaming to different services: each network trace (A-F) is scaled to 60%/90%/120% of the baseline video encoding bitrate

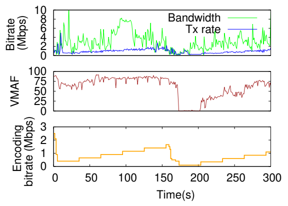


Figure 5.9: OBS-dynamic increases encoding bitrate slowly when bandwidth increases.

5.5 Using Livelyzer for Network Rate Adaptation Analysis

Next, we study how different broadcasting apps adapt their upstream transmission to cope with network dynamics, and the resulting impact on performance. As mentioned in §5.2.1, there can be different ways of adjusting the amount of data to send to the network, *e.g.,* dropping frames or reducing encoding bitrates. To understand the adaptation behavior, we measure the performance evolution across time and correlate it with the corresponding prevailing network bandwidth condition.

We focus on the "Concert" video with a medium content complexity across the three videos we have (Figure 5.3). We leverage the network condition emulation feature of Livelyzer (§5.3.1) to replay six real-world network bandwidth traces. For each of the six cellular uplink traces (denoted as A, B, C, D, E, and F, whose coefficients of variation are 1.11, 0.90, 0.84, 0.65, 1.08, and 0.69, respectively), we create three variants as follows for each broadcasting app. We proportionally scale the per-second bandwidth values in a trace such that the average bandwidth of the resulting scaled trace becomes either 60%, 90%, or 120% of the average of the encoding bitrate time series output by that app under plentiful uplink network conditions (§5.4.1). This, in total, creates $6 \times 3 = 18$ different network conditions for replay for each broadcasting app.

5.5.1 Using Third-party Broadcasting App: OBS

We start with understanding the third-party OBS broadcasting app. We consider both the default OBS and OBS with dynamic bitrate adaptation mode enabled (denoted as OBS-dynamic). Starting with OBS version 24 back in 2019, "dynamic bitrate mode" was added as an optional scheme to replace the default rate adaptation scheme. We use these two schemes to stream videos to *S1*, *S2*, and *S3*. For *S2*, we focus on its default normal streaming mode (§5.4.2).

5.5.1.1 Default OBS

Figure 5.6 overviews the quality of the video streaming from OBS to different services under the 18 different network conditions described above. For comparison, we also show the baseline video quality under ideal network conditions measured in §5.4. In general, the higher the average bandwidth, the higher the video quality we observe. The instantaneous network bandwidth and its variability over time also also have an impact on video quality. For example, trace D has the lowest coefficient of variation, leading to a relatively better video quality than others. *Overall, the VMAF values show considerable degradation compared to the results under no bandwidth constraints (§5.4.2): the (25^{th} percentile, median) VMAF averaged over the 18 network conditions is decreased by (64%, 31%), (62%, 28%), and (66%, 34%) compared to the baseline for S1, S2, and S3, respectively.* Besides, we still observe low quality especially in the tail of the distribution (*e.g.,* 5^{th} and 25^{th} percentiles of the VMAF distribution) even when the average network bandwidth for a network condition is high (*e.g.,* 120% of the encoding bitrate). The average stall ratio (defined in §5.3.4) across different traces is 3.5%, 1.0%, and 2.1% for *S1*, *S2*, and *S3*, respectively (not shown in the figures).

To understand the root cause for such performance degradation, we plot the evolution of network bandwidth, data sending rate, video quality, as well as the effective frame rate of a typical live video ingest session across time, as shown in Figure 5.7. This figure shows an example of how OBS performs when the upload bandwidth availability changes over time according to one trace. As shown, there are still many low-bandwidth periods even when the average bandwidth is high (120% of the encoding bitrate in this case). In high-bandwidth periods, even if the bandwidth is higher than the baseline encoding bitrate, due to the real-time nature of the live video stream, the additional bandwidth availability cannot be used to further improve the video quality. We also see a relationship among the network bandwidth, VMAF, and the effective frame rate (effective FPS, defined in §5.3.4): when the network experiences low-bandwidth periods, the effective FPS in the top ABR track (TAT) is very low (e.g., zero), leading to low VMAF values and choppy video quality. The measured low effective FPS is likely because some frames get dropped before reaching the ABR server.

To further explore the above, we instrument the OBS source code[4] to collect frame management information. We find that *the default OBS broadcasting app drops frames when the network bandwidth becomes insufficient to support the configured video encoding bitrate. We also note that the frame drop process is bursty – sequences of consecutive frames are dropped. Some bursts can be as large as 2s worth of frames, leading to poor video experience during that time*. We also examined the TAT created by different streaming servers. We find that *S1* and *S2* duplicate frames to maintain a constant high frame rate, although the effective frame rate (§5.3.4) still remains low. *S3* adopts a different strategy – it does not fill the gaps in the sequence with duplicate frames and uses discontinuous presentation times (PTS) to indicate the presence of gaps so that the player knows when to render each frame.

5.5.1.2 OBS dynamic bitrate mode

We now study the performance of this new mode under the same network conditions as in Figure 5.6. Overall, the video quality is improved compared to the default OBS rate adaptation (see Figure 5.8). However, there are some scenarios when the quality becomes worse, such as when using trace E with its average bandwidth scaled to 60% of the encoding bitrate. The stall ratio is 4.3%, 0.8%, and 2.1% for *S1*, *S2*, and *S3*, respectively.

To understand why OBS-dynamic sometimes has worse performance than the default OBS adaptation, we examine the different performance metrics across time. Figure 5.9

[4]Unlike many other broadcasting apps, OBS is open-source

shows an example run for this broadcast setting. We find that the network resource frequently becomes under-utilized: even if during many periods the bandwidth is higher than the baseline OBS encoding bitrate, the broadcasting app does not leverage the increased bandwidth to reach the baseline bitrate. The VMAF can stay low even after the network bandwidth rapidly improves from a low value, *e.g.,* even if the bandwidth is only very low at a few points, the near-zero poor VMAF lasts more than 30s.

To understand the root cause for the above behavior, we further instrument the corresponding dynamic bitrate adaptation module of OBS. Specifically, we log the encoding bitrate decision over time, as shown in the bottom subfigure of Figure 5.9. We can see that the encoding bitrate increases relatively infrequently (every $\sim$30s), and it increases very little at each step. By examining its source code, we find that although OBS-dynamic reduces its encoding bitrate whenever the network condition degrades from good, it only increases the encoding bitrate very gradually over time, when the network connection starts recovering from a poor bandwidth condition. Specifically, it reduces the encoding bitrate when the measured frame buffer occupancy is high and sets the new encoding bitrate to the buffer drain rate, which approximates the available network bandwidth. When the current encoding bitrate is lower than the baseline bitrate, the scheme only checks whether it is safe to increase encoding bitrate every 30s based on the frame buffer occupancy: if the buffer occupancy is low, it would increases the encoding bitrate. Worse, every time OBS-dynamic decides to increase the bitrate, it only increases the bitrate by a fixed amount ($\frac{maxBitrate}{10}$) and keeps probing until it reaches $maxBitrate$, which is the default encoding bitrate specified by the system/user, regardless of the current network condition. *As a result, even when the network condition already becomes good right after* *a temporal outage at around 170s, it takes OBS more than 100s to fully recover to the default encoding bitrate under high network bandwidth conditions.* In §5.6, we show how this rate adaptation logic can be improved by demonstrating our proposed rate adaptation scheme.

5.5.2 Using Browser-based Broadcasting Apps

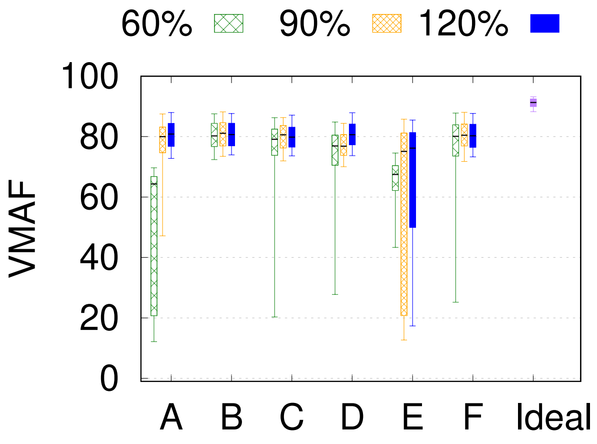
We now study the browser-based broadcasting apps, specifically the *S1-Web* and *S2-Web* broadcasting apps. *S3* does not support sending streams from a browser and requires either a third-party broadcasting app or the service's own mobile app for broadcasting.

5.5.2.1 *S1-Web*

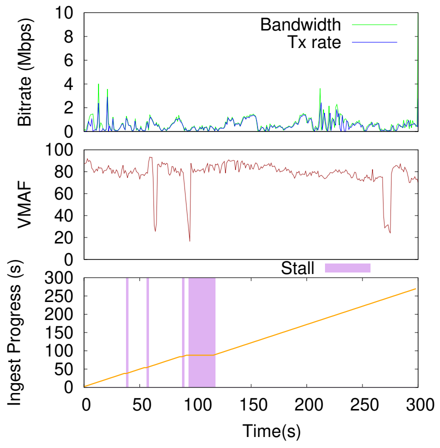
As shown in Figure 5.10(a), under many settings, *S1-Web* still has high quality video encoding despite network bandwidth constraints. However, this leads to high stalls, as shown in Figure 5.10(b), an example run under trace B scaled to 60% of the original video encoding bitrate. The average stall over different traces is 11% – the highest stall ratio across different broadcasting apps that we studied. This suggests that *S1-Web* tends to maintain a relatively high data sending rate, which can overshoot the network when the bandwidth is limited, leading to a high stall ratio despite maintaining a relatively high video encoding quality.

5.5.2.2 *S2-Web*

S2-Web has very different behavior from *S1-Web*. As shown in Figure 5.11(a), most of the scenarios exhibit low video quality, *e.g.,* even the 95th percentile VMAF values



((a)) Quality distribution



((b)) Example run

Figure 5.10: Browser-based broadcast to *S1*.

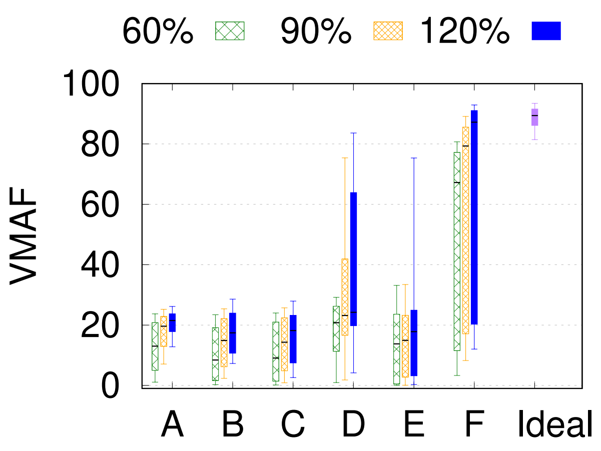
of the first six settings are all less than 30. On deeper exploration, we find that once the bandwidth increases after a period of drops, *S2-Web* still keeps sending data at a low rate instead of increasing it to the baseline data rate (∼2Mbps, §5.4.1), as exemplified by Figure 5.11(b). The average stall ratio (0.7%) is much lower compared to *S1-Web*.

5.5.3 Using Mobile-based Broadcasting Apps

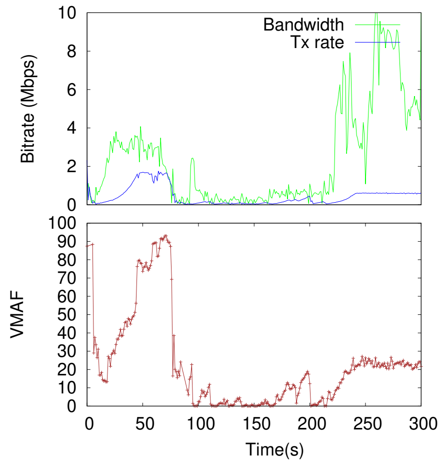
We next study the *S1* and *S3* mobile app broadcast performances. The *S2* mobile app broadcast feature requires the streaming account to have more than 1000 subscriptions, making it difficult to conduct experiments, and is not studied.

5.5.3.1 *S1-Mobile*

Figure 5.12(a) shows the video quality distribution. We find that the broadcasting app increases the data sending rate slowly when the network conditions recover, missing



((a)) Quality distribution



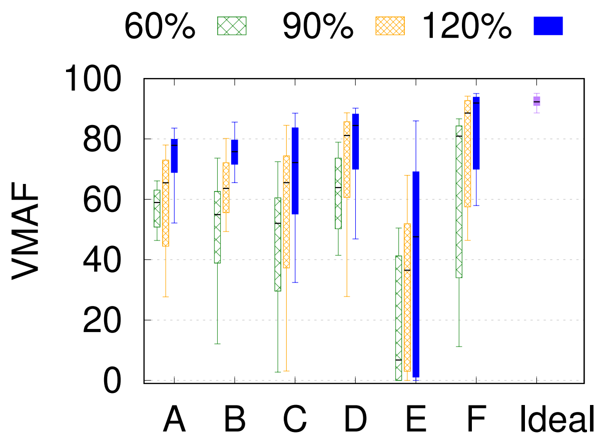
((b)) Example run

Figure 5.11: Browser-based broadcast to *S2*.

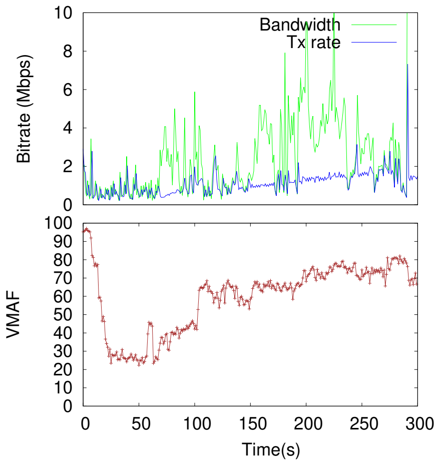
opportunities to increase the encoding bitrates and thereby the video quality, similar to the behavior of OBS-dynamic. Figure 5.12(b) shows such an example: when the network condition recovers at t $\approx$ 70s, the broadcasting app's data sending rate increases very slowly, making the VMAF recovery slow as well. The stall ratio is 4.2% on average across different settings.

5.5.3.2 *S3-Mobile*

We observe relatively high video quality, as shown in Figure 5.13(a). However, we also observe severe stalls, 6.3% on average. Figure 5.13(b) shows an example run where a stall occurs when the network bandwidth drops at t $\approx$ 180s. The video quality decrease indicates that *S3-Mobile* does adapt to the network bandwidth reduction by reducing the encoding bitrate. But the stall keeps occurring, which suggests that *S3-Mobile*'s rate adaptation is sub-optimal and can be improved (in this case, it still overshoots a little, causing

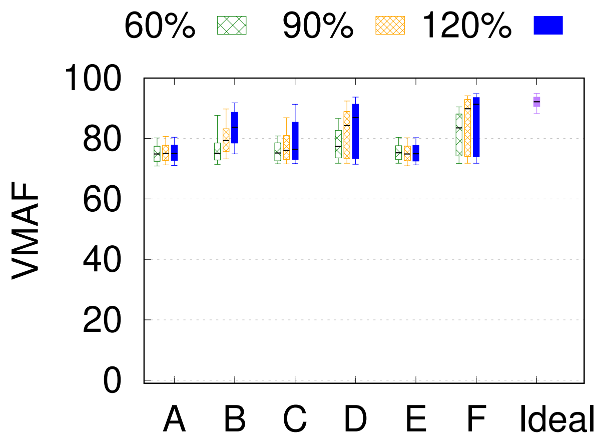


((a)) Quality distribution

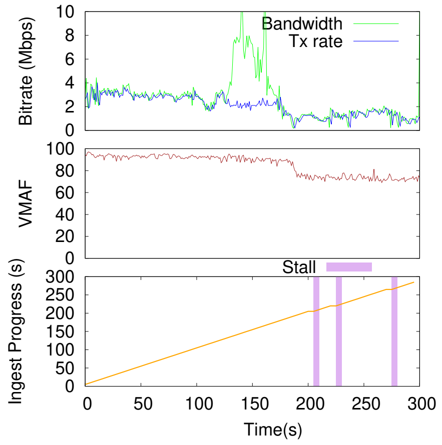


((b)) Example run

Figure 5.12: Mobile-based broadcast to *S1*.



((a)) Quality distribution



((b)) Example run

Figure 5.13: Mobile-based broadcast to *S3*.

an extra stall at t $\approx$ 275s).

To summarize, our measurements show a vast diversity of performances across different scenarios, suggesting different broadcasting apps and services choose different points in the design spaces for the live streaming ingest pipeline. This reinforces the need for tools like Livelyzer to analyze systems and understand their performance profiles as well as their strengths and weaknesses.

5.6 Improving Rate Adaptation Logic

It is vital to deliver the content on the first mile at high quality with minimal impairments, as it becomes the source reference used for the ABR track encoding and streaming delivery to end users, and its quality constrains the end-user QoE (§5.2.1). However, the limited bandwidth and variability on the first mile make this task difficult, as shown by our characterization of existing designs (§5.5). In this section, we conduct a what-if analysis to explore the potential for improving the video quality on the first mile by suitable rate adaptation to better adapt to network conditions.

For ABR streaming, rate adaptation has been studied extensively on the distribution path from the server to the client, with the latter dynamically selecting from different variants over time. On the ingest path, there is a single variant being transmitted, and any adaptation would involve dynamically changing the encoding bitrate. Conceptually, an adaptation scheme that tailors the video bitrate to the actual network bandwidth variability should be able to deliver better video QoE. However, there has been much less research on the rate adaptation for the live streaming ingest case.

§5.5 shows that while the existing broadcasting apps seem to have rate adaptation schemes built in, their performances could be further improved. For example, §5.5.1.2 indicates that when the bandwidth increases from a low level, OBS-dynamic's encoding bitrate always increases by a fixed delta at 30s intervals, irrespective of the actual network bandwidth, leading to suboptimal performance (*e.g.,* Figure 5.9).

To understand the potential improvements possible with a better adaptation logic, we design a simple proof-of-concept rate adaptation scheme that better adapts to changes in the available bandwidth. Our algorithm works as follows. In every epoch, we compute a new encoding bitrate for the next epoch based on the predicted uplink bandwidth for the next epoch. The new encoding bitrate is calculated as $min\{(1-\eta) \times BW, maxBitrate\}$ where BW is the predicted throughput for the next epoch, $\eta \in (0,1)$ is a tunable parameter introduced to control the aggressiveness of the adaptation algorithm in terms of bandwidth consumption, and $maxBitrate$ is the maximum video encoding bitrate specified by the system/user.

We implement this new adaptation logic in OBS the open-source broadcasting app. We denote it as OBS-adapt and compare it with the default OBS and OBS-dynamic adaptation schemes. We use the following bandwidth prediction approach. In our network bandwidth trace-driven experiments, we take the average of the bandwidth values for the past N epochs as the estimated network bandwidth for the next epoch for input to OBS-adapt. We empirically set η to 25%, N to 4, and the epoch to be 2s.

Figures 5.14, 5.15, and 5.16 show the video quality associated with streaming from OBS to *S1*, *S2*, and *S3*, respectively, under the 18 different network conditions described earlier for OBS. As shown, OBS-adapt noticeably improves the video quality across different network conditions. The (25^{th} percentile, median) VMAF averaged over the 18 network conditions and three services for OBS-adapt improves by (29.9, 21.7) and (14.5, 12.1) compared to OBS and OBS-dynamic, respectively. The average stall ratio is similarly low for all three schemes, with OBS-adapt being slightly lower: 2.2%, 2.4%, and

1.7%, for OBS, OBS-dynamic, and OBS-adapt, respectively.

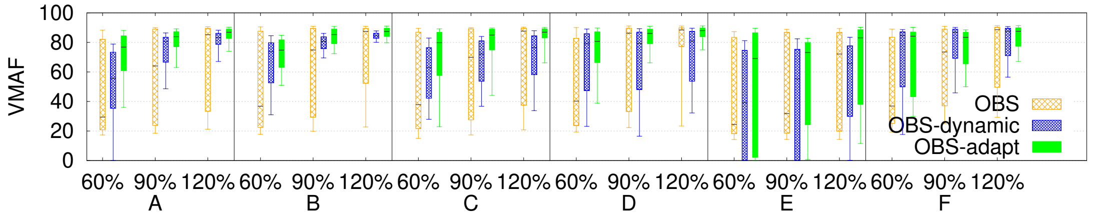


Figure 5.14: Ingest performance comparison of the default and improved OBS streaming to *S1*.

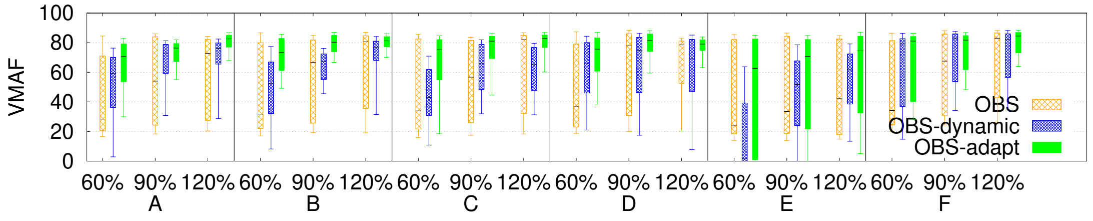


Figure 5.15: Ingest performance comparison of the default and improved OBS streaming to *S2*.

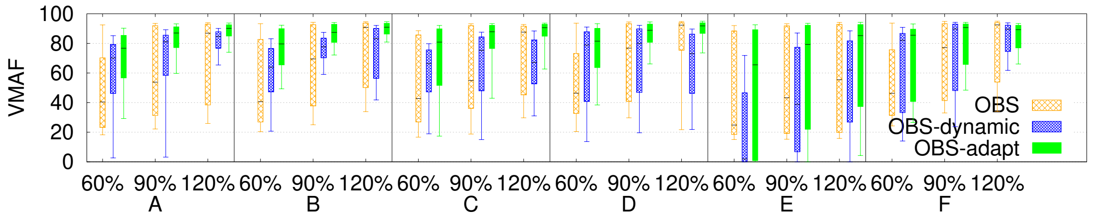


Figure 5.16: Ingest performance comparison of the default and improved OBS streaming to *S3*.

To better understand the reason for the observed performance improvements, we consider an example experimental run under one bandwidth profile. As shown in Figure 5.17, OBS-adapt's VMAF values remain high compared to OBS and OBS-dynamic. Although

OBS-dynamic also adjusts its encoding bitrate to cope with the network condition, compared to OBS-adapt, it frequently under-utilizes the available network resources. Specifically, it selects a much lower encoding bitrate than the available network bandwidth, due to its specific adaptation logic when the available network bandwidth increases from a low value (*e.g.,* at t ≈ 70s and t ≈ 155s). In contrast, OBS-adapt is able to better adapt to the same changing bandwidth conditions, leading to better video quality. The default OBS adaptation scheme drops frames instead of adjusting the encoding bitrate, leading to frequent very low quality periods.

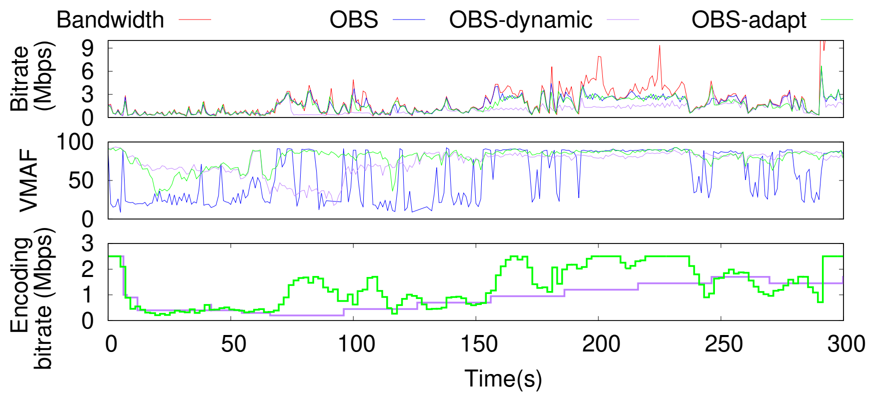


Figure 5.17: Example run showing the network bandwidth, data rate, video quality, and encoding bitrate decision evolution over time.

The above results clearly show that even a relatively straightforward adaptation strategy that fully accounts for network bandwidth changes can significantly improve the delivered quality on the ingest path compared to the current state of the art, even under challenging network conditions. Note that developing an overall optimized adaptation strategy on the ingest path is not straightforward and involves various challenges, distinct from the adaptation on the distribution path. We leave the development of a full-fledged ingest adaptation strategy to future work.

5.7 Summary

This chapter explores the first-mile ingest design and performance of live streaming. We develop Livelyzer, a generalized active measurement and black-box testing framework for analyzing this component in popular live streaming software and services under controlled settings. We use Livelyzer to characterize the ingest behavior and performance of several live streaming platforms. We identify broadcasting app rate adaptation design deficiencies that lead to poor ingest performance due to poor coordination between application policy and network performance, and propose network-aware best practice application design recommendations to improve the same.

CHAPTER VI

Harbor: Hybrid Architecture for Collaborative Vehicular Sensing

This chapter further studies the mobile application design by looking at an emerging live video analytics application, edge-assisted vehicular sensing, in a collaborative setting by having multiple vehicles sending data to a shared edge server for view merging. We develop a hybrid system architecture adaptable to different vehicle-to-infrastructure (V2I) and vehicle-to-vehicle (V2V) network connectivity by integrating a set of cross-layer optimizations.

6.1 Introduction

Autonomous vehicles leverage various on-board 3D vision sensors (*e.g.,* LiDAR and stereo cameras) to continuously sense the surrounding environment. Running computer vision algorithms [120, 92, 164] on their on-board computers to analyze these 3D video streams, autonomous vehicles are expected to have a better understanding of the physical world than human drivers and improve traffic safety and efficiency by making more informed driving decisions [17, 28, 27].

However, a single vehicle can only sense a limited range, and its view may be further restricted due to occlusion. One promising solution to overcome these limitations is collaborative sensing, where multiple vehicles share sensor data using wireless networks, given that today's vehicles are increasingly equipped with WiFi and cellular interfaces [3, 7, 1]. Collaborative sensing would thus benefit autonomous driving and Advanced Driving Assistance Systems (ADAS) by merging sensor data from different vehicles to form a complete view. Video analytics tasks (*e.g.,* drivable space detection [67] and object detection [151, 92]) can then be performed based on the merged 3D data, whose results can finally be shared among all the cars through network communications.

There exist a few studies to enable collaborative sensing [162, 128, 90], but they are not flexible enough to handle dynamic traffic and network conditions, suffering from performance issues (§6.2.2). Most base their design on a vehicle-to-vehicle (V2V) system architecture, where vehicles exchange data among each other to share sensor data streams. Despite its autonomous nature without external infrastructure support, the V2V architecture heavily relies on the wireless network established by vehicles, whose capacity needs to be shared by all vehicles and is likely limited in high-mobility scenarios. V2V was also shown to suffer from scalability issues [162]. Another limitation of V2V is its inability to leverage more powerful in-house computation resources, missing opportunities to benefit from recent technological advancements in cloud and edge computing.

A related line of solutions proposes to leverage a powerful edge or cloud server for view merging and analytics by letting each vehicle upload its sensor data, without vehicles talking to each other [162]. However, this vehicle-to-infrastructure (V2I) architecture also has limitations. V2I communication is often unreliable: even cellular networks, currently the most universal mobile network infrastructure, still have coverage issues, especially in rural areas. The performance of V2I links can be limited and fluctuates significantly in high-mobility scenarios [93, 94, 167].

In this chapter, we argue that a hybrid system architecture is needed to cope with varying wireless network connectivity and resource in high-mobility driving scenarios. Specifically, we aim to jointly harness the benefit of both V2V and V2I, to better utilize the available network resources and adapt the system modality dynamically. For example, in rural areas, one vehicle may not be able to access the server (*i.e.,* a helpee vehicle) due to the poor coverage of its ISP. The vehicle can leverage V2V to send its sensor data to another vehicle with V2I access (*i.e.,* a helper vehicle) and ask to forward the data to the server. The server then performs analytics based on the merged 3D video stream and sends the video analytics results back to each vehicle, either directly through V2I (for helper vehicles) or through V2I and V2V forwarding (for helpee vehicles).

The above mechanism seems intuitive. However, we are faced with several challenges when designing such a hybrid system (§6.2.3). How to efficiently assign helper vehicles for helpee vehicles in a strategic and balanced manner that improves the overall system performance? How to balance the quality and timeliness of video analytic tasks, especially when the links in a hybrid system are highly heterogeneous, and the analytic result can flow over the same V2V wireless medium shared with sensor data transmission? When a vehicle receives remote analytic results from the server, how to judiciously combine the local analytic and remote analytic results to take full advantage of both? We next highlight the key design decisions of our proposed system called Harbor.

• As a hybrid system architecture, Harbor keeps track of the state of each vehicle through a series of control messages and dynamically establishes and tears down V2V connections based on the system state. To maintain system connectivity and reduce interruption time under high vehicular mobility, we periodically adjust the V2V connections and employ multipath routing to increase V2V connectivity.

• Harbor strategically assigns helper vehicles to helpee vehicles by exploring different assignments in a simulation-based fashion and estimating the V2V and V2I links' performance on every helpee vehicle's end-to-end path for each possible assignment to pick the best one. The assignment decision is made dynamically based on factors that are both highly correlated with V2V and V2I link performances and easily accessible via operating systems and networking APIs (§6.3.2).

• To ensure timely delivery of video analytic results in a heterogeneous environment where different vehicles' point cloud upload time in a round can differ a lot, Harbor keeps deadline-awareness to decide when to start the frame merging process and analytic task on the server, instead of always waiting for frames from all vehicles in the round to arrive. Harbor also leverages MAC-layer prioritization to prevent analytic result messages from being delayed by the sensor data transmission over the same wireless V2V medium (§6.3.3).

• Harbor judiciously combines local and remote analytic results by examining for each area the fidelity of the local and remote point clouds quantified by the point density. It also leverages the certainty information in the analytic results to choose the side whose analytic result for a specific area has a higher certainty (§6.3.4).

We implement Harbor by integrating the above design decisions in a holistic system (§6.4), and systematically evaluate its performance by comparing it with a few baseline mechanisms (§6.5). Our key evaluation results consist of the following.

• Compared to using only V2V or V2I, Harbor reduces the end-to-end drivable space detection latency by at least 18.6% and up to 45.9%, without sacrificing the detection accuracy.

• By strategically assigning helper vehicles to helpee vehicles, Harbor reduces the tail latency by 38.8% (up to 43.8%), compared to naïve assignment schemes.

• By having server-side deadline-awareness and V2V MAC-layer prioritization, Harbor reduces the tail latency by 25.6% (up to 42.6%) compared to naïve analytics result delivery.

• By combining local and remote detection results, Harbor improves the average drivable space detection accuracy by 34.96% (16.02%) compared to using only local (remote) detection results.

Overall, Harbor is, to our knowledge, the first system that jointly uses V2V and V2I for collaborative vehicular sensing. Compared to existing collaboration schemes, Harbor offers several benefits such as more reliable data transfer and combined local and remote video analytic results. As a hybrid system, Harbor efficiently bridges V2I-disconnected vehicles by strategically pairing them with V2I-connected vehicles through V2V connectivity. Harbor also leverages server-side deadline awareness and MAC-layer prioritization to reduce analytic result delivery time.

6.2 Background and Motivation

Autonomous vehicles rely on various on-board 3D vision sensors to understand the physical world consisting of road segments, pedestrians, cyclists, other cars, *etc.*, to make correct driving decisions. For example, LiDAR (Light Detection and Ranging) [34, 39] is a major on-board vision sensor, which fires laser lights at different angles and measures how long it takes for the lights to return to the sensor after reflection from objects, based on which it calculates the distance of these objects and generates 3D point clouds to represent the surroundings. Different software modules will further process the collected point cloud data and make appropriate driving decisions.

6.2.1 Benefits of Collaborative Sensing

As mentioned in §6.1, collaborative sensing would benefit autonomous driving by merging the point clouds from different vehicles to form a complete view. Figure 6.1 shows a concrete example where the point cloud data are generated from a state-of-the-art autonomous driving simulator [31]. As shown, in a single vehicle's point cloud data (marked as blue), there are three blind spots caused by occlusion. After merging this point cloud with another one from a nearby vehicle (marked as green), two of the three blind spots can be eliminated.

6.2.2 V2V or V2I? Why Not Both?

Limitations of V2V. In a V2V system, vehicles exchange data for sharing data streams [128]. Depending on the specific V2V system design, either all the vehicles receive all the point cloud data from other vehicles and perform analytics, or only one of the

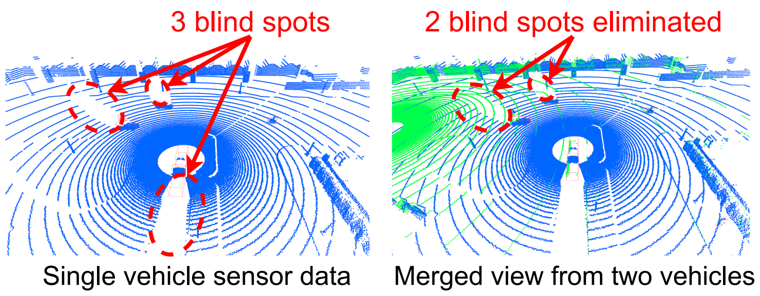


Figure 6.1: A LiDAR point cloud example showing the benefits of collaborative sensing by merging a nearby vehicle's data (green) to a single vehicle (blue).

vehicles receives data, performs analytics, and disseminates the analytics result to other vehicles. Despite its autonomous nature without external infrastructure support, V2V heavily relies on the wireless network established by vehicles, whose capacity needs to be shared by all cars and is likely limited in high mobility scenarios. While some WiFi standards like 802.11ad can theoretically achieve a bandwidth of up to a few Gbps, they suffer from low coverage (about 10 meters) that cannot accommodate constantly moving vehicles. Long-distance wireless communication technologies such as DSRC/802.11p [84] can achieve 3-27 Mbps, and current commercial products tend to achieve up to 6 Mbps, which is not enough to support a large number of vehicles transmitting their sensor data at the same time. V2V was also shown to suffer from scalability issues [162]. Another limitation of V2V is its inability to leverage more powerful cloud/edge computation resources.

Limitations of V2I. V2I systems leverage a powerful edge or cloud server for view merging and analytics by letting each vehicle upload its sensor data to the server, without vehicles talking to each other [162]. However, V2I communications are not always reliable everywhere: even cellular networks–currently the most universal mobile network infrastructure–still have coverage issues [30, 38, 40]. The performance of V2I links can

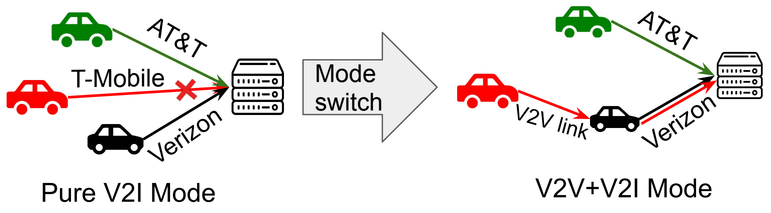


Figure 6.2: An example of joint using V2V and V2I to bridge a disconnected/poor-performing car (red).

often be limited and fluctuates a lot in high-mobility scenarios [93, 94, 167]. According to our analysis of the measurement data from recent work [108, 162], the LTE uplink bandwidth in driving scenarios is lower than 1.5 Mbps (the lowest bandwidth requirement for uploading compressed point cloud data) during 11% of the time. The percentage of low-performance periods for the whole V2I system can be further amplified as each vehicle in the system can experience such limited performance at different times.

Joint Use of V2V and V2I. The limitations of only using either V2V or V2I motivates us to consider a joint use of both. Figure 6.2 shows an example where in a rural area, the red car may not be able to access the server due to the poor coverage of its ISP. In this case, it can leverage V2V to send its sensor data to the black car with direct server access and ask to forward the data to the server through V2I. In the following sections, we refer vehicles with direct server access as helper vehicles or *helpers*, and vehicles with no V2I connectivity or experiencing temporary disruption on their V2I links as helpee vehicles or *helpees*.

6.2.3 Challenges

However, designing such a hybrid system architecture that jointly uses V2V and V2I also involves several challenges.

• As the vehicles are on the move and each may join and leave the collaboration at any time, the V2V and V2I connections need to be flexibly managed in a dynamic manner. Also, the system needs to quickly identify disconnected vehicles and pair them with other cars to forward their sensor data to minimize their network interruption time.

• When a vehicle's V2I connection gets disconnected, there could be multiple vehicles that can help forward its sensor data. Each helper may have different levels of abilities in terms of helping a specific helpee, and there could also be multiple disconnected vehicles. How to efficiently assign helpers for helpees?

• For a hybrid architecture, helpees' sensor data and detection results need to go through both V2V and V2I links. The V2V network is a shared wireless medium, where there is contention between the analytics result message delivery and the sensor data transmission. This contention could cause small result messages to be delayed by bulk sensor data transfers. How to eliminate this contention and reduce result message delivery delay without affecting the sensor data transmission throughput?

• Vehicles' point cloud data will be sent to the server for view merging and analytics, and in the meantime, vehicles also perform local analytics on their point cloud. How to combine the analytics results from both sides?

6.3 Harbor Design

We propose Harbor, a hybrid collaborative vehicular sensing system architecture that enables multiple vehicles to leverage their V2V or/and V2I network access to share their 3D point cloud sensor data. Compared to existing collaborative vehicular sensing systems, Harbor aims to reduce the 3D vehicular video analytics latency and improve analytics accuracy in an efficient and scalable manner, especially under dynamic wireless network and mobility conditions.

We next describe how to address the aforementioned key challenges of Harbor (§6.2.3): How to flexibly manage V2V and V2I connections in a dynamic manner (§6.3.1)? How to efficiently assign a helper vehicle for each helpee vehicle (§6.3.2)? How to balance the quality and timeliness of video analytics tasks (§6.3.3)? When a vehicle receives remote analytics results from the server, how to combine the local and remote analytics results (§6.3.4)?

6.3.1 Hybrid System Architecture

Figure 6.3 shows the system architecture of Harbor. Every vehicle in Harbor runs various vehicle-side modules (*e.g.,* partition, adaptive encoding, *etc.*), and the server runs server-side modules. At any specific time, vehicles fall into two disjoint sets: a helper set and a helpee set. Vehicles in the helper set have V2I connectivity, whereas vehicles in the helpee set do not. Helpers and helpees form a V2V wireless network so that helpees can leverage the V2V medium to send their point clouds to the server through helpers' V2I links, which also upload helpers' point clouds.

• **Data Plane Operations.** As shown in Figure 6.3, before uploading each point cloud

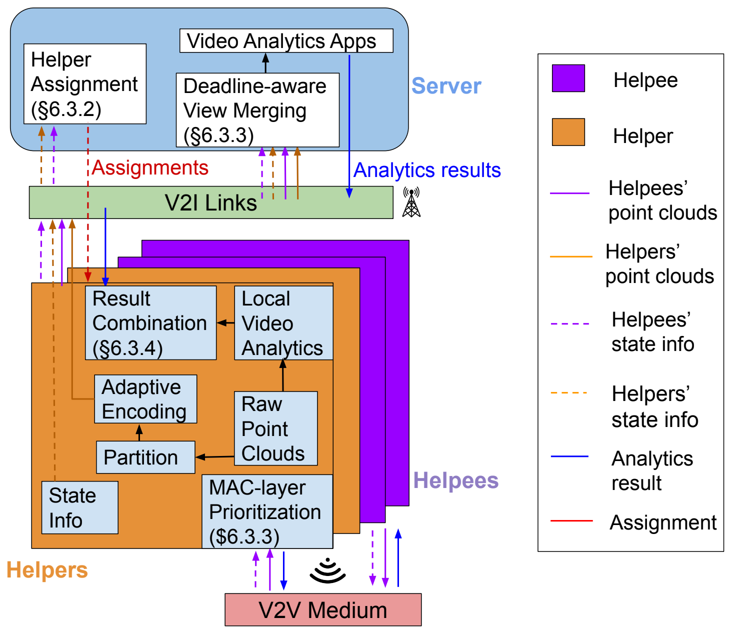


Figure 6.3: System Architecture of Harbor.

frame, partition and encoding are performed to reduce the data size for transmission. On receiving a round of point clouds from different vehicles[1], the server maps each point cloud frame collected from different positions to a unified coordinate system, harnessing the vehicles' sensor position and orientation information reported by vehicles (described shortly). It then merges these frames and performs video analytics tasks such as object detection and drivable space detection to generate analytics results. In the rest of the chapter, we use drivable space detection as an example application given its maturity and little sensitivity to merged point clouds[2]. Drivable space detection extracts the road plane from a point cloud, partitions the plane space into grids, and marks each grid as drivable

[1]Like existing work [162], we assume timestamps of point cloud capture on different vehicles are synchronized. In reality, point cloud data generation time across vehicles can have slight misalignment, which can be solved by existing techniques [77].

[2]Existing 3D object detection models, on the other hand, are designed for a single vehicle, whose performances on merged point clouds are poor.

or occupied. Finally, the detection result is sent back to vehicles. Meanwhile, each car continues to produce local detection results[3]. Upon receiving detection results from the server, vehicles then strategically fuse the received result with their local detection results (§6.3.4).

• **Control Plane Message Exchanges.** Harbor needs to assign helpers to helpees (§6.3.2) to optimize the network resource allocation for fast vehicular point cloud data uploading. A naïve way to do this is to let each helpee find a helper in a distributed manner. However, such distributed pairing lacks a holistic view of the network topology and V2I&V2V resources in the system. Harbor chooses to use a centralized method, where the server collects vehicle state information (Figure 6.3) to track vehicles' various statistics and makes assignment decisions accordingly. Dashed lines in Figure 6.3 show the control plane message exchanges in Harbor. Helpers send their state information periodically to the server, including locations, point cloud data generation timestamps, the computation time for result combination, V2I bandwidth measurements, and V2V network routing tables stored in operating systems. Helpees broadcast their state information using the wireless V2V network and rely on helpers to forward it to the server. The server then computes a best assignment using some of the above information (§6.3.2) and informs the concerned helpers. These helpers get notified that they are assigned to help some of the others, and each of them informs the helpee it is about to help to establish a V2V connection for point cloud forwarding. The server detects the helper and helpee set changes by updating the corresponding vehicle state information. Note that helpees in the system will keep probing its V2I link performance periodically and become helper again when their V2I connectivity

[3]Local detection is useful because remote detection results can sometimes arrive late and local unencoded sensor data have high quality.

resumes.

• **Grouping Vehicles to Scale Better.** The complexity of assignment computation/data forwarding can increase drastically when the number of vehicles increases. To make Harbor more scalable, we apply a geo-location-based grouping method, where region boundaries are pre-defined in the map offline, and vehicles inside each region form a group. Then the vehicles inside each group (rather than all vehicles in the system) perform collaborative sensing within the group. Harbor regions are defined as $100m \times 100m$ grids based on the typical LiDAR sensor range, and typical vehicle number within this area is small[4]. More sophisticated region partition (*e.g.,* dividing region based on road sectors) is left for future work. Such grouping makes Harbor scalable because **1)** control messages (*e.g.,* location/routing) only need to be forwarded by nodes within the group, reducing the message overhead and **2)** computing helper assignment is faster with smaller input size.

• **Adaptation to Changing Network Conditions.** Harbor leverages V2I and V2V networks, both of which can be highly dynamic. In order to cope with fluctuating wireless network bandwidths and V2V channel congestion, vehicles perform point cloud encoding adaptation. Harbor first uses existing bandwidth estimation technique for real-time point cloud streaming [162] to estimate the upload bandwidth for each vehicle. Based on the estimated upload bandwidth, each vehicle adjusts the encoding level of its next sending frame, whose bandwidth requirement after encoding is lower than and the closest to the estimation.

• **Handle high mobility of vehicles.** As vehicles can move at high speeds, Harbor needs to update helper assignments to cope with helper performance dynamics and remain ro-

[4]Based on the average traffic and vehicle speed in city areas in the U.S. [33], in a $100m$ road, the average number of vehicles is only 6.

bust communications against V2V network topology changes. To make prompt assignment updates, Harbor server periodically computes the best assignment and updates the assignment to vehicles in the system. Infrequent assignments may fail to capture vehicles' position change, producing bad assignments that hurt system performance, while doing assignments too frequently incurs high computation and communication overhead. To find a good balance, we set the period to be 200ms because 1) vehicle positions won't change too much in each period[5] and 2) the computation and communication overhead is small. To make Harbor robust against V2V network topology change, we leverage multipath routing (§6.4) to have backup routing paths for nodes in the V2V network. Having a backup routing path enables the communication between vehicles when the primary path is stale due to topology change caused by mobility.

• **V2V Fallback.** Under some extreme cases, all of the vehicles in close regions can have no V2I connectivity (*e.g.,* due to no cellular deployment in some rural areas or there is no server available). In such scenarios, Harbor chooses to fall back to a V2V-based collaboration where one vehicle acts as a server (through a leader election process) for data aggregation and detect result delivery. Harbor can resume using the hybrid V2V+V2I mode once some vehicles become helpers (*i.e.,* reconnect to the server) and continue to perform the collaboration seamlessly.

6.3.2 Strategic Helper Assignment

One crucial question we need to answer is how to efficiently assign helpers for helpees. This is challenging because different helpers have different V2I and V2V conditions when

[5]Vehicles at 110 km/h only moves around 6 m during this period.

assigned to a specific helpee. In addition, there could be multiple helpees at the same time, and one of them's helper selection decision may affect the performance of another's helpee-helper pair in such a shared V2V wireless medium. Ideally, we would like to do the assignment so that the overall performance of the system is maximized, which in collaborative sensing means the "slowest" vehicle's point cloud upload time would be minimized.

We first identify three significant factors that impact the performance of the V2V and V2I links on each helpee's end-to-end (E2E) path to the server, which consists of a helpee-to-helper path and a helper-to-server path.

• **Physical distance.** In a wireless V2V network, the physical distance between a helpee and a helper has an impact on the throughput of the helpee-to-helper path, especially when they are only one hop away from each other. When there are multiple hops between the helpee and helper, this single indicator of the V2V connection throughput becomes insufficient, and V2V network interference (described below) would be another indicator.

• **V2I network bandwidth.** The V2I bandwidth of a helpee's helper can also impact the point cloud data upload time, especially when the helpee-to-server path becomes the bottleneck of the E2E helpee-to-server path. The V2I bandwidth of a helper will also be shared by itself and one or more helpees.

• **V2V network interference.** In wireless networks, hosts can experience interference when multiple surrounding nodes are actively generating network traffic [82]. Scenarios like multi-hoping between helper and helpee nodes and multiple helpees transmitting data to the same helper will hurt the performance of a helpee-helper connection.

Symbol	Meaning
e_i	Helpee vehicle i
r_j	Helper vehicle j
$a_k = (e_i, r_j)$	A (helper, helpee) assignment pair
$A = \{a_1, a_2, ..., a_n\}$	An assignment is a set of assignment pairs
$P(a_k) = e_i \to ... \to r_j$	The network path for a_k (helpee i to helper j)
n	Total number of helpees
m	Total number of helpers
$D(e_i, r_j)$	Physical distance between helpee i and helper j
$S_{dist}(A), S_{bw}(A), S_{intf}(A)$	Distance,V2I bandwidth, V2V interference score of assignment A
$IC(v_i, X)$	Interference count produced by vertices in graph X to vertex v_i
$IC(P(a_k), X)$	The sum of interference counts for all vertices in path $P(a_k)$

Table 6.1: Notations for helper assignment algorithm.

6.3.2.1 Assignment Algorithm Design

This section discusses how Harbor optimizes its helper assignment by considering all three above impacting factors. Overall, Harbor takes a simulation-based approach by exploring and evaluating different assignments. Specifically, for each possible assignment, Harbor calculates a score based on the above factors. The assignment with the highest score would finally be selected to pair helpees with concerned helpers. Harbor assigns helpers in a way that each helpee only needs one helper[6] but one helper can be assigned to help multiple helpees. Following this rule, given a system with m helpers and n helpees, there are m^n possible assignments in total.

For an assignment A, its score is defined as Eq. 6.1 below. Assuming the total number of helpees is n, an assignment $A = \{a_1, a_2, ..., a_n\}$ contains one or multiple assignment pairs from each helpee e_i to its helper r_j (Table 6.1).

[6]Assigning multiple helpers for a helpee may sometimes be useful but is complicated to realize and out of the scope of this study.

$$Score(A) = S_{dist}(A) + S_{bw}(A) + S_{intf}(A) \tag{6.1}$$

We first calculate the distance, V2I bandwidth, and interference score of each pair and then aggregate each type of score using an aggregation function f, as shown in Eq. 6.2.

$$S_{factor}(A) = f(S_{factor}(a_1), ..., S_{factor}(a_n)) \tag{6.2}$$

While different aggregation functions can be applied, Harbor uses the harmonic mean to aggregate scores of different assignment pairs, which helps filter out assignments with low-score assignment pairs to prevent the whole collaboration system being slowed down by a single "slow" vehicle, as harmonic mean is very sensitive to small-value outliers.

We next elaborate on how to calculate the score for each individual impacting factor, including distance, V2I bandwidth, and wireless interference. Since distance, bandwidth, and interference have different units and scales, we design each score function (S_{dist}, S_{bw}, and S_{intf}) in a way that maps the corresponding factor to a value in $[0, 1]$ to give each factor/score equal importance. Table 6.1 summarizes the notations used to calculate different scores of an assignment.

• **Distance Score.** Intuitively, the wireless throughput is better when two nodes physically reside closer to each other. Therefore, the assignment should prefer a helper closer to a specific helpee to forward its data to the server. The distance score for a pair $a_i = (e_i, r_j)$ is quantified by comparing the physical distance between e_i and r_j with the longest possible distance between e_i and any other helper in the system. Specifically, for an assignment pair $a_i = (e_i, r_j)$, its distance score is calculated by

$$S_{dist}(a_i) = 1 - \frac{D(e_i, r_j)}{max\{D(e_i, r_k)\}_{k=1}^{m}} \tag{6.3}$$

From Eq. 6.3, we can see that with a smaller physical distance between e_i and r_j, the score will be higher, indicating a nearby node is preferred for potential better V2V bandwidth. Also, the score value lies in between 0 and 1, as we desire. After calculating $S_{dist}(a_i)$ for all a_i in A, the distance score for the assignment can be calculated using Eq. 6.2.

• **Bandwidth Score.** At high level, a higher V2I bandwidth (after sharing with helpees) should have a higher score as it can potentially reduce the data upload time. If a helper r_j is assigned to help one (or more) helpee(s), we first calculate its amortized V2I bandwidth $\frac{BW(r_j)}{1+N(r_j)}$ where $N(r_j)$ is the number of assigned helpees to r_j. The amortized V2I bandwidth measures the share of r_j's total V2I bandwidth for each vehicle that uses it this V2I link. We then consider a low threshold BW_{low} (*i.e.,* bandwidth lower than BW_{low} can hardly support transmitting sensor data) and a high threshold BW_{high} (*i.e.,* bandwidth that is sufficient to transmit data at high quality). $S_{bw}(a_i)$ will reach 0 if $\frac{BW(r_j)}{1+N(r_j)} < BW_{low}$, indicating select this helper will congest the network. Similarly, $S_{bw}(a_i)$ will reach 1 if $\frac{BW(r_j)}{1+N(r_j)} > BW_{high}$. When the helper's amortized V2I bandwidth is in between BW_{low} and BW_{high}, a linear fit is used to scale the bandwidth score of a connection whose V2I bandwith lies in between the low and high thresholds (Eq. 6.4). We set $BW_{low} = 1.5Mbps$ and $BW_{high} = 12Mbps$ based on the bandwidth requirement for transmitting point cloud at 10 fps with different encoding levels[7] (§6.3.4).

[7]A typical point cloud encoded with 8 (12) quantization bits is 18 (140) KB.

$$S_{bw}(a_i) = \begin{cases} 0 & \text{if } \frac{BW(r_j)}{1+N(r_j)} < BW_{low} \\ 1 & \text{if } \frac{BW(r_j)}{1+N(r_j)} > BW_{high} \\ \frac{\frac{BW(r_j)}{(1+N(r_j))} - BW_{low}}{BW_{high} - BW_{low}} & \text{otherwise} \end{cases} \tag{6.4}$$

• **Interference Score.** Overall, the interference score is designed so that a higher score is generated when the assignment's interference is smaller. To fulfill this, we propose a graph-based method to quantify the amount of interference by leveraging a simple observation: senders and receivers will interfere when they are in each others' coverage range. Specifically, we define a term called *Interference Count*, IC, to quantify the interference. At a high level, $IC(v_i, X)$ measures the total number of interfering nodes from a network topology (represented as a graph) X to a node vertex v_i (v_i can either be a helper or a helpee) and $IC(a_i, X)$ measures the interference produced by X to a pair a_i. To calculate the interference count for a_i, we sum over all the interference count for the nodes in the network path $P(a_i)$ for the pair a_i, *i.e.,* $IC(a_i, X) = \Sigma_{v_i \in P(a_i)} IC(v_i, X)$. The entire graph, including the network paths of all assignment pairs, can be constructed from the control messages received by the server, which include the routing tables on vehicles (§6.3.1). When no valid network path exists for a_i (*i.e.,* the helpee cannot connect to the helper through V2V), the total score $Score(A)$ for this assignment is set to 0. If the network paths are valid, then the interference score function maps the interference count for a_i to a value between 0 to 1. The exact formula to calculate the interference score for a_i is shown in Eq. 6.5.

$$S_{intf}(a_i) = 1 - \frac{IC(a_i, A) - IC(a_i, a_i)}{IC(a_i, G) - IC(a_i, a_i)} \tag{6.5}$$

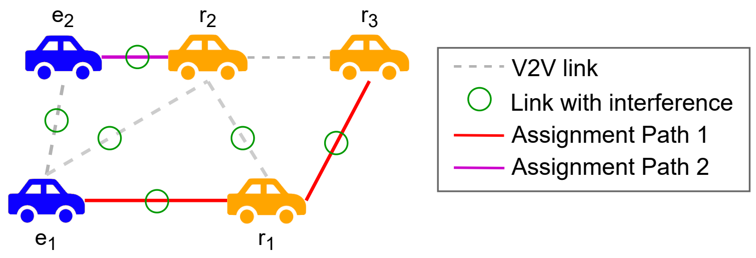


Figure 6.4: An example of graph-based interference score calculation.

In Eq. 6.5, A represents the sub-graph produced by the active transmitting and receiving nodes in the assignment. G represents the graph formed by assuming all nodes are actively generating/receiving network traffic. $IC(a_i, a_i)$ calculates the interference caused only by the nodes in the network path of a_i. Similarly, $IC(a_i, A)$ is the interference caused by the actual assignment A and $IC(a_i, G)$ is the maximum possible interference for the pair a_i.

Take Figure 6.4 as an example, there are two assignment pairs in the assignment $A = \{a_1, a_2\}$, where $a_1 = (e_1, r_3)$ and $a_2 = (e_2, r_2)$. Their network paths are $p_1 = P(a_1) = e_1 \to r_1 \to r_3$ and $p_2 = P(a_2) = e_2 \to r_2$. To calculate $IC(a_1, A)$, we sum up all interference sources for all nodes in the network path p_1, *i.e.*, $IC(a_1, A) = \Sigma_{v_i \in p_1} IC(v_i, A) = IC(e_1, A) + IC(r_1, A) + IC(r_3, A) = |\{e_2, r_1\}| + |\{e_1\}| + |\{r_1\}| = 2 + 1 + 1 = 4$. Similarly, we can calculate $IC(a_1, a_1) = |\{r_1\}| + |\{e_1\}| + |\{r_1\}| = 1 + 1 + 1 = 3$ and $IC(a_1, G) = |\{e_2, r_1, r_2\}| + |\{e_1, r_2, r_3\}| + |\{r_1, r_2\}| = 3 + 3 + 2 = 8$. Finally we have the interference score $S_{intf}(a_1) = 1 - \frac{4-3}{8-3} = 0.8$. Similarly, $IC(a_2, A)$ can be calculated and the interference score $S_{intf}(A)$ can be obtained using Eq. 6.2.

With all the three scores calculated, the score for the assignment A can be obtained from Eq. 6.1. Among all the possible assignments, Harbor uses brute force search to select the assignment with the highest score. Note that Harbor only maps the destination helpers to helpees and leaves the network path from helpees to helpers to the routing algorithm. This destination-only mapping, along with the region-based grouping (§6.3.1) mechanism, can greatly reduce the search space and make the brute force search solvable within a short amount of time[8].

6.3.3 Timely Delivery of Detection Results

As vehicles need to digest the point cloud data in real time, it is vital to deliver the remote detection result back to each individual vehicle in time. Stale detection results may not be useful because the driving environment can change drastically within a few seconds. To achieve timely delivery of the detection result, Harbor harnesses both application-level deadline awareness and MAC-layer prioritization.

6.3.3.1 Deadline Awareness

As introduced in §6.3.1, adaptive encoding helps prevent wireless links from being over-occupied. However, some vehicles' point cloud frames may still not be uploaded in time when 1) the V2I/V2V bandwidth is incapable of supporting the lowest bitrate version of the point cloud or 2) the adaptation algorithm make sub-optimal encoding decisions due to bandwidth estimation errors. If the server waits for frames from all vehicles (generated in the same round) to arrive before performing view merging (*i.e.,* an *all-or-nothing* approach), it could happen that the E2E latency would be inflated by stragglers whose frame uploading is too slow. Figure 6.5 shows an example where the upload time of the frames captured in the same round can differ by more than 0.75s across 6 vehicles due to the

[8]More sophisticated algorithm-level optimizations (*e.g.,* dynamic programming) may be able to further reduce the algorithm complexity, and such optimizations are left as future work.

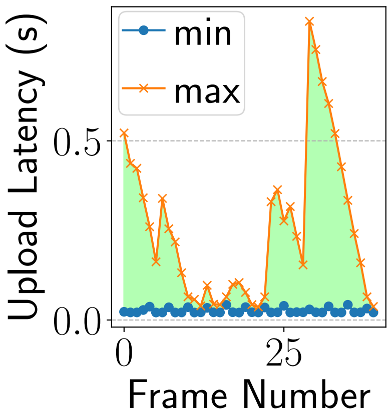


Figure 6.5: An example showing latency variation across vehicles.

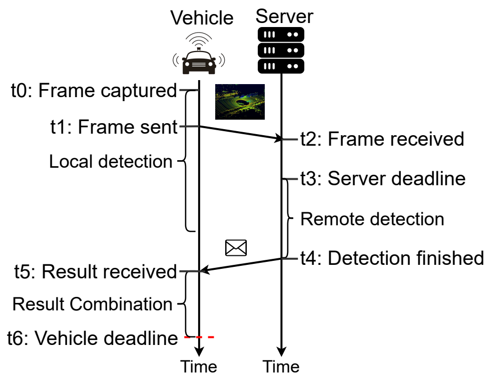


Figure 6.6: Timeline of Harbor's vehicle-side and server-side data processing.

aforementioned reasons. Harbor addresses this latency heterogeneity by incorporating a deadline by which frame merging and detection must start (even if not receiving the frames from all the vehicles in this round).

Specifically, Harbor's server determines this deadline in the following way. Each vehicle has a fixed E2E latency requirement T_{e2e} (§6.4) for each point cloud frame since the frame is captured from the sensor. Following the timelines in Figure 6.6, the server can calculate the vehicle-side deadline $t_6 = t_0 + T_{e2e}$. Based on a vehicle's local computation time T_{comb} to combine the detection result, the server further estimates $t_5 = t_6 - T_{comb}$. All the vehicle-side timestamps are embedded into control messages sent to the server (§6.3.1). The server then estimates the downlink one-way delay T_{owd} to get $t_4 = t_5 - T_{owd}$. Finally, based on its detection time T_{remote}, the server computes the deadline timestamp $t_3 = t_4 - T_{comp}$ at which it must start merging all the currently received sensor data and doing detection. Since there are multiple vehicles in the system and their time parameters can have differences, the server uses the minimal t_3 as the estimated deadline.

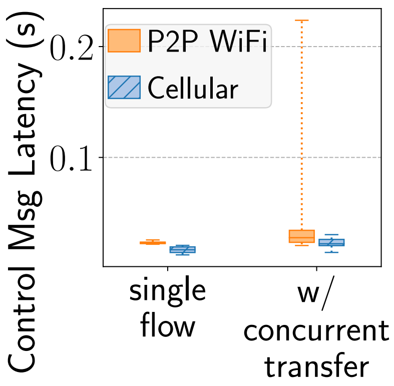


Figure 6.7: Impact of network types on the detection result delivery latency.

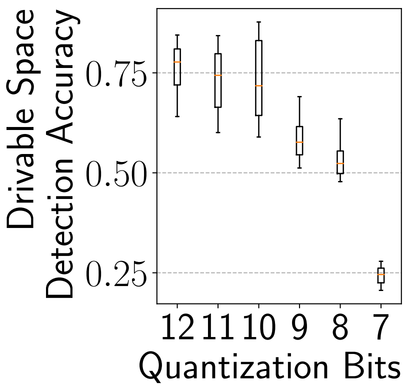


Figure 6.8: Impact of quantization bits on the detection accuracy.

6.3.3.2 MAC-layer Prioritization

Different from previous work [162], where only a cellular V2I downlink is required to transmit each result message, in a hybrid system that involves both V2V and V2I links, data messages and result messages compete for transmission from opposite directions in the same V2V wireless medium. Figure 6.7 illustrates that the latency of light-weight detection results can be largely inflated if there is concurrent data-intensive transmission in a peer-to-peer V2V wireless network. This is due to medium contention when multiple nodes in the same wireless network are trying to transmit data simultaneously. As shown, in contrast, such drastic tail latency increase is not observed on a cellular link.

To quickly delivery result messages in this challenging situation, prioritization of result messages should be done at the MAC layer. This is because the traffic coming from different hosts contend for the shared wireless medium, and the logic to access the shared medium is controlled by the MAC layer. We propose a MAC-layer message prioritization design, which prioritizes different types of messages using priority queues. For queues with higher priorities, we adjust the MAC-layer parameters (§6.4) to enable a higher probability of accessing the medium. This MAC-layer prioritization design allows the lightweight but latency-sensitive detection result messages to be prioritized over data-intensive sensor data transmission, achieving lower end-to-end latency while maintaining the network throughput for sensor data transmission.

6.3.4 Combining Local and Remote Detection Results

Upon receiving the remote detection results from the server, each vehicle needs to effectively combine the remote detection result with its local detection result. Previous work [162] does not consider how the remote detection result gets fused into the local detection result. Should the vehicles always accept the remote detection result whenever there is one?

No, they should not, and here is why. Due to the adaptive encoding of point clouds (Figure 6.8), the accuracy of detection can drop when the compression level is high (a small quantization bits value means a high compression level) under poor network conditions. Because different cars can have different network conditions, the frames arriving at the server in a round may have been encoded with different compression levels. This may cause the detection accuracy to drop in certain physical areas when merging these frames. Therefore, blindly accepting the remote detection is not enough.

To further illustrate the need for the detection results from both sides, Figure 6.9 visualizes a scenario where using neither local detection nor remote detection is sufficient. Local detection at the yellow vehicle has a large area of occlusion. For the remote detec-

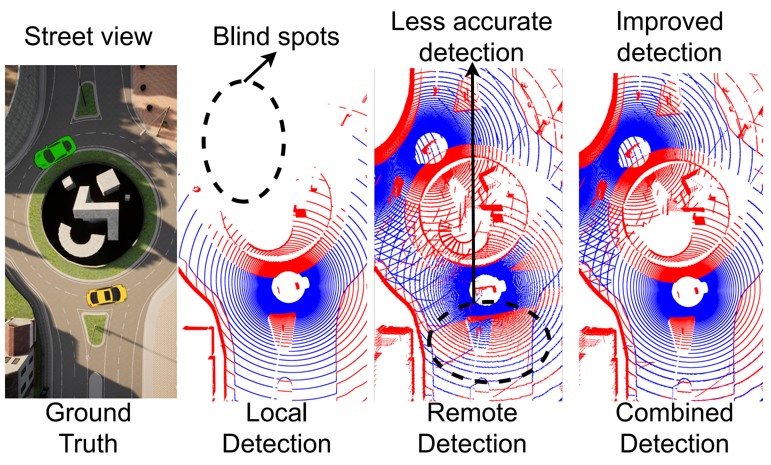


Figure 6.9: An example for local and remote detection result combination (drivable areas are marked as blue and occupied areas are marked as red).

tion, the occlusion is eliminated by fusing data from the green vehicle with its own data. The results of remote detection for some regions near the yellow vehicle, however, are less accurate. This can happen when the frame sent by the yellow car is encoded with a high compression level, causing the merged point cloud at the server to have less resolution, affecting the detection accuracy on the area nearby. If we combine the local result with the remote result by taking the local detection result close ($<$ 30m) to the yellow vehicle and accepting remote detection on other areas, both blind spots and areas with inaccurate detection are eliminated. However, this mentioned combination is manual and specific to one frame, and in Harbor we need a generic approach to automatically combine local and remote detection results to achieve better drivable space detection accuracy.

Motivated by this example, we develop a method showing that strategically combining local detection with remote detection can improve the overall accuracy for drivable space detection. Because of the physical property of LiDAR sensors, distant areas will contain fewer points and information, making the point density at distant areas lower. However, on the server side, after merging sensor data from multiple vehicles, the point density can be much larger than a single vehicle's view. Based on this observation, we propose a point density-based strategy to combine local detection with remote detection. The high-level idea of the strategy lies in two aspects: 1) use local detection for nearby areas and remote detection for faraway areas (with higher point density), 2) use remote results if the nearby areas are occluded.

Specifically, the detection results are occupancy grids where each grid is labeled as drivable/occupied/unknown (§6.3.1). For each grid, the vehicle can choose to use its local/remote detection label. When a vehicle receives the detection result from the server, it first compares the point density of local points with the remote points[9] starting from radius $d = 0m$ and finds the first ring $(d, d + 5)$ such that the remote point density in this ring is $t\%$ denser than the local points. Then it accepts the remote detection results on the grids where the distance to the vehicle is larger than the inner radius d_0 of the ring. The density threshold $t\%$ is set to 50% to ensure remote detection contains a fair amount of additional information than the local detection. Besides, for other grids in the detection results whose distances to the vehicle are smaller than d_0, they also accepts the remote detection results if that region is labeled as "unknown" in local detection, otherwise (*i.e.,* the region labeled as "known": either "drivable" or "occupied"), the vehicle keeps its own detection decision. The intuition behind this is that if a nearby region is labeled as "unknown", it is very likely to be caused by occlusions, and accepting remote detection result can gain more information about the occluded area. Our point density-based technique can be generalized to other detection apps with some modifications. Take collaborative object detection as an

[9]The server sends back point density info along with the detection result.

example, vehicles can choose to use local/remote bounding boxes if the distances between objects' bounding boxes and themselves are smaller/larger than d_0.

6.4 Implementation

• **Server side** performs helper assignment, sensor data merging, and drivable space detection. Harbor's drivable space detection uses the Random Sample Consensus (RANSAC) [64] algorithm to extract the road plane and mark the points on the road plane as "drivable" (other points are marked as objects). Then it converts the road plane into an occupancy grid of size $1m \times 1m$ and labels each grid as either "drivable", "occupied" or "unknown". To make Harbor's assignment scheme more robust to small changes in score, we introduce a stability threshold θ to decide when we want to switch to a new assignment. The server will enforce the new assignment only if its score is higher than the old score by over θ. In our evaluation, we empirically set $\theta = 0.4$.

• **Vehicle side.** V2V control message exchanges are implemented over UDP due to the broadcast requirement. V2V sensor data exchanges and V2I communications are implemented over TCP. To cope with the highly dynamic nature of vehicles, we apply multipath routing upon the Optimized Link State Routing Protocol (OLSR) [57] to provide a backup path when the primary route is not valid. We use Draco [32] to encode point cloud data, and encoding adaptation is performed by setting different quantization bits values. We apply a distance-based partition for data partition: each vehicle partition the point cloud data within a fixed threshold distance 50m to its location as each vehicle's sensor data is denser at places close to its center and contains more information. Our data partition module can be replaced with more sophisticated data partition algorithms [162] easily. MAC-layer prioritization is implemented in Linux kernel v5.8.0 [35]. Specifically, we create a higher priority queue for delivering detection results in addition to the normal MAC-layer frame processing queue (inside Linux *mac80211* module). Whenever a packet from the user-space gets processed at the MAC layer, it checks whether it is a detection result packet and puts it into the corresponding queue for transmission. We adjust the contention window ($CW_{min} = 2$) and arbitrated inter-frame spacing ($AIFS = 1$) to increase the probability of successfully transmitting the packets.

To better understand the latency requirement for collaborative sensing, we derive a 500 ms latency threshold by considering several safety-critical scenarios from previous work [162, 128]. In an unprotected left turn, the ego-vehicle has to receive the detection results within 543 ms. In the high-speed overtaking scenario, assuming the LiDAR range [39] is 80m and drivable space detection results within half LiDAR range will be useful, the end-to-end detection results need to be delivered within 559 ms. By further considering a 50 ms processing delay for other modules in the vehicle (*e.g.,* path planning), we set $T_{e2e} = 500$ ms as a hard deadline for the detection results (§6.3.3) and mark results delivered with larger than 500 ms latency as overdue.

6.5 Evaluation

We demonstrate the benefits of Harbor by experimentally evaluating its performance and comparing it with a few baseline schemes, under realistic traffic scenarios and network conditions. We quantify improvements in terms of drivable space detection accuracy and latency by conducting both trace-driven emulation and live vehicular experiments.

6.5.1 Experimental Setup and Methodology

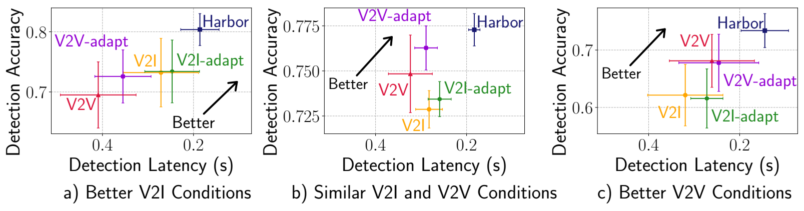


Figure 6.10: Emulation-based experimental results for E2E performance of Harbor and baseline schemes (an edge cloud is used as the server).

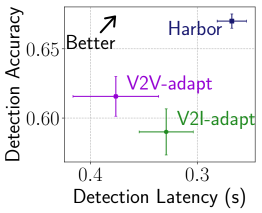


Figure 6.11: E2E live vehicular experiments.

We consider the following evaluation metrics.

• **E2E detection latency:** This is the duration between when a point cloud is captured on a vehicle to when the drivable space detection result is ready for the vehicle to use.

• **Detection accuracy:** This quantifies the difference between the actual detection result with ground truth. Specifically, we compare the predicted grid labels with ground-truth labels. Detection accuracy is defined as Eq. 6.6 where TP (True Positives) is the number of grids whose prediction and ground-truth are both drivable. Similarly, TN (True Negatives) is the number of grids whose prediction and ground-truth are both occupied or unknown.

$$Accuracy = \frac{TP + TN}{Total\_number\_of\_grids} \tag{6.6}$$

We compare Harbor with the following baseline schemes. **V2I:** each vehicle only uses V2I and cannot join the collaboration. **V2V:** all vehicles send sensor data to a single vehicle for drivable space detection. **V2I-adapt:** the above V2I scheme with Harbor's adaptive encoding (§6.3.1). **V2V-adapt:** the above V2V scheme with Harbor's adaptive encoding. In our V2V baseline implementation, one of the vehicles receives data, performs detection, and disseminates the results to other vehicles.

Trace-driven emulation. Due to the challenge of conducting real-world driving experiments in diverse scenarios, most of our experiments are trace-driven emulation. Our Traffic traces include vehicle trajectory and LiDAR point cloud data collected from a photorealistic autonomous driving simulator [31] in different driving scenarios (Table 6.2). By replaying these trajectory traces in Mininet-WiFi [36], the different mobility patterns of vehicles will help us exercise different V2V network conditions. Our V2I network traces consist of real LTE and 5G uplink network traces collected by driving vehicles [108, 162] to emulate various V2I network conditions. The experiments are conducted on a Linux machine with 16-core 2.6GHz CPU and 32GB memory, running Mininet-WiFi, which creates different numbers of wireless nodes and a server node. Each wireless node runs a Harbor vehicle instance and has two network interfaces: one 802.11g WLAN interface for the communication between vehicles, and another Ethernet interface for the connection with the server node. We use Linux *tc* to replay the V2I network traces over this Ethernet interface to emulate cellular V2I. The point cloud capture rate is configured as 10 fps in both emulation and live vehicular experiments (described below).

Live vehicular experiments. We also conduct small-scale live vehicular experiments by driving real vehicles. Each vehicle is equipped with a laptop which runs a Harbor vehicle instance. We also use a powerful server machine with a 32-core Intel Xeon CPU and 128 GB memory to run the Harbor server instance. In this set of experiments, we have two helpers and one helpee. For each laptop, we tether a smartphone to it. The smartphones have GPS and each runs an app [37] to read locations (the smartphones tethered to helpers

Setting	Value
# of vehicles	2 - 20
Speed of vehicles	0 - 70.0 km/h
Average V2I bandwidth	5.36 - 55.88 Mbps
Traffic scenes	urban roundabout, urban&rural intersections, urban&rural road segments, entrance ramp
Point cloud dataset size	157.09 GB

Table 6.2: Summary of experiment settings.

also provide cellular Internet access for V2I communications). The laptops form an ad-hoc network using their WiFi interfaces for V2V communications. In this setup, although we still replay sensor data traces due to the cost of installing LiDAR sensors, we use real networks and create real vehicle movements, which results in more realistic V2V and V2I network conditions.

6.5.2 End-to-end Performance

We start with showing the E2E performance results. Table 6.2 summarizes the configurations such as number of vehicles, mobility patterns, and network conditions for all the experiments we performed. We measure both the detection accuracy and latency for each scheme.

Trace-driven emulation. We vary the number of vehicles, mobility patterns, and V2I network conditions, which in total create 100 different settings. We further categorize these settings into 3 classes: (1) better V2I conditions, (2) similar V2I and V2V conditions, and (3) better V2V conditions. The categorization is based on average V2I bandwidth, V2V distances and the duration of helpee vehicles.

Figure 6.10 shows the E2E performance of Harbor and baseline schemes, for the aforementioned three categories of settings. In the aggregated results, we show the mean latency in the x axis and mean detection accuracy in the y axis. The error bar shows the standard deviation of each scheme under various settings. Because of the heterogeneous setting configurations like vehicle mobility and V2I network bandwidth, all baseline schemes experience large variation across settings. As shown, Harbor greatly improves the detection latency without sacrificing the detection accuracy. The mean latency is improved by 39.3% and 36.7% compared to V2V-adapt and V2I-adapt, respectively. The improvements over V2V and V2I are even higher: 45.9% and 42.0%, respectively.

In fact, Harbor even slightly improves the detection accuracy compared to the baselines. Specifically, on average, it improves the accuracy by 5.08%, 7.66%, 3.48%, and 7.76%, compared to V2V, V2I, V2V-adapt, and V2I-adapt, respectively. The reasons are two-fold. First, Harbor tries to bridge more vehicles together despite their V2I/V2V disconnections by jointly using their V2V and V2I links, leading to a more complete aggregate view. Second, Harbor reduces the E2E latency of remote detection results delivery, making the results more likely to be delivered in time hence utilized.

Field tests with real vehicles. We evaluate Harbor by running experiments driving three real vehicles on a University campus, following the live vehicular experimental setup described in §6.5.1. Figure 6.11 shows the experiment results: Harbor outperforms two baseline schemes, V2I-adapt and V2V-adapt[10], by reducing 18.6% and 29.9% of detection latency and improving 8.0% and 5.78% on detection accuracy. Note that in real-world driving tests, we have observed that the RTT between commercial cellular network to a server is 97.4 ± 1.49ms, which is much longer than a real vehicle-to-edge communication

[10]We didn't include the results for V2I and V2V as our emulation results indicate that they already perform worse than V2I-adapt and V2V-adapt

latency (about 20ms [163]). Compared to emulation experiments, Harbor's improvements on latency are a bit lower due to higher base V2I communication latency.

Case study. Figure 6.12 shows an example case demonstrating how Harbor adapts to V2I disconnections and network bandwidth changes. In this case, two vehicles that initially have V2I connections experience bad V2I conditions and become helpees at different times, and there is another helper with a consistent V2I connection. The first vehicle disconnects from the server at t $\approx$ 12s, and the second one disconnects at t $\approx$ 22s. Both vehicles only experience a latency spike that lasts for a short duration ($<$0.25s), and quickly recover from the disconnection by pairing with the helper. For example, when helpee 0 switches to using the V2V network, its encoding bitrate decreases as the V2V bandwidth is lower than its previous V2I bandwidth. Similarly, the helper also adapts to the network condition change by reducing its encoding bitrate, as it now shares part of its uplink bandwidth to helpee 0. As a result, both the helper and helpees keep their latency close to where both had V2I connectivity. We also measure the disruption to the collaborative application by analyzing how fast Harbor can resume low latency transmission if a vehicle loses its V2I connectivity. In this case, the latency for consecutive frame transmission resumes to less than 100 ms within 0.4s, which only affects the next three frames' detection latency on a 10fps rate.

Scaling to large-scale deployment. We evaluate how Harbor scales to environments with a large number of vehicles. We proportionally increase the number of helpees with the number of total vehicles in the system to keep a consistent disconnection ratio. Figure 6.13 shows the performance of Harbor changes with an increasing number of vehicles. Harbor outperforms V2V-adapt and V2I-adapt schemes by 38.5%, 54.48% in average detection

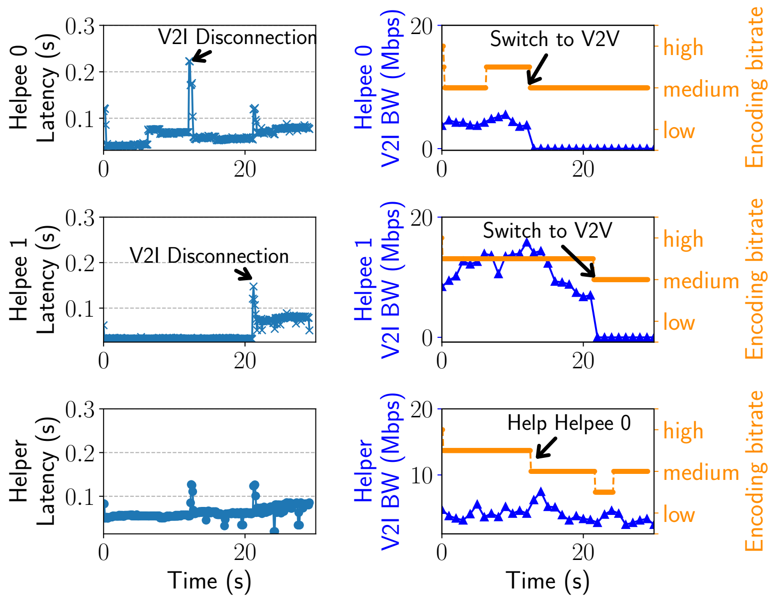


Figure 6.12: Case study experiment.

latency, and 10.34%, 16.11% in detection accuracy respectively. We observe that applying a V2I-based scheme is indeed more scalable than a pure V2V scheme, as shown before [162]. Harbor achieves better scalability compared to V2I by using the V2V medium for helpees and applying region-based grouping to reduce network message overhead. Note that for other baselines, the detection accuracy degrades with more vehicles. This is because the scheme is not scalable enough to support in-time delivery of detection results, and fewer vehicles can benefit from collaboration. However, as shown in Figure 6.13, Harbor consistently improves the detection accuracy with more vehicles participating in the collaboration.

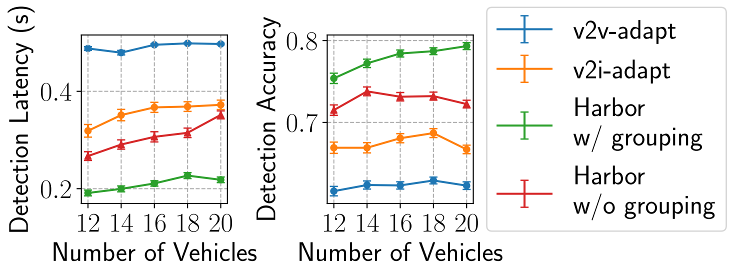


Figure 6.13: Benefits of Harbor's region-based grouping.

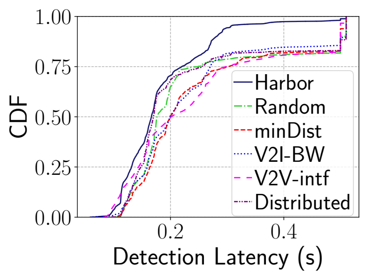


Figure 6.14: Comparison of Harbor's assignment strategy and other assignment strategies.

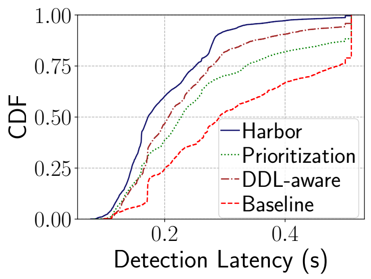


Figure 6.15: Benefits of Harbor's detection result delivery strategy.

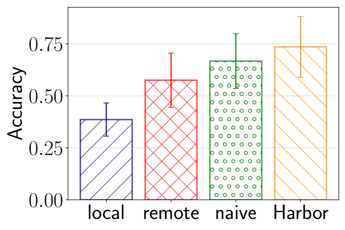


Figure 6.16: Benefits of Harbor's combination of local and remote detection results.

6.5.3 Strategic Helper Assignment

In this section, we use trace-driven emulation to examine the benefits of one of Harbor's key design decisions: strategically assign a helper vehicle for each helpee vehicle. To this end, we randomly sampled 20% of the tested settings. To make a fair comparison, we enable all other optimization features in Harbor and change only the assignment scheduling logic to have five different baseline assignment schemes.

We compare Harbor's assignment scheme with the following baselines: (1) **Random:** randomly assign helpers to helpees; (2) **Min-distance:** assign helpers by minimizing the total distance between helper-helpee pairs; (3) **V2I-BW:** assign helpers by selecting the helpers with higher V2I bandwidths; (4) **V2V-interference:** assign helpers to helpees by maximizing the interference score on the V2V wireless network; (5) **Distributed:** each helpee broadcasts a "seek-for-helper" message in the V2V network, which will be responded by received helpers, and selects the helper whose response is delivered first. In this scheme, vehicles can avoid transmitting some control message (*e.g.,* routing).

Figure 6.14 shows the benefits of Harbor's strategic helper assignment leveraging different information sources, where it improves the detection latency over all other baseline assignment schemes. In fact, Harbor's strategic assignment benefits mostly in reducing tail latency. The 90^{th} and 95^{th} percentile latency has improved by 43.8% and 38.8%, respectively. As the baseline assignment algorithms often optimize a single factor, they cannot perform well in all settings. For instance, although the "Distributed" scheme avoids some overhead by transmitting fewer control messages, its performance is still lags Harbor due to the lack of a holistic view of the network resources. This reinforces the need for a centralized method to manage the resources (§6.3.1) by mapping helpees to helpers to realize better performance. By considering all different factors that affect V2I and V2V data transmission, Harbor becomes more robust under various driving and network conditions, thus improving the tail latency greatly. In addition to the improvement of the tail latency, Harbor also achieves a slightly higher detection accuracy than other assignment schemes.

6.5.4 Timely Detection Result Delivery

To showcase the benefits of Harbor's fast detection result delivery strategy, we experiment under heterogeneous vehicular network settings. We select the 15% of our tested settings and compare Harbor's strategy with three baseline strategies: (1) baseline, which is Harbor without the entire fast detection result delivery design (§6.3.3), (2) prioritization, which adds detection result prioritization at the MAC layer to the baseline. (3) deadline-aware (DDL-aware), which adds server-side deadline-awareness in frame merging to the baseline. In our micro-benchmark, we include settings to cover heterogeneous network conditions in two aspects: (1) V2I bandwidth difference across vehicles and (2) the number of simultaneous helpees that use V2V medium.

Figure 6.15 shows the benefits of Harbor's decisions in fast detection result delivery. As shown, both MAC-layer result message prioritization and server-side deadline awareness reduce the average detection latency by over 19.0%. Harbor achieves a higher latency improvement of 38.33% by leveraging both techniques. Harbor also improves 90^{th} percentile latency by 25.6% and 42.6%, compared to prioritization and DDL-aware. DDL-aware slightly outperforms prioritization in most experiments because it optimizes result delivery for all participant vehicles, while MAC-layer message prioritization only reduces latency for helpees. Harbor also achieves the highest detection accuracy among these four schemes.

6.5.5 Detection Results Combination

We compare Harbor's judicious combination of local and remote detection results with three baselines: 1) using only local detection, 2) using remote detection results whenever possible, and 3) a naïve combination method, which uses local results if the distance to an area is within 50% of the LiDAR range and uses remote results otherwise.

Figure 6.16 shows the benefits of Harbor's point density-based combination of local and remote detection results (§6.3.4). As expected, remote detection aggregates sensor data from multiple vehicles improves the detection accuracy by eliminating blind spots for single vehicle sensor data. And combining local and remote data can further improve the detection accuracy by leveraging high accuracy detection on close areas with high-quality data in the local detection. Moreover, by further strategically aggregating local and remote detection, Harbor can consistently achieve better drivable space detection accuracy with an average increase of 16.02% compared to remote-only detection and improves 6.87% over the naïve combination method.

6.6 Summary

We propose Harbor, a hybrid system architecture for collaborative vehicular sensing. The design of Harbor leverages a cross-layer assignment algorithm to allocate vehicles with V2I connectivity to other vehicles without it. Harbor also optimizes detection result delivery latency with app-layer deadline awareness and MAC-layer message prioritization and uses strategic result combination to improve detection accuracy. Evaluation results show that Harbor reduces the drivable space detection latency by up to 45.9%, without sacrificing the detection accuracy.

CHAPTER VII

Conclusion and Future Work

The mobile Internet has rapidly evolved over the past decade, with an ever-growing diversity of end systems and applications. However, this increased complexity of different components and protocol layers makes it more challenging to achieve high network utilization and meet the diverse QoE requirements for mobile applications. As a result, despite the richness of diverse network resources, the performance of today's mobile applications still falls behind expectations. My dissertation is dedicated to addressing this challenge. We demonstrate that with a better understanding of the various entities and different layers of the increasingly complex mobile Internet, we can identify unique performance problems and leverage such knowledge to develop network transport protocols with cross-device awareness and application adaptation strategies with cross-layer considerations for better mobile app performance.

Specifically, to understand and solve the new challenges brought by diverse end systems, we explore wearable networking where a mobile end host does not always directly access the Internet with end-to-end TCP/IP and oftentimes relies on a paired mobile device as the gateway. We first identify the limitations of existing networking stacks on wearable systems by conducting the first in-depth investigation of the networking performance of Wear OS, one of the most popular OSes for smartwatches and potentially other wearable systems. Our measurement study reveals that the existing Wear OS suffers from serious performance issues regarding key aspects that distinguish wearable networking from smartphone networking. To mitigate the identified performance impairment, we design, implement, and evaluate several readily deployable transport management solutions and demonstrate that wearable networking performance can be improved with a better understanding of the end system diversity and heterogeneous wireless links.

To leverage the opportunity for multi-device provision and jointly use the network interfaces of nearby mobile devices, we propose MPBond, an efficient system allowing multiple personal mobile devices to collaboratively fetch content from the Internet. Inspired by the success of MPTCP, MPBond applies the concept of distributed multipath transport, where multiple subflows can traverse different devices. We develop device/connection management schemes, a buffering strategy, a packet scheduling algorithm, and a policy framework tailored to MPBond 's architecture to efficiently utilize the heterogeneous network resources. We evaluate MPBond using real-world mobile devices, networks, and applications and demonstrate that a cross-device transport protocol considering the interaction of multiple mobile devices and heterogeneous wireless links improves the mobile application performance.

To optimize the application design to adapt to varying network conditions to improve mobile application performance, we explore the emerging live streaming application, which is both bandwidth-intensive and latency-sensitive. We aim to understand how commercial live streaming broadcast and distribution platforms such as Youtube and Facebook perform over mobile networks. Specifically, we look at the upstream ingest path from the broadcasting app to the video server, which is responsible for capturing the video content with a camera, encoding it, and transmitting it over cellular or Wi-Fi uplinks. Delivering high-quality video over mobile uplinks in real time is challenging, and there exists little related research. To this end, we develop Livelyzer, a tool to analyze the first-mile ingest path of commercial live streaming, and provide best-practice suggestions to developers. Our study demonstrates that existing live video upload apps incur poor coordination between the application decisions and network conditions, and schemes that better adapt real-time encoding rates to network bandwidths can improve QoE.

To further explore the optimization spaces for mobile application design, we look at another emerging live video analytics application: collaborative vehicular sensing. We design a hybrid system architecture that leverages both the direct communication between vehicles (V2V) and direct communication between vehicles and the remote server (V2I) to better utilize the available network and compute resources. We develop methods for the dynamic establishment of different V2V and V2I channels to better adapt to heterogeneous network resources, and an algorithm that efficiently relays sensor data considering the available V2V and V2I resources. We demonstrate that a flexible and adaptive sensor data sharing application architecture considering the underlying wireless networking protocols improves collaboration sensing performance for autonomous driving.

7.1 Limitations and Future Work

We have learned that cross-device network transport and cross-layer application adaptation improve mobile app performance. While we only touched a few networking and application settings in this dissertation, we encourage future studies to further explore this guideline with new types of apps, networks, and devices on the increasingly complex mobile Internet.

• *New types of apps.* Different apps have different traffic patterns and QoE requirements. As new types of apps such as AR/VR are getting popular, it would be good to have a deep understanding of their performance over mobile networks and develop transport- and app-layer optimizations to improve QoE, especially in multi-user settings. These apps are also computation-intensive, calling for joint considerations of computation and networking for performance optimization. Besides, there are also UDP-based mobile applications such as video conferencing, where reacting to packet losses in the transport- and app-layer is a major challenge, especially over multiple network paths.

• *New types of networks.* New network technologies such as 5G cellular and 802.11ad wireless have recently entered the world. Existing studies have shown the challenges in applying existing transport- and application-layer networking solutions to such millimeter-wave networks [109, 166]. Redesigning these higher-layer protocols for these networks would be a future direction, especially in multipath and multi-device settings.

• *New types of devices.* In this dissertation, we explored a wide range of devices including smartphones, smartwatches, and connected vehicles. However, these are just a few drops in the ocean of IoT. Other devices such as smart glasses, voice controllers, smart thermostats, and drones need mobile network access for different use cases. Understanding their unique networking characteristics is beneficial to the design of suitable transport protocols and application adaptation strategies.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Tesla Model S software release notes v5.8. https://www.tesla.com/sites/default/files/blog_attachments/software_update_5.8.pdf, 2013.

[2] People for Whom One Cellphone Isn't Enough. https://www.wsj.com/articles/people-who-use-two-cellphones-1396393393, 2014.

[3] ATT and Audi to Wirelessly Connect all 2016 Model Year Vehicles. https://about.att.com/story/att_and_audi_to_wirelessly_connect_all_2016_model_year_vehicles.html, 2015.

[4] Doing the Two-Smartphone Shuffle. https://geekdad.com/2015/02/two-smartphone-shuffle/, 2015.

[5] How Often Does the Average American Replace His or Her Smartphone? https://www.fool.com/investing/general/2015/07/15/how-often-does-the-average-american-replace-his-or.aspx, 2015.

[6] 3 Reasons Why You Should Own A Second Cell Phone. https://www.forbes.com/sites/forbesmarketplace/2016/03/17/3-reasons-why-you-should-own-a-second-cell-phone/#7e3b43595c4c, 2016.

[7] Mercedes-Benz mbrace. https://www.mbusa.com/vcm/MB/DigitalAssets/pdfmb/mbraceservicebrochures/1527_MBfactsheet_0814_KH_v2.pdf, 2016.

[8] MPTCP v0.91 Release. http://multipath-tcp.org/pmwiki.php?n=Main.Release91, 2016.

[9] 8 Frugal Reasons to Have Two Phones. https://www.thefrugalgene.com/frugal-phones/, 2017.

[10] Cicret Bracelet. https://cicret.com/wordpress/, 2017.

[11] "Multiple phone personality" is trending. https://hackernoon.com/multiple-phone-personality-is-trending-2c1670bd7367, 2017.

[12] Telegram for Android Wear 2.0. https://telegram.org/blog/android-wear-2-0, 2017.

[13] The netfilter.org project. https://www.netfilter.org/, 2017.

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

[15] ZenWatch Remote Camera. https://play.google.com/store/apps/details?id=com.asus.rcamera2, 2017.

[16] Cisco Visual Networking Index: Forecast and Trends, 2017-2022 White Paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html, 2018.

[17] How autonomous vehicles could save over 350K lives in the US and millions worldwide. https://zdnet.com/article/how-autonomous-vehicles-could-save-over-350k-lives-in-the-us-and-millions-worldwide/, 2018.

[18] Market share of smart wristwear shipments worldwide by operating system from 2015 to 2020. https://www.statista.com/statistics/466563/share-of-smart-wristwear-shipments-by-operating-system-worldwide/, 2018.

[19] Monsoon Power Monitor. https://www.msoon.com/online-store, 2018.

[20] Smartwatch Market Size, Share, Growth, Industry Report, 2018–2023. https://www.psmarketresearch.com/market-analysis/smartwatch-market, 2018.

[21] Specifications. The building blocks of all Bluetooth devices. https://www.bluetooth.com/specifications, 2018.

[22] Using Your Old Smartphone as a Mobile Hotspot. https://www.hellotech.com/blog/using-old-smartphone-as-mobile-hotspot/, 2018.

[23] Cisco visual networking index: global mobile data traffic forecast update, 2017–2022. 2019.

[24] *MPWear* github repository. https://github.com/XiaoShawnZhu/MPWear, 2019.

[25] *WearMan* github repository. https://github.com/XiaoShawnZhu/WearMan, 2019.

[26] Exoplayer. https://google.github.io/ExoPlayer, 2019.

[27] Study shows autonomous vehicles can help improve traffic flow. https://phys.org/news/2018-02-autonomous-vehicles-traffic.html, 2019.

[28] Autonomous vehicles for safety. https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety, 2020.

[29] *MPBond* github repository. https://github.com/XiaoShawnZhu/MPBond, 2020.

[30] ATT Maps - Wireless Coverage. https://www.att.com/maps/wireless-coverage.html, 2021.

[31] Carla: Open-source simulator for autonomous driving research. https://carla.org/, 2021.

[32] Draco 3D Graphics Compression. https://google.github.io/draco/, 2021.

[33] INRIX Global Traffic Scorecard - Last-Mile Speed. https://inrix.com/scorecard/, 2021.

[34] Lidar — Wikipedia. https://en.wikipedia.org/wiki/Lidar, 2021.

[35] Linux kernel v5.8. https://github.com/torvalds/linux/tree/v5.8, 2021.

[36] Mininet-wifi: Emulator for software-defined wireless networks. https://github.com/intrig-unicamp/mininet-wifi, 2021.

[37] Share GPS. http://jillybunch.com/sharegps/, 2021.

[38] T-Mobile Coverage Map. https://www.t-mobile.com/coverage/coverage-map/, 2021.

[39] Velodyne LiDAR HDL-32E sensor. https://velodynelidar.com/products/hdl-32e/, 2021.

[40] Verizon Coverage Map. https://www.verizon.com/coverage-map/, 2021.

[41] I.-T. P. 910. Subjective Video Quality Assessment Methods for Multimedia Applications, 2008.

[42] A. Abbas. Introducing VMAF percentiles for video quality measurements, 2020.

[43] Adobe. Adobe's Real Time Messaging Protocol. `https://www.adobe.com/content/dam/acom/en/devnet/rtmp/pdf/rtmp_specification_1.0.pdf`, 2012.

[44] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: a system solution for sharing i/o between mobile systems. In *MobiSys*. ACM, 2014.

[45] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning wireless network power management. *Wireless Networks*, 11(4):451–469, 2005.

[46] G. Ananthanarayanan, V. N. Padmanabhan, L. Ravindranath, and C. A. Thekkath. Combine: leveraging the power of wireless peers through collaborative downloading. In *MobiSys*. ACM, 2007.

[47] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for internet hosts. *ACM SIGCOMM Computer Communication Review*, 29(4):175–187, 1999.

[48] R. Braden. Requirements for internet hosts-communication layers. 1989.

[49] L. S. Brakmo and L. L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[50] J. Chauhan, S. Seneviratne, M. A. Kaafar, A. Mahanti, and A. Seneviratne. Characterization of early smartwatch apps. In *PerCom Workshops*. IEEE, 2016.

[51] D. Chen, K. G. Shin, Y. Jiang, and K.-H. Kim. Locating and tracking ble beacons with smartphones. In *CoNEXT*. ACM, 2017.

[52] Q. Chen, S. Tang, Q. Yang, and S. Fu. Cooper: Cooperative perception for connected autonomous vehicles based on 3d point clouds. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 514–524. IEEE, 2019.

[53] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone energy drain in the wild: Analysis and implications. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):151–164, 2015.

[54] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone energy drain in the wild: Analysis and implications. volume 43, pages 151–164. ACM New York, NY, USA, 2015.

[55] X. Chen, T. Grossman, D. Wigdor, and G. Fitzmaurice. Duet: Exploring joint interactions on a smart phone and a smart watch. In *ACM CHI*, 2014.

[56] 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 *WearSys workshop*, pages 33–38. ACM, 2015.

[57] T. Clausen, P. Jacquet, C. Adjih, A. Laouiti, P. Minet, P. Muhlethaler, A. Qayyum, and L. Viennot. Optimized link state routing protocol (olsr). 2003.

[58] G. Developers. Real-time communication for the web. `https://webrtc.org`, 2020.

[59] Facebook. FB: How to Go Live on Mobile? `https://www.facebook.com/business/help/1884140525218868`, 2020.

[60] Facebook. How do I go live from my Facebook Page? `https://www.facebook.com/help/1916203341847533`, 2020.

[61] Facebook. How do I set up streaming software to work with Facebook? `https://www.facebook.com/help/755943624557739`, 2020.

[62] Facebook. What are the video format guidelines for live streaming on Facebook? `https://www.facebook.com/help/1534561009906955`, 2020.

[63] FFmpeg. A complete, cross-platform solution to record, convert and stream audio and video, 2021.

[64] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[65] I. O. for Standardization. Iso/iec 23000-19:2020 information technology — multimedia application format (mpeg-a) — part 19: Common media application format (cmaf) for segmented media. `https://www.iso.org/standard/79106.html`, 2020.

[66] D. I. Forum. DASH-IF Live Media Ingest Protocol Technical Specification, 26 February 2021. `https://dashif-documents.azurewebsites.net/Ingest/master/DASH-IF-Ingest.html`, 2021.

[67] A. Frickenstein, M.-R. Vemparala, J. Mayr, N.-S. Nagaraja, C. Unger, F. Tombari, and W. Stechele. Binary dad-net: Binarized driveable area detection network for autonomous driving. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2295–2301. IEEE, 2020.

[68] R. Friedman, A. Kogan, and Y. Krivolapov. On power and throughput tradeoffs of wifi and bluetooth in smartphones. *IEEE Transactions on Mobile Computing*, 12(7):1363–1376, 2013.

[69] 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.

[70] M. Grüner, M. Licciardello, and A. Singla. Reconstructing proprietary video streaming algorithms. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020.

[71] Y. Guo, F. Qian, Q. A. Chen, Z. M. Mao, and S. Sen. Understanding on-device bufferbloat for cellular upload. In *IMC*. ACM, 2016.

[72] 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*, pages 141–153. ACM, 2017.

[73] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *MobiSys*. ACM, 2014.

[74] O. L. Haimson and J. C. Tang. What makes live events engaging on facebook live, periscope, and snapchat. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, pages 48–60, 2017.

[75] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan. Mp-dash: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 129–143. ACM, 2016.

[76] D. T. G. Hao. Mixer's Faster Than Light streaming protocol explained. `https://dotesports.com/streaming/news/mixers-faster-than-light-streaming-protocol-explained`, 2019.

[77] Y. He, L. Ma, Z. Jiang, Y. Tang, and G. Xing. Vi-eye: semantic-based 3d point cloud registration for infrastructure-assisted autonomous driving. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 573–586, 2021.

[78] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale. Weardrive: Fast and energy-efficient storage for wearables. In *USENIX ATC*, 2015.

[79] 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 *SIGCOMM*. ACM, 2013.

[80] C. V. N. Index. Forecast and methodology 2017–2022. *Cisco: San Jose, CA, USA*, 2019.

[81] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling Bufferbloat in 3G/4G Networks. In *IMC*. ACM, 2012.

[82] A. Kashyap, S. Ganguly, and S. R. Das. A measurement-based approach to modeling link capacity in 802.11-based wireless networks. In *Proceedings of the 13th annual ACM international conference on Mobile computing and networking*, pages 242–253, 2007.

[83] L. Keller, A. Le, B. Cici, H. Seferoglu, C. Fragouli, and A. Markopoulou. Microcast: Cooperative video streaming on smartphones. In *MobiSys*. ACM, 2012.

[84] J. B. Kenney. Dedicated short-range communications (dsrc) standards in the united states. *Proceedings of the IEEE*, 99(7):1162–1182, 2011.

[85] K.-H. Kim and K. G. Shin. Improving tcp performance over wireless networks with collaborative multi-homed mobile hosts. In *MobiSys*. ACM, 2005.

[86] H. Kolamunna, I. Leontiadis, D. Perino, S. Seneviratne, K. Thilakarathna, and A. Seneviratne. A first look at sim-enabled wearables in the wild. In *Proceedings of the Internet Measurement Conference 2018*, pages 77–83, 2018.

[87] H. Kolamunna, I. Leontiadis, D. Perino, S. Seneviratne, K. Thilakarathna, and A. Seneviratne. A first look at sim-enabled wearables in the wild. In *IMC*. ACM, 2018.

[88] R. Krashinsky and H. Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 119–130. ACM, 2002.

[89] E. Krings. What is RTMPS and Why is it Important to Secure Streaming?, 2021.

[90] S. Kumar, L. Shi, N. Ahmed, S. Gil, D. Katabi, and D. Rus. Carspeak: a content-centric network for autonomous driving. *ACM SIGCOMM Computer Communication Review*, 42(4):259–270, 2012.

[91] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, and N. Dai. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. In *Proceedings of MobiCom 2017*, pages 409–421. ACM, 2017.

[92] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12697–12705, 2019.

[93] H. Lee, J. Flinn, and B. Tonshal. Raven: Improving interactive latency for the connected car. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 557–572. ACM, 2018.

[94] L. Li, K. Xu, T. Li, K. Zheng, C. Peng, D. Wang, X. Wang, M. Shen, and R. Mijumbi. A measurement study on multi-path tcp with multiple cellular carriers on high speed rails. In *SIGCOMM*, pages 161–175. ACM, 2018.

[95] Z. Li, A. Aaron, I. Katsavounidis, A. Moorthy, and M. Manohara. Toward a practical perceptual video quality metric, 2016.

[96] Z. Li, C. Bampis, J. Novak, A. Aaron, K. Swanson, A. Moorthy, and J. D. Cock. Vmaf: The journey continues, 2018.

[97] 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 13th International Conference on emerging Networking EXperiments and Technologies*, pages 147–159. ACM, 2017.

[98] J. Y. Lin, T.-J. Liu, E. C.-H. Wu, and C.-C. J. Kuo. A fusion-based video quality assessment (fvqa) index. In *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2014 Asia-Pacific*, pages 1–5. IEEE, 2014.

[99] R. Liu, L. Jiang, N. Jiang, and F. X. Lin. Anatomizing System Activities on Interactive Wearable Devices. In *APSys*, 2015.

[100] 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, 2016.

[101] S. Liu, T. Başar, and R. Srikant. Tcp-illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation*, 65(6-7):417–440, 2008.

[102] T.-J. Liu, Y.-C. Lin, W. Lin, and C.-C. J. Kuo. Visual quality assessment: recent developments, coding applications and future trends. *APSIPA Transactions on Signal and Information Processing*, 2, 2013.

[103] 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, 2017.

[104] X. Liu, B. Han, F. Qian, and M. Varvello. Lime: understanding commercial 360° live video streaming services. In *Proceedings of the 10th ACM Multimedia Systems Conference*, pages 154–164, 2019.

[105] X. Liu, Y. Yao, and F. Qian. Rethink phone-wearable collaboration from the networking perspective. In *ACM WearSys*, 2017.

[106] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang. When good becomes evil: Keystroke inference with smartwatch. In *CCS*. ACM, 2015.

[107] H. Miao and F. X. Lin. Tell your graphics stack that the display is circular. In *HotMobile*, 2016.

[108] A. Narayanan, E. Ramadan, R. Mehta, X. Hu, Q. Liu, R. A. Fezeu, U. K. Dayalan, S. Verma, P. Ji, T. Li, et al. Lumos5g: Mapping and predicting commercial mmwave 5g throughput. In *Proceedings of the ACM Internet Measurement Conference*, pages 176–193, 2020.

[109] A. Narayanan, X. Zhang, R. Zhu, A. Hassan, S. Jin, X. Zhu, X. Zhang, D. Rybkin, Z. Yang, Z. M. Mao, et al. A variegated look at 5g in the wild: Performance, power, and qoe implications. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2021.

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

[111] C. Nicutar, D. Niculescu, and C. Raiciu. Using cooperation for low power low latency cellular connectivity. In *CoNEXT*, pages 337–348. ACM, 2014.

[112] 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 *MobiCom*. ACM, 2016.

[113] A. Nikravesh, Y. Guo, X. Zhu, F. Qian, and Z. M. Mao. Mp-h2: A client-only multipath solution for http/2. In *MobiCom*. ACM, 2019.

[114] Y. Niu, Y. Li, D. Jin, L. Su, and A. V. Vasilakos. A survey of millimeter wave communications (mmwave) for 5g: opportunities and challenges. *Wireless networks*, 21(8):2657–2676, 2015.

[115] S. Oh, A. Kim, S. Lee, K. Lee, D. R. Jeong, S. Y. Ko, and I. Shin. Fluid: Multi-device mobile platform for flexible user interface distribution. In *MobiCom*. ACM, 2019.

[116] S. Oh, H. Yoo, D. R. Jeong, D. H. Bui, and I. Shin. Mobile plus: Multi-device mobile platform for cross-device functionality sharing. In *MobiSys*. ACM, 2017.

[117] C. Olaverri-Monreal, P. Gomes, R. Fernandes, F. Vieira, and M. Ferreira. The see-through system: A vanet-enabled assistant for overtaking maneuvers. In *2010 IEEE Intelligent Vehicles Symposium*, pages 123–128. IEEE, 2010.

[118] 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.

[119] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure. Experimental evaluation of multipath tcp schedulers. In *ACM SIGCOMM Capacity Sharing Workshop (CSWS)*. ACM, 2014.

[120] A. Paigwar, Ö. Erkent, D. Sierra-Gonzalez, and C. Laugier. Gndnet: Fast ground plane estimation and point cloud segmentation for autonomous vehicles. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2150–2156. IEEE, 2020.

[121] 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.

[122] C. Peng, G. Shen, and Y. Zhang. Beepbeep: A high-accuracy acoustic-based system for ranging and localization using cots devices. *ACM Transactions on Embedded Computing Systems*, 11(1):4, 2012.

[123] F. Qian, V. Gopalakrishnan, E. Halepovic, S. Sen, and O. Spatscheck. Tm 3: flexible transport-layer multi-pipe multiplexing middlebox without head-of-line blocking. In *CoNEXT*. ACM, 2015.

[124] F. Qian, B. Han, J. Pair, and V. Gopalakrishnan. Toward practical volumetric video streaming on commodity smartphones. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. ACM, 2019.

[125] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *MobiSys*. ACM, 2011.

[126] D. Qiao and K. G. Shin. Smart power-saving mode for ieee 802.11 wireless lans. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 1573–1583. IEEE, 2005.

[127] Y. Qin, S. Hao, K. R. Pattipati, F. Qian, S. Sen, B. Wang, and C. Yue. Abr streaming of vbr-encoded videos: characterization, challenges, and solutions. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 366–378, 2018.

[128] H. Qiu, F. Ahmad, F. Bai, M. Gruteser, and R. Govindan. Avr: Augmented vehicular reality. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 81–95. ACM, 2018.

[129] V. Radu, P. Katsikouli, R. Sarkar, and M. K. Marina. A semi-supervised learning approach for robust indoor-outdoor detection with smartphones. In *SenSys*. ACM, 2014.

[130] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *ACM SIGCOMM*, 2011.

[131] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *NSDI*. USENIX, 2012.

[132] E. Rescorla, H. Tschofenig, and N. Modadugu. The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. `https://tools.ietf.org/id/draft-ietf-tls-dtls13-01.html`, 2018.

[133] Selenium. Selenium automates browsers, 2021.

[134] A. Sharma, V. Navda, R. Ramjee, V. N. Padmanabhan, and E. M. Belding. Cool-tether: energy efficient on-the-fly wifi hot-spots using mobile phones. In *CoNEXT*. ACM, 2009.

[135] H. Shi, Y. Cui, X. Wang, Y. Hu, M. Dai, F. Wang, and K. Zheng. Stms: Improving mptcp throughput under heterogeneous networks. In *USENIX ATC*, pages 719–730, 2018.

[136] M. Siekkinen, T. Kämäräinen, L. Favario, and E. Masala. Can you see what i see? quality-of-experience measurements of mobile live video broadcasting. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 14(2s):1–23, 2018.

[137] M. Siekkinen, E. Masala, and T. Kämäräinen. A first look at quality of mobile live streaming experience: the case of periscope. In *Proceedings of the 2016 Internet Measurement Conference*, pages 477–483, 2016.

[138] I. Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *IEEE multimedia*, 18(4):62–67, 2011.

[139] O. Studio. Open Broadcast Software. `https://obsproject.com/`, 2020.

[140] P. Sun, M. Yu, M. J. Freedman, and J. Rexford. Identifying performance bottlenecks in cdns through tcp-level monitoring. In *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack*, pages 49–54. ACM, 2011.

[141] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescape. Broadband Internet Performance: A View From the Gateway . In *ACM SIGCOMM*, 2011.

[142] J. C. Tang, G. Venolia, and K. M. Inkpen. Meerkat and periscope: I stream, you stream, apps stream for live streams. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 4770–4780, 2016.

[143] D. Tse and P. Viswanath. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[144] Twitch. Twitch Recommended Software for Broadcasting. `https://help.twitch.tv/s/article/recommended-software-for-broadcasting?language=en_US`, 2020.

[145] umlaeute. v4l2loopback - a kernel module to create V4L2 loopback devices. `https://github.com/umlaeute/v4l2loopback`, 2020.

[146] B. Wang, X. Zhang, G. Wang, H. Zheng, and B. Y. Zhao. Anatomy of a personalized livestreaming system. In *Proceedings of the 2016 Internet Measurement Conference*, pages 485–498, 2016.

[147] H. Wang, T. T.-T. Lai, and R. Roy Choudhury. Mole: Motion leaks through smartwatch sensors. In *MobiCom*. ACM, 2015.

[148] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[149] Wikipedia. Reed–Solomon error correction.

[150] Wikipedia. QR code. `https://en.wikipedia.org/wiki/QR_code`, 2020.

[151] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 129–137, 2017.

[152] J. Xu, Q. Cao, A. Prakash, A. Balasubramanian, and D. E. Porter. Uiwear: Easily adapting user interfaces for wearable devices. In *ACM MobiCom*, 2017.

[153] S. Xu, E. Petajan, S. Sen, and Z. M. Mao. What you see is what you get: measure abr video streaming qoe via on-device screen recording. In *Proceedings of the 30th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 60–66, 2020.

[154] S. Xu, S. Sen, and Z. M. Mao. Csi: inferring mobile abr video adaptation behavior under https and quic. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[155] S. Xu, S. Sen, Z. M. Mao, and Y. Jia. Dissecting vod services for cellular: performance, root causes and best practices. In *Proceedings of the 2017 Internet Measurement Conference*, pages 220–234, 2017.

[156] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. Investigating transparent web proxies in cellular networks. In *International Conference on Passive and Active Network Measurement*, pages 262–276. Springer, 2015.

[157] Y. Yang and G. Cao. Characterizing and optimizing background data transfers on smartwatches. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2017.

[158] Youtube. Create a live stream on mobile – Android – Youtube Help. `https://support.google.com/youtube/answer/9228390?hl=en`, 2020.

[159] Youtube. Create a live stream via webcam – Youtube Help. `https://support.google.com/youtube/answer/9228389?hl=en&ref_topic=9257984`, 2020.

[160] Youtube. How to Live Stream On Youtube – How Youtube Works. `https://www.youtube.com/howyoutubeworks/product-features/live/#youtube-live`, 2020.

[161] Youtube. YouTube Live verified encoders. `https://support.google.com/youtube/answer/2907883?hl=en&ref_topic=9257984#zippy=%2Csoftware-encoders`, 2020.

[162] X. Zhang, A. Zhang, J. Sun, X. Zhu, Y. E. Guo, F. Qian, and Z. M. Mao. Emp: Edge-assisted multi-vehicle perception. In *ACM MobiCom*, 2021.

[163] P. Zhou, W. Zhang, T. Braud, P. Hui, and J. Kangasharju. Arve: Augmented reality applications in vehicle to edge networks. In *Proceedings of the 2018 Workshop on Mobile Edge Communications*, pages 25–30, 2018.

[164] Y. Zhou and O. Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 2018.

[165] X. Zhu, Y. E. Guo, A. Nikravesh, F. Qian, and Z. M. Mao. Understanding the networking performance of wear os. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(1):1–25, 2019.

[166] X. Zhu, Y. Jin, F. Qian, and Z. M. Mao. Poster: Experimental evaluation of tcp congestion control over 60ghz wlan. In *Proceedings of the 2019 on Wireless of the Students, by the Students, and for the Students Workshop*, pages 18–18, 2019.

[167] X. Zhu, S. Sen, and Z. M. Mao. Livelyzer: analyzing the first-mile ingest performance of live video streaming. In *Proceedings of the 12th ACM Multimedia Systems Conference*, 2021.

[168] X. Zhu, J. Sun, X. Zhang, Y. E. Guo, F. Qian, and Z. M. Mao. Mpbond: efficient network-level collaboration among personal mobile devices. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 364–376, 2020.