DETECTION AND ALLEVIATION OF LAST-MILE WIRELESS LINK BOTTLENECKS

DOCTORAL THESIS

NIMANTHA THUSHAN BARANASURIYA

NATIONAL UNIVERSITY OF SINGAPORE

2016

DETECTION AND ALLEVIATION OF LAST-MILE WIRELESS LINK BOTTLENECKS

NIMANTHA THUSHAN BARANASURIYA

BSc. Engineering (Hons), University of Moratuwa, Sri Lanka

Advisor:

Dr. Seth Gilbert (National University of Singapore)

Mentors:

Dr. Venkat Padmanabhan (Microsoft Research India)

Dr. Vishnu Navda (Microsoft Research India)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE, SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2016

To my wife, Dilini, my daughter, Inuki, and my parents,

Thusanthi and Nimal.

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

[illegible]

Nimantha Thushan Baranasuriya

June 2016

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Seth Gilbert, for his wonderful support and guidance throughout my PhD program. I am privileged to have received the opportunity to pursue my studies with him because the things that I learned from him, both technical and non-technical, are invaluable. He has always been there to see me through the rough patches I hit during my PhD and I could turn to him for advice whenever I needed it. He has continuously inspired me to think differently and has shaped the way I approach and tackle tough problems.

I would like to convey my sincere gratitude to Dr. Venkat Padmanabhan and Dr. Vishnu Navda for giving me the rare opportunity of contributing to several exciting projects that appear in this dissertation. I enjoyed our collaborations immensely and I owe the success of these projects to their close guidance and mentorship. The art of doing impactful systems research is the most important lesson I learned from them. I would also like to thank them for hosting me at Microsoft Research India thrice during the course of my PhD.

I would like to thank Dr. Calvin Newport and Dr. Jayanthi Rao for their support on the first project I worked on during my PhD program. I thank Rajat Kateja and Rajdeep Das, with whom I collaborated on the `DiversiFi` and `Kwikr` projects, respectively. It is my absolute pleasure to have worked with all of them.

I am thankful to Dr. Chan Mun Choon, Dr. Ben Leong, and Dr. Kannan Srinivasan for agreeing to serve in my thesis committee, in spite of their busy schedules. I appreciate their most valued feedback on the improvement of this dissertation. I thank Dr. Damith Rajapakse for all the support and advice he provided during my stay at NUS, and inspiring me to create and deliver presentations that standout.

I would like to thank my wonderful friends in Singapore and my lab mates of the CIR lab: Abhijeet, Anuja, Chengwen, Girisha, Hande, Kajanan, Kartik, Lahiru, Marcel, Mobashir, Mostafa, Padmanabha, Pravein, Sumanaruban, Vara and Xiangfa. I am lucky to have met such people because they supported me all throughout my PhD by helping me on my projects, proofreading and giving feedback on my papers, and providing useful tips on living in Singapore. Our lab is, without a doubt, the friendliest in NUS, where everyone helps out each other in times of difficulties and celebrate each other's achievements. We've had countless technical, political, and philosophical discussions that I truly enjoyed partaking in.

Thanks are due to my parents, Thusanthi and Nimal. They were elated when I got the opportunity to enroll at NUS and have always been by my side during my PhD being an endless source of love, support and motivation. I thank my wife, Dilini, for her unconditional love, for being my pillar of strength, for encouraging me during hard times, and for giving her fullest support to complete my studies. I would also like to thank my daughter, Inuki, for her love and for making life meaningful and enjoyable. I dedicate this dissertation to them. I thank my two sisters, Chamilka and Chamali, my brothers-in-law, Nuwan and Rusiru, my nephew, Minuka, and my niece, Nethuki, for their love, support, and guidance all throughout this period.

Last but not least, I would like to thank National University of Singapore and the Singapore government for giving me the opportunity to live in this great country and pursue my PhD at one of the world's best universities. I have thoroughly enjoyed my time at NUS and for that I am eternally grateful to everyone who helped me even in the smallest of ways.

Abstract

Increasingly, a significant portion of users connect to the Internet via wireless connections, i.e., cellular or Wi-Fi. This phenomenon can be attributed to the proliferation of hand-held devices. With the advent of these devices, network service providers have to increase the performance of the last-mile connection so that users have uninterrupted, high-bandwidth access to the Internet.

In this thesis, we first show the impact of the last-mile connection on the user-perceived quality of end-to-end network connections by conducting large-scale measurement studies. Specifically, we consider two last-mile technologies: cellular (3G and LTE) and Wi-Fi, and quantify their effect on real-time data streams such as VoIP calls. These measurement studies show that last-mile wireless links can often be the bottleneck link in a given path, but the existence of WAN path bottlenecks is substantial as well. Therefore, it is of great importance to identify where network bottlenecks lie in order to take the appropriate remedial actions.

In this regard, first, we present `QProbe`, a lightweight tool that provides diagnostic information about a user's cellular link. `QProbe` identifies whether an end-to-end network path is bottlenecked by the cellular link or the Internet path with $> 85\%$ accuracy. Next, we present a suite of techniques named `Kwikr` that Wi-Fi connected clients can use to detect various forms of pathology conditions that might be present in their Wi-Fi connections, with $> 90\%$ accuracy.

Real-time streaming is an important class of applications that is used by millions of people all over the world. Despite their importance, our measurement studies show that quite often Wi-Fi last-mile connections cause quality degradations in real-time streams. To address such issues, in the latter part of this thesis, we present two systems that make real-time streaming reliable and robust on Wi-Fi last-mile links by alleviating bottleneck conditions that occur on them.

First, we show how we can alleviate Wi-Fi bottlenecks in real-time streaming applications by allowing them to perform “Wi-Fi-aware” bandwidth adaptation using a novel system named `KwikrAdapt`. Next, we present another Wi-Fi bottleneck alleviation system, named `DiversiFi`, that improves the quality of real-time streams by performing cross-link packet replication across multiple Wi-Fi links, while ensuring the overhead of doing so is kept low. Both these systems significantly improves the quality of real-time streams and makes them robust.

Contents

List of Figures

List of Tables

List of Publications

1. Nimantha Baranasuriya, Seth Gilbert, Calvin Newport, and Jayanthi Rao. Aggregation in Smartphone Sensor Networks. In *Proceedings of the 10th IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS 2014.

2. Nimantha Baranasuriya, Vishnu Navda, Venkat N. Padmanabhan, and Seth Gilbert. QProbe: Locating the Bottleneck in Cellular Communication. In *Proceedings of the The 11th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT 2015. (Best Short Paper)

3. Rajat Kateja, Nimantha Baranasuriya, Vishnu Navda, and Venkat N. Padmanabhan. DiversiFi: Robust Multi-Link Interactive Streaming. In *Proceedings of the The 11th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT 2015. (Co-primary author)

4. Nimantha Baranasuriya, Rajdeep Das, Venkat N. Padmanabhan, Christoffer Rödbro, and Seth Gilbert. Kwikr: Fast Bandwidth Adaptation Using Wi-Fi Hints. (Under submission)

Part I

Introduction and Background

Chapter 1

Introduction

We are experiencing an unprecedented growth in wireless network-connected consumer electronics in the current era. Global smartphone sales have risen from 122.3 million units in 2007 to 1.2 billion units in 2014 [9]. That is an increase of 884% in just 7 years. In fact, the latest statistics show that smartphones have now surpassed PC sales [7]. If we consider other types of wireless devices such as tablet-PCs, smart TVs, *etc.*, which have not been included in the above figures, it is easy to see that they have taken the world by storm.

The rapid proliferation of these devices has introduced new classes of challenging problems that network administrators and service providers have to deal with. The main commonality of these devices is that they have to be connected to the Internet for them to do useful things, and they use wireless communication mediums to meet those requirements. In simple terms, almost all these devices connect to the Internet using either Wi-Fi or cellular data connections. Therefore, unlike earlier times, wireless technologies have become the preferred mode of Internet access as opposed to wired mediums. This has opened up avenues for new standardizations and immense growth in both Wi-Fi and cellular data technologies all over the world.

The focus of this dissertation is on presenting novel tools and techniques that can detect bottlenecks and alleviate them on last-mile wireless links. A last-mile wireless link, as Figure 1.1 depicts, is either the link from a cellular base station to the client, or the Wi-Fi link from the Wi-Fi access point to the client, when the client is connected to the Internet via a cellular (3G/LTE) connection or a Wi-Fi connection, respectively. At a high level, in this dissertation, we attempt to answer the following three questions pertaining to last-mile

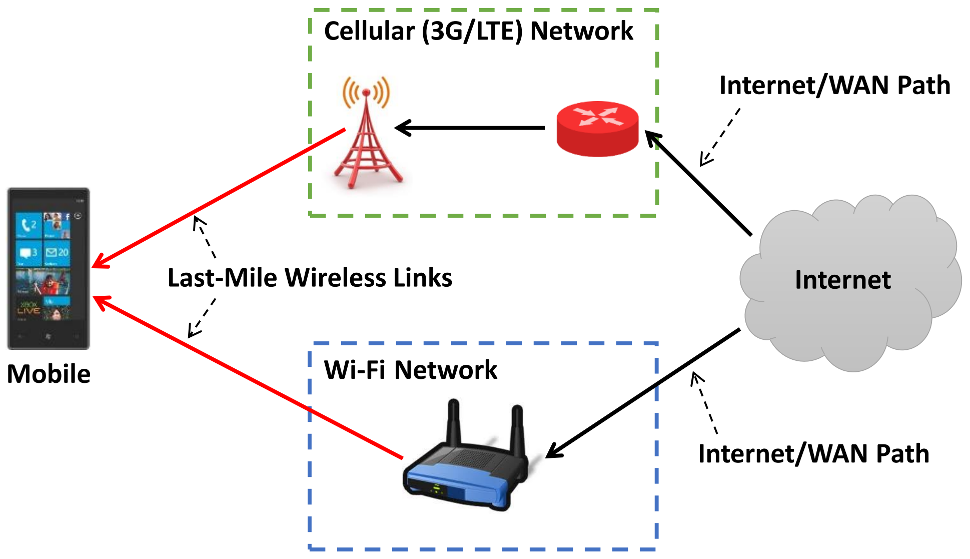


Figure 1.1: Last-mile wireless links: the link from either a cellular base station to a client, or the link from a Wi-Fi access point to a client.

wireless links.

1.1 How Bad Can the Last-Mile Link Be?

Despite the state-of-the-art wireless communication technologies that are in operation today, whenever users experience poor performance in a network application, they typically tend to blame it on the last-mile connection. If they are connected to a Wi-Fi network and they believe that the Wi-Fi link is working fine, then the blame typically falls on the Internet Service Provider (ISP) link that connects the users' homes/offices to the Internet. This results in users requesting service visits from their ISPs, which can be quite costly. The same phenomenon applies for cellular data users as well.

To study how poor last-mile links affect end-user experience, we conducted several large-scale measurement studies on both cellular and Wi-Fi last-mile links. In addition, we analyzed a dataset we obtained from a large Voice over Internet Protocol (VoIP) service provider.

The first experimental study involves cellular data users and we study how congestion at the cellular base station affects the quality of real-time data streams such as Skype calls.

We show that users experience poor quality calls when they are connected to a base station that is under a high workload. In addition, we present data from a measurement study, involving over 600 users across 33 countries over 2 months, that we conducted to study the performance of cellular networks across the world. This study shows that the cellular link as well as the WAN path can be bottlenecked in a given network path, and hence the cellular link is not the culprit for poor performance all the time.

In a similar fashion, we conducted a second measurement study using 274 users we recruited from 22 different countries to study how Wi-Fi links impact the quality of VoIP streams. We built a distributed system called `NetTest` where users send and receive simulated VoIP data streams over their Wi-Fi connections. We collected data for over 2 months from the `NetTest` system. Analysis of this dataset helps us study the impact of the last-mile Wi-Fi link on the quality of the VoIP calls end users experienced.

Finally, we present an analysis of a dataset we obtained from a large VoIP service provider, which contains one year worth of calls that originated from hundreds of millions of users, amounting to billions of talk-time minutes. In this analysis, we compare the number of poor-quality calls that Ethernet-connected clients faced compared to Wi-Fi-connected clients and show that Wi-Fi-connected users experience significantly a higher fraction of poor-quality calls.

1.2 Can We Detect Last-Mile Wireless Link Bottlenecks?

A given network path that terminates at a wireless client comprises two distinct components: the last-mile wireless link, and the Internet/WAN path[1] (see Figure 1.1). Though users typically blame the last-mile connection at times of distress, it is quite an anecdotal belief. There are many settings where the Internet path can be the culprit for quality degradations that users experience in their applications. For example, users might experience problems because their traffic gets routed through an Autonomous System (AS) that regulates certain types of traffic, or the Internet path might have a congested router, or the user's application

[1]We use the terms "Internet path" and "WAN path" interchangeably.

might be using a heavily loaded server, *etc.* Therefore, the Internet path cannot always be ruled out as the bottleneck source when a problem surfaces.

When there is a quality degradation in an application, it is advantageous if the end user can diagnose the issue and attribute the problem to a particular hop in the end-to-end network path in question. For example, say that Alice is using a 3G data connection on her smartphone and she is video conferencing with Bob over Skype. If Alice sees blurry video for most part of the call, it can be due to several reasons: Alice might be connected to a congested base station, or the data center that is used for the call can be heavily loaded, or a router in the Internet might be having hardware issues that is slowing down the packet processing rate, *etc.* If Alice's application can run a diagnostic test and find out where the problem is, it can take the most appropriate remedial action to alleviate the issue; if the 3G connection is the problem, it can request Alice to switch to a Wi-Fi network to place the call; if the problem lies in the Internet path, Skype can attempt to route around the problem by choosing a different route or by changing the data center that is being used for the call.

Though a bottleneck detection tool would be quite useful as explained above, existing techniques do not perform well in wireless networks. Cellular networks have unique characteristics like per-device queue management, different queue service models from that of the core Internet, *etc.* that introduce inaccuracies in existing techniques [103]. There exists many tools in the literature that can detect Wi-Fi bottlenecks (*e.g.,* [90, 57]). However, these are not practically usable because these techniques either need the Wi-Fi adapter to operate in monitor mode, hindering communication, or need access to special infrastructure that is not always available. Therefore, bottleneck detectors designed for wireless networks should address the above mentioned challenges. Moreover, their overhead should be minimal to ensure that they do not disrupt the network when they perform their detections.

To this end, we introduce a network diagnostic tool named `QProbe` that attributes a network bottleneck to either the cellular link or the Internet path when the user experiences a quality degradation in an application. `QProbe` exploits the difference in queue service models in the core Internet and cellular base stations to perform the diagnosis—the Internet routers use FIFO queues, whereas cellular base stations use per-device queues and a fair queue servicing algorithm to service each of the queues [99, 103]. `QProbe` performs the diagnosis by studying the dispersion of packets of a carefully designed packet train. It is lightweight (uses less than 4 KB of data) and requires only $\sim$ 700 ms to run. Through simulation studies, controlled experiments and a measurement study done by recruiting over 600 users from 33 countries and 51 ISPs, we show that `QProbe` is capable of attributing network bottlenecks to either the cellular link or the Internet path with $> 85\%$ accuracy.

Having presented a technique for bottleneck detection on cellular clients, we next turn to Wi-Fi clients. We present a suite of techniques, named `Kwikr`, that can detect bottlenecks on Wi-Fi links. `Kwikr` can be used by any device (*e.g.,* Smartphones, laptops, tablets, *etc.*), on any platform (e.g., Windows, Linux, iOS, Android, *etc.*) and does not require special privileges or infrastructure to run. Our suite of detectors entails a novel active measurement technique that uses Internet Control Message Protocol (ICMP) ping messages of different Differentiated Services Code Point (DSCP) values to detect congestion, and simple and straightforward detectors to detect mobility, handoffs, and weak links by monitoring Received Signal Strength Indicator (RSSI) and the Basic Service Set Identification (BSSID) of the user's Wi-Fi link. Our detector suite is capable of detecting the above mentioned Wi-Fi pathology conditions with $> 90\%$ accuracy.

1.3 How Can Last-Mile Wi-Fi Bottlenecks Be Alleviated?

Real-time streaming is an important class of applications that is used by millions of people to be in touch with their loved ones using VoIP applications like Skype, to have immersive audio video conferences, to play online interactive games, amongst many others. These applications have stringent demands (e.g., low round-trip times (RTTs) and packet loss rates) on the network that are sometimes not met by wireless last-mile links (e.g, 3G/LTE, Wi-Fi). This is underscored by our measurement studies, in which we show that last-mile Wi-Fi links give rise to a significant number of poor quality VoIP calls. Therefore, we present two complimentary approaches that alleviate Wi-Fi bottlenecks and improve the reliability and robustness of real-time streams.

One of the key factors behind the quality degradation of real-time streaming on Wi-Fi networks is the way that bandwidth adaptation in real-time streaming applications functions. Real-time streaming applications (e.g., Skype) send UDP traffic because this class of traffic is delay sensitive and does not require the reliability that TCP offers. Therefore, these applications need to perform bandwidth adaptation in the application layer so that data streams do not exceed the available bandwidth of network paths. The state-of-the-art bandwidth adaptation techniques used by real-time streaming applications adapt the bandwidth of data streams based on end-to-end network path characteristics. We show that this is suboptimal given the dynamic nature of last-mile Wi-Fi links, and argue that real-time streaming applications can do improved bandwidth adaption by being aware of the status of the underlying Wi-Fi link.

The key insight here is that the client connected to the Wi-Fi link can look deeply into Wi-Fi link-level information to understand the condition of the Wi-Fi link. Then, this information can be passed onto real-time streaming applications to take "Wi-Fi-aware" bandwidth adaption decisions. To this end, we propose a user-level module named `KwikrAdapt` that real-time streaming applications can use to perform improved bandwidth adaptation. This module utilizes the information provided by `Kwikr` to be aware of the condition of the Wi-Fi link. Using this information, it is capable of accurately attributing performance problems of real-time streams to the Wi-Fi link, and then tweaking bandwidth adaptation techniques in ways that will allow for improved adaptation. Through our experiments, we show that we can improve the quality of real-time streams by allowing them to achieve higher data rates in the presence of Wi-Fi congestion, adapt proactively in mobility and weak link conditions, and ramp-up the data rate of the flows faster after a handoff completes.

Another factor that imposes quality degradations in real-time streams is the nature of how Wi-Fi links function. Due to the over-the-air broadcast mode of operation of Wi-Fi, it is prone to pathologies like interference from other Wi-Fi networks and devices like microwave ovens that share the same spectrum as Wi-Fi, heavy congestion at access points, poor quality channel conditions, *etc.* Therefore, researchers have proposed techniques wherein clients can improve user experience by leveraging multipath diversity by connecting to heterogeneous technologies such as Wi-Fi and cellular [27, 35], or by selecting the best link to use by switching between multiple links [69, 45].

To alleviate quality issues in real-time streams caused by bottlenecked Wi-Fi connections, we propose a *hedging* technique that enables a device to maintain *two* simultaneous Wi-Fi access point associations. Our experiments show that hedging can significantly improve the quality of real-time data streams by replicating packets across the concurrent Wi-Fi connections, while minimizing the overhead of replication. Our solution, named `DiversiFi`, is applicable in enterprise settings such as offices, schools, malls, airports, *etc.*, where a single entity maintains a multitude of Wi-Fi access points. In such places, `DiversiFi` can cut down bad quality data-streams by up to 2.24×.

1.4 Contribution Summary

In summary, this thesis makes the following contributions.

1. Through large-scale measurement studies, we show that, as one might expect, last-mile wireless links can often be the bottleneck in given network paths, causing poor user experiences. We also show that WAN paths can be bottlenecked in a non-negligible fraction of the paths in the Internet.

2. We present tools and techniques that can detect whether a given last-mile wireless link, either cellular or Wi-Fi, is bottlenecked or not.

   In the cellular context, we present a lightweight probing technique, named `QProbe`, to disambiguate between WAN and cellular (3G\LTE) bottlenecks. Our experiments show that it can locate the bottleneck with high accuracy.

   In the Wi-Fi context, we present a user-level module, named `Kwikr`, that detects various types of Wi-Fi pathological conditions, *e.g.,* congestion, mobility, handoffs, and weak links, that cause a Wi-Fi link to get bottlenecked.

3. As our measurement studies show that last-mile wireless links, more often than not, give rise to poor real-time streaming performance in Wi-Fi networks, we present two complimentary systems to alleviate such issues and improve the reliability of real-time

streams in Wi-Fi networks.

First, we present a system, named `KwikrAdapt`, that enables real-time streaming applications provide better quality streams by performing “Wi-Fi-aware” bandwidth adaptation.

Next, we present a system, named `DiversiFi`, that improves the reliability and robustness of real-time streams by replicating packets across multiple Wi-Fi links, while ensuring the replication overhead is kept low.

1.5 Overview of the Thesis

The rest of this thesis is structured as follows. We first discuss the related work in Chapter 2. Next, we study the impact of the last-mile connection on real-time data streams in Chapter 3. Then, we present network diagnostic tools that can detect bottlenecks in cellular and Wi-Fi connections in Chapters 4 and 5, respectively. Next, in Chapter 6, we discuss how Wi-Fi bottlenecks can be alleviated by performing improved bandwidth adaptation of real-time streams on Wi-Fi-connected clients. In Chapter 7, we discuss the novel technique of hedging in which devices can enjoy better quality data streaming by connecting to two different Wi-Fi access points and replicating packets across them. Finally, we conclude and discuss future work in Chapter 8.

Chapter 2

Related Work

In this chapter, we provide an overview of the literature that is related to our work. We focus on work that is relevant to bottleneck detection, congestion control, and real-time streaming, in wireless networks.

2.1 Detecting Network Bottlenecks

Bottleneck detection techniques attempt to find the bottlenecked hop of a given network path, if such a hop exists. This is important for both network operators, application service providers, and end-users. For example, assume that an application service provider (*e.g.,* YouTube) can detect that a hop between one of its servers and a client is bottlenecked. Then, it can start serving the client using a replica located in another data center, thus potentially routing around the bottleneck. Due to such advantages, there has been significant interest in the literature in designing bottleneck detection techniques. We survey some of the most important pieces of work in this section.

2.1.1 Detecting Bottlenecks in the Internet

Hu *et al.* present a technique in [50], named Pathneck, that is capable of locating bottlenecks in the Internet. They use an active measurement technique, wherein they use packet trains that comprise measurement packets and load packets to do the detection. Load packets resemble the behavior of regular traffic in the Internet, while the measurement packets trigger responses from routers they reach. Their measurement technique is based on the arrival times of such router responses. Therefore, the uniqueness of their solution is that they do not require access to the destination to do the detection. Instead, they specify varying Time To Live (TTL) values on the measurement packets in the train and base their inference on the times the “ICMP TTL exceeded in transit” replies arrive at the source for the measurement packets of which TTL expires while on transit to the destination.

Similar to Pathneck, Cartouche [47] and Packet Tailgating [80] also use packet trains to detect bottlenecks in the Internet. Cartouche uses packet trains that comprise packets of different payload sizes and the technique is built on how network paths handle packets of different sizes. Using this approach, they can measure the bandwidth of any segment of the network path, and hence the bottleneck location can be inferred from those measurements. Packet Tailgating employs a similar approach to Pathneck, wherein they use packet trains consisting of load packets and measurement packets to detect bottlenecks. However, they let their load packets expire instead of the measurement packets as done in Pathneck.

2.1.2 Detecting Bottlenecks in Cellular Networks

To the best of our knowledge, the only work that has looked at detecting bottlenecks in the cellular context is presented by Schiavone *et al.* in [82]. Their detection methodology is based on the passive monitoring or DATA and ACK packets of TCP flows *inside* the operator’s network [82], which is similar to the technique used by [90] in Wi-Fi networks. Therefore, the downside to this scheme is the requirement of vantage points that end-users typically do not have access to.

2.1.3 Detecting Bottlenecks in Wi-Fi Networks

A bottleneck link detection technique for Wi-Fi home networks was introduced by Sundaresan *et al.* [90, 89]. Their solution is based on passively monitoring TCP flows at the Wi-Fi access point and then determining whether the bottleneck is in the Internet path or confined to the Wi-Fi network. Kanuparthy *et al.* showed that user-level probes can discover the pathologies of low signal-to-noise ratio (SNR), hidden terminals, and congestion [57] in Wi-Fi networks.

Their inferences are based on the one-way delays of packets that the client transmits to a server, which is connected to the Wi-Fi access point via a LAN connection.

Apart from the bottleneck detection tools mentioned above, there are systems that provide useful information in diagnosing and troubleshooting Wi-Fi network faults. NetPrints [20] is a system that diagnoses problems in home networks due to misconfigurations of devices including wireless routers. They use crowdsourcing techniques to collect device configurations and compare snapshots of working and non-working configurations by using decision trees to diagnose the root cause of issues that users face. Adya *et al.* presented an architecture and techniques that can diagnose faults in enterprise networks such as connectivity problems, performance problems, security issues, and authentication problems [19]. Rayanchu *et al.* presented a system named Airshark that can detect interference from non-Wi-Fi devices such as cordless phones, microwave ovens, *etc.* by using commodity Wi-Fi hardware [77]. WiSlow [62] uses low-level Wi-Fi network information (e.g., frame checksum errors and MAC-layer data rate) and also additional information (e.g., MAC-level ACKs) to determine the nature of impairments suffered by a wireless link (e.g., interference from a baby monitor versus from a microwave oven).

2.1.4 Estimating Capacity and Available Bandwidth of Network Paths

Though capacity and available bandwidth estimation techniques do not locate bottlenecks, they help us detect the existence of them. In the past decade, a myriad of techniques (*e.g.,* [52, 60, 53, 40]) have been proposed for estimating the capacity and the available bandwidth of network paths. Note that these two properties are different from one another; the capacity of a path refers to the maximum data rate that it can support, and the available bandwidth is the bandwidth available for a new flow that wants to use that path. Moreover, capacity is time invariant whereas available bandwidth is time variant.

Capacity Estimation via Packet-Pair Principle. A significant portion of capacity estimation techniques use the packet-pair principle [52, 60], where a server sends two back-to-back packets and the receiver estimates the capacity based on the time gap of the packet arrivals. As this technique is susceptible to cross-traffic interference, researchers have proposed techniques like sending trains of packets of various sizes (*e.g.,* Bprobe [30]), or filtering to discard samples that do not relate to the bottleneck link capacity (*e.g.,* Nettimer [63], Pathrate [37]).

Capacity Estimation via Packet Size-Delay Relationship. Another prevalent technique is to use the relationship between the packet size and the delay to estimate the capacity, *e.g.,* pathchar [39], pchar [71], and clink [33]. A shortcoming of these tools is their inapplicability in a variety of settings because they rely on ICMP time-exceeded messages from routers for delay measurements—some subnets in the Internet completely block ICMP messages.

Available Bandwidth Estimation via PRM. The literature on measuring available bandwidth can be mainly divided in to two groups: packet rate method (PRM) and packet gap method (PGM). The PRM method works by sending a train of probe packets at different rates. The basic principle is that if the sending rate is lower than the available bandwidth of the path, the receiver will receive the packet train at a rate similar to the sending rate. However, if the sending rate is higher than the available bandwidth, the receiving rate will be lower and the one way delay trend of the packets increases. PRM method estimates the available bandwidth by finding the sending rate at which a transition occurs between the two modes. Examples of tools that use PRM are Pathload [53], pathChirp [40], PTR [51], and TOPP [67].

Available Bandwidth Estimation via PGM. In contrast, PGM tools use the change in spacing between pairs of equal-sized probe packets at the receiver to estimate the volume of cross-traffic that shares the network path between the sender and the receiver. Then, this estimate is subtracted from the capacity of the path to estimate the available bandwidth (*e.g.,* Spruce [87], Delphi [79], and IGI [51]).

Available Bandwidth Estimation in Broadband Access Networks. The above available bandwidth estimation techniques perform poorly in broadband access networks due to rate regulations, non-FIFO packet scheduling, *etc.* Lakshminarayanan *et al.* devised a tool named ProbeGap [64] that provides accurate estimates despite the aforementioned challenges.

2.1.5 Differentiation

The tools we present in Chapters 4 and 5 are different from the work discussed above in several ways. First and foremost, we are not interested in estimating the capacity or the available bandwidth of a given network path, as done by tools that we surveyed in Section 2.1.4. Rather, our focus is to detect whether the last-mile wireless link or the Internet path is the bottleneck in a given network path in which user applications perform poorly.

Prior work has shown that existing bottleneck detection techniques do not provide accurate results in cellular networks [103]. This is mainly because they were not designed to cope with the unique properties of cellular networks; cellular base stations maintain per-device queues and employ proportional fair scheduling. Therefore, our work in Chapter 4 is distinguished from the above body of work in terms of its focus on (a) cellular-terminated paths, and (b) a deployable tool that can run on off-the-shelf client devices (e.g., iPhone), which do not provide access to any low-level network information. Furthermore, we do not require measurement vantage points within the network infrastructure.

The work we present in Chapter 5 is different from the literature we surveyed in Section 2.1.3 in many ways. Our tools can be easily deployed on off-the-shelf client devices such as laptops, smartphones, *etc.* Existing techniques, although they provide highly accurate bottleneck condition detections, require modifications in the Wi-Fi access point, special-purpose hardware, or running the Wi-Fi interface in monitor mode, that inhibits wide scale deployability.

2.2 Congestion Control in Wireless Networks

Congestion control is an important building block in network communications that is either built into the protocol (*e.g.,* TCP) or done in the application layer (*e.g.,* real-time streaming applications that use UDP). We can go as far as to say that one of the most important pillars that keeps the Internet alive is the congestion control techniques used by all the data flows that traverse the Internet.

When traditional congestion control techniques, which were mainly designed for wired networks (*e.g.,* LANs), were put into use in wireless networks, researchers found that they underperform by a great margin due to the dynamic nature of wireless networks. So there has been significant interest in designing congestion control techniques that cope well with the unique challenges introduced by wireless networks. We survey some of the papers related to this field in this section.

2.2.1 TCP Improvements

There is a long history of work on the impact of random wireless errors on TCP congestion control [25]. A number of solutions have been proposed over the years, including splitting the end-to-end TCP connection to isolate the wireless hop [24], snooping on TCP acks to detect and recover from wireless packet loss locally [26], and enhancing end-to-end TCP to differentiate between wireless errors and congestion loss [43]. Arguably, however, local mechanisms such as link-layer retransmissions (ARQ) and MAC-layer rate adaptation have obviated the need for any fixes at the transport layer.

Since TCP Reno, there have been a number of proposals and also widely used implementations of TCP aimed at performing delay-based congestion control [29], improving high-speed operation [46], and supporting background transfers [93, 78]. In the cellular context, a rate-based congestion control framework for cellular networks was introduced by Leong *et al.* [59]. There has also been work on TCP-friendly rate control intended for streaming applications [92].

Performance-Oriented Congestion Control (PCC) [36] advocates replacing a fixed response to congestion signals with randomized controlled trials, wherein a sender conducts micro-experiments to learn which actions lead to an improvement in performance. The congestion control decisions that are then taken depend on the observed performance improvement.

2.2.2 Rate Anomaly

The rate anomaly problem in Wi-Fi occurs due to the differences in the MAC-layer data rates used by clients [48]. For example, if one client has a lower MAC-layer data rate due to a weak signal from the access point, the performance of all the clients will be considerably degraded because the access point requires more time to serve a client that uses a low MAC-layer rate. This problem has received much attention in the literature, with several proposals to address the problem by equalizing the airtime usage of competing flows. These proposals require the cooperation of the infrastructure (i.e., the Wi-Fi access points) to determine the number of competing flows [58] and/or low-level modifications to Wi-Fi operation such as changing the channel contention behavior [49] or adjusting the frame size [104].

2.2.3 Forecasting

Another segment of prior work has looked at congestion control strategies that forecast the future performance of a network path and adapt a data flow's data rate accordingly. One such approach found in the literature is Sprout [99], which proposes a stochastic forecasting based approach for congestion control in cellular networks. The receiver monitors the packet arrival times to reason about the network conditions and this information is used by the sender to regulate the data rate of the flow. Sprout leverages per-station queues maintained at the cellular base stations, which isolates the self-interaction of a flow's packets from the impact of cross-traffic.

Similar to Sprout, PROTEUS [102] predicts network performance, like throughput, packet loss, and one-way delay, based on past performance of those metrics. They show that bandwidth adaptation decisions taken based on such predictions can help increase the user-perceived quality of real-time streams such as in video conferencing applications.

2.2.4 Utilizing Sensor Hints

Finally, [76] uses hints from sensors such as GPS and accelerometer to determine the mobility mode of a device and perform actions such as MAC-layer rate adaptation and access point selection accordingly. Though such a scheme increases the energy utilization of smartphones, [76] shows that such techniques provide significant improvements in the network performance.

2.2.5 Differentiation

Our work in Chapter 6 is different from the work we surveyed in Section 2.2 in many ways. First, we do not consider the wireless corruption problem itself but instead focus on other wireless-related issues such as detecting wireless congestion and alleviating the impact of the MAC-layer data rate and handoff-induced packet loss. Furthermore, most of the above techniques operate end-to-end, and hence do not have the benefit of direct visibility into the situation at a Wi-Fi link (e.g., the cessation of congestion or the change in mobility status). Next, we do not attempt to forecast network performance and we do not rely on external sensors to infer the status of the Wi-Fi link. Our focus in Chapter 6 is to improve bandwidth adaptation of real-time streams by using information about the Wi-Fi link rather than being oblivious to such data and operating end-to-end. As we aim for deployability on off-the-shelf devices, we do not require access point, device or network stack modifications, as is the case for most of the prior work.

2.3 Real-Time Streaming in Wireless Networks

Real-time streaming applications are in abundance in the current era. Users rely on them to play online games, to keep in touch with their friends and family through VoIP, and to experience immersive audio/video conferences. As we will discuss in length later in Chapter 3, real-time streams suffer numerous performance degradations in wireless networks. Therefore, researchers have introduced new techniques, systems, and even standards to improve the quality of real-time streams on wireless networks. We survey some of the most notable contributions in the literature in this section.

2.3.1 Network Support for Streams

Network support for real-time flows has received a lot of attention over the years, and has even led to the creation of standards such as DiffServ [28] and 802.11e [6], which provide prioritized service to packets based on the type of service (ToS) bits carried in the header. Therefore, these standards allow real-time streams to experience lower latencies when the Wi-Fi link is congested, for example. The ToS bits, however, are typically not preserved across administrative boundaries on an end-to-end path, and hence not used extensively by real-time streaming applications.

2.3.2 Multi-Link Association

Having clients connect to and maintain multiple links simultaneously has been explored in several pieces of prior work with the motive of improving the performance of data flows. Some of this work has considered links spanning heterogeneous technologies such as Wi-Fi and cellular [27, 35] while other work has focused on links over a homogeneous technology such as Wi-Fi (e.g., [32] enables multiple Wi-Fi associations by virtualizing a single Wi-Fi NIC in software). One motivation for combining links is to enhance performance through bandwidth aggregation, especially in the backhaul [21, 56]. A different motivation is to enable seamless handoffs through a make-before-break strategy for managing connections to multiple access points [34], with multi-path TCP [75] providing the glue.

[95] also exploits the diversity of multiple links, either using multiple Wi-Fi NICs or using a single one but switching it between links, but without requiring any changes to the existing Wi-Fi hardware. This work focuses on the uplink, and employs a static switching strategy that cycles through the available links in a round-robin fashion and furthermore employs coding to recover from (non-bursty) packet loss. Other work [94] has assumed knowledge of the statistical models of the packet loss for each link and has accordingly derived the optimal switching strategy.

2.3.3 Reliability via Link Diversity

Improving reliability of real-time streams through link diversity has also been considered in prior work. Multi-radio diversity (MRD) [68] uses multiple radios to simultaneously receive the same transmission and then combines the individual copies, which may each be errored, to try and reconstruct the transmitted packet. Leveraging such *receiver diversity* would require the radios to expose the raw decoded bits to the upper layers of the stack, which is a departure from the currently-deployed Wi-Fi hardware and firmware. Complementary work such as Source-Sync [74] on *sender diversity* involves synchronized transmissions from multiple access points to a receiver to improve reliability and throughput. Again, this would be hard to accomplish with the currently-deployed Wi-Fi hardware.

2.3.4 Other Strategies

There has also been work on multipath streaming over WAN paths [44, 23, 96], with a coded or duplicated stream of packets being spread across multiple paths. Coding and duplication imposes an overhead over the source stream and furthermore it is assumed that the receiver would be able to receive the streams sent on the multiple paths concurrently.

Finally, fine-grained link *selection*, wherein a client switches rapidly from one link/channel to another, has also been studied [69, 45]. Switching is different from diversity because diversity means that, for example, if the client does not receive a packet over one link, it can still receive the missed packet over another link (by taking advantage of network-side buffering), thus achieving significantly better loss recovery. In contrast, switching links based on a link selection strategy could only possibly help in the delivery of future packets.

2.3.5 Differentiation

In the context of our focus in Chapter 7, there are several key differentiating factors when compared to the work surveyed above. Firstly, our goal is to leverage the diversity of multiple Wi-Fi links to improve reliability, hence our focus is on real-time streaming rather than TCP flows, which is the case of most of the above work. Furthermore, we take the advantage of the diversity provided by multiple paths, without incurring the overhead of switching between links or duplicated transmission, unless there is actually a packet loss. Secondly, in contrast to the above body of work, ours is a software-only solution that does not depend on any new hardware capability, does not perform proactive switching, and also addresses the downlink direction. Our solution actually observes packet loss on a link and then switches to a different link *reactively*.

We do derive from the link switching techniques mentioned in Section 2.3.4 in that we also advocate switching between links/channels rapidly. However, our focus is on diversity rather than selection, in that the client derives the benefit of multiple links.

Chapter 3

Motivation

Most often than not, network performance issues are typically blamed on the last-mile link. In this chapter, we take a closer look at this anecdotal claim by studying the effect of the last-mile wireless link performance on the user-perceived quality of real-time data streams in end-to-end network paths. To be precise, we examine how cellular and Wi-Fi last-mile technologies affect real-time data streams such as VoIP calls.

Section 3.1 studies the performance issues introduced by cellular links in Skype calls, and presents results from a measurement study that we conducted to study how cellular networks perform globally. Section 3.2 first presents an analysis of a dataset we obtained from a large VoIP service provider that contains data from hundreds of millions of users throughout a period of one year. Next, we present an analysis of a dataset that we gathered by deploying a geographically distributed measurement testbed that simulated VoIP calls over Wi-Fi between 274 real-world users throughout a period of two months.

The measurement studies we present in this chapter make the following contributions.

1. We show that, as one might expect, last-mile wireless links can often be the bottleneck in given network paths, causing poor user experiences.
2. We also show that WAN paths can be bottlenecked in a non-negligible fraction of the paths in the Internet.
3. Therefore, these studies highlight the importance of detecting bottleneck locations carefully, because the last-mile wireless link is not *always* the bottleneck link. (This forms the basis for our discussion in bottleneck detection strategies in Part II of this thesis.)

4. Finally, our studies show that last-mile wireless links, more often than not, give rise to poor real-time streaming performance, especially in Wi-Fi networks. (We discuss in Part III of this thesis how such issues can be alleviated.)

3.1 Impact of Cellular Bottlenecks

In this section, we present results from experimental studies and a measurement study that we conducted to study the severity of performance issues introduced by cellular links in real-time data streams such as VoIP calls, and the prevalence of such issues.

The experimental studies were conducted in India using cellular connections we obtained from India's most widely subscribed cellular service provider. The goal of these studies is to understand how the load at the cellular base station would affect the quality of an ongoing data stream, *e.g.,* a Skype call.

Our measurement study was conducted globally with the participation of 642 users across 33 countries. The goal of this study is to evaluate the performance of cellular networks on a global scale.

3.1.1 Impact of Base Station Load on TCP Throughput

We first conduct simple experiments to study how the load at the last-mile cellular link affects the end-to-end TCP throughput of a data flow.

Experiment Setup and Results

We installed a TCP server on a desktop PC that sends full-throttle TCP traffic to clients that connect to it. This PC was connected to the Internet using a high-bandwidth up-link (10 Mbps) connection. We then installed a Windows Phone application on three Nokia Lumia 920 smartphones that connects to our TCP server and receives TCP traffic for a specified time period and finally displays the average TCP throughput. The smartphones were connected to the Internet via 3G data connections from the same service provider. We

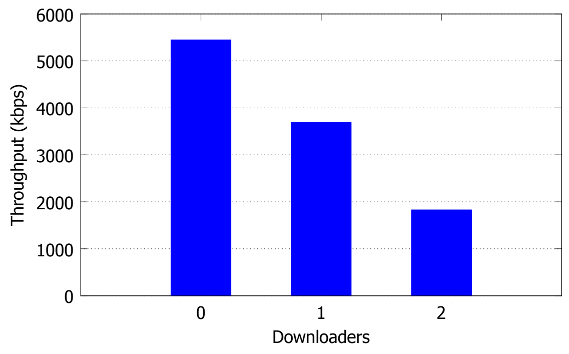


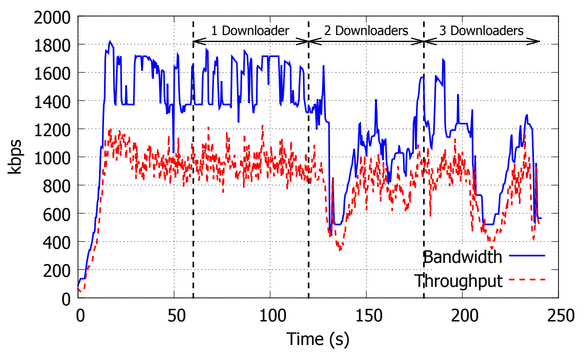
Figure 3.1: The TCP throughput observed at a device when we increased parallel TCP bulk downloaders from 0 to 2. With increasing load at the base station, the observed throughput decreases.

ensured that all three of them were connected to the same base station by analyzing the base station identifier we obtained from the Qualcomm QxDM tool [11]. We measured the TCP throughput at one of the devices in three different settings: when 0, 1, and 2 other phones were downloading TCP traffic alongside the first phone. Figure 3.1 summarizes the results. It can be clearly seen that increasing load at the base station causes the observed TCP throughput to drop. For example, the throughput dropped by 66.38% when we added 2 parallel downloaders as compared to the setting where no parallel downloaders were in operation. (Similar results were observed in [31].)

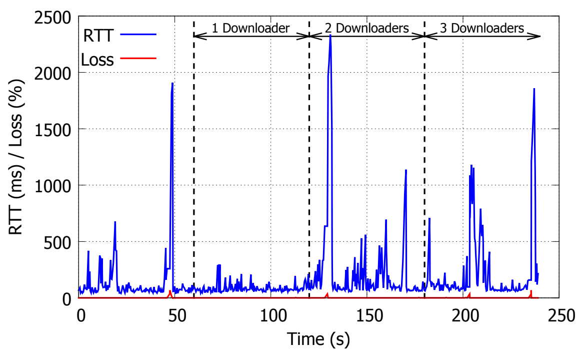
3.1.2 Impact of Base Station Load on Real-Time Data Streams

Next, we analyze how the load at the base station can affect the quality of real-time data streams. Specifically, we analyze the performance of Skype video calls with varying load at the base station.

During a Skype audio/video call, the Skype client at either end estimates the bandwidth of the network path between the two clients and monitors a host of other network parameters such as the link throughput, RTT of the packets, and packet losses. These parameters are fed into Skype's call quality controlling mechanism that adapts the call's quality by changing the audio/video codecs and adjusting the number of Forward Error Correction (FEC) bits used. Rapid fluctuations of the measured network parameters ultimately cause the user-perceived



(a) Estimated Bandwidth and Observed Throughput



(b) RTT and Packet Loss

Figure 3.2: Network parameters of a Skype video call: (a) the fluctuation of the estimated bandwidth and the observed throughput, (b) the fluctuation of RTT and packet losses, of the call's network path. Increasing load at the base station causes quality degradations in real-time streams.

call quality to be poor. For example, users can perceive audio lags if the playback jitter between two successive audio samples is more than 150 ms [12].

Experiment Setup and Results

We used a special Skype client for these experiments that outputs a log at the end of a call containing network parameters it observed during the call. We ran this client on a laptop that was connected to the Internet using a 3G connection. We made this client place a video call to an ordinary Skype client running on a desktop PC that was connected to the Internet using a well-provisioned LAN connection. While the call was going on, we added a new TCP data downloader using the setup in Section 3.1.1 at every one minute mark of the call. Figure 3.2 presents the results we observed.

Figure 3.2(a) shows the fluctuation of the estimated bandwidth and the observed throughput of the network path between the two Skype clients. The bandwidth and the throughput are less variable when there were 0 and 1 downloaders. However, When we increase the number of downloaders beyond that, high variations in the bandwidth and throughput can be observed, which ultimately result in poor call quality. Figure 3.2(b) paints a similar picture where high RTT variations and spikes are observed when the load at the base station is high.

Note that when the number of TCP downloaders were 0 and 1, the Skype flow was not impacted because of the way that the base station manages the flows. When there were only one TCP flow and the Skype flow, the base station has to schedule packets across only two queues. When the TCP flows increase, the base station has to maintain and schedule more queues, and hence the Skype flow observes delays, which results in the behaviors shown in Figure 3.2.

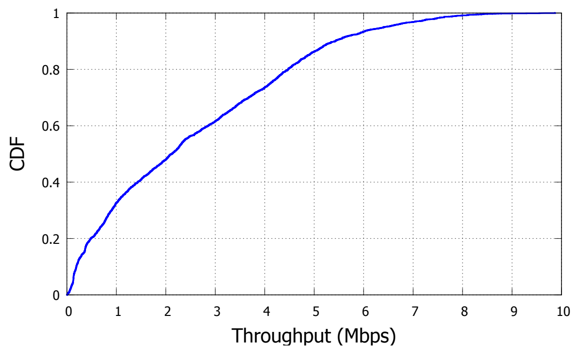
These results show that cellular last-mile technologies can indeed be bottlenecks in end-to-end network paths. The load at a cellular base station can be highly variable with time as it has to cater to both voice and data users who connect to it. Hence, these experiments show that higher loads at the base station would cause the cellular link to be the bottleneck in the end-to-end path.

3.1.3 Cellular Performance in the Wild

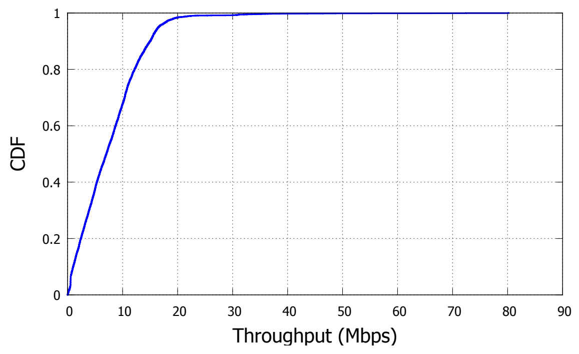
We conducted a large-scale measurement study to analyze cellular network performance in the wild with the aid of an iPhone application (see Section 4.4.3 for details). In a nutshell, in each run of an experiment on our app, it conducts two TCP throughput measurements with the two "closest"[1], well-provisioned Azure servers that we deployed in 15 geographically distributed Microsoft data centers. We calculate the throughput by observing the amount of TCP bytes received in a 5 s time period. We ran our measurement study for 2 months and in this period, we received data from 642 users across 33 countries and 51 cellular service providers.

Figures 3.3(a) and (b) show the CDF distributions of the TCP throughputs we observed on 3G clients and LTE clients, respectively. The 90^{th} percentiles are 5.42 Mbps and 15 Mbps on 3G and LTE, respectively. Typically, a link should have an approximate bandwidth of 1.2 Mbps for Skype to support High Definition (HD) video calls [86]. 36% of our measurements on 3G and 11% on LTE do not meet this requirement, meaning that these clients will experience

[1]Determined by sampling the TCP RTT times between the client and all the servers we deployed.



(a) 3G



(b) LTE

Figure 3.3: CDF distribution of all the TCP throughputs we observed in our measurement study on: (a) 3G users; and (b) LTE users.

Table 3.1: Network Bottlenecks in the Wild.

Technology	**Network Paths**	**Cellular Bottlenecked Paths**	**WAN Bottlenecked Paths**
3G	2573	215 (8.4%)	97 (3.8%)
LTE	5480	441 (8.1%)	837 (15.3%)

non-HD quality video streams if they were to place Skype calls using their cellular data connections.

It is also interesting to study whether the cellular link is the bottleneck link or not in a given network path. To achieve this objective in our measurement study, before we analyzed a network path, we obtained two reference throughput measurements as discussed above. Next, we obtained the TCP throughput of the path that we were analyzing by conducting a TCP throughput measurement with the server the client was connected to. We classified the network path to be bottlenecked by the cellular link only if all three throughputs were low. If the two reference throughputs were high but the throughput of the path was low, we determined that the network path was bottlenecked by the WAN. To determine what constitutes a low throughput, we looked at all throughput measurements obtained for 3G and LTE networks separately, and chose the 25th percentile as the threshold (732 kbps and 3149 kbps for 3G and LTE, respectively). Table 3.1 summarizes our results.

Based on this analysis, we find that 8.4% and 8.1% of the network paths we studied are bottlenecked by the cellular link on 3G and LTE, respectively. Similarly, we find that 3.8% and 15.3% of the total network paths were bottlenecked by the Internet path on 3G and LTE, respectively. Hence, out of the total bottlenecked runs observed in the wild (215 + 97 for 3G, and 441 + 837 for LTE), 68.9% and 25.7% were caused by the wireless link in 3G and LTE, respectively. These results show that the cellular link can become the bottleneck in a significant number of cases. However, it need not be blamed every time a performance problem is perceived because it is also possible that the Internet path is causing the problem. Therefore, whenever a path is bottlenecked, a more nuanced approach is needed to identify the location of the bottleneck responsible for poor performance, which is the focus of discussion in Chapter 4.

3.2 Wi-Fi Streaming Problems

We often hear anecdotes of users facing poor VoIP call quality when connected over a Wi-Fi link. In this section, we seek to quantify the problem by studying whether in real-world deployments Wi-Fi-connected clients encounter worse performance than wired clients.

3.2.1 Analysis of a Dataset from a Large VoIP Service Provider

We begin by analyzing a year's worth of data from a large VoIP service provider, which serves hundreds of millions of users and carries over a billion talktime minutes each day. Our analysis focuses solely on the question of whether the Wi-Fi link is a significant contributor to poor call quality. Hence, we do not disclose any other aspect of the service.

At the end of each call in this VoIP service, the user is invited at random to rate their call experience on a 5-point scale. If the user actually chooses to respond, the provided rating is recorded. We define the two lowest ratings on the 5-point scale as "poor" and accordingly calculate the poor call rate (PCR) as the fraction of calls that are rated as poor. For the reasons noted above and also because there may be a bias in when users choose to provide a rating (e.g., they may be more likely to do so when they have a poor call experience), we do not focus on the PCR itself but instead examine how factors such as Wi-Fi connectivity

Table 3.2: Change in PCR relative to the baseline in the dataset we obtained from a large VoIP service provider. '+' denotes a better (lower) PCR while '-' denotes a worse (higher) PCR. Wi-Fi-connected clients experience more poor calls when compared to wired-connected clients.

	Subset	EE	EW	WW
1	All Subnets and devices	+27.7%	+1.6%	-18.4%
2	/24 Subnets with #E≥#W	+31.9%	+6.3%	-11.9%
3	PC-class devices	+34.2%	+12.9%	-5.4%
4	/24 Subnets with #E≥#W	+36.6%	+15.1%	-3.1%

impact the PCR.

As the baseline, we first compute PCR_{all} based on all user-rated calls during 2014. We then compute the relative difference between PCR_{all} and PCR_X, for a subset, X, of the calls, as $PCR^{\Delta}_{all,X} = \frac{PCR_{all} - PCR_X}{PCR_{all}} * 100\%$. For example, if $PCR_{all} = 10\%$, and $PCR_X = 8\%$ and $PCR_Y = 15\%$ for subsets X and Y, respectively, then we would compute $PCR^{\Delta}_{all,X} = \frac{10-8}{10} * 100 = +20\%$ and $PCR^{\Delta}_{all,Y} = \frac{10-15}{10} * 100 = -50\%$. In other words, the PCR for subset X is 20% better than the baseline whereas that for Y is 50% worse.

The VoIP client determines the network interface used for Internet access and reports this with the other data it sends back to the central server. Hence, we can determine the connectivity type used by the endpoints of each call. Given our interest in analyzing the impact of the Wi-Fi link on the PCR, we separate the calls based on the type of last-hop connectivity of the communicating peers, and focus on the two dominant types: Wi-Fi and Ethernet. We then compute the PCR when both peers are on Ethernet (labeled "EE"), both are on Wi-Fi ("WW"), and one is on Ethernet and the other on Wi-Fi ("EW"). As shown in row #1 of Table 3.2, relative to the baseline, $PCR^{\Delta}_{all,EE} = +27.7\%$ and $PCR^{\Delta}_{all,WW} = -18.4\%$, while $PCR^{\Delta}_{all,EW} = +1.6\%$ lies in the middle. In other words, having Ethernet-connected clients at both ends yields the best (lowest) PCR, Wi-Fi-connected clients at both ends yields the worst (highest) PCR, and Ethernet on one end and Wi-Fi on the other end yields an intermediate PCR.

Although the significantly higher PCR for Wi-Fi-connected peers compared to Ethernet-connected ones points to Wi-Fi being the contributing factor, there are a couple of concerns with drawing this conclusion. The first concern is that since clients on Wi-Fi tend to be mobile, they would likely connect from a much more diverse set of locations than those on Ethernet. For instance, while the latter might be largely confined to well-connected locations such as enterprises and homes, the former would include more challenging environments such as airports and malls, where the backhaul connection itself might be constrained. To mitigate any consequent bias, we only consider pairs of /24 subnets (corresponding to the two communicating peers) for which there are at least as many data points (i.e., user-rated calls) for EE as there are for WW. Thus, subnets that primarily or overwhelmingly host only Wi-Fi clients are excluded, thereby avoiding the concern noted above. As shown in row #2 of Table 3.2, when only such (presumably better-connected) subnets are considered, the PCR improves across the board (i.e., for both Ethernet- and Wi-Fi-connected clients) relative to the full set reported in row #1. However, there is still a significant difference between the PCR for Ethernet-connected clients and Wi-Fi-connected ones.

A second concern is that many Wi-Fi-connected clients would be inexpensive, low-end smartphones and tablets that might suffer from deficiencies in hardware (e.g., under-powered CPUs, low-quality microphones and speakers) that could negatively impact user-perceived call quality. Such poor calls would then be mistakenly attributed to Wi-Fi as the underlying cause. To mitigate this concern, we consider the subset of PC-class devices, which includes a mix of Ethernet- and Wi-Fi-connected desktop and laptop machines. As shown in rows #3 and #4 of Table 3.2, the PCR for the PC-class devices is better (lower) across the board than that for the overall population, more so when we consider just the (presumably better-connected) subnets noted above. However, there is still a significant difference in PCR between Ethernet-connected clients and Wi-Fi-connected clients.

In summary, the PCR for Wi-Fi-connected clients is significantly worse than that for Ethernet-connected clients, with the difference between the two being about 40% relative to the baseline in all cases reported in Table 3.2. We conclude, therefore, that the Wi-Fi link is a significant contributor to poor call quality, and hence is it worthwhile exploring how Wi-Fi can be made more reliable, which is our focus in the work we present in Chapter 7.

Table 3.3: Poor Call Rates for different call categories in the dataset we obtained from our `NetTest`-based measurement study. Calls from a (Wi-Fi-connected) `NetTest` node to another Wi-Fi-connected `NetTest` node have a higher PCR than those to a well-connected Azure node.

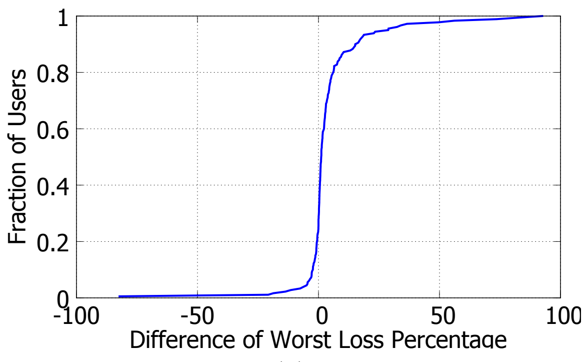
Call Type	Total Calls	Poor Call Rate (%)	Δ in PCR w.r.t EW type (%)
EW	6953	5.22	-
WW	1240	7.98	+52.87
EW-Relayed	798	42.11	-
WW-Relayed	233	62.66	+48.8
Total	9224	10.23	-

3.2.2 Measurements from a Distributed Testbed

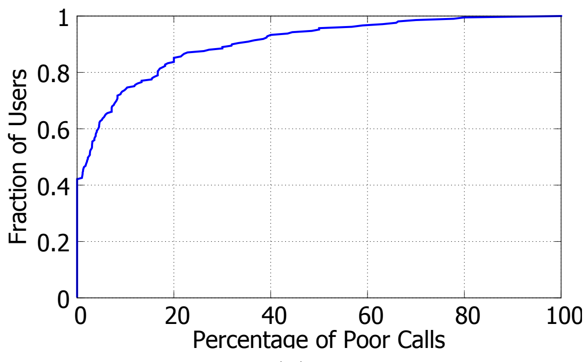
Our analysis of data from a large VoIP service provider in the previous section did not allow us to control the calling pattern or report the *absolute* PCR. To overcome these limitations, we conducted a distributed measurement study, wherein we recruited 274 users[2] across 22 countries to install on their Wi-Fi-connected Windows PC/laptop a simple measurement tool called `NetTest` that we developed. In addition, we install `NetTest` on 10 well-connected machines distributed across Microsoft Azure's data centers worldwide. `NetTest` ran VoIP-like streams (64 kbps, 20 ms inter-packet spacing, 2 min duration) between various pairs of the participating clients. We orchestrated the pattern of these calls, for instance, to have a particular Wi-Fi-connected client connect, in turn, to another Wi-Fi connected client or to a well-connected node on Azure, with these connections happening either directly or through a relay in the cloud. This pattern of connections is designed to mimic typical connectivity patterns in a VoIP service. We conducted this measurement study for 2 months.

Based on our dataset of 9224 simulated calls, we analyze the voice quality of these calls by running the packet traces through a G.711 codec, and using the degree of interpolation and extrapolation of voice samples to estimate Poor Call Rate (PCR), in accordance with well established models [17, 18]. We found the overall PCR, in *absolute* terms, to be 10.23%.

[2]We obtained IRB approval for our study.



(a)



(b)

Figure 3.4: (a) CDF of differences of the average worst packet loss percentage for calls placed by each user to Wi-Fi-connected or Azure clients. (b) CDF of percentage of poor calls observed per user. Users experienced higher losses in the calls that were placed to Wi-Fi-connected clients. Majority of the users experienced at least one poor call and 16.3% of the users experienced a PCR of ∼ 20% or higher.

Table 3.3 shows a breakdown of PCR across different call categories. The call types follow the naming convention used in the earlier section, *i.e.,* "EW": one peer is on Ethernet and the other on Wi-Fi; "WW": both peers on Wi-Fi. The "-Relayed" suffix indicates calls that are relayed through a cloud server because a direct path between the two peers could not be established.

Consistent with our above analysis on data from a large VoIP service, we find that calls from a (Wi-Fi-connected) `NetTest` node to another Wi-Fi-connected `NetTest` node have a higher PCR than those to a well-connected Azure node (7.98% vs. 5.22%, which corresponds to a 50% *relative* difference).

This pattern of Wi-Fi performing worse than wired connection is also seen in the network-level packet loss rate. We divide each 2-minute call trace into 5-second windows and compute the loss rate for the worst window. (See Section 7.2 for the reasoning behind this analysis.) For each client, we then compute the average of such loss rate for calls to Wi-Fi-connected clients and to Azure nodes, and take the difference between the two. In Figure 3.4(a) we plot a CDF of this difference in the loss rate for each user. More than 72% of the users faced higher packet loss rates for calls to Wi-Fi-connected clients compared to wired clients.

To understand how the poor calls are distributed across the users in our dataset, we compute the percentage of poor calls each user experienced and plot the CDF in Figure 3.4(b). While 42.1% of the users did not experience any poor calls, 16.3% of the users had a poor call percentage of ~20% or higher. This implies that the problem of poor streaming quality is widespread and, in fact, quite severe for a sizable fraction of users.

3.3 Chapter Summary

In this chapter, we presented empirical data to quantify the effect of last-mile links on user-perceived quality of real-time data streams. Our results show that users experience poor quality streams when utilizing wireless links (e.g., cellular and Wi-Fi) more so when compared to their wired last-mile counterparts.

However, as noted in this chapter, the last-mile connection might not be the culprit for all instances where users experience quality degradations. Hence, it is important to attribute network issues to the actual bottlenecked link, *i.e.,*the last-mile link or the Internet path, which is the focus of the next part of this dissertation.

Our measurement studies also showed that real-time streaming applications exhibit poor performance on last-mile wireless networks, especially in the case of Wi-Fi. Part III of this dissertation discusses how such performance issues can be alleviated.

Part II

Bottleneck Detection

Part Overview

In this part of the dissertation, we present tools and techniques that can detect whether a given last-mile wireless link, either cellular or Wi-Fi, is bottlenecked or not. Our tools and techniques are designed and implemented giving priority to wide-scale deployability, and hence they can be used on a host of off-the-shelf devices and operating systems.

In Chapter 4, we first introduce a bottleneck detection tool for cellular-connected clients. This tool, named `QProbe`, is capable of detecting whether the cellular wireless link or the WAN path is the cause of a user-perceived performance problem in an application (*e.g.,* a YouTube video that keeps buffering).

Next, in Chapter 5, we present a suite of detectors that is capable of detecting Wi-Fi bottlenecks. This suite, named `Kwikr`, is capable of analyzing whether a Wi-Fi link is bottlenecked by detecting the presence of several pathological conditions like congestion, mobility, handoffs, and weak links.

Chapter 4

Detecting Bottlenecks in Cellular Communication

4.1 Introduction

Cellular connections at times are frustratingly slow. While it might be natural for users to blame the cellular link for the poor performance, doing so is not always appropriate. Indeed, modern cellular connections are often faster than Wi-Fi (e.g., [35] reports that LTE outperforms Wi-Fi 40% of the time). Therefore, we need a more nuanced approach to identifying the location of the bottleneck.

Broadly, we would like to know whether the bottleneck responsible for poor performance lies in the "last-mile" cellular wireless link or elsewhere in the WAN path. Knowing this is key to remediating the problem. For example, if the problem is *not* in the last-mile wireless link, one might route around the WAN bottleneck, say by picking a different replica (e.g., a different CDN server). On the other hand, if the cellular link is the bottleneck, the user may have to look for an alternate connection (e.g., Wi-Fi) or have the application adapt (e.g., by downsizing media content). Any such adaptation undertaken by the client would be over and above any remediation (e.g., load balancing) performed by the cellular provider itself, thus having the advantage of being provider-independent.

The problem of locating the bottleneck in the context of cellular-connected clients is challenging for several reasons. First, cellular base stations employ per-station queues with proportional fair (PF) scheduling [61], unlike the FIFO queuing and servicing assumed in past work on identifying and estimating bottleneck capacity on wired Internet paths (e.g., [50, 64, 38, 53, 87]). Therefore, as pointed out in [103], these techniques do not provide accurate results in cellular networks. Second, cellular data plans tend to have tight data caps, so the bottleneck detection algorithm needs to be lightweight in terms of data usage, avoiding expensive data-intensive probing. Third, cellular networks tend to be closed, under the tight control of the operator, leaving clients with little visibility into the state of the network (e.g., the network load) and no access to a vantage point within the cellular network.

In this chapter, we present `QProbe`, a lightweight, active probing technique to locate the bottleneck link on an end-to-end path, specifically, to decide whether the bottleneck lies on the wired WAN path or on the cellular last-mile. `QProbe` works by detecting whether there is queuing on either segment, hence its name.

This detection is enabled by the *very different behavior of FIFO queuing on the wired path and PF scheduling at the cellular base station.* Routers in the Internet serve packets using the FIFO model, which is in stark contrast to PF schedulers at cellular base stations that maintain per-device queues and serves them in a round-robin fashion to ensure fairness across the clients that are connected to the base station.

When a train of small probe packets is sent by `QProbe` running on a remote host, these tend to get clumped together into *back-to-back bursts* when there is queuing at the cellular base station, much more so than when there is queuing on the wired path. On the other hand, when TTL-limited (large) load packets are interspersed with the small probe packets, then the inter-packet spacing between the latter tends to get *stretched* when there is queuing on the wired path but is unaffected by queuing at the basestation. These 2 signals — back-to-back bursts and stretching — are used in combination by `QProbe` to locate the bottleneck.

We make three main contributions in this section:

1. We have designed a novel, lightweight probing technique to disambiguate between WAN and cellular (3G\LTE) bottlenecks in time on the order 700 ms.

2. Using simulations and controlled experiments, we validate the `QProbe` technique and show that it can locate the bottleneck with high accuracy.

3. We analyze data from our deployment of `QProbe` as an iPhone app to over 600 users spread across 33 countries and 51 operators and show that `QProbe` classifies the

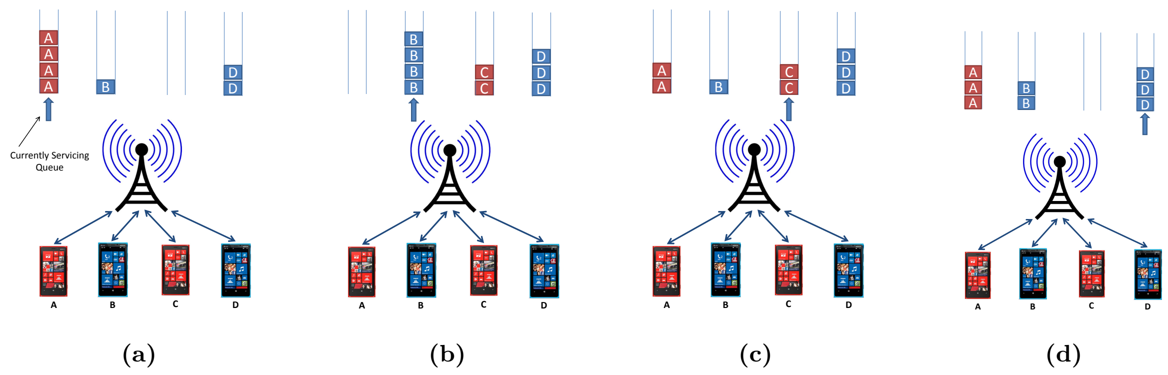


Figure 4.1: Operation of a PF scheduler at a cellular base station. Each device has its own queue at the base station and the base station iterates through each queue and schedules one or more packets from it. While one queue is being served, packets destined to other devices will be queued in their respective queues. This behavior causes a client's packets to be sent back-to-back when the base station is congested.

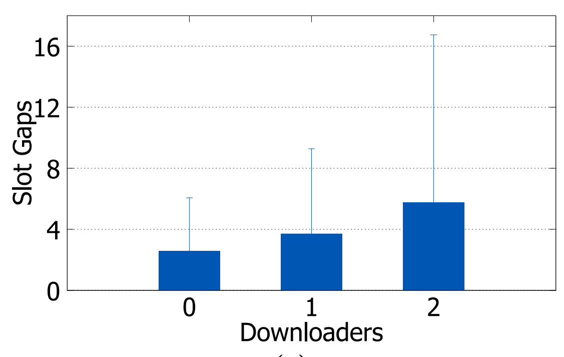
bottleneck with greater than 85% accuracy.

4.2 Background

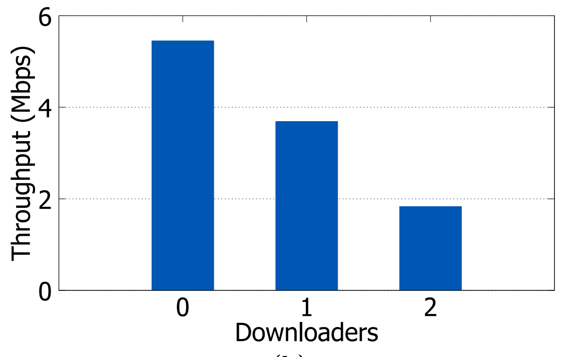
In this section, we highlight the unique characteristics of the downlink packet scheduler in cellular base stations. We then provide insight on how these are leveraged in `QProbe`.

Unlike Wi-Fi and wireline routers, cellular base stations employ a centralized *proportional fair scheduler* (or "PF-scheduler") that determines which client(s) should be served in each scheduling interval (also referred to as a time slot or TTI) [99, 103]. Each slot is typically a few milliseconds long (2 ms in 3G). Multiple clients can be simultaneously served in a single time slot by assigning different orthogonal codes (time-frequency resource blocks) for 3G (LTE) clients. Figure 4.1 depicts the operation of a PF scheduler.

The objective of the PF-Scheduler is to improve the aggregate network throughput, while also being fair to the clients. Instead of strict round-robin scheduling, the PF-Scheduler picks a client that maximizes the ratio of instantaneous rate to the average allocated rate for each client over a certain time window. This ensures that no client is starved, while also keeping the aggregate network throughput healthy.



(a)



(b)

Figure 4.2: (a) Slot gaps between assignments, and (b) Overall throughputs observed by the measurement device with different number of background downloaders. When the load at the base station increases, the frequency at which clients are served reduces.

As the number of clients in the network increases, the slot allocations for a client tend to get spaced apart in time, as the base station is serving other clients. However, when a client is scheduled, multiple packets buffered at the base station can be transmitted back-to-back, depending on the bitrate associated with the instantaneous link quality and the average allocated rate from the recent past.

We wanted to understand how cellular congestion impacts the slot assignments in real networks. However, as described earlier, this is challenging since slot assignments are decided by the base station, while by contrast clients have no knowledge of how many other users are present in the network. In addition, the cellular stack on the client typically does not expose link and physical layer information up the network stack.

To overcome this challenge and gather slot-level data, we used the QxDM diagnostics tool from Qualcomm on a rooted Windows Phone device. We performed a bulk download over 3G on one device, adding 0, 1, or 2 other downloaders in the background. (We ran these experiments in the middle of the night to avoid interference from other users.)

Figure 4.2(a) clearly shows that as the number of downloaders increased, the inter-slot gaps also increased, from a median of 2 ms to 6 ms with the addition of just two downloaders; the throughput also dropped correspondingly (Figure 4.2(b)). During peak hours in crowded settings, we have seen inter-slot gaps even larger than 6 ms.

Note that cross-traffic on a WAN link can also cause large gaps to be introduced between packets destined to a client. The key difference, however, is that with cellular networks, when the PF-Scheduler chooses a client for downlink transmission, multiple packets can be scheduled back-to-back, especially if the packets are small and the inter-slot gap is large. `QProbe` leverages this for its detection, especially since inter-packet gaps can easily be measured by the client.

4.3 Design

In this section, we begin with a description of the design constraints for the bottleneck detection algorithm. We focus on localizing the problem to one of two parts in an end-to-end path, and describe our methodology for each part separately. Finally, we put it all together and describe the `QProbe` technique that combines these design elements.

4.3.1 Design Requirements

Our focus in this work is on diagnosing pathological cases when the observed throughput from an Internet server to a cellular-connected client is quite low. The measurement technique should be able to pinpoint the location of the bottleneck to one of the two segments in an end-to-end path: (a) wired WAN path; or (b) cellular wireless last hop. The wired path includes the Radio Network Controller (RNC) optical backbone within the provider's network as well as the WAN beyond. Typically, the conventional wisdom is that in a well-engineered network, the RNC network is unlikely to be the bottleneck relative to the air interface (wireless hop) [101].

For the purpose of deployability, we do not assume that we have vantage points within the cellular network or along the WAN path for additional measurements, nor do we assume access to a cohort of cooperating clients. Hence, we focus on designing an end-to-end measurement technique between a server on the Internet and an individual cellular-connected mobile device. Moreover, since cellular connections are metered, a key constraint is to ensure that the data usage incurred for our measurements is quite low and is unlikely to exacerbate congestion.

Our goal is to have a technique that works across different platforms and does not require any new operating system (OS) or network capabilities, and can be easily deployed in existing platforms as a user-level application. This is different from prior techniques such as LoadSense [31], which relies on passive low-level measurements from the cellular stack.

4.3.2 Detection Methodology

At a high level, `QProbe` technique consists of actively sending probe traffic downstream from an Internet server to a mobile device, and analyzing the probes received at the client-side to diagnose the problem. We send a train of equally spaced tiny probe packets to the client and observe the arrival times. The small-sized probes ensure that the probing traffic is light.

When these packets arrive at the basestation, if the wireless link is not congested, the basestation would not be backlogged and so the packets are likely to be delivered immediately. In this case, we expect to see the inter-packet spacing at the receiver to be very similar to that at the transmitter. However, with an increase in wireless cross-traffic, the basestation would have to service several other clients, and the probe packets are likely to get queued up at the basestation. Two successive probes would get queued up at the basestation if the inter-packet spacing at the sender is smaller than the times when the client gets scheduled by the PF-scheduler. When the PF-scheduler chooses to service the client to which the probe traffic is destined, multiple probe packets can be delivered depending on the quantum of service allocated to the client.

Moreover, since the packets are small-sized, the chances of multiple probe packets being delivered back-to-back are very high. Thus, to detect wireless congestion, we look for occurrences of increase in inter-packet spacing followed by one or more back-to-back packets[1]. As observed experimentally in the previous section, the higher the congestion, the larger the spacing between two consecutive scheduling opportunities. Thus, we expect to see more back-to-back packets delivered during each scheduling opportunity.

[1]When two consecutive probes arrive within a single transmission time interval (2 ms), we count them as back-to-back.

The idea of using a number of back-to-back probes as a measure of wireless congestions relies on the fact that probes arrive spaced apart in time at the basestation. However, cross-traffic on the WAN path can also cause probe packets to get queued up at a WAN link, and arrive at the basestation without any gaps between them. In such a case, differentiating the WAN effects from wireless congestion is difficult.

To isolate the effects of WAN cross-traffic, we introduce *load packets*, one or more large MTU-sized packets in-between two successive probes. The objective of the load packets is to ensure that the probe packets arrive spaced apart in time at the basestation. But, load packets introduce two new problems. Since they are large, the data consumption for measurement traffic increases. More importantly, at high wireless congestion, as the per-client allocation during each scheduling opportunity reduces, multiple probe packets may not get scheduled at the same time even if they are queued up at the basestation. To address these problems, we employ a commonly used TTL-based approach to drop the load packets at an IP hop close to the cellular network before they reach the basestation. We set the TTL of the load packets to $hop\_count - 1$, where hop_count is the number of hops in the path between the server and the client. (We find the hop count by getting the server to send a series of packets with differing TTL values that are also indicated in the packet payloads. Appendix A outlines the specifics of our hop count finding algorithm.)

Load packets provide two benefits. Not only do they ensure that probe packets arrive spaced apart in time at the receiver, they also help in detecting WAN link bottlenecks. In cases where the WAN link is the bottleneck, the load packets can introduce additional delay due to the packet transmission time of MTU-sized packets over a wired bottleneck link. As a result, the packet spacing between successive probe packets can increase. We measure this effect using a metric called *stretch-factor*, which is computed as the ratio of time duration of the packet train at the receiver to that at the sender. The duration of the train refers to the time between the last and the first probe packets.

Ideally, when there are no bottlenecks anywhere, the packet spacings are similar, and the *stretch-factor* should be close to 1. When there are bottlenecks in the WAN path, additional spacing is introduced due to the transmission of the load packets, and the total duration

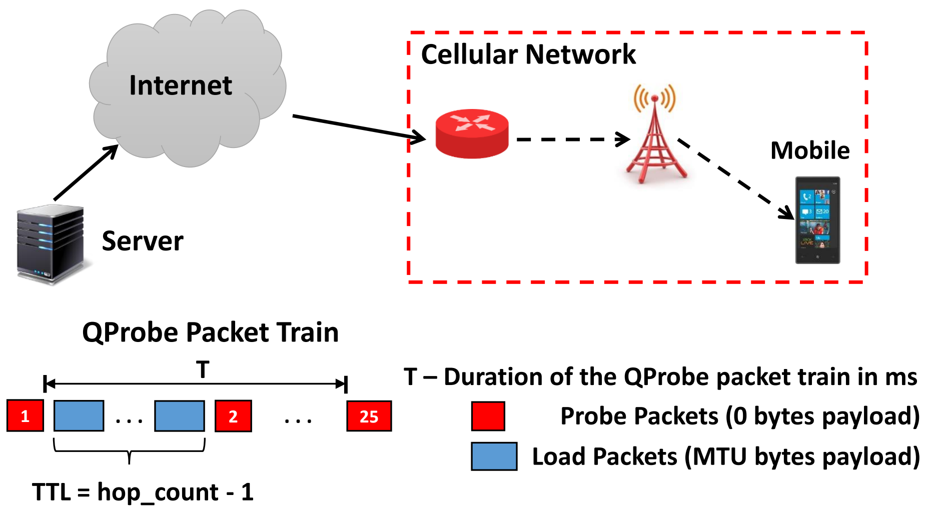

Internet
Cellular Network
Mobile
Server
QProbe Packet Train
T
1
...
2
...
25
TTL = hop_count - 1
T – Duration of the QProbe packet train in ms
Probe Packets (0 bytes payload)
Load Packets (MTU bytes payload)

Figure 4.3: `QProbe` packet train with tiny probe packets and large load packets. TTL for load packets expires 1 hop before the cellular network and get dropped en route.

of the train gets stretched at the receiver, resulting in a *stretch-factor* value greater than 1. With wireless bottlenecks, we still expect low *stretch-factor* (close to 1) since the PF-scheduler can deliver multiple probe packets back-to-back after each occurrence of a large gap due to cross-traffic at the base station.

The overall construction of the `QProbe` packet train is shown in Figure 4.3. We send a sequence of 25 probe packets spaced equally apart. Multiple load packets of MTU size with TTL set to $hop\_count - 1$ are sent in between two consecutive probe packets. The spacing between two load packets as well as the spacing between a probe packet and a load packet is set to 1 ms. The user-level `QProbe` client application records the timestamps of the received packets and runs our detection algorithm to determine the bottleneck location. Since we do not know the wireless conditions, we run `QProbe` train for different values of spacings (4 ms to 8 ms) between the probe packets to get a better estimate of the condition. Therefore, the total data usage for running 5 `QProbe` trains is only 3.5 KB, and completes in just 720 ms. All the above parameters were chosen carefully by conducting experiments where they were varied to find the most appropriate values that provided the best comprise in the cost and the accuracy of the technique.

Our detection algorithm uses two features to determine whether it is a wireless problem or a

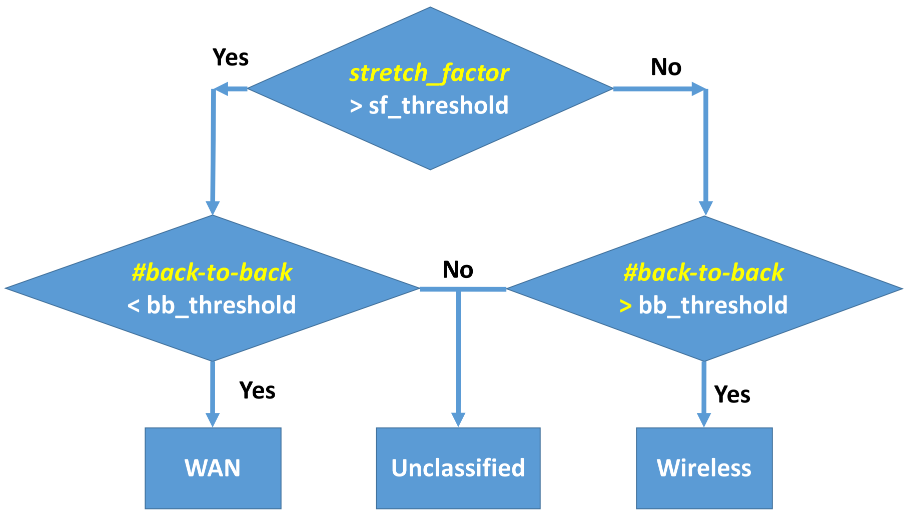


Figure 4.4: `QProbe` Algorithm

WAN path problem. When the number of back-to-back packets observed is high and the *stretch-factor* metric is low, we classify it as a wireless problem, and whenever the number of back-to-back packets is low and the *stretch-factor* is high, we classify it as a WAN bottleneck. Otherwise, we let the problem remain unclassified.

For each `QProbe` run for a certain probe packet spacing, we record the packet arrival timestamps, and run the detection algorithm. The detection algorithm for `QProbe` is described in a flow-chart shown in Figure 4.4. There are two thresholds – `sf_threshold` and `#bb_threshold` that are used to determine the different bottleneck scenarios.

Computing QProbe Thresholds: Using controlled experiments (see Section 4.4.2), we evaluated `QProbe` by varying packet spacings from 4 ms to 8 ms in increments of 1 ms for 489 problem cases, out of which 233 were bottlenecked by the wireless link and the remaining 256 cases experienced WAN bottlenecks. For each of these runs, the ground truth is known – whether the problem is wired or wireless. We trained a 10-fold cross-validation decision tree to predict the bottleneck using *stretch-factor* and the number of back-to-back packets as the two features[2]. We use the thresholds obtained from the tree that is generated from this training phase for classification of `QProbe` runs done on 3G.

[2]We used a decision tree for its simplicity in usage. We believe that other classification algorithms could be used for this purpose as well but we do not study the benefits of doing so.

Since LTE throughput ranges are different from 3G, the thresholds for LTE are trained separately. However, we did not have a sufficient number of LTE connections to perform controlled runs. Therefore, we used a subset of the LTE runs that we obtained from our large-scale measurement study (Section 4.4.3) to build decision tree models for LTE, and derived the thresholds for the two metrics.

Reasons for Unclassified: There can be instances of wireless bottleneck cases that have high *stretch-factor* along with high back-to-back packets. These occur mostly due to transient congestion occurring in both wired and wireless paths. In addition, some wired bottleneck cases can have a low *stretch-factor* and low back-to-back packets. We believe this can occur due to congestion in the wired backhaul portion of the cellular network. Since load packets get dropped at the IP cellular gateway, any congestion in the wired path beyond this point are unlikely to impact the probe packets and thus does not cause the two metrics to increase significantly. Both the above two categories are hard to identify due to lack of ground truth information. In our algorithm, we treat them as unclassified runs.

4.4 Evaluation

We validate the `QProbe` technique using LTE simulations in NS3, and controlled experiments in two 3G networks in India. We then go on to deploy `QProbe` as an iPhone app and collect data from the wild for a large number of cellular operators for both 3G and LTE networks.

Using NS3, we created a real-world topology consisting of a multi-hop wired WAN path that connected to an LTE basestation having a PF-scheduler. The WAN path contained 17 hops, which is the mean number of hops in the data we gathered from the wild (more details in Section 4.4.3). The link speeds of the WAN path were set to 1 Gbps. We connected a server on the wired endpoint to generate probing traffic destined to a cellular client at the other endpoint. We then created different wireless congestion levels by increasing the number of background bulk TCP downloaders in the cell, varying the load level from low (6 downloaders) to medium (9 downloaders) to high (13 downloaders). The number of downloaders for each congestion level was decided such that the TCP throughput observed at the `QProbe` client resembled the 75th, 50th, and 25th percentiles of the TCP throughputs we observed in the LTE measurements of our dataset (Section 4.4.3).

To create WAN bottlenecks, we first varied the capacity of one of the wired hops from 1 Mbps to 5 Mbps. Next, we introduced a TCP cross-traffic flow sent and received by a pair of nodes that shared a hop in the wired path between the `QProbe` server and the cellular-connected client. We varied the throughput of the TCP flow such that it used 40% to 80% of the shared hop's capacity.

As part of `QProbe`, we sent a 100 ms probe train consisting of 25 packets with 4 ms inter-packet spacing from the server to the cellular-connected `QProbe` client.

4.4.1 Simulations

Figure 4.5(a) shows the number of back-to-back packets received at the client side, and Figure 4.5(b) shows the corresponding *stretch-factor* for the same conditions. At high loads, almost 80% of the probe traffic arrive back-to-back, thus matching our hypothesis. In addition, *stretch-factor* remains constant for the most part, with only a marginal increase of 10% at high load. Thus, both metrics behave as expected with wireless congestion.

We analyze the benefit of using load packets in between probe packets in distinguishing WAN and wireless bottlenecks. An intermediate WAN link can become a bottleneck for two reasons: (a) due to low capacity, and (b) due to low available bandwidth owing to cross-traffic. Our objective is to detect WAN bottlenecks for both these cases.

As seen in Figure 4.5(c), the *stretch-factor* does not change when there are no load packets in the probe train, but with the introduction of load packets, the stretch factor increased by a factor of 9 when the bottleneck bandwidth is as low as 1 Mbps. Similarly, in the cross-traffic scenario, with the increase of the volume of cross-traffic, Figure 4.5(d) shows that the *stretch-factor* shows an increase of more than 25%.

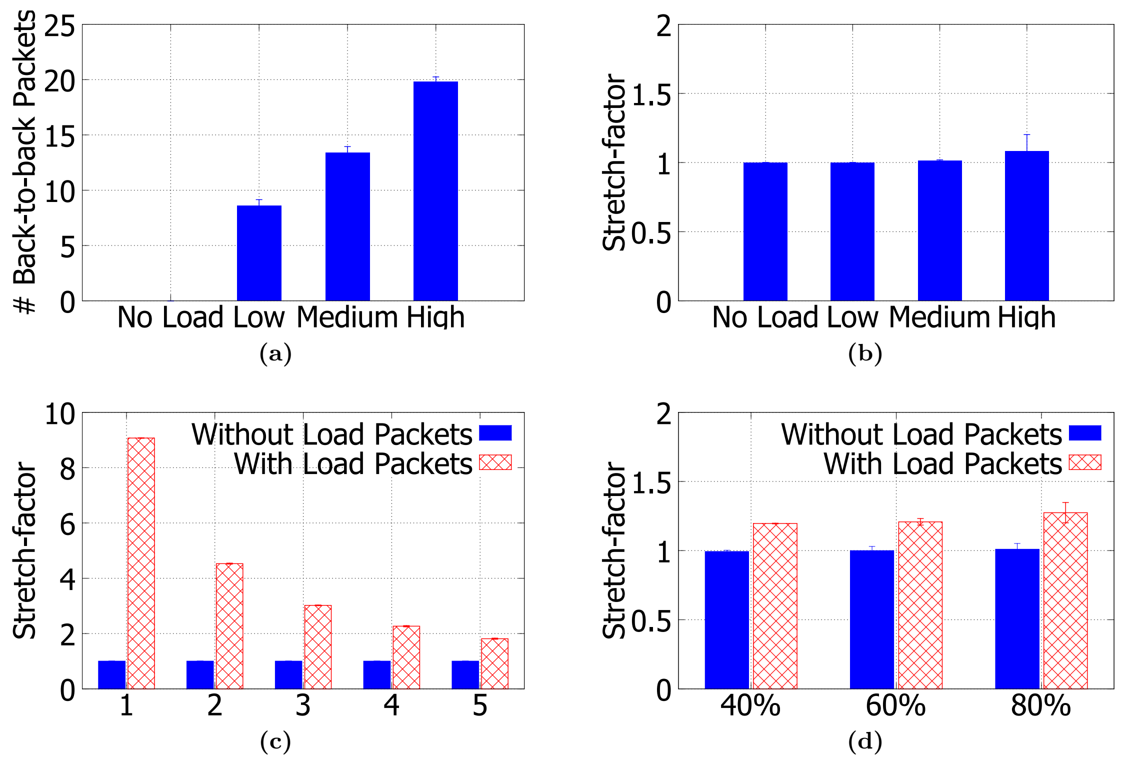


Figure 4.5: (a) Number of back-to-backs, and (b) *stretch-factor* with varying wireless congestion levels. *stretch-factor* for WAN link bottlenecks due to (c) varying link bandwidth (in Mbps) and (d) varying cross-traffic volumes. As the load at the base station increases, the number of back-to-backs observed by the client increases, however, the *stretch-factor* remains low. Load packets help us detect WAN bottlenecks because they increase the *stretch-factor* when the WAN path is bottlenecked by either links having low bandwidths or cross-traffic.

4.4.2 Controlled Experiments

To evaluate `QProbe` on real networks, we conducted controlled experiments on two 3G operators – BSNL and Airtel in Bengaluru, India, with several runs corresponding to the wireless and WAN bottleneck scenarios. Using five co-located smartphones, we generated heavy background wireless traffic in the cell via simultaneous TCP bulk downloads. Next, we connected a laptop running the `QProbe` client to the same basestation using a 3G USB dongle, and conducted 233 `QProbe` runs from a well-provisioned server hosted on the Azure data center in Singapore. Note that these runs are bottlenecked by the wireless link due to the presence of background downloaders. The average throughput the `QProbe` client observed under this bottleneck condition is 945 Kbps.

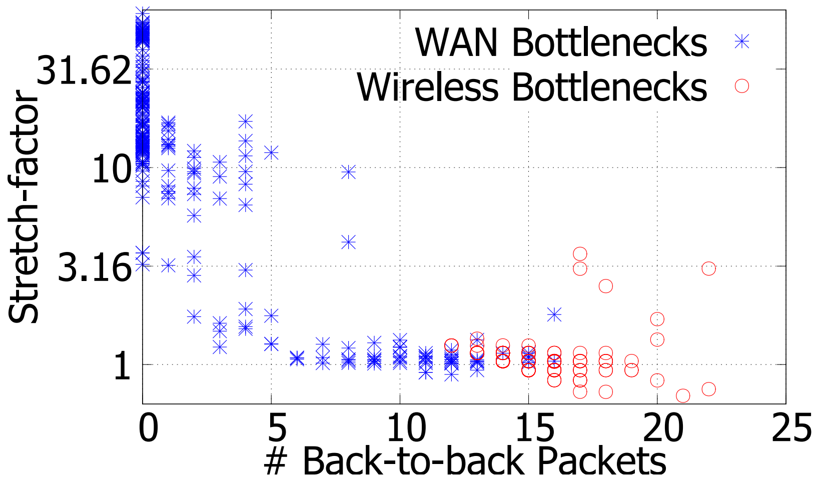


Figure 4.6: # Back-to-back packets and *stretch-factor* for 489 `QProbe` runs. (Y-axis is in log scale.) WAN bottlenecks result in low # back-to-back packets and high *stretch-factor* while wireless bottlenecks show the opposite behavior.

To gather runs for WAN bottlenecks, we also deployed the `QProbe` server on 34 PlanetLab servers that had bottlenecked WAN paths[3] and conducted 256 `QProbe` runs from them. The average throughput the `QProbe` client received from the PlanetLab servers is 819 Kbps.

To minimize interference from other traffic in the same cell, we ran all our experiments late at night. To ensure the load packets are not accounted for byte usage of the cellular-connected client, we had to set the TTL to $hop - count - 2$.

Figure 4.6 depicts the *stretch-factor* and the number of back-to-back packets of all the runs we conducted for a 4 ms inter-packet spacing for the probe packets. This plot verifies our simulation results, in that we see more back-to-back packets and a small *stretch-factor* in the presence of a wireless bottleneck, whereas a wired bottleneck result in fewer back-to-back packets and a higher *stretch-factor*.

Using the thresholds obtained from the training phase (Section 4.3.2), `QProbe` algorithm classified 94.7% of the 489 runs, while 5.3% of the runs did not satisfy the two threshold conditions, and thus remained unclassified for reasons described in Section 4.3. For the runs that were classified, the accuracy of detecting both wired and wireless bottlenecks is more

[3]We verified this by running TCP measurements with these servers using a high-bandwidth wired link from the same service provider as the cellular ISP.

Table 4.1: Summary of the dataset.

#Users	642
Data collection period	2 months
#QProbe runs	8116
#Countries	33
#Cellular providers	51

than 97.4%, thus showing that with a simple, easy to measure client-side algorithm, we can accurately detect the bottleneck location.

Note that in our evaluations, we calculate `QProbe`'s accuracy with respect to the number of classified runs and not the total runs. This is because `QProbe` is not capable of detecting some conditions like bottlenecks inside the cellular operator's network, as explained in Section 4.3.2. This makes using the total runs as the baseline unsuitable.

For validation, we also ran `QProbe` without background traffic at night time and collected 833 runs for 3G and 989 runs for LTE. `QProbe` indeed classified only 8.2% and 1.9% as wireless bottlenecks for 3G and LTE, respectively, since these periods are lightly loaded.

4.4.3 Large-scale Measurement Study

We developed `QProbe` as an iOS application on both iPads and iPhones, and deployed it in the Apple App Store [73]. To detect bottlenecks in different end-to-end paths, we used 15 well-provisioned Microsoft Azure servers that were deployed in different data centers, located in the US, South America, Europe, Singapore, China, Japan, and Australia. We also deployed the `QProbe` server on 51 geographically spread PlanetLab servers.

We collected data from this deployment via users of the `QProbe` iOS application. We advertised the app through social platforms as well as Amazon MTurk to get more participation [4]. Table 4.1 summarizes our dataset. We have made our dataset publicly available at [72].

[4]We obtained IRB approval for this study.

Table 4.2: `QProbe` runs for different radio technologies and the number of runs that were bottlenecked by either the wireless link or the WAN path as determined by the groundtruth.

Technology	Runs	Wireless Bottlenecks	WAN Bottlenecks
3G	2573	215 (8.4%)	97 (3.8%)
LTE	5480	441 (8.1%)	837 (15.3%)

The 8116 runs were spread across different cellular radio access technologies as shown in Table 4.2. We only analyzed data from the dominant radio technologies present in this study: 3G (WCDMA, HSDPA, and EVDO RevA) and LTE.

When users run the `QProbe` app, the app connects to the two closest Azure servers, referred to as reference servers, based on TCP RTTs. Then, the app conducts bulk TCP download measurements with these two reference servers sequentially. Next, the app runs `QProbe` experiments from 6 randomly chosen servers sequentially. Thus, each run of the app provides 6 `QProbe` measurements. For each of these 6 servers, the application first does TCP throughput measurements and then receives 5 `QProbe` packet trains, having probe packet spacings of 4 ms to 8 ms in increments of 1 ms. It logs the arrival timestamps of the probe packets and uploads these timestamps, along with throughput measurements and carrier information like the MNC, MCC codes, and the operator's name to a central server. We analyze this data offline to evaluate `QProbe`'s bottleneck detection accuracy.

Obtaining the Groundtruth: We use multiple throughput measurements conducted during each run of the app to estimate ground truth. The basic idea is that whenever low throughput is consistently observed with both reference servers and the server the `QProbe` train is sent from, we blame it on the last-mile wireless link. Otherwise, we blame the specific instance of low throughput on the WAN path from that specific server. To determine what constitutes low throughput, we looked at all throughput measurements obtained for 3G and LTE networks separately, and chose the 25th percentile as the threshold (732 kbps and 3149 kbps for 3G and LTE, respectively).

Note that this is not the most ideal way of obtaining the groundtruth due to several reasons. First, all our throughput measurements can be low due to a common bottlenecked

Table 4.3: Confusion matrix for 3G. The overall bottleneck detection accuracy is 85.2%.

		`QProbe` Classification	
Ground Truth		Wireless	WAN
Wireless	187	161 (86.1%)	26 (13.9%)
WAN	76	13 (17.1%)	63 (82.9%)

Table 4.4: Confusion matrix for LTE. The overall bottleneck detection accuracy is 86.6%.

		`QProbe` Classification	
Ground Truth		Wireless	WAN
Wireless	330	307 (93%)	23 (7%)
WAN	708	116 (16.4%)	592 (83.6%)

WAN path link and not only because of a bottlenecked cellular link as assumed above. Second, using thresholds on throughputs to decide what constitutes a bottlenecked link might not be accurate because the throughput available in cellular networks might vary significantly. However, as we did not have access to the individual devices that took part in our measurement study, this is the best strategy that we could use to reason about the groundtruth. In fact, if we had a more accurate way of obtaining the groundtruth, we believe that the accuracy of `QProbe` in this measurement study would be even higher, as shown in our controlled experiments.

Bottleneck Detection Accuracy: Using the threshold parameters obtained from training, we classified the bottlenecked runs in the measurement study. `QProbe` classified 84.3% and 81.2% of 3G and LTE runs, respectively. For those that were classified as either wireless or WAN path, the overall accuracy of bottleneck detection is over 85% for both 3G and LTE. Table 4.3 and Table 4.4 provide the confusion matrix for 3G and LTE, respectively.

The Need for Two Metrics: Note that if we were to use the number of back-to-back packets or the *stretch-factor* in isolation to classify the runs, all of them can be classified. However, in doing so, the classification accuracy drops significantly. For example, if `QProbe` used only the number of back-to-backs or the *stretch-factor* for classification, the accuracy reduces by 14.4% and 15.5%, respectively. For 3G, the two corresponding accuracy reductions are 14.6% and 21.7%. Therefore, though the two metric based classification does not classify a fraction of the runs, it achieves a far higher classification accuracy than using a single metric. Furthermore, using the two metrics allow `QProbe` to not classify runs that are bottlenecked due to reasons that `QProbe` is incapable of detecting.

4.5 Chapter Summary

Cellular-connected users often blame poor end-to-end network path performances on the last-mile wireless connection. However, as shown in our measurement studies in the previous chapter, the WAN path can be the bottleneck in many cases. In this chapter, we presented `QProbe`, a simple but effective tool that can be used by cellular clients to detect bottlenecks in end-to-end network paths. `QProbe` locates the bottleneck location by leveraging the unique characteristics of the PF-schedulers used in cellular basestations. While being a lightweight probing technique requiring only $\sim$ 700 ms to run and less than 4 KB data usage by the probe packets, `QProbe` locates the bottleneck in real-world 3G and LTE networks with more than 85% accuracy.

Chapter 5

Detecting Bottlenecks in Wi-Fi Communication

5.1 Introduction

Wi-Fi networks are inherently dynamic in nature due to the over-the-air broadcast mode of operation. This may potentially introduce many pathological conditions in the network that will lead to performance problems in user applications. For example, if many clients are connected to the same Wi-Fi access point (e.g., in a café that provides free Wi-Fi for its patrons), the clients' traffic will cause congestion to manifest at the router causing buffer bloats. Alternatively, due to channel fading, clients might be a connected to a Wi-Fi connection with poor RSSI. Such conditions cause poor performance in applications because the capacity of the Wi-Fi link will be reduced under such conditions. The consequences are more severe in delay-sensitive applications, such as real-time streaming, because the imposed quality degradations on such applications are hard to mask from users (*e.g.,* lagging audio or video in Skype calls).

However, as we discussed in Chapter 3, though Wi-Fi connected clients experience poor quality VoIP streams more than LAN-connected clients, the Wi-Fi link might not be the bottleneck link all the time because in some cases, it is probable that the bottleneck link is in the Internet path. It is important to know whether the Wi-Fi link is the bottleneck in an end-to-end network path because this knowledge helps user applications perform better. For example, real-time streaming applications can tune bandwidth adaptation based on whether the Wi-Fi link is bottlenecked or not to increase call quality (as we discuss later in Chapter 6).

Therefore, prior work has looked at detecting the presence of pathological conditions in the Wi-Fi link to determine whether the Wi-Fi link is bottlenecked (*e.g.,* [57, 19, 77, 90]). However, such techniques are hard to realize in practice for two major reasons: they require the Wi-Fi adapter to operate in monitor mode to allow it to scan the whole Wi-Fi spectrum (*e.g.,* [77]); or they require special infrastructure to function, like a special server connected to the Wi-Fi access point (*e.g.,* [57]), or a Wi-Fi access point running a modified operating system (*e.g.,* [90]).

Note that the `QProbe` tool we presented in the previous chapter does not work in Wi-Fi networks because it is designed specifically for cellular networks, wherein we exploited the per-device queue management and PF scheduling properties of cellular base stations to detect the bottleneck location. As Wi-Fi access points do not have those unique properties, `QProbe` will not be able to detect bottlenecks on Wi-Fi-connected clients. Moreover, our focus in this chapter is not only on detecting whether the Wi-Fi link is bottlenecked, but also on identifying the root cause of the bottleneck conditions.

To this end, we present a user-level module, named `Kwikr`, that is capable of deducing whether the Wi-Fi link is bottlenecked by detecting the presence of Wi-Fi pathological conditions such as wireless congestion, mobility, handoffs, and weak signals. The key novelty of `Kwikr` is that its detectors leverage simple active and passive measurements that can be performed on all types of off-the-shelf devices (*e.g.,* laptops, smartphones, tablet-PCs, *etc.*) and operating systems (OSs) (*e.g.,* Windows, Linux, Android, iOS, *etc.*), and does not require modified or additional hardware. After `Kwikr` completes a run of its detectors, it packages the detectors' outputs in a form that we call "Wi-Fi hints" that are passed onto applications that are interested in being aware of the status of the Wi-Fi link.

To detect the presence of wireless congestion, *i.e.,* queue build-up, at the Wi-Fi access point, `Kwikr` uses a novel, lightweight, active measurement technique that we call `Ping-Pair`. In `Ping-Pair`, the client sends two ICMP ping messages with differing Differentiated Service Code Point (DSCP) values to its default gateway, *i.e.,* the Wi-Fi access point. We show that the RTT difference of the pair of pings can be used as a reliable signal to detect congestion with high accuracy. To detect mobility, handoffs, and weak links, we use standard, straightforward inferences based on passively observable signals like RSSI and the BSSID the device is connected to.

We make three main contributions in this chapter:

1. We have designed a user-level module, named `Kwikr`, that first detects various types of Wi-Fi pathological conditions, such as congestion, mobility, handoffs, and weak links that cause a Wi-Fi link to get bottlenecked. It then informs applications about the status of the Wi-Fi link using Wi-Fi hints.
2. We implement `Kwikr` as a user-level module that can be deployed on a wide variety of devices and operating systems.
3. Through empirical studies, we show that `Kwikr` is capable of detecting the presence of the above mentioned pathological conditions with over 90% accuracy.

5.2 Background

In this section, we give a brief introduction to the Differentiated Services Code Point (DSCP) field of the IP header and how it enables differentiated services for packets in the Internet. We also discuss how DSCP values are dealt with at Wi-Fi access points that are Wi-Fi Multimedia (WMM) enabled. We base the design of our `Ping-Pair` technique (discussed later in Section 5.3.1) on these principles.

5.2.1 DSCP and DiffServ

The IP header contains a 8 bit field that was originally called Type of Service (ToS). Newer standards redefine the ToS filed as a combination of two fields: one that carries a 6 bit Differentiated Services Code Point (DSCP) field, and one containing a 2 bit Explicit Congestion Notification (ECN) field. The DSCP field, defined in RFC 2474 [41], is used for Differentiated Services (DiffServ) in networks, including the Internet, to provide different types of services for different classes of network traffic. For example, real-time streaming applications can mark the DSCP field of its packets to request high priority services with

Table 5.1: Commonly used DSCP values.

DSCP	**Decimal**	**Meaning**	**Drop probability**	**Equivalent IP precedence value**
101 110	46	Expedited forwarding (EF)	N/A	101 Critical
000 000	0	Best effort	N/A	000 - Routine
001 010	10	AF11	Low	001 - Priority
001 100	12	AF12	Medium	001 - Priority
001 110	14	AF13	High	001 - Priority
010 010	18	AF21	Low	010 - Immediate
010 100	20	AF22	Medium	010 - Immediate
010 110	22	AF23	High	010 - Immediate
011 010	26	AF31	Low	011 - Flash
011 100	28	AF32	Medium	011 - Flash
011 110	30	AF33	High	011 - Flash
100 010	34	AF41	Low	100 - Flash override
100 100	36	AF42	Medium	100 - Flash override
100 110	38	AF43	High	100 - Flash override

the goal of keeping latencies lower.

The design rationale of DiffServ is simple. It requires traffic to be distributed into multiple classes, and the routers in the network that support DiffServ simply implement a forwarding policy for each class of traffic. So in essence, if an application sets the DSCP field of its packets to high priority, when they reach a DiffServ-enabled router they get higher precedence in the scheduling process. RFC 2475 [42] outlines the commonly used DSCP values shown in Table 5.1. The two ICMP ping messages in our `Ping-Pair` technique use the DSCP values of 46 and 0.

It is worthwhile noting that DiffServ policies are typically honored only within certain network boundaries. Therefore, though DSCP policies might be in effect within a home or an office network, when the packets go out to the Internet, some ASs will either reset the DSCP field to the default value (0) or else treat every traffic class as the same irrespective of the set DSCP value.

5.2.2 IEEE 802.11e Standard

The IEEE 802.11e standard proposes a set of QoS enhancements to the IEEE 802.11 standard through the Enhanced Distributed Channel Access (EDCA) mechanism. EDCA supports four traffic classes (also called as Access Categories). In increasing order of priority, the four Access Categories are: Background, Best Effort, Voice, and Video. This categorization allows Wi-Fi stations to wait for differing amounts of time before they try to access the channel. For example, a station with high priority (*e.g.,* Voice) traffic would wait less than a station having low priority (*e.g.,* Best Effort) traffic before they attempt to send frames.

The Wi-Fi alliance's Wi-Fi Multimedia (WMM) specification contains a subset of enhancements proposed by the 802.11e standard. Wi-Fi access points that are WMM-enabled, support EDCA and maintain 4 queues in the downlink direction—one for each access category. The downlink scheduling policy of the access point always serves frames in the high priority queues first before serving frames in the low priority ones.

A key assumption of our congestion detector, `Ping-Pair`, which we discuss in the next section, is that the Wi-Fi access point is WMM-enabled. This is increasingly the case because, since the 802.11e standard came out in 2005, it has been widely adapted by most Wi-Fi access point manufactures. Typically, on WMM-supported access points, WMM is enabled by default. This is the case in OpenWRT as well. As users typically do not modify their default access point settings, we believe that depending on WMM-enabled routers does not inhibit the practicality of the `Ping-Pair` technique. We conducted a measurement study to evaluate this hypothesis in which we analyzed 171 unique Wi-Fi access points of all types of makes and models across 14 countries belonging to 166 users, and found that 77.2% of them are WMM-enabled[1].

5.2.3 DiffServ and WMM

When a WMM-enabled router receives a frame carrying an IP packet with the default DSCP value, it will be queued in its Best Effort queue. Likewise, if the IP packet carries

[1]We obtained IRB approval for this study.

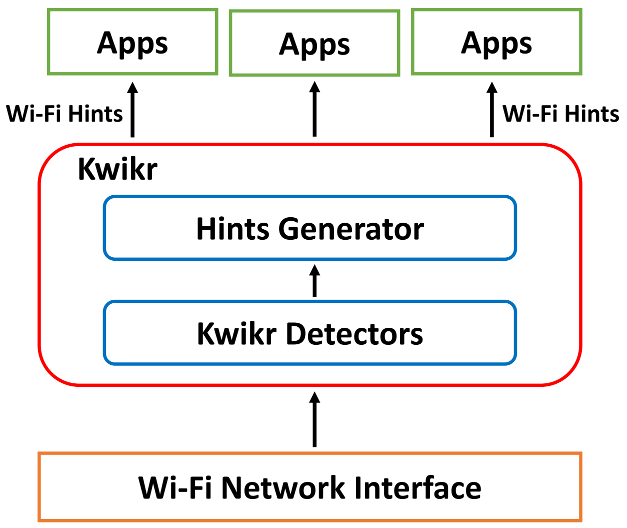


Figure 5.1: A high-level architectural diagram of `Kwikr`. `Kwikr` first detects the presence of various pathological conditions in the Wi-Fi link via its detector suite. Then, it generates information about the status of the Wi-Fi link in the form of Wi-Fi hints. These hints are then passed onto applications.

the Expedited Forwarding DSCP value, then it gets queued in the Video queue. As the scheduling policy gives precedence to the Video queue over Best Effort, any frames sitting in the Video queue get scheduled first to provide the lower latencies that DiffServ requires.

We exploit this behavior in our design of the `Ping-Pair` technique, which will be discussed next in Section 5.3.

5.3 Design

We now discuss the design elements of `Kwikr`'s suite of detectors and how they are exposed to end-user applications in the form of *Wi-Fi hints*. We design `Kwikr` as a network module that runs as a user-level library. This mode of operation allows us to deploy it on a wide variety of platforms and operating systems such as Windows, Linux, and even more restrictive ones like Android and iOS. Once invoked, the `Kwikr` module runs the detectors and Wi-Fi hints generators continuously at an interval specified by an application that requires Wi-Fi hints.

A high-level architectural diagram of `Kwikr` is shown in Figure 5.1. `Kwikr` first gets raw data from the Wi-Fi network interface that includes RTT measurements of ICMP ping messages, RSSI, and BSSID data. Then, `Kwikr`'s detector suite consumes this data and detects the presence of various pathological cases, such as wireless congestion, mobility, handoffs, and weak links, that cause the Wi-Fi link to be bottlenecked. Next, it runs simple inferences on the detectors' results and generates useful information, which we call Wi-Fi hints, about the status of the Wi-Fi link. Finally, these hints are passed onto applications that are interested in changing their behavior depending on how good or bad the Wi-Fi link is. For example, in the in the next chapter (Chapter 6), we discuss how real-time streaming applications can leverage Wi-Fi hints to improve their bandwidth adaptation strategies of data streams.

In Section 5.3.1, we first discuss the design aspects of the `Kwikr` detectors. Next, we discuss how Wi-Fi hints are inferred in Section 5.3.2.

5.3.1 Pathology Detectors

When many clients are connected to the same access point (*e.g.,* in a café, airport), the throughput that a data flow would observe in the Wi-Fi link reduces. Moreover, delay-sensitive traffic (*e.g.,* real-time streams) would observe increased jitter and losses. This is because, the clients' data flows will cause increased queuing delays as frames have to wait in the access point's queue to get served. This is what we call congestion. Our goal is to detect this phenomenon by using a simple, lightweight active measurement technique that a user can run on any device and operating system. We present a measurement technique, named `Ping-Pair`, that achieves this goal by using a pair of carefully crafted ICMP ping messages.

In addition, we make use of passive measurements to detect mobility, handoffs and weak link conditions in the Wi-Fi link. These pathologies reduce the throughput that a data flow would observe in the Wi-Fi link as well due to changing channel conditions, temporary disruptions, and channel fading. We detect these conditions by using standard inferences on passively observable signals like RSSI and BSSID of the Wi-Fi connection. While these detectors are straightforward applications of well-known techniques, the combining them and our `Ping-Pair` detector to design one cohesive, user-level detector suite is something that has not been addressed in prior work.

Design Requirements

Our focus is to design a suite of detectors that is capable of detecting wireless congestion, mobility, handoffs, and weak link pathological conditions. For the purpose of practicality, we do not require the Wi-Fi adapter to operate in monitor mode or special-purpose hardware. Therefore, our goal is to design techniques that an end user can run on any type of device or operating system—a capability that is not present in the state-of-the-art.

For the techniques to potentially benefit user applications (*e.g.,* real-time streaming applications), our detectors need to be able to run as a user-level library. Therefore, we do not assume elevated OS privileges, and this allows us to deploy our techniques on stock devices, such as non-rooted Android smartphones.

The Ping-Pair Technique

To detect the presence of wireless congestion, *i.e.,* queue build-up at the Wi-Fi access point, we present a simple, active measurement technique that we call `Ping-Pair`. It uses a carefully crafted pair of ICMP ping messages to perform the detection, and hence the name.

The `Ping-Pair` technique works as follows. We send two ICMP ping requests to the client's default gateway, which is the Wi-Fi access point the client is connected to. One of the requests carries a DSCP value set to high priority and the other's DSCP value is set to default priority. The access point will then send ICMP ping responses to both requests and we use the arrival times of the responses to compute the round trip times (RTTs) for both. Next, we take the difference of the two RTT values and compare it with a threshold. If the difference is higher than the threshold, we conclude that the Wi-Fi link is bottlenecked. This process is outlined in Algorithm 5.1.

Let us now discuss how this technique works. The `Ping-Pair` technique solely relies on the DiffServ and WMM standards we discussed in Section 5.2. First of all, note that the `Ping-Pair` ICMP messages are sent and received only on the last-mile Wi-Fi network. Therefore, this technique helps us study the performance of the Wi-Fi link in isolation. This would not be the case if we were to use any infrastructure in the Internet, like a cloud server.

Algorithm 5.1: The `Ping-Pair` Algorithm

```
1  $m1 \leftarrow$ createIcmpPingRequest()
2  $m1.DSCP = 46$ //Expedited Forwarding (high priority)
3  $m2 \leftarrow$ createIcmpPingRequest()
4  $m2.DSCP = 0$ //Best Effort (default priority)
5  //Send the ping requests and record the sent timestamps
   $ts_{m1} \leftarrow$ sendIcmpPingRequest($m1$)
6  $ts_{m2} \leftarrow$ sendIcmpPingRequest($m2$)
7  //Receive the ping responses and record the received timestamps
   $tr_{m1} \leftarrow$ receiveIcmpPingResponse()
8  $tr_{m2} \leftarrow$ receiveIcmpPingResponse()
9  //Calculate the RTTs of the ping messages
10 $RTT_{m1} \leftarrow tr_{m1} - ts_{m1}$
11 $RTT_{m2} \leftarrow tr_{m2} - ts_{m2}$
12 $RTT\_difference \leftarrow RTT_{m2} - RTT_{m1}$
13 if $RTT\_difference > RTT\_difference\_threshold$ then
14     return Wi-Fi link is congested
15 end
16 else
17     return Wi-Fi link is not congested
18 end
```

Next, as the ICMP messages are confined to the Wi-Fi network, the DiffServ policies are honored as the DSCP values do not get overwritten, provided that the Wi-Fi access point is WMM enabled.

When the access point receives the two ICMP ping requests, it creates the corresponding responses by carrying over the headers of the requests. The responses are then scheduled on two different queues based on the DSCP values that are set on the IP headers. Hence, the high priority ICMP response gets scheduled in the Video access category queue and the default priority ICMP response in the Best Effort queue. Note that the traffic coming through the WAN interface to the access point, destined to the clients in the Wi-Fi network, also gets queued in the Best Effort queue. Why is this? As we discussed in Section 5.2.1, DiffServ policies are typically not honored in the Internet, which essentially is a cross-boundary DiffServ domain. Therefore, all incoming packets to the access point get queued in the Best Effort queue as the DSCP field would have been overwritten to the default value by some hop on the WAN path irrespective of the DSCP value the sender set at transmission time[2]. This means that the ICMP response for the high priority request gets served immediately as the Video queue is almost always empty and it is given precedence by the scheduling algorithm. The ICMP response for the default priority request gets served only after the frames that were queued before it in the Best Effort queue are transmitted.

Let us assume that the Wi-Fi link is not congested. In this case, both Video and Best Effort queues will be empty or contain a few frames. Therefore, the RTTs of the `Ping-Pair` messages will be almost identical, and hence their difference will be low. Now let us assume that the Wi-Fi link is congested, implying that the Best Effort queue contains a significant number of frames that are waiting to be transmitted. In this case, the high priority ping request will get its response promptly but the default priority request's response will be delayed by the time the access point requires to serve all the frames that were queued before it. Therefore, this would cause the difference of the RTTs of the `Ping-Pair` messages to be high.

Note that there are two major factors that make the `Ping-Pair` technique effective. First, WMM is widely supported worldwide, as we discussed earlier in Section 5.2.2. Next, DiffServ is rarely used by applications as they do not gain anything from doing so because DSCP values get set to default when packets cross AS boundaries.

To validate the `Ping-Pair` technique, we varied the number of TCP flows in a Wi-Fi network from 0 to 6 to create differing amounts of congestion in the network, and then monitored the number of frames that were in the access point's downlink Best Effort queue. Simultaneously, we conducted the `Ping-Pair` test on a laptop connected to the same Wi-Fi access point.

[2]For the same reason, applications typically do not bother setting the DSCP field of outgoing packets as DiffServ does not come into effect when the packets traverse WAN paths.

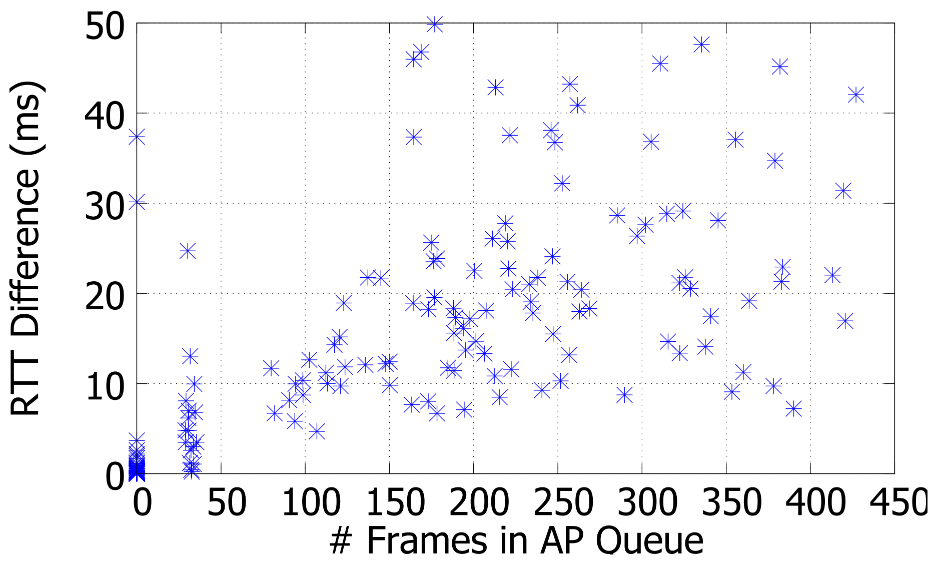


Figure 5.2: RTT difference of `Ping-Pair` for varying number of frames queued in the access point's downlink Best Effort queue. The RTT difference is high when the queue contains a high number of packets, and vice versa.

Figure 5.2 clearly shows that the RTT difference is high when the access point's queue gets backlogged. So by using a threshold on the RTT difference, we can detect the presence of wireless congestion by using the `Ping-Pair` technique.

Detecting Other Pathological Conditions

The three other pathological conditions that `Kwikr` detects are: mobility, handoffs, and weak links. The detection of these are straightforward and can be done by using passively observable and easy-to-obtain measurements.

Mobility. Mobility can arise especially on hand-held Wi-Fi devices such as smartphones. When clients are mobile, the wireless link characteristics change over time and this may cause poor user experiences in applications. For example, in Chapter 6, we show that mobility can cause the quality of Skype calls to become poor. Therefore, we incorporated a RSSI-based mobility detection technique in `Kwikr`. The idea behind it is simple. As shown in Figure 5.3, the RSSI profiles of stationary and mobile clients are quite different. Hence, we can detect mobility by computing the variance of the RSSI values observed in a small time window and

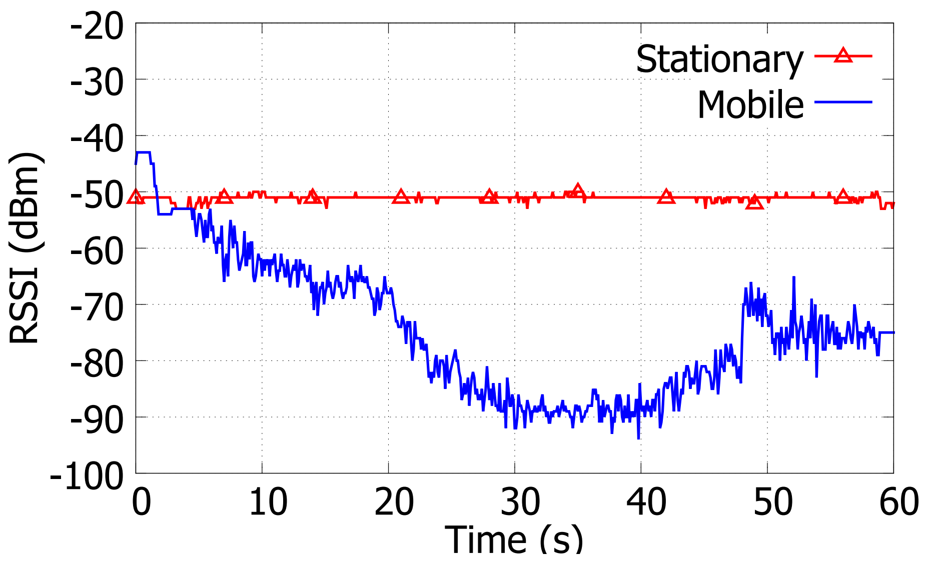


Figure 5.3: RSSI profiles for mobile and stationary clients. RSSI profiles are quite different on mobile and stationary clients, implying that frequent fluctuations of RSSI can detect mobility.

repeating this while sliding the window forward. This forms a reliable mobility detection technique, albeit a slow one since we have to observe the RSSI fluctuations for a few seconds to detect mobility with high accuracy.

We can of course use other sensors like Accelerometers available on smartphones to detect mobility, as is done in [76]. However, we do not investigate this approach in this work for two reasons: (i) continuous use of sensors reduces the energy efficiency of the detector and `Kwikr` is meant to be run continuously while an application that uses it is in operation; (ii) additional sensors might not be available on all types of devices (*e.g.,* laptops), and hence relying on them would violate our design requirements. Prior work has also looked at detecting mobility using PHY layer information: Channel State Information (CSI) vector and Time-of-Flight (ToF) [88]. However, in their approach, the access point determines the mobility status and uses that to improve client roaming, rate control, frame aggregation, and MIMO beam-forming. This is not applicable in our setting, because we require end-user applications to detect the mobility status and as they operate in the application layer, they do not have access to PHY layer information.

The advantage of using RSSI for mobility detection is that it is possible to obtain it on all devices and OSs via simple API calls. Moreover, as it is a passive measurement, it does not cost anything extra to retrieve it as modern OSs monitor RSSI continuously.

Handoffs. A handoff in Wi-Fi occurs when a client disassociates with one access point and associates with another. This is quite common in enterprise networks where one entity has deployed and maintains multiple access points under the same SSID, *e.g.,* in universities, offices, airports, *etc.* Handoffs is a useful feature in Wi-Fi that allows clients to move freely without having to worry about manually connecting to a new Wi-Fi access point every time they lose connectivity.

Albeit useful, Handoffs cause a temporary outage in packet reception at the client because it takes some time for a handoff to complete, *i.e.,* to disassociate from the previously connected access point and connect to a new one. This outage causes poor user experiences, especially in real-time streaming applications, as we show in Chapter 6. Therefore, it is useful to know when a handoff occurs because applications can use this information to change the way they recover from the transient network outage caused by a handoff such that they improve user experience.

In `Kwikr`, we use a simple technique to detect handoffs. Every Wi-Fi access point has a globally unique MAC address called the BSSID. When a client connects to an access point, the client stores the BSSID of the access point it is connected to in its connection properties. It is possible to retrieve this parameter on most OSs with simple API calls (*e.g.,* Windows, Linux, Android and iOS). Hence, our handoff detection technique is to simply monitor whether the connected BSSID value changes over time.

Weak Links. A client's Wi-Fi connection becomes weak when the client is at the edge of connectivity of the access point. Due to channel fading, if a client is at the edge of connectivity, user applications will experience poor effects because the Wi-Fi link might lose frames due to bit errors that occur in frames during transmission.

Detecting that a client's Wi-Fi connection is weak can be done by monitoring the RSSI of the Wi-Fi signal. In `Kwikr`, similar to our mobility detection technique, we monitor the average

RSSI value of a time window. When the average RSSI observed in a certain window is less than a threshold, `Kwikr` outputs that the link is weak. To perform continuous detection, we repeat this approach by sliding the window forward.

The threshold we use is the RSSI value at which the TCP throughput of a TCP flow drops by more than 75% relative to the throughput at a strong-link location. In most networks and settings we conducted this trial at, we noticed that this value is -80 dBm, and hence use that in our evaluations.

5.3.2 Wi-Fi Hints Generation Process

After the detector suite completes a run, `Kwikr` uses the detectors' results and generates useful information about the status of the Wi-Fi link that in turn is passed onto applications that are interested in obtaining this information. We call such information as *Wi-Fi hints*. `Kwikr` provides the below mentioned hints to applications.

- The onset and conclusion timestamps of congestion in the Wi-Fi link. This hint enables applications to be aware about time frames during which the Wi-Fi link is congested.
- In a congested time period, the fraction of queuing delay at the Wi-Fi access point that is self-inflicted by the flow. Using this hint, an application can determine its data flow's contribution to the total queuing delay it observes. For example, a low contribution factor implies that the network is congested due to cross-traffic flows.
- Client mobility status—mobile or stationary. This hint allows applications to change their behavior depending on whether the client is mobile or not.
- Wi-Fi link strength—strong or weak. Applications can dynamically adapt their behavior depending on the signal strength of the Wi-Fi connection.
- Timestamp at which the last handoff occurred. This hint enables applications to disregard any transient connectivity drops they observe that coincide with handoff events.

We discuss how each of the aforementioned Wi-Fi hints are generated and passed onto applications in this section. We defer the discussion of a concrete application of Wi-Fi hints, which shows how Wi-Fi hints can be used to improve the quality of real-time streams, to Chapter 6.

Onset and Conclusion of Wi-Fi Congestion

As explained earlier, `Kwikr` uses the `Ping-Pair` technique to infer the presence of congestion (Section 5.3.1). At the first time instant it detects that the Wi-Fi link is congested, it generates a binary hint that carries the congestion flag set to true and the timestamp at which congestion was first detected. When the `Ping-Pair` technique observes that the Wi-Fi link's congestion has concluded, a new hint is generated with the binary congested flag set to false and the timestamp at which it was detected. These two hints allow applications to be aware of congestion episodes in the Wi-Fi link and adapt to such conditions accordingly.

Self-Inflicted Queuing Delay Fraction

When a Wi-Fi link is congested, it manifests in the form of increased delays and packet losses at the clients. The `Ping-Pair` technique helps us find the total queuing delay at the Wi-Fi access point when a Wi-Fi link is congested. This hint lets an application know how much of its traffic contributed to the queuing delay. Such information is of great importance in real-time streaming applications, where bandwidth adaptation strategies can be tweaked to use more or less bandwidth depending on the real-time stream's contribution to the total queuing delay. We use the *Queue Occupancy Metric (QOM)* to reason about an application's contribution to the queuing delay and it is calculated based on the `Ping-Pair` technique's output. The QOM value is sent to the application as a hint during congested time periods so that it can change its behavior accordingly.

Let's discuss an example to see how the QOM metric is calculated. Assume that Skype is using `Kwikr` and detects that there is congestion, and it wants to know how much it is contributing to the congestion. Consider the access point queue depicted in Figure 5.4. It contains two types of packets: blue-colored Skype packets destined to the client running a Skype flow, and red-colored packets, labeled A-E, destined to other clients in the network.

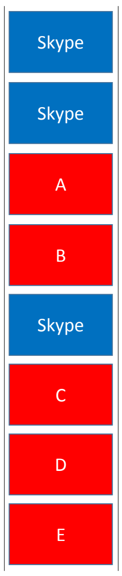


Figure 5.4: An example of a queue at a Wi-Fi access point. The blue packets are destined to a client running a Skype flow and the red packets (labeled A to E) belong to flows generated by other clients of the network.

In this case, our goal is to allow Skype to infer that it is responsible for a smaller fraction of the overall queuing delay because the majority of the packets in the queue are cross-traffic. We do this as follows.

Recall that the `Ping-Pair` technique returns the total time (say T) that was required by the access point to serve all the packets that were sandwiched between the ping responses in the queue.

$$T = \text{time to serve Skype packets (say } T_S) + \text{time to serve other packets (say } T_O) \quad (5.1)$$

T_S can be estimated by taking the summation of the transmission time required for the Skype packets in the queue and their channel access delays. For simplicity, we assume that the per-packet channel access delay (α) is the same for each packet.

Let s_i be the size of the i^{th} Skype packet. To calculate T_S, Skype needs to keep track of the packets it receives during the time period in which a `Ping-Pair` sample is collected. This can be easily achieved via timestamps. Assume that the first and the last packet that Skype receives during the `Ping-Pair` period are j and k, respectively. Say the MAC-layer data rate of the Wi-Fi link is D. Then,

$$T_S = \frac{\sum_{i=j}^{k} s_i}{D} + (k - j + 1) \times \alpha \tag{5.2}$$

Next, we compute QOM as follows.

$$QOM = \frac{T_S}{T} \tag{5.3}$$

To estimate α, we conducted experiments in which we varied the load on a Wi-Fi link and computed the channel access delay using a Wi-Fi channel sniffer. In our evaluations, we use a fixed value of 0.125 ms for α as a heuristic. D can be obtained on most operating systems via a simple system call.

As it is practically challenging to estimate α on the fly, our simple strategy of using a fixed value we obtained experimentally is good enough for several reasons: (i) usually α is non-negligible when compared to the transmission delay of packets; (ii) in the case that α is comparable to the transmission delay because D is large, then we do not expect the Wi-Fi link to be bottlenecked in the first place. Therefore, our heuristic would suffice in most settings.

Putting everything together, `Kwikr`, after obtaining a `Ping-Pair` sample, gathers the other parameters required in Equation 5.2. Next, it computes QOM using Equation 5.3 and returns it as a hint that can be used by applications, for example to do Wi-Fi-aware bandwidth adaptation as we will discuss in detail in Chapter 6.

To validate the QOM metric, we congested a Wi-Fi network by using competing TCP cross-traffic flows, and made a laptop receive CBR flows of varying data rates (200 kbps to 1200 kbps, in increments of 200 kbps), one after the other. The laptop collected 25 `Ping-Pair` samples while each flow was being received and the QOM metric for each sample was computed as discussed above. Figure 5.5 depicts the average QOM percentage for each

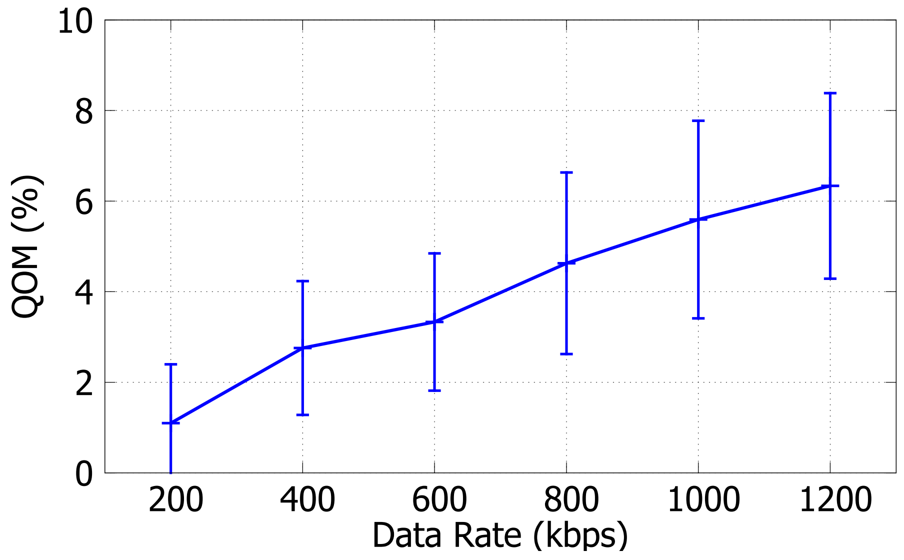


Figure 5.5: QOM for CBR flows of varying data rates in a congested Wi-Fi network. At higher data rates, the observed QOM percentages are higher because high data rate flows send more packets that would increase the number of packets in the access point's queue.

of the data rates we used. As expected, the QOM percentage increases with the data rate. Note that QOM does not double when we double the data rate, because doubling the data rate affects both T_S and T in Equation 5.3.

Mobility, Link Strength, and Handoffs

To generate these hints, we use the outputs of the `Kwikr` detectors directly. As discussed earlier, `Kwikr` uses its RSSI and BSSID based mobility, weak links, and handoffs detectors, and the output of these detectors are passed as hints to the interested applications. In the case of mobility and link strength, the hints are sent in a binary form—mobile/stationary for mobility and strong/weak for link strength. In the case of a handoff, it sends the timestamp at which the handoff occurred.

5.4 Implementation

We implemented the `Kwikr` module using C#.Net and tested it on Windows 10 and Windows 8.1 operating systems. Additionally, we implemented it in C and tested it on Ubuntu 12.04, and Android 4.4 and higher. In all these OSs, our implementation was done as a user-level library that any application can use to get information about whether the Wi-Fi link experiences any pathological conditions (such as congestion, mobility, handoffs, and weak links) that would make the Wi-Fi link a bottleneck.

`Kwikr`'s congestion detection technique, `Ping-Pair`, operates by sending a pair of ICMP ping packets with different DSCP values. In all the operating systems we considered, a user-level application that needs to send ICMP messages has to do so by using raw sockets and creating the complete packet, with the headers (ICMP and IP) and payload, manually. However, this requires elevated security privileges because opening of raw sockets is a privileged operation. This inhibits deployability of `Kwikr` because it is designed to run in user-space even on platforms on which elevated privileges cannot be obtained, like Android and iOS.

To overcome this issue, we implemented the `Ping-Pair` technique using the inbuilt commandline ping tool that comes pre-installed on all OSs as an OS utility. The Ping tool executes with elevated privileges and can be invoked by applications that run in user-space. On all OSs, several commandline arguments can be passed in to the utility, and one of them is the DSCP value. For example, on Android and Linux, running 'ping <IP address> -Q 184', pings the specified IP address with the DSCP field set to Expedited Forwarding (high priority). On Windows, the equivalent command is 'ping <IP address> -v 184'. We defer a comparison of the congestion detection accuracy of raw sockets and the ping tool to the next section.

To detect mobility, handoffs, and weak links, the passive measurements we require are RSSI and the BSSID the client is connected to. On Windows, we obtain these parameters via the Native Wi-Fi API [70]. On Linux, these parameters are exposed through commandline utilities such as `iwlist` and `iwconfig`. The WifiInfo API [22] provides the RSSI and the BSSID of the connected Wi-Fi connection on Android.

In all our implementations, we provide APIs for applications to subscribe to Wi-Fi hints they are interested in receiving. We also provided an API through which the applications can pass in the payload size of each packet it receives. This data is required for the QOM metric calculation, as outlined in Section 5.3.2.

5.5 Evaluation

We evaluate the accuracy of the detector suite of `Kwikr` in this section. We do not evaluate the Wi-Fi hints generation process because it is merely a way of packaging the detector suite's results in a way that applications can make the best use of the information. The evaluations were carried out in different Wi-Fi environments in India and Singapore.

5.5.1 Wi-Fi Congestion Detection Accuracy

We start by studying the accuracy of the `Ping-Pair` technique in detecting congestion in the Wi-Fi network.

To obtain the groundtruth as to whether the Wi-Fi link is congested, *i.e.,* if there is a queue build-up at the access point, we instrumented OpenWRT (Chaos Calmer 15.05) to log the number of frames in the downlink queue at every time instant and installed it on a Netgear WNDR3800 Wi-Fi access point. This model supports WMM and it is switched on by default by OpenWRT. We deployed our C implementation of the `Ping-Pair` technique on a laptop running Ubuntu 12.04 and collected 30 `Ping-Pair` measurements. We repeated this process while increasing the number of TCP flows generated by other clients in the Wi-Fi network from 0 to 7 to create varying amounts of congestion. We conducted these experiments on both 2.4 GHz and 5 GHz bands as the Netgear WNDR3800 access point supported dual-band operation.

Note that during the time period a `Ping-Pair` sample is taken, the number of frames in the queue might change. Therefore, we look at all the OpenWRT log statements in this time period and classify that there is persistent queueing if more than 90% of the log statements show a non-empty queue. Otherwise, we classify the queue as an empty or a lightly-loaded queue. This serves as the groundtruth for our detector.

We discarded `Ping-Pair` samples in which any of the two pings had timed out. In the remaining samples, we calculated the RTT difference as discussed in Section 5.3.1. Next, we trained a 10-fold cross-validation decision tree classifier on this data and the groundtruth. The thresholds the classifier used are 3.4 ms for the 2.4 GHz band and 2.6 ms for the 5 GHz

Table 5.2: Confusion matrix for congestion detection via the `Ping-Pair` technique on the 2.4 GHz band. The overall congestion detection accuracy is above 90%.

		`Kwikr` Classification	
Ground Truth		Empty Queue	Persistent Queue
Empty Queue	116	106 (91.4%)	10 (8.6%)
Persistent Queue	117	11 (9.4%)	106 (90.6%)

Table 5.3: Confusion matrix for congestion detection via the `Ping-Pair` technique on the 5 GHz band. The overall congestion detection accuracy is above 95%.

		`Kwikr` Classification	
Ground Truth		Empty Queue	Persistent Queue
Empty Queue	104	98 (94.2%)	6 (5.8%)
Persistent Queue	135	5 (3.7%)	130 (96.3%)

band. We believe that this difference in thresholds is due the 5 GHz band being typically less "noisier" than the 2.4 GHz band.

Tables 5.2 and 5.3 show the confusion matrices of the `Ping-Pair` detector for the 2.4 GHz and 5 GHz bands, respectively. In both bands, `Kwikr`'s congestion detection accuracy is over 90%. The detector does make mistakes in scenarios where the transmission of ping requests or responses fail and are retransmitted by either the access point or by the client. In such cases, the `Ping-Pair` technique will output a high RTT difference while the access point queue might be empty. Also, the `Ping-Pair` messages will have a low RTT difference if the queue is lightly loaded *i.e.,* containing a very few number of frames. However, as we looked at whether the queue is empty or not as our groundtruth, such cases would be classified as persistent queuing cases, and that causes mistakes in the classifier. As shown in the confusion matrices, the occurrence of such mistakes is small. This shows that `Kwikr`'s `Ping-Pair` technique is a reliable detector of Wi-Fi congestion, albeit the simplicity and lightweight properties it entails.

To increase accuracy of the detector, multiple `Ping-Pair` samples can be taken at a given time instant. For example, in Chapter 6, when `Ping-Pair` is utilized, we take three samples and take the majority decision as the final detection result. This simple technique helps the detector be more robust and accurate.

5.5.2 Raw Sockets vs Ping Utility

We discussed in Section 5.4 that raw sockets cannot be used by `Kwikr` to send ICMP ping messages on platforms like Android that do not allow user-level applications and libraries to execute with elevated privileges. So in such platforms, we use the operating system's inbuilt commandline ping utility to implement the `Ping-Pair` technique. In this section, we compare these two approaches of implementing `Ping-Pair`, *i.e.,* raw sockets and ping utility.

To get raw sockets to function on Android, we flashed CyanogenMod, a customizable, open-source Android ROM, on a Samsung Galaxy S3 smartphone. On CyanogenMod, user applications can execute with elevated privileges because the installation of it 'roots' the phone. On the same phone, we modified our C implementation of `Ping-Pair` to conduct the `Ping-Pair` experiment using raw sockets and then by calling the inbuilt OS ping utility. Next, it recorded the RTT difference that each method calculated. We collected 200 such samples on Android both when the Wi-Fi link was not congested and congested by introducing TCP flows to the network.

Figure 5.6 represents the average of the 200 samples collected by the two approaches of implementing the `Ping-Pair` technique. As it clearly shows, both when the Wi-Fi network is not congested and is congested, the RTT difference they calculate and their variances are very similar. Therefore, we conclude that on platforms on which raw sockets cannot be used, we can make use of the OS-inbuilt ping utility to implement our `Ping-Pair` technique, thus making the detector widely deployable.

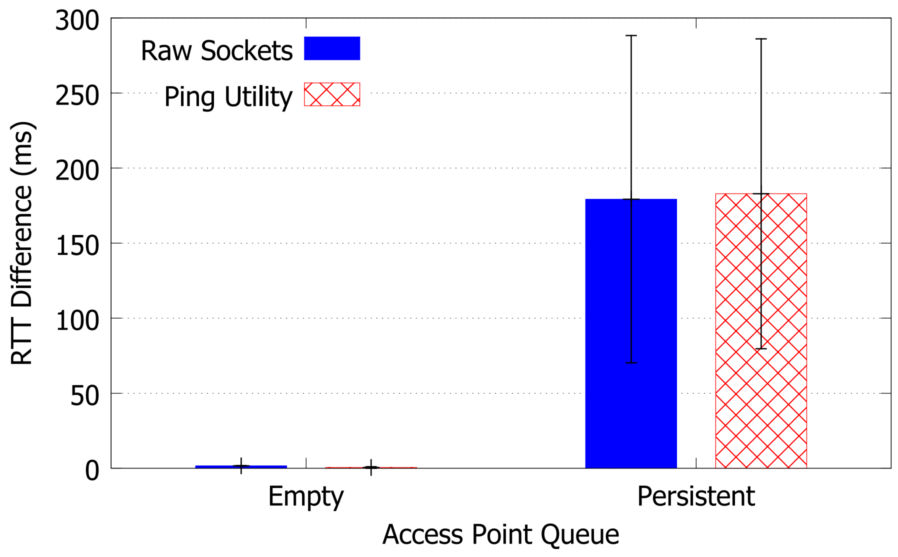


Figure 5.6: RTT difference of the `Ping-Pair` messages when using raw sockets and the OS-inbuilt ping utility for the implementation. The observed RTT differences are quite similar in both approaches.

5.5.3 Mobility Detection Accuracy

To evaluate `Kwikr`'s simple RSSI-based mobility detector's accuracy, we used our C#.Net implementation of `Kwikr` on a Windows Surface Pro 2. We gathered 20 RSSI traces, each 1 minute long, containing samples taken every 100 ms while being mobile. The mobility profiles were random and included ones in which we walked from a location where the signal was strong to one where it was weak and vice versa; we paced around the access point without going too far away, etc. We also gathered traces while keeping the client stationary at different locations and signal strength levels.

For each trace, we used a sliding window approach to calculate the RSSI variance. We used window sizes ranging from 1 s to 10 s in increments of 1 s and for each window size, the window was slid forward by 100 ms. Using this approach, we obtained the RSSI variance for all the windows in our traces and each window was labeled either stationary or mobile based on the experiment setting. Then, we built a 10-fold cross-validation decision tree model on this data to build a mobility classifier.

Table 5.4 shows the confusion matrix of the detector for a window size of 10 s. It shows

Table 5.4: Confusion matrix for mobility detection. The overall detection accuracy is above 90%.

		`Kwikr` Classification	
Ground Truth		Stationary	Mobile
Stationary	15262	13939 (91.3%)	1323 (8.7%)
Mobile	19646	1978 (10.1%)	17668 (89.9%)

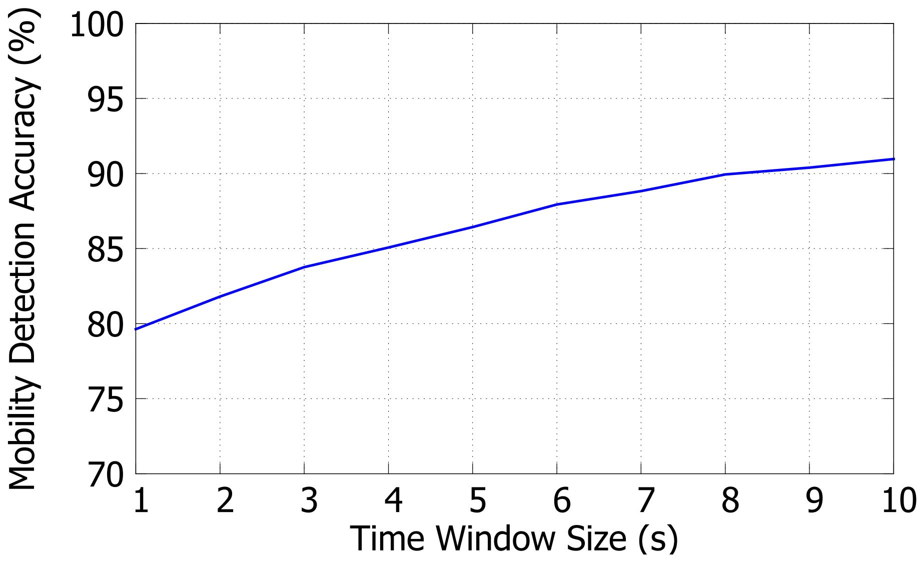


Figure 5.7: Mobility/Stationary detection accuracy for varying time window sizes. The detection accuracy increases with the size of the time window.

that `Kwikr` has an overall mobility detection accuracy of around 90%. The detector makes mistakes when the RSSI variance is high in stationary settings and vice versa. When the client is stationary, there can be transient RSSI fluctuations that are caused by external causes, *e.g.,* a person walking between the access point and the client. RSSI variance can be low in settings when the client's mobility profile is not that significant, *e.g.,* walking a few steps back and forth.

Figure 5.7 plots the overall accuracy of `Kwikr`'s mobility detector for time window sizes ranging from 1 s to 10 s. This shows that to be at least 90% of the time correct in predicting mobility, a client has to observe RSSI variance for at least 8 s. The accuracy increases with the time window size, meaning that the longer the client observes RSSI variances for, the higher the accuracy. In fact, for a time window size of 20 s, the accuracy is over 96%.

This raises the question of whether a detector that has to wait for 8 s to give an answer is good enough. The answer depends on the application in which `Kwikr` is meant to be used. For example, a VoIP application that wants to be mobility-aware to do smart bandwidth adaptation can cope with such a delay, as we will discuss in the next chapter (Chapter 6).

5.5.4 Detection of Other Pathologies

Handoffs. Whenever a client goes through a handoff scenario, the BSSID the client is connected to changes as soon as the handoff is completed. This is because every Wi-Fi access point has a unique physical address, which serves as its BSSID. Across all the locations in which we conducted our experiments (NUS campus, Microsoft Research India office, and an apartment complex), we monitored the BSSID during 50 handoff events and as expected, we noted that the BSSID changed every single time a handoff event occurred.

Weak Links. We conducted 50 TCP throughput measurements, wherein we monitored the TCP throughput at a client while moving away from the Wi-Fi access point and logged the RSSI of the signal and the TCP throughput continuously. The average RSSI value at which the TCP throughput dropped below 75% with respect to the throughput observed when the client was in close proximity to the access point was $\sim$ 80 dBm.

5.6 Chapter Summary

Wi-Fi-connected clients experience poor performance in applications due to many pathological conditions that can arise in Wi-Fi networks. In this section, we presented `Kwikr`, a user-level module that comprises a suite of techniques that can detect wireless network congestion, mobility, handoffs and weak links. It then generates useful Wi-Fi hints about the status of the Wi-Fi link that are then passed onto applications that are interested in knowing the condition of the Wi-Fi link. `Kwikr`'s detector suite is lightweight and can be deployed on any platform, even on those that are more restrictive like Android. Through our experimental studies, we showed that `Kwikr` can detect the presence of these pathologies with more than 90% accuracy. In the next chapter, we discuss an application of `Kwikr` where we use the Wi-Fi hints that `Kwikr` produces to improve bandwidth adaptation strategies used by real-time streaming applications.

Part III

Bottleneck Alleviation

Part Overview

Real-time streaming applications exhibit poor performance on Wi-Fi networks when compared to wired networks, as we showed in our measurement studies in Chapter 3. This is because these applications have stringent demands, *e.g.,* low packet loss and delay requirements, that are sometimes not met by Wi-Fi networks.

In this part of the dissertation, we present two complimentary and orthogonal solutions to improve the reliability of real-time streams in Wi-Fi networks. First, in Chapter 6, we present a user-level module, named `KwikrAdapt`, that enables real-time streaming applications perform improved bandwidth adaptation by being aware about the condition of the Wi-Fi link. Our module is capable of helping real-time streaming applications detect the root cause of quality degradations in the streams, and then adapt accordingly to improve the quality of the streams.

Next, in Chapter 7, we introduce a system that improves the reliability and robustness of real-time streams by replicating packets across multiple Wi-Fi links. Our system, named `DiversiFi`, allows clients maintain two parallel Wi-Fi connections and then replicate packets of the stream across these connections in an on-demand fashion. We show that this strategy significantly improves the quality of real-time streams.

Chapter 6

Bottleneck Alleviation via Bandwidth Adaptation

6.1 Introduction

Real-time interactive streaming has been growing in importance, spanning both traditional applications such as audio-video (AV) conferencing (e.g., Microsoft Skype, Google Hangouts) and newer ones such as cloud-based app streaming (e.g., Amazon AppStream) and gaming (e.g., Sony PlayStation Now). The performance of such real-time streaming applications is highly sensitive to network performance, being a function of not just the data rate supported but also the delay, delay jitter, and packet loss rate.

Fluctuation in the available network bandwidth presents a particular challenge. Unlike on-demand streaming, where a multi-second playout buffer can be employed to absorb and smooth out much of this variation, the tight deadline for real-time interactive streaming (e.g., an RTT of no more than 300 ms for VoIP and 60-100 ms for gaming) rules out a large playout buffer. Therefore, it becomes critical to estimate network bandwidth and track its variation effectively.

Typically, interactive streaming applications use UDP, instead of TCP, due to the real-time constraints they entail, and they do congestion control in the application layer. While TCP friendliness [92] helps real-time streaming ensure fair sharing of bandwidth with legacy applications based on TCP, it does not guard against queue build-up and the consequent spike in delay and latency, which could hurt real-time streaming. To address this problem, there has also been work on conservative approaches that back off at the first sign of incipient queuing [78], and ramp up slowly. However, in a situation where the congestion is due to other traffic, not self-congestion, such a conservative strategy would likely hurt the real-time flow without any benefit in terms of reduced latency.

The crux of the problem is that real-time streaming applications' bandwidth estimation, like TCP's congestion control, operates end-to-end, without direct knowledge of the internal state of the network. Increasingly, however, the clients engaged in real-time communication tend to be connected over a Wi-Fi network. Furthermore, the Wi-Fi link is often, though not always, the bottleneck link that determines the fate of the end-to-end real-time flow. The combination of these two factors means that the client is often in a position to directly observe the state of the bottleneck link. By leveraging the Wi-Fi hints obtained, as discussed earlier (Chapter 5), bandwidth estimation and adaptation can be made faster and more effective.

To this end, we present `KwikrAdapt`, which leverages Wi-Fi hints produced by `Kwikr` (see Chapter 5) for improved bandwidth estimation of real-time streams. The high-level goal of `KwikrAdapt` is to complement currently used end-to-end bandwidth adaptation strategies in real-time streaming applications and improve upon them using last-mile Wi-Fi link information when and where it makes sense. Using `Kwikr` hints, `KwikrAdapt` first determines whether an end-to-end flow is experiencing performance issues due to pathological conditions in the Wi-Fi link, such as congestion, mobility, weak-links, and handoffs. If so, `KwikrAdapt` proceeds to tweak the bandwidth adaptation technique of the flow using `Kwikr` hints. If not, implying that the bottleneck is elsewhere on the end-to-end path, it falls back on the default behavior for the bandwidth adapter.

`KwikrAdapt` improves bandwidth adaptation based on (a) local congestion on the Wi-Fi link, which indicates the onset and conclusion of congestion or interference due to other sources (as distinct from self-congestion), (b) contribution factor, which represents the flow's contribution in making the Wi-Fi link a bottleneck, (c) mobility and weak links, which can indicate the likelihood of a drop in link performance, and (d) handoffs, which often cause localized disruption in the form of packet loss.

We have evaluated `KwikrAdapt` in a range of Wi-Fi settings, including offices, coffee shops, airports, conference venues, etc. We also conducted controlled experiments in a lab setting.

Based on these experiments, we have found that `KwikrAdapt` helps improve streaming performance *safely*, i.e., without causing negative side-effects, either to the flow itself or to other traffic. Our results show that Wi-Fi-aware bandwidth adaptation of real-time streams provide far higher quality than the conventional end-to-end based adaptation strategies. For instance, `KwikrAdapt` allows real-time streams to operate at relatively higher data rates while incurring similar delays as low data rate flows in congested Wi-Fi networks; reduces the worst loss percentage by about $\sim 70\%$ in mobile settings; and reduces the time required to ramp-up to the original data rate of a flow by $3.1\times$ when a handoff occurs.

In summary, our contributions in this work include:

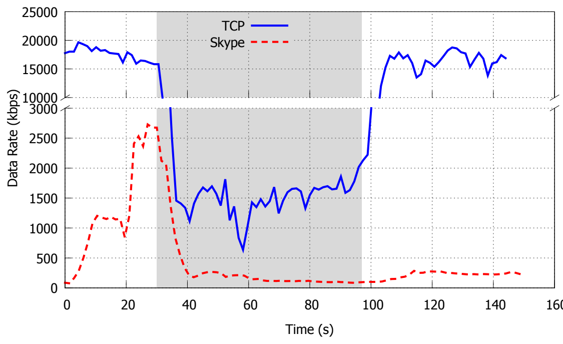
- A system, named `KwikrAdapt`, and its implementation that improves the quality of real-time streams by using Wi-Fi hints provided by `Kwikr` to make bandwidth adaptation "Wi-Fi-aware".
- An empirical analysis to show the significant benefits of `KwikrAdapt`.
- The implementation of `KwikrAdapt` in the popular Skype application.

6.2 Motivation

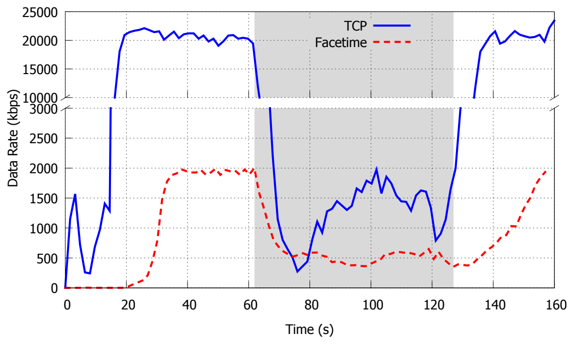
To illustrate the issues with bandwidth estimation for real-time streaming, we focus on three examples of popular AV conferencing applications: Skype, FaceTime, and Hangouts. For Skype, we have access to an instrumented version that provides a log containing detailed metrics pertaining to a call at the end of each call. The metrics we extracted from these log files are the call throughput and the per-packet round trip times. We obtain the throughputs of FaceTime and Hangouts calls via packet captures on Wireshark [100].

6.2.1 Congestion

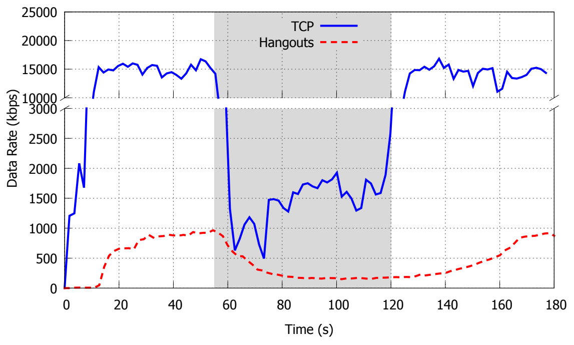
Consider Figure 6.1(a), which shows the data rate of both a Skype AV stream and of a foreground TCP flow. Congestion, in the form of cross-traffic (generated by 6 devices, each performing a TCP bulk transfer), is introduced and withdrawn on the Wi-Fi bottleneck link



(a) Skype



(b) FaceTime



(c) Hangouts

Figure 6.1: Congestion response of (a) Skype, (b) FaceTime, and (c) Hangouts vis-a-vis that of TCP. The shaded region in each plot depicts the period in which congestion was present in the Wi-Fi link in the form of cross-traffic TCP bulk transfers. In all cases, TCP flows achieve a much higher data rate in the presence of congestion when compared to the VoIP flows.

at the points in time marked by the shaded area. At the onset of congestion, the data rate of both Skype and TCP plummets, but TCP soon recovers to a level of almost 1.5 Mbps while Skype remains stuck at the much lower level of about 200 Kbps.

Skype adopts a conservative approach, apparently with a view to limiting queuing delay and avoiding packet loss. However, in this instance, such a conservative approach does not really help with the RTT for the Skype flow remaining high throughout the congestion episode (Figure 6.2) despite the sharp cutback in Skype's data rate, dropping from about 2.8 Mbps soon after the onset of congestion to about 200 Kbps beyond t=30 s. The reason is that the congestion is due to other traffic, not self-congestion due to the Skype flow itself. Thus, Skype is being needlessly conservative, without deriving any benefit.

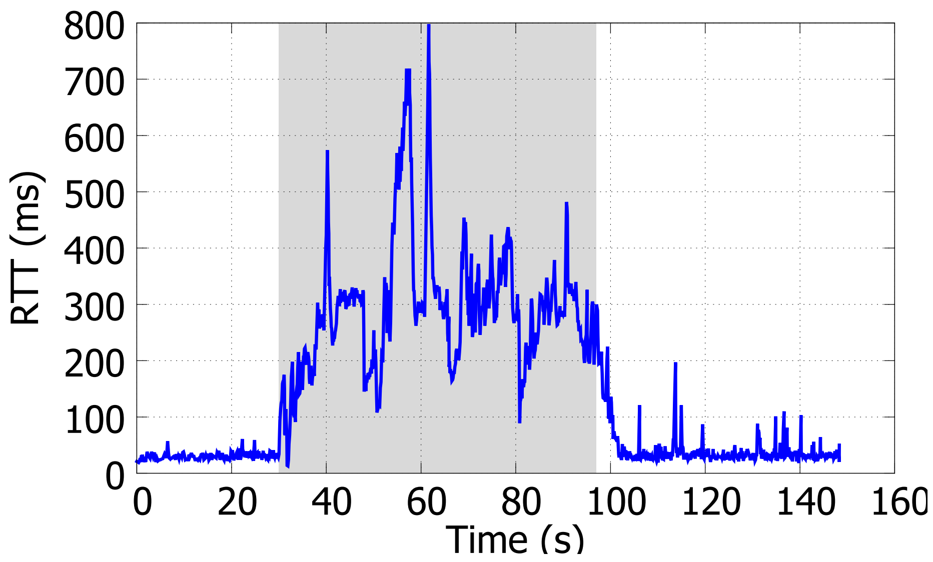


Figure 6.2: The RTT of the Skype packets. The shaded region depicts the period in which the Wi-Fi network was congested. Being conservative and operating at a much lower data rate does not help Skype to bring its RTTs down.

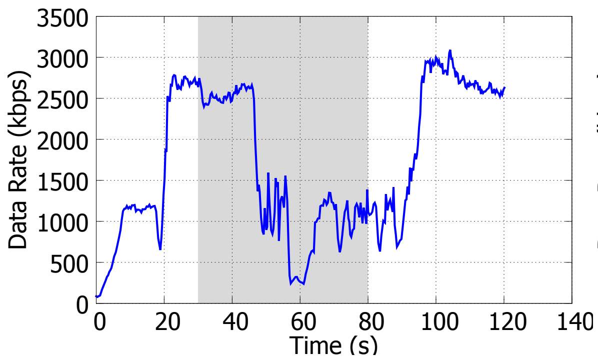
Similarly, even when the congestion episode has concluded, Skype is slow to recover, as shown in Figure 6.1(a), because it is unaware that there has been a step change in the congestion state of the bottleneck link.

The other two real-time streaming applications—FaceTime and Hangouts—also show similar conservative behavior, as depicted in Figures 6.1(b) and 6.1(c). Of these three VoIP applications, FaceTime seems to employ a more aggressive approach in bandwidth adaptation because it recovers faster than Skype or Hangouts at the conclusion of the congestion episode. But even that takes about 25 s to ramp up to the data rate that was present before the onset of congestion.

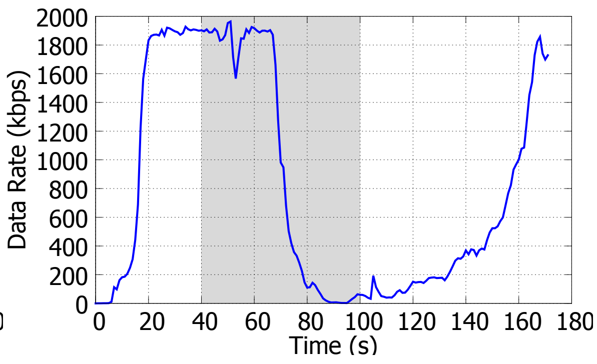
Since we do not have access to detailed metrics for these applications, we are not in a position to plot other network metrics like RTT.

6.2.2 Mobility

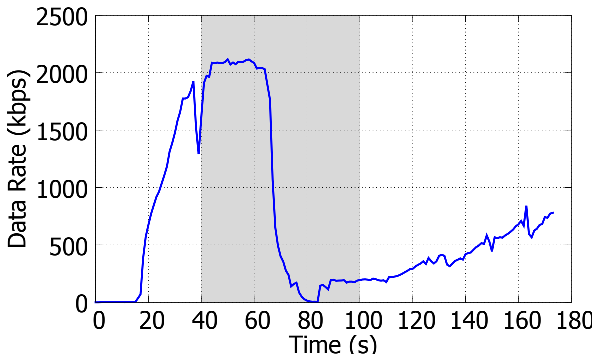
Figure 6.3 depicts how VoIP streams react to mobility in Skype, FaceTime, and Hangouts. The shaded area shows the period during which the client moved away from the Wi-Fi access



(a) Skype



(b) FaceTime



(c) Hangouts

Figure 6.3: Mobility response (a) Skype, (b) FaceTime, and (c) Hangouts. The shaded region in each plot depicts the period in which the client was mobile. All VoIP applications suffer performance issues when the client is mobile.

point and came back to the original position.

As clearly shown in Figure 6.3, all flows vary their rates quite significantly when the client is mobile. These frequent fluctuations happen because the bandwidth adaptation techniques used by these applications are not aware that the underlying channel conditions are changing due to mobility. Hence, the decisions that they take are suboptimal and ultimately result in poor user experience.

Another thing to note is that after the mobility period is over, all flows take a significantly long period of time to ramp up to their original rates. This is akin to the behavior we observed in Section 6.2.1.

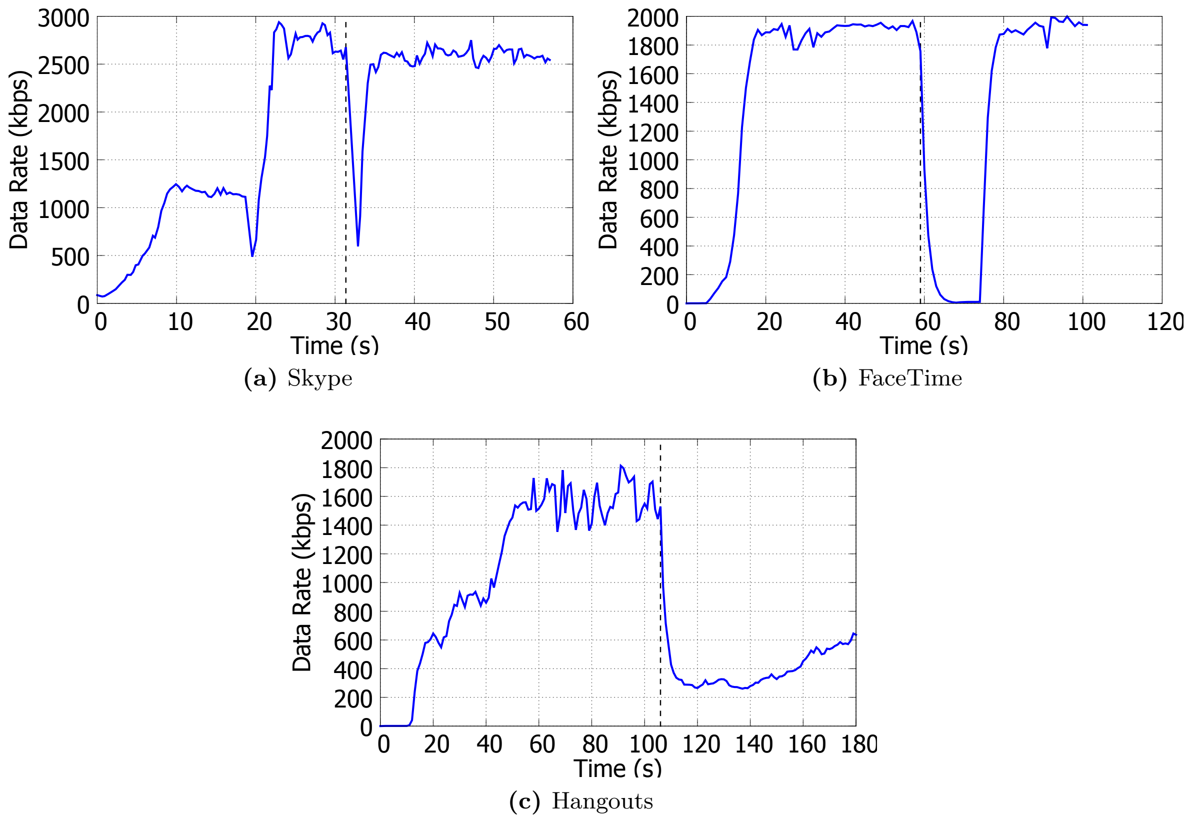


Figure 6.4: Handoffs response of (a) Skype, (b) FaceTime, and (c) Hangouts. The vertical dashed line shows the point in time when the handoff occurred. Although Skype recovers promptly from a handoff, the other VoIP applications take a longer time to recover.

6.2.3 Handoffs

Figure 6.4 shows how Skype, FaceTime, and Hangouts cope with handoffs. Skype's bandwidth adaptation technique does take into handoffs into account because it instantly resumes transmitting at the original rate when the handoff completes. FaceTime is slower than Skype in recovering from a handoff as it stops transmitting video and sends audio for a few seconds when the connection drops, and then ramps up aggressively while enabling video. Hangouts, on the other hand, is very slow in increasing the data rate after a handoff completes.

6.2.4 Discussion

The above experiments highlight an important problem in the state-of-the-art bandwidth adaptation techniques used by real-time streaming applications like Skype, FaceTime and Hangouts. These techniques operate end-to-end, and hence are oblivious to various link impairments that commonly occur in the last-mile Wi-Fi link.

In the case of congestion, applications see that the RTT of their packets are increasing and decide to be conservative thinking that the problem is self-inflicted, which does not result in improved performance. If they know that the increased RTTs are caused by congestion in the network, they could adapt differently and also ramp up faster to the original data rate at the end of the congestion period.

Client mobility imparts quality degradations in Wi-Fi networks because mobility causes channel conditions to change over time. This causes inaccuracies in current bandwidth adaptation strategies because they are oblivious to such changes due to their end-to-end mode of operation. If applications are aware that the client is mobile, they can be more conservative, specially when the signal strength of the Wi-Fi link becomes weaker. Moreover, when mobility ceases and channel conditions improve, applications can increase their throughputs to deliver better call quality experiences to users.

In the case of handoffs, applications lose packets during the period in which a handoff is in progress. However, such losses are transient and more often than not, a handoff improves channel quality. As current bandwidth estimation techniques operate using an end-to-end approach, they cannot differentiate these losses from losses that occur due to a multitude of reasons in the network. Hence, they unnecessarily drop their rate and ramp up slowly to the original rate, with the exception of Skype. If they are aware of handoffs, they could avoid the slow ramp-up phase to improve call quality.

These observations show that if bandwidth adaptation techniques can be made "Wi-Fi-aware", they can take better-informed decisions to improve user experience. The goal of `KwikrAdapt` is to do this by utilizing Wi-Fi hints that can make applications, such as Skype, FaceTime, and Hangouts, aware about the underlying status of the Wi-Fi connection. We discuss the design and implementation of `KwikrAdapt` in the following sections.

6.3 Design

The goal of `KwikrAdapt` is to help real-time streaming applications improve upon their current bandwidth adaption techniques by taking into account the condition of the Wi-Fi link. Specifically, it helps real-time streaming applications to use their existing algorithms for bandwidth adaptation when the Wi-Fi link is not bottlenecked in any manner, and tweak their behavior if the Wi-Fi link is causing performance issues. The actions that `KwikrAdapt` take in tweaking the adaptation scheme can be both proactive and reactive. For example, if the client becomes mobile or is using a weak Wi-Fi connection, `KwikrAdapt` takes proactive measures to alleviate probable packet losses and delays that the client might experience. If the client is not mobile and is not using a weak Wi-Fi connection but still is experiencing packet losses and increased delays, `KwikrAdapt` first determines whether the Wi-Fi link is introducing these issues. If so, it determines whether the issues are self-inflicted or caused by cross-traffic in the Wi-Fi network, and then helps the bandwidth adaptation technique tweak its behavior such that it can deal with the Wi-Fi issues as best as it can. Such actions are reactive measures.

`KwikrAdapt` takes the above mentioned actions based on the Wi-Fi hints it receives from `Kwikr`. Recall that `Kwikr` provides information about the status of the Wi-Fi connection via hints (Chapter 5). `KwikrAdapt` periodically receives hints (*e.g.,* every 500 ms in our experiments) and depending on those, it decides whether the bandwidth adaptation algorithm needs any tweaking via either reactive or proactive measures as we discussed above.

6.3.1 Problem Attribution

As discussed earlier, in the context of real-time streaming, our goal in `KwikrAdapt` is to complement existing end-to-end bandwidth adaptation schemes used by real-time streaming applications by allowing them to leverage Wi-Fi hints when it is relevant, and use their default behavior otherwise. In other words, a bandwidth adaptation scheme should use its

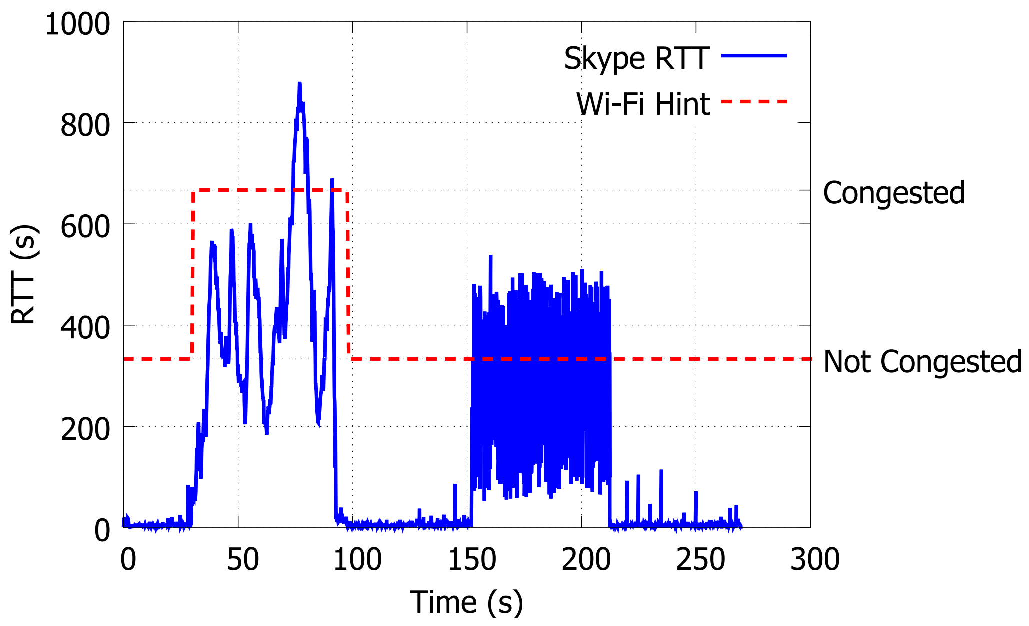


Figure 6.5: Skype RTT during a call. The first segment of increased RTTs are caused by Wi-Fi congestion and the second is caused by inducing latencies into the Internet path. Using Wi-Fi hints, `KwikrAdapt` can determine that the Wi-Fi link is the cause of only the first segment of poor performance.

default behavior when the Wi-Fi link is not the bottleneck source, and use our Wi-Fi hints to tweak the adaptation when it is. This is quite important in the case where `KwikrAdapt` needs to take reactive measures in tweaking the adaptation technique. Therefore, we first need to determine whether a performance issue in a real-time stream is caused by the Wi-Fi link. To do this, `KwikrAdapt` correlates the congestion-related Wi-Fi hints of `Kwikr` with network metrics of the call (*e.g.,* RTT, packet losses) in real-time. If the correlation is high, then `KwikrAdapt` determines that the Wi-Fi link is indeed the bottleneck link and instructs the bandwidth adaptation technique to use the hints to adapt accordingly. Otherwise, the adaptation technique falls back to the default behavior.

Consider Figure 6.5 for example. It shows the RTT values of a Skype call and the congestion-related Wi-Fi hints that `Kwikr` produced in parallel with the call every 500 ms. The first segment of elevated RTTs (32 s to 92 s) was caused by congestion in the Wi-Fi network by adding competing cross-traffic flows. The second segment (152 s to 212 s) was caused by inducing delays, using the Microsoft Network Emulator tool, into the outgoing link of the

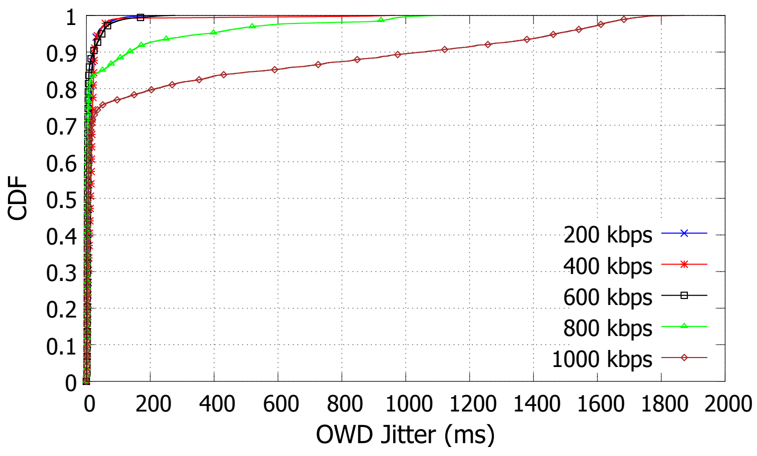
Skype client at the other end of the call. As can be seen clearly in the figure, the Wi-Fi hints correctly show that the first performance issue is caused by the Wi-Fi link and the second is not. Therefore, based on this information, the behavior of the real-time stream's adaptation technique can be changed accordingly—it can be tweaked to use the hints, as explained above, during the first period of network issues and use the default behavior in the second period.

6.3.2 Being Less or More Conservative in Choosing a Data Rate

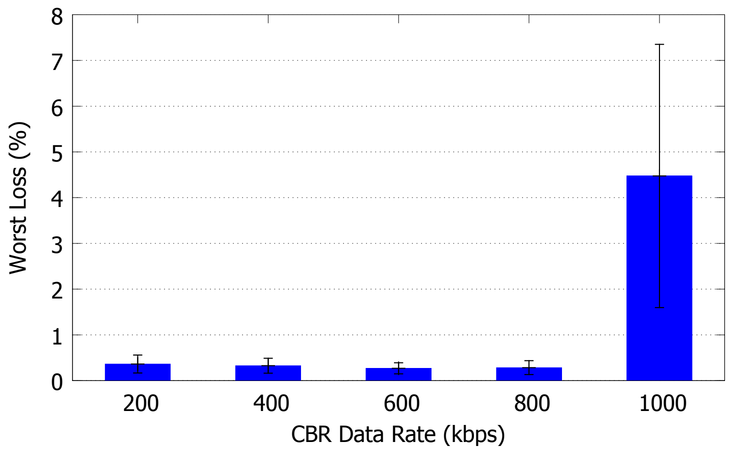
If the Wi-Fi link is congested, and the real-time stream's contribution to congestion is low, we claimed in Section 6.2.1 that being conservative and dropping the transmission rate of the flow does not help to reduce packet delays and losses. To study this, we made a client receive Constant Bit Rate (CBR) flows at varying data rates (200 kbps to 1000 kbps in increments of 200 kbps) while keeping the Wi-Fi network congested by introducing 6 TCP flows to the network. Figure 6.6(a) depicts the One-Way Delay (OWD) jitter profiles of the flows. It shows that though the 800 and 1000 kbps flows experience high delays, the other flows experience quite similar delays. This validates our claim. In the presence of network congestion, a flow should reduce its rate as operating at high data rates would incur high delays. However, instead of being quite conservative, as is the case in the state-of-the-art techniques employed by current VoIP applications, a flow could very well operate at a less-conservative data rate while incurring similar delays as a low data rate flow. For example, Figure 6.6(a) shows that the jitter profiles of the 200 kbps and 600 kbps flows are quite similar. Essentially, a flow gains better stream quality by operating at a less-conservative data rate while paying the same price (in delays) as a more conservative flow.

Our claim is further validated by Figure 6.6(b) that shows the average worst loss percentage the flows experienced. We calculate the worst loss percentage by dividing each of our traces into 5 s windows and taking the highest packet loss observed in them[1]. This plot clearly shows that once again the worst loss percentages are low and quite similar in the 200 to 800

[1]There is evidence that worst degradation in a (short) call is a significant determinant of the overall user-perceived call quality [98].



(a)



(b)

Figure 6.6: (a) OWD jitter profiles, and (b) Worst loss percentages of CBR flows operating at varying data rates in a congested Wi-Fi network. 200 kbps to 600 kbps flows observe similar OWD jitter profiles and losses. This implies that a flow gains better stream quality by operating at a less-conservative data rate (600 kbps) while paying the same price (in delays and losses) as a more conservative flow (200 kbps).

kbps flows but is high in the 1000 kbps flows.

Therefore, in a congested setting, `KwikrAdapt` utilizes the QOM hint of `Kwikr` to determine the contribution of a real-time flow to the observed congestion. If QOM is high *i.e.,* congestion is self-inflicted, then the bandwidth adaptation technique should drop the transmission more conservatively as the flow is responsible for the observed delays. However, if QOM is low *i.e.,* delays are not self-inflicted and caused by other flows in the network, then the real-time flow needs to reduce the data rate less conservatively, as being conservative does not help improve performance. Hence, `KwikrAdapt` helps a real-time flow adapt dynamically by first being aware of Wi-Fi congestion and then being aware of whether the observed delays and losses are self-inflicted or not.

6.4 TFRC Implementation

We implemented `KwikrAdapt` on Windows using C#.Net as a Wi-Fi hints subscriber to our Windows-based `Kwikr` implementation. It subscribed to all the hints that are provided by `Kwikr` and requested that hints be delivered every 500 ms in our implementation. We should note that `Kwikr` hints can be generated at any interval but the overhead in doing so should be taken into account by the application. For example, if the interval is set to 500 ms, the detectors would be invoked twice every second. This means that the `Ping-Pair` technique would run twice a second, and hence the client receives 4 ICMP ping messages in a second. If the real-time flow operates at 64 kbps (a typical voice-only call), the overhead in running the detectors is 8%, as the flow receives 50 packets in a second (assuming the standard 160 byte payload size). However, if the flow operates at about 1 Mbps (a video and voice call), the overhead incurred by `KwikrAdapt` is much less.

As we do not have access to the code bases of popular VoIP applications like Skype, FaceTime, Hangouts, *etc.*, we could not modify their bandwidth adaptation techniques to use `KwikrAdapt` and study the gains of doing so. Therefore, we use TCP Friendly Rate Control protocol (TFRC) [92] for this purpose. TFRC is a well-understood bandwidth adaptation mechanism for unicast flows. As in state-of-the-art bandwidth adaptation mechanisms, the receiver monitors network parameters like the RTT, and packet loss of the incoming flow, and then feeds them back to the sender by sending period reports via a control channel. The sender then adapts the sending rate based on these reports. We implemented a real-time streaming application that sends data at a given data rate using the TFRC protocol. Our implementation was done on Windows using C#.Net following the TFRC specification documented in RFC 5348 [92].

The TFRC protocol does not react to increasing packet delays if no packets are lost. Therefore, we modified the protocol followed by the receiver such that it considers any packet that has a One-Way Delay (OWD) of more than 150 ms lost. This is justified by the fact that the maximum tolerable OWD for VoIP is 150 ms [12].

6.4.1 TFRC with KwikrAdapt

TFRC's rate selection mechanism is based on a single equation which takes in two input parameters: RTT, and packet loss event rate. We make several changes to the default TFRC behavior to study the benefits of `KwikrAdapt`. First, we changed the structure of the report the receiver sends back to the sender to contain two new fields: *Mobility Status*, and *Link Status*. Next, we changed the protocol to adapt to different pathological conditions in theWi-Fi network as follows.

Congestion. If the receiver observes that the flow is exhibiting poor performance (high RTTs and/or packet losses), and it is caused by the Wi-Fi link, it uses the QOM hint to find its level of contribution to the problem. If QOM is quite small (under 10%), the receiver reports a lower than actual loss event rate to make the sender transmit the flow at a relatively higher data rate than what the default protocol would use. If the QOM is high or the delays and losses are not caused by the Wi-Fi link, the receiver reports the actual loss event rate so that the sender uses the default TFRC behavior.

Mobility and Weak Links. If `KwikrAdapt` gets a non-stationary mobility hint from `Kwikr`, it sets the Mobility Status field of the receiver report to 1, and the sender immediately lowers the sending rate to avoid any potential disruptions of the stream. Once the mobility status becomes stationary, the receiver clears the flag upon which the sender will increase the data rate to the rate at which it was operating at prior to the start of the mobile period. While the user is mobile, if they walk away from the access point and the link becomes weak, the TFRC receiver will receive the weak link hint from `KwikrAdapt` and then it will set the Link Status field to 1 of the outgoing reports. The sender will be even more conservative in this case and again return back to the original data rate if and when the receiver informs that the link is strong.

Handoffs. In the case of a handoff, the TFRC receiver will observe packet losses which will cause the loss event rate value of the report to be set to a non-zero value. This in turn makes the sender lower the data rate of the flow. When using `KwikrAdapt`, the TFRC receiver knows when a handoff occurs and can conclude that any losses that coincide with the handoff event is actually caused by the handoff event. In such cases, the receiver ignores losses and does not update the loss event rate parameter of the report. This allows the sender to keep transmitting at the initial rate, and thereby avoiding the slow ramp-up phase that follows a data rate drop in the default behavior.

6.5 Evaluation using TFRC

We deployed our sender-side TFRC implementation on a Microsoft Azure server in Singapore and the client-side implementation on a Windows Surface Pro 2 running Windows 8.1. We capped the data rate at the sender to 1.2 Mbps, the typical bandwidth requirement for a high definition quality Skype video call. In each run of our experiments, the sender sends a TFRC flow to the receiver for 60 seconds using the default behavior as specified in RFC 5348 [92], and then followed by another 60 second flow using the TFRC version that uses `KwikrAdapt`. We conducted our experiments in India and Singapore on different Wi-Fi networks and settings such as office spaces, universities, and apartment complexes.

6.5.1 Adaptation in Congestion Scenarios

In our TFRC implementation that uses `KwikrAdapt`, when the receiver experiences increasing delays and/or loss rates, it first detects whether the problem is due to the Wi-Fi link, and if so, its contribution to the congestion using the QOM hint. If the Wi-Fi link is congested and QOM is low, `KwikrAdapt` allows the receiver to report a loss rate that is lower than the actual loss rate. This makes the sender transmit the flow at a less-conservative data rate. Furthermore, after the conclusion of the congestion episode, the receiver reports a loss rate of 0 so that the sender can immediately start transmitting at the original rate it was using prior to the onset of congestion. This removes the need for the slow ramp-up phase,

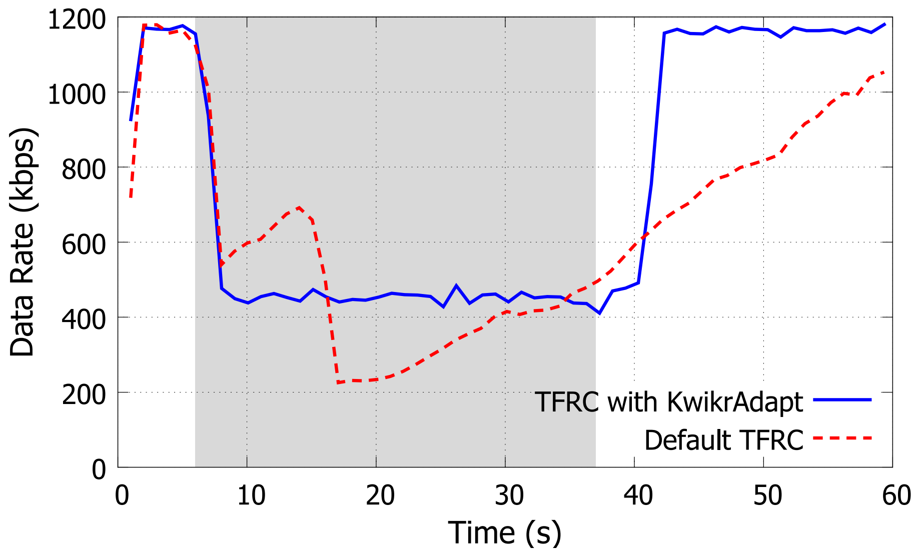


Figure 6.7: Data rate fluctuations of Default TFRC and TFRC with `KwikrAdapt` flows. The shaded area depicts the period in which the Wi-Fi network was congested. Being aware of Wi-Fi congestion, `KwikrAdapt` allows the TFRC flow to achieve a higher and an even data rate throughout the congestion episode.

and hence provides better quality streams to the user. Using this setup, we collected 50 traces and introduced congestion for ~ 30 seconds by adding 6 TCP cross-traffic flows to the network while the flows were in operation.

Figure 6.7 depicts the data rate fluctuations of a Default TFRC flow and a TFRC with `KwikrAdapt` flow. Both flows react to congestion by dropping their data rates, however as `KwikrAdapt` detects that the congestion is caused by the Wi-Fi link and the problem is not self-inflicted, it makes the flow operate at a higher and an even data rate throughout the congestion period. After the congestion period ends, the default behavior requires about ~ 28 s to ramp-up to the original rate. However, TFRC with `KwikrAdapt`, having a better understanding of what caused the problem in the first place and knowing that Wi-Fi link is congestion-free again, allows the flow to ramp-up within 2 s, reducing the ramp-up time by $14\times$. The average reduction of the ramp-up time across all our traces is $12.2\times$.

Figure 6.8 shows a CDF curve of the OWD jitter the two flows experienced during the congestion period in all our 50 traces. (This plot shows similar behavior to Figure 6.6(a).) As it clearly depicts, the jitter profiles of the two flows are quite similar. Furthermore, the

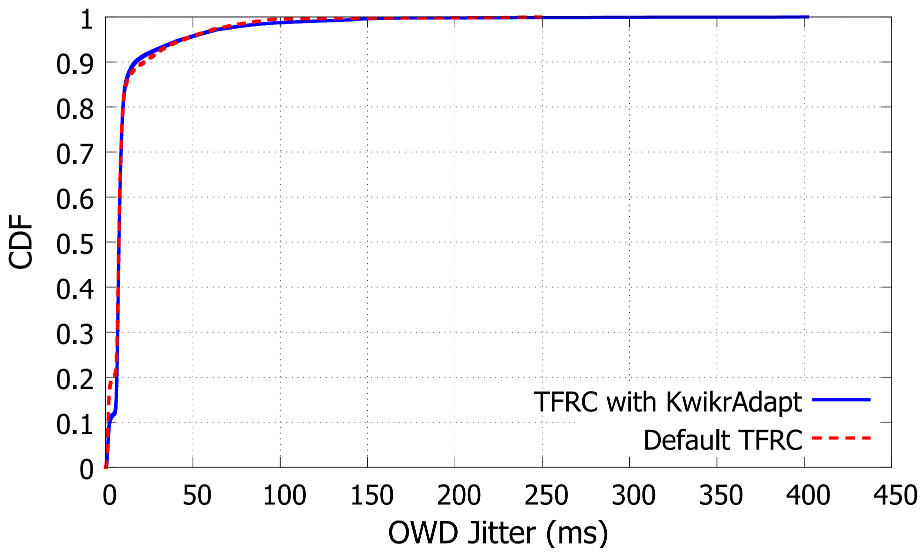


Figure 6.8: OWD jitter profiles of Default TFRC and TFRC with `KwikrAdapt` flows. Both protocols observe similar packet delays even though the TFRC with `KwikrAdapt` flow operated at a much higher data rate than the default protocol.

average packet loss rates during the congestion period are 1.22% and 1.85% for Default TFRC and TFRC with `KwikrAdapt`, respectively. This data shows that being less-conservative when the Wi-Fi network is congested does not incur an additional cost, but in return the quality of the stream can be improved as it is able to operate at a relatively higher data rate.

Impact on Cross-Traffic Flows. Next, to study what happens to cross-traffic flows when `KwikrAdapt` makes a real-time flow less-conservative in backing off in the presence of congestion, we studied the aggregate throughput the 6 TCP flows garnered during the congestion period. The average aggregate throughputs were 18276.16 kbps and 17914.76 kbps when the Default TFRC and TFRC with `KwikrAdapt` flows were in operation, respectively. This shows that the less-conservative nature of `KwikrAdapt` does impact cross-traffic flows, but the throughput drop, which is about 360 kbps, is not significant and is almost equal to the average difference in the data rates the two TFRC variants use during congestion. In fact, as this 360 kbps reduction gets distributed across all the cross-traffic flows, Figure 6.9 shows that the 6 individual TCP flows experience minimal impact due to the less-conservative nature of `KwikrAdapt`.

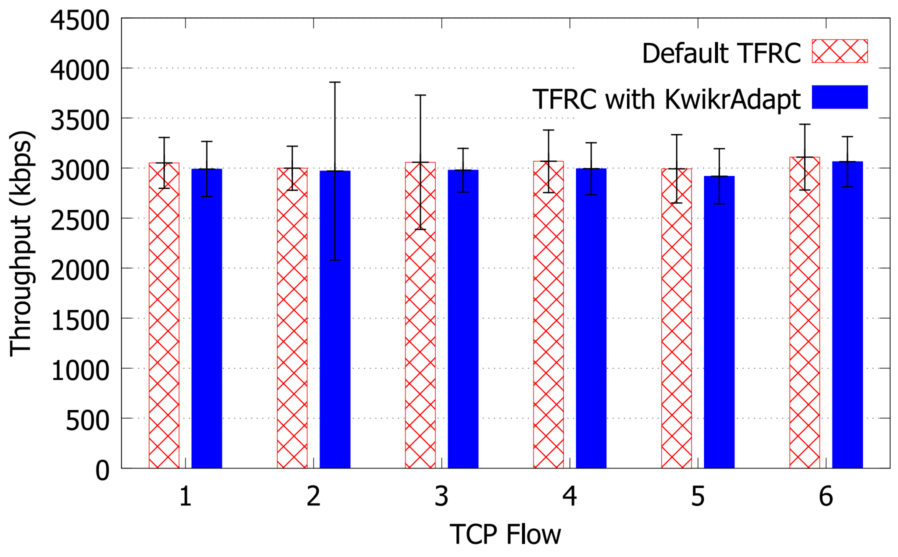


Figure 6.9: Average TCP throughputs of the 6 TCP cross-traffic flows when the Default TFRC and TFRC with `KwikrAdapt` flows were in operation. The less-conservative nature of `KwikrAdapt` causes minimal impact on all of the individual cross-traffic flows.

Handling WAN Bottlenecks. Recall that our goal in `KwikrAdapt` is to allow a real-time flow to use its default bandwidth adaptation scheme when the bottleneck is not in the Wi-Fi network, and tweak the adaptation algorithm based on Wi-Fi hints only when the Wi-Fi link is the bottleneck source. We conducted experiments where we first introduced Wi-Fi bottlenecks (by introducing competing TCP cross-traffic flows into the Wi-Fi link), and thereafter WAN bottlenecks (by inducing packet losses and delays into the outgoing link of the sender via the Microsoft Network Emulator tool). Under such conditions, Figure 6.10 shows how Default TFRC and TFRC with `KwikrAdapt` flows performed.

The first shaded region (6 s to 21 s) depicts a 15 s time window in which the Wi-Fi link was congested. As `KwikrAdapt` is capable of detecting this, it allows the flow to operate at a less-conservative data rate and ramp-up to the original rate as soon as the cross-traffic flows were removed. In contrast, the default protocol makes the flow cut its data rate quite conservatively and takes $\sim$ 20 s to ramp-up to the original data rate. The second shaded region (61 s to 76 s) shows a 15 s period wherein the WAN link was bottlenecked. In this case, as `KwikrAdapt` finds that the observed performance issue of the flow is not caused by the Wi-Fi link, it follows the default behavior of TFRC. This shows that `KwikrAdapt`

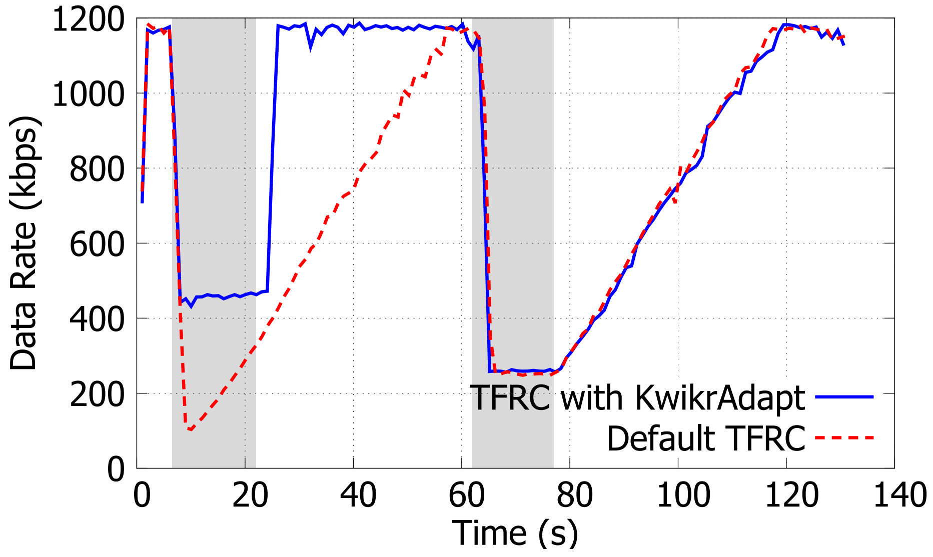


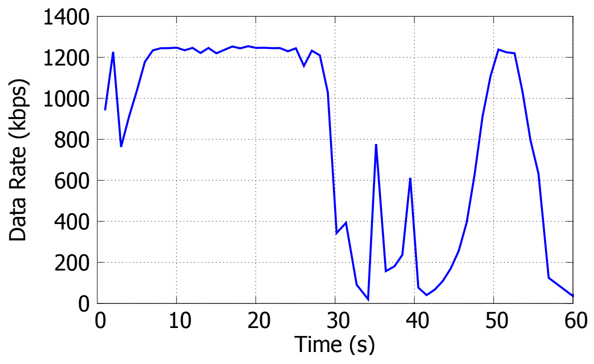
Figure 6.10: The data rate fluctuations of Default TFRC and TFRC with `KwikrAdapt` flows during two bottleneck situations. The first shaded region (6 s to 21 s) depicts a 15 s time window in which the Wi-Fi link was congested by using competing TCP cross-traffic flows. The second shaded region (61 s to 76 s) shows a 15 s period wherein the WAN link was bottlenecked by inducing packet losses and delays. `KwikrAdapt` allows the flow to be less-conservative in cutting its rate in the former period, and follow the default TFRC behavior in the latter.

is capable of helping real-time flows to take the most appropriate bandwidth adaptation decisions based on the root cause of the observed quality degradation.

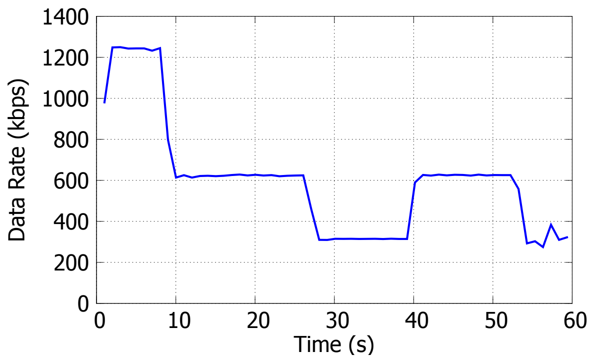
6.5.2 Adaptation in Mobility Scenarios

In this section, we evaluate the performance of real-time streams when the client is mobile while receiving the streams. We used a variety of mobility patterns in our experiments. For example, we walked back and forth both being in close proximity and quite a distance away from the access point, we walked slowly and quickly between locations where we had strong and weak signal strengths, and we walked starting from close proximity to the access point to the edge of connectivity, amongst others.

In our `KwikrAdapt`-enabled TFRC implementation, when the sender receives a receiver report with the Mobility Status or Link Status flags set to 1, it halves the operating rate. This is a proactive measure and is in stark contrast to the reactive measures that state-of-the-art

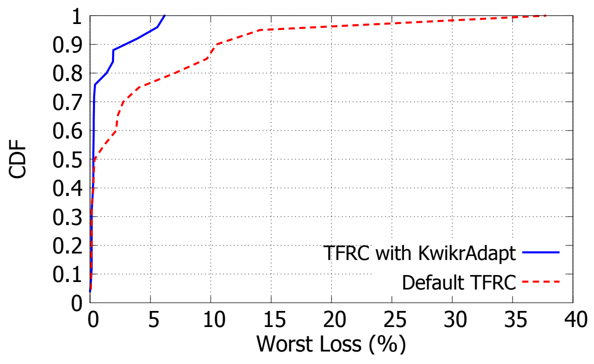


(a) Default TFRC

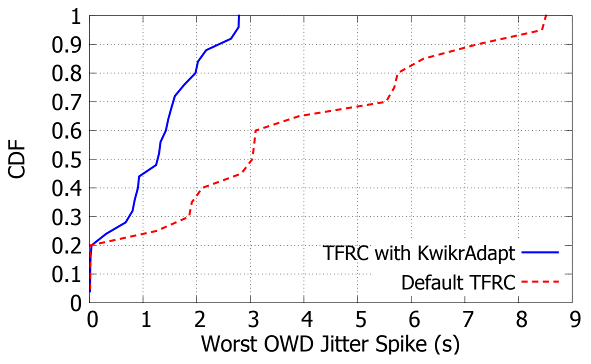


(b) TFRC with `KwikrAdapt`

Figure 6.11: Data rate fluctuations of (a) Default TFRC, and (b) TFRC with `KwikrAdapt` flows in the presence of mobility. Being aware that the client is mobile, `KwikrAdapt` provides a smooth and evened out streaming experience to the user.



(a) Worst Loss (%)



(b) Worst OWD Jitter

Figure 6.12: CDF of (a) the worst 5 s loss percentage, and (b) the worst OWD jitter spike, of a default TFRC flow and `KwikrAdapt`-enabled TFRC flow in a mobile setting. The proactive adaptation of `KwikrAdapt` helps to significantly reduce both metrics, thus providing high quality streams.

bandwidth adaptation techniques follow, *i.e.,* they drop the rate only when they experience a performance problem like jitter spikes or packet losses. We evaluate the effectiveness of our proactive adaptation technique in this section.

Figure 6.11 depicts how the data rate of a default TFRC flow and a `KwikrAdapt`-enabled TFRC flow fluctuate in the presence of mobility. In Figure 6.11(b), notice that `KwikrAdapt` allows the TFRC flow to make smooth step changes in the data rate to better suit channel conditions. For example, at t=9 s, it detects that the user is mobile and halves the rate. At t=26 s, as the Wi-Fi link becomes weak, it halves the rate yet again. The rate is increased at t=39 s when the link becomes stronger. This method of operation is very different from the default TFRC protocol shown in Figure 6.11(a), which is akin to the state-of-the-art bandwidth adaptation techniques we discussed in Section 6.2.2. As it is not aware that underlying channel conditions are changing, it keeps trying to ramp up after a jitter spike ends or losses decrease to find out that the Wi-Fi link cannot handle the new rate and has to drop the rate, and this happens repeatedly. Ultimately, such behavior results in poor user experiences.

To illustrate the problem further, consider Figure 6.12(a) that plots the worst 5 s packet loss percentage of the traces that we gathered. That is, we divide each one minute trace into 5 s periods, and calculate the packet loss percentage of each period and plot the highest loss of each trace. This analysis is justified by the fact that users typically remember the worst period of a stream they experience. The 90th worst loss percentiles of default TFRC and `KwikrAdapt`-enabled TFRC flows are 10.82% and 3.11%, respectively, implying that the proactive behavior of `KwikrAdapt` leads to much lower losses. Moreover, the plot shows that the default TFRC protocol sometimes experiences worst loss percentages as high has 37%, which would cause the streams to be of poor quality.

The worst one way delay jitter spike the flows experienced in Figure 6.12(b). As it clearly shows, `KwikrAdapt` helps reduce one way delay jitter spikes significantly. The 90th worst one way delay jitter percentiles of default TFRC and `KwikrAdapt`-enabled TFRC flows are 7.35 s and 2.46 s, respectively. Furthermore, note that the tail of the default TFRC curve in Figure 6.12(b) is quite long, which shows that in some cases there have been delay spikes of more than 8 s. Such occurrences cause sever outages in the flows that lead to poor user experiences.

6.5.3 Adaptation in Handoff Scenarios

To evaluate how `KwikrAdapt` can help a flow regain its original data rate rapidly after a handoff, we collected 50 traces in which we triggered a handoff at random points in time in the flow for both default TFRC and the `KwikrAdapt`-enabled TFRC protocols. Figure 6.13

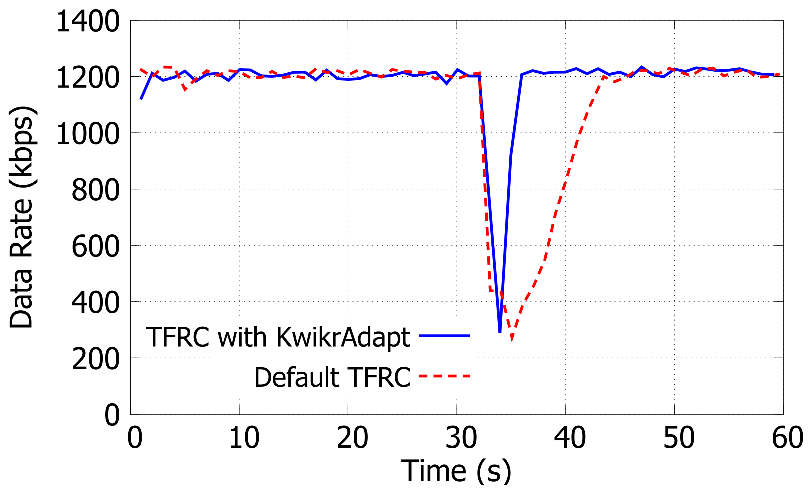


Figure 6.13: A data rate plot of a default TFRC flow vis-a-vis that of a `KwikrAdapt`-enabled TFRC flow. At 32 s, a handoff is triggered on both flows. Being aware of the handoff, `KwikrAdapt` allows the flow to ramp-up to the original data rate faster, while the default protocol takes a longer time to do so.

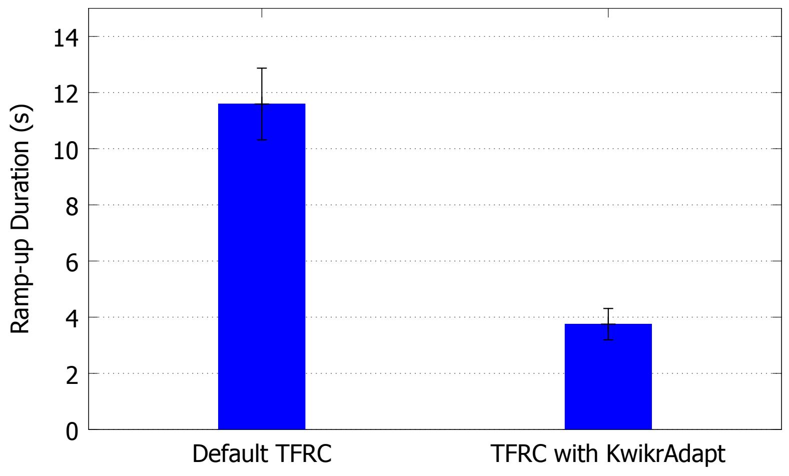


Figure 6.14: Average time the ramp-up phase requires after a handoff for the default TFRC and `KwikrAdapt`-enabled TFRC protocols. `KwikrAdapt` reduces the ramp-up duration by 3.1×.

shows how the two bandwidth adaptation mechanisms respond to a handoff. The default TFRC protocol goes through a slow ramp-up of the data rate after the handoff because it is unaware that the handoff is the root cause of the losses it observed in the flow. Therefore, the ramp up takes 14.45 s to complete. However, with `KwikrAdapt`, the flow gains the original rate in just 3.9 s. Thus, with `KwikrAdapt`, we can cut down the time required to ramp up to the original rate after a handoff by 3.7×.

Figure 6.14 plots the average time the flows take to ramp up after a handoff occurs in all the 50 traces we gathered. As can be clearly seen, as `KwikrAdapt` allows the TFRC protocol to realize the observed losses were caused by a handoff event, the flow can ramp up faster. The average reduction in the ramp up phase in `KwikrAdapt`-enabled TFRC is 3.1×.

As the user-perceived quality of a real-time stream is directly proportionate to the data rate, the fast ramp up that `KwikrAdapt` provides allows applications to send high quality audio and video faster, and thereby improving the overall quality of the stream.

6.6 Skype Implementation

In addition to using TFRC, we also evaluate the benefits of Wi-Fi-aware bandwidth adaptation by integrating `KwikrAdapt` in Skype. This section outlines how we implemented `KwikrAdapt` in Skype and the modification we did in Skype's code base.

6.6.1 Handling Wi-Fi Congestion

Skype's bandwidth estimation is based on a Kalman Filter. At a high level, a Kalman Filter estimates the state of a system, combining a predictive model with ongoing noisy observations. We integrate the `Ping-Pair` information by treating congestion as an indication that the observations (i.e., the packet delay) is noisier than normal since the cross-traffic is corrupting the data. Hence we report a higher variance to the Kalman Filter, which correspondingly decreases its response to the congestion.

In more detail, Skype uses an Unscented Kalman filtering (UKF) [55, 97] of the leaky bucket state-space model (see [81]). UKF is based on the Unscented Transform, a method for computing the statistics of a random variable that undergoes non-linear transformation. It operates by sampling the state distribution at selected *sigma points* to obtain a Gaussian representation which is amenable for propagation through a Kalman Filter.

The model parameters are channel data backlog $N(k)$ and serving bandwidth $BW(k)$, making the approximated Kalman filter observation equation:

$$d(k) = \frac{N(k)}{BW(k)} + e(k) \tag{6.1}$$

where k is a packet index and $d(k)$ is the packet one-way delay, calculated from transmission and reception timestamps and compensated for sender/receiver clock offset and channel propagation delay through a minimum tracking mechanism. $e(k)$ is the Kalman filter observation noise, and this is our attack point: when the `Kwikr` hint derived from `Ping-Pair` indicates that there is a queue build-up not caused by Skype, we increase its variance and allow it to have non-zero positive mean.

By using a non-zero positive mean, we signal that not all delay is caused by Skype, making it converge towards a higher bandwidth. And by increasing the noise we introduce greater uncertainty, causing less weight to be given to the delay observation, thus slowing down adaption speed. In the unscented Kalman filter, both effects are achieved by offsetting the positive sigma-point corresponding to the observation noise, by adding a term to the base observation noise variance σ_e^2:

$$\chi = \frac{\hat{N}(k)}{B\hat{W}(k)} + \lambda\sqrt{\sigma_e^2 + \beta c^2} \tag{6.2}$$

where χ is the sigma-point in question, $\hat{N}(k)$ and $B\hat{W}(k)$ are the current Kalman Filter estimates of the backlog and the bandwidth, respectively, λ is given by the degrees of freedom in the unscented Kalman filter, β is an empirically tuned boost factor (we found $\beta = 4$ to be adequate), and c is the delay due to other traffic (i.e., cross-traffic).

Therefore, while the measured $d(k)$ increases in Equation 6.1, by offsetting the sigma-point for the positive observation noise, we ensure that the bandwidth estimate $BW(k)$ does not needlessly collapse, or could even increase if $BW(k)$ is low and c is high.

6.6.2 Handling Wi-Fi Link Fluctuations

To handle link fluctuations, we use a basic two-state model for the Wi-Fi link with "good" and "bad" states. When the link is good, the bandwidth estimation operates normally; when the link is bad, the bandwidth estimate is adjusted accordingly.

We studied the relationship between RSSI and MAC-layer data rate by empirically obtaining 3200 data samples. We see a rapid transition from a high data rate of over 60 Mbps to a low rate of under 10 Mbps as the RSSI transitions from about -77 dBm to about -87 dBm. This rapid transition lends support to the two-state link model and suggests a simple detection procedure: a threshold of -79 dBm separates the good link state from the bad. We add hysteresis by computing a 3-second average of RSSI, with sampling done every 100 ms (i.e., 30 samples), and compare this with the threshold.

When a link is bad, we adjust the estimated available bandwidth. Notice that this is different from the case of congestion (see Section 6.6.1) where cross-traffic corrupted the bandwidth observations and so we increased the variance of the data input to the Kalman Filter. Here the bandwidth observations remain accurate (i.e., the link really is changing in quality), and our goal is to damp the fluctuations in the data rate due to changing link quality. This would provide users a lower but consistent quality as opposed to wild swings in quality.

Therefore, when a link is bad, we modify the bandwidth estimation produced by the Kalman Filter. This modified estimate is then used by Skype to choose a data rate. Specifically, we apply a multiplicative correction factor, multiplying the estimate by the ratio of the average MAC layer data rate seen when the link is bad to that seen when the link is good. For instance, if the average bad rate is 10 Mbps while the average good rate is 50 Mbps, the bandwidth estimate obtained from the Kalman Filter is multiplied by a correction factor of 0.2. This limits the magnitude of the swings in bandwidth that would have otherwise been present.

Finally, we impose a lower bound of 100 Kbps on the bandwidth estimate; this rate is, in our experience, the minimum needed for a passable AV call. This does *not* mean that 100 Kbps is the minimum stream rate; the application is free to send at a lower rate, say by switching to an audio-only call.

6.6.3 Handling Wi-Fi Handoffs

As noted previously (see Sections 6.2.3), in Skype, the receiver automatically suppresses any reports of packet loss during handoffs, because no reports get through during the period of disconnection. Thus we do not have to do anything special in this regard based on a `Kwikr` handoff hint.

Next, we consider the case of a handoff from a poor cell to a good cell (poor→good). This is the common case for handoffs, as handoff is typically triggered only when the current connection has become too poor to sustain (whether because of load in the cell or because the client has moved to the edge of the cell). Since the bandwidth available in the new cell could be quite different from what the Skype bandwidth estimator had converged to in the old cell, we reset the Kalman Filter to a seed bandwidth of 250 Kbps and then trigger bandwidth rediscovery. (Note that we work with a higher seed bandwidth of 250 Kbps here compared to the floor of 100 Kbps used in Section 6.6.2 because we expect bandwidth to improve after a handoff whereas there we expect bandwidth to be low in a weak link situation.) This procedure enables Skype to attain a higher bandwidth in the new cell much more quickly than it would otherwise have.

We also consider the reverse, where the client hands off from a cell where it enjoyed high bandwidth to one that is very congested and so offers low bandwidth (good→poor). In such a case, there is the risk of overshooting the available bandwidth, resulting in delay and packet loss. However, it turns out that Skype's conservative approach to bandwidth estimation, wherein it backs off sharply in the face of (congestion-induced) delay, ensures that no such overshooting happens. (Indeed, this conservative behaviour, which is unnecessary in some cases, is what motivated the adaptation presented in Section 6.6.1.)The bandwidth estimated by Skype falls sharply, regardless of whether bandwidth rediscovery is triggered. This obviates the need for anything special to be done in the case of a good→poor handoff.

6.6.4 Implementing `KwikrAdapt` on Skype on Android

Our main implementation effort is centered on Skype on Android. This application is largely written in C, with a UI shell in Java that loads a C library using JNI. Our implementation spanned the following three components (the lines of code of our implementation are noted within parentheses):

1. **Platform Interface (PI)** (50 LOC): this is a platform-specific module responsible for interfacing with the OS to obtain the BSSID, RSSI, and MAC data rate information.
2. **Probing Module (PM)** (1400 LOC): this is also platform-specific and orchestrates all `Kwikr` detections, including `Ping-Pair` , handoffs, etc.
3. **Bandwidth Estimator (BE)** (50 LOC): this module is platform-independent and the only component that lies in the data path, i.e., actually receives the stream packets. BE implements `KwikrAdapt`'s modifications to the Unscented Kalman Filter based bandwidth estimator.

The operation starts with PM being invoked at the beginning of a call. PM then obtains information about the default gateway (the Wi-Fi access point), which it probes at regular intervals to get the queuing delay. Each probe records the start/stop time (to determine the number of packets sandwiched between the `Ping-Pair`), queuing delay and the instantaneous MAC data rate (to infer the transmission speed). This information is then passed on to `Kwikr`, which counts the number of Skype packets sandwiched between the start and stop times, and computes the contribution of Skype to the queuing delay. Finally, this information is used to adjust the UKF, as discussed in Section 6.6.1.

As discussed in the chapter on `Kwikr`, raw sockets cannot be used with root privileges. So our implementation uses the built-in `ping` utility on Android, which supports setting the TOS bits. Finally, our implementations of the detections and adaptations for link fluctuations and handoffs also center around the same 3 modules as noted above.

6.7 Evaluation using Skype

We ran several controlled experiments of `KwikrAdapt`-enabled Skype to evaluate the effectiveness of the improved bandwidth adaptation mechanism, looking at congestion, handoff, and mobility scenarios. We also look at scenarios where `KwikrAdapt`-enabled clients co-exist with non-`KwikrAdapt`-enabled clients, showing that the improvement is not coming at the expense of other users.

6.7.1 Congestion

We first show that `KwikrAdapt` enables a better response to congestion. We ran 50 experiments, half with unmodified Skype clients and half that were `KwikrAdapt`-enabled. We initiated a three minute calls where we introduced congestion in the form of TCP flows during the middle minute.

Figure 6.15(a) shows a representative execution, where the shaded region represents congestion. We see that Skype with `KwikrAdapt` maintains a higher data rate during the congested period, since it is determined that Skype is not a significant cause of congestion. It then recovers quickly when the congestion ends.

Figure 6.15(b) shows a CDF of the average data rate for each call, and we see that overall `KwikrAdapt` enables a higher data rate. On average, the calls with `KwikrAdapt` had a 20% higher data rate. We also see in Figures 6.15(c) and 6.15(d) that the round-trip time and loss are not hurt by `KwikrAdapt`. (Instead of a CDF, for clarity here we simply note a few percentiles.)

Thus, we see that `KwikrAdapt`-enabled Skype avoids backing off sharply when congestion is due to cross-traffic and recovers more quickly.

We also consider the scenario where the congestion is self-inflicted, i.e., where Skype really needs to reduce its bandwidth usage. We ran experiments where the bandwidth was artificially throttled mid-stream, and initiated three minute calls with both regular and `KwikrAdapt`-enabled Skype. We see in Figure 6.16 that `KwikrAdapt` does not affect normal

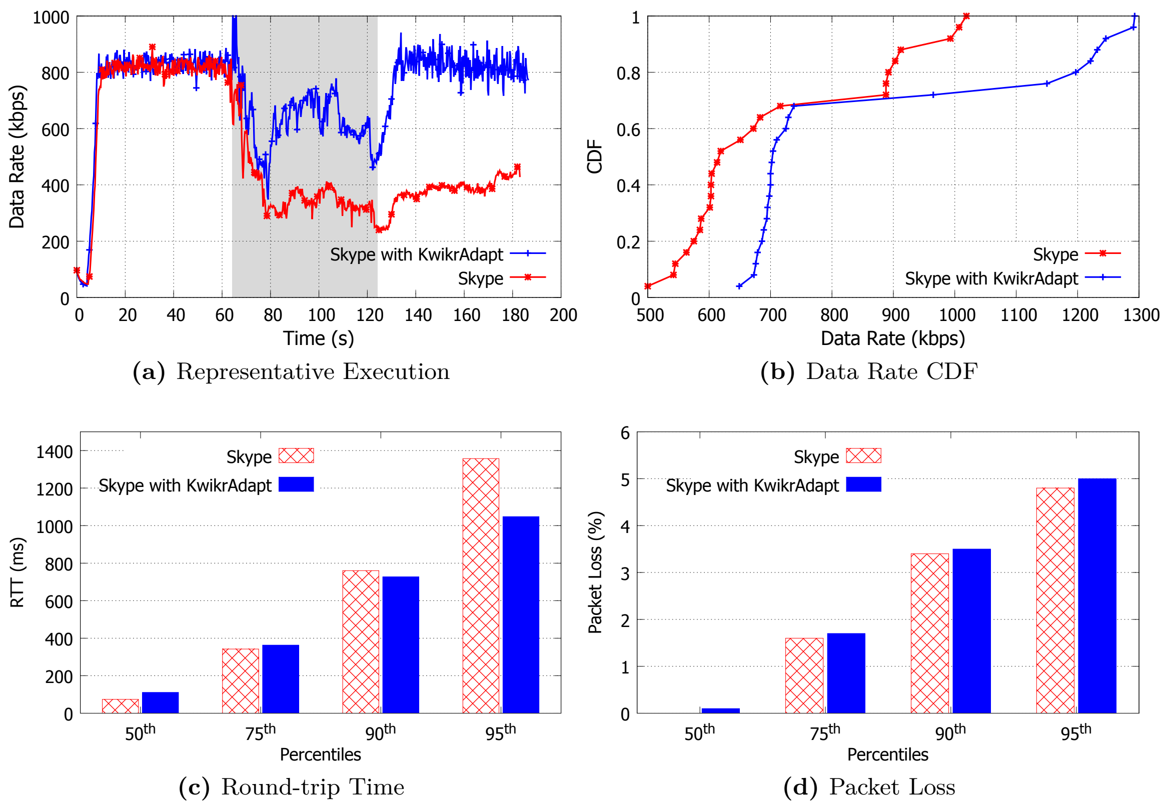


Figure 6.15: Adaptation in congested scenarios using `KwikrAdapt`. Figure (a) shows a representative execution showing how `KwikrAdapt` responds more aggressively to congestion (during the shaded interval). Figure (b) shows a CDF of data rate, showing the higher data rate achieved by `KwikrAdapt` . Figures (c) and (d) shows that the round-trip time and packet loss are not hurt by the more aggressive adaptation strategy.

bandwidth adaptation when congestion is self-inflicted.

Finally, we looked at the performance of `KwikrAdapt` when co-existing with other clients. We ran 30 experiments in which two clients made simultaneous two-minute calls—for ten of the calls, both clients were `KwikrAdapt`-enabled; for another ten calls, one client was `KwikrAdapt`-enabled and the other was not; for the final ten calls, both were unmodified Skype clients. We see in Table 6.1 that in general, co-existence does not have a significant impact on data rate. The non-`KwikrAdapt`-enabled clients are essentially unaffected by co-existing with `KwikrAdapt` clients. By contrast, two `KwikrAdapt`-enabled clients running together do appear to be slightly affected.

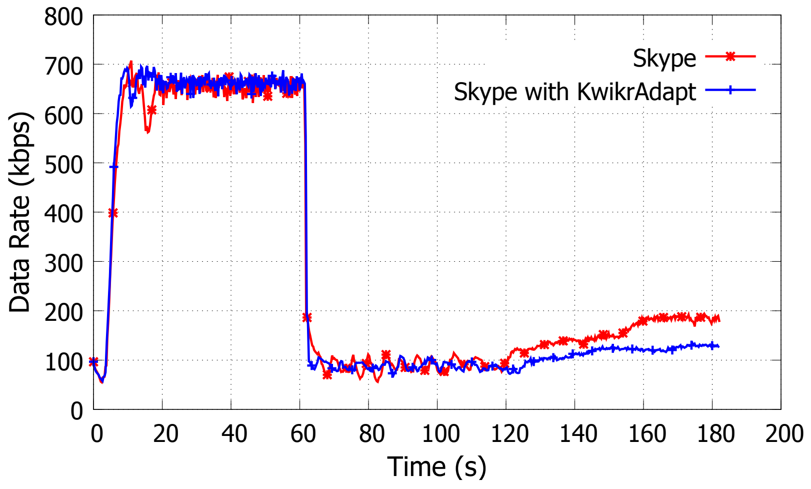


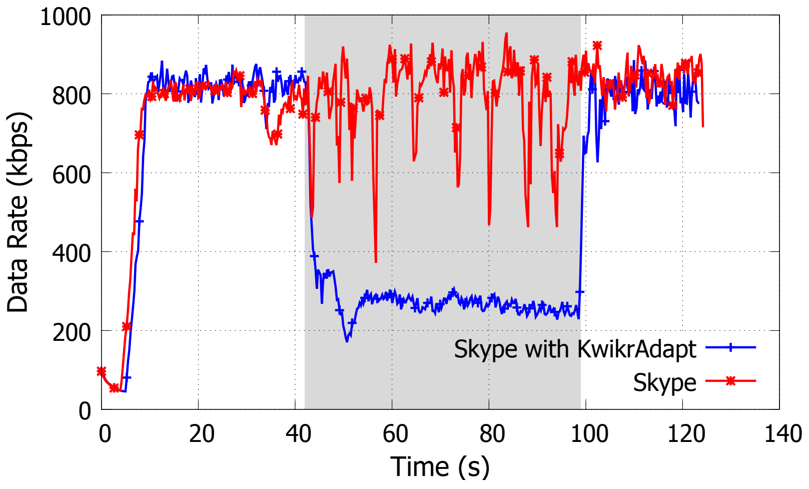
Figure 6.16: When congestion is self-inflicted, `KwikrAdapt` reduces the rate appropriately.

Table 6.1: `KwikrAdapt` flows co-existing with other flows. Table contains the data rate for the measured flow when running simultaneously with the specified background flow.

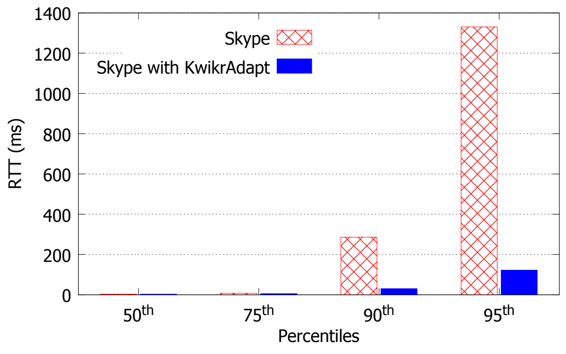
	Background Flow	
Measured Flow	Skype	Skype with KwikrAdapt
Skype	512 kbps	507 kbps
Skype with KwikrAdapt	542 kbps	483 kbps

6.7.2 Link Fluctuations

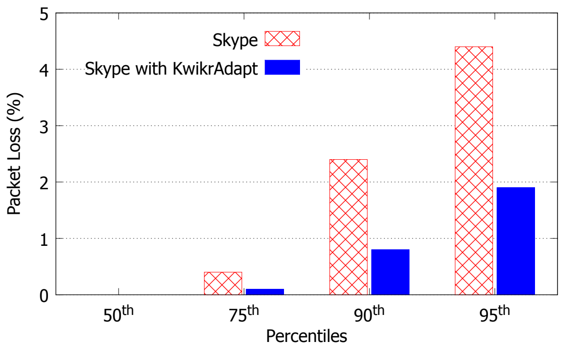
Next, we look at mobility. We made two minute calls where we transitioned between a strong link and a weak link and back. There are 20 such calls for both the regular Skype client and the `KwikrAdapt`-enabled Skype client. Figure 6.17(a) gives a representative execution. Regular Skype's bandwidth estimate remains high (because it has built-in mechanisms to reject delay spikes and not treat these as indications of congestion) but its data rate fluctuates as the underlying link rate fluctuates. In contrast, `KwikrAdapt` detects the link fluctuations and successfully smooths the data rate, reducing the significant fluctuations suffered by the regular Skype client. While the total throughput may be reduced, in this case there are significant advantages to reducing the data rate: the `KwikrAdapt`-enabled client sees a lower round-trip time (see Figure 6.17(b)) and a lower packet loss rate (see Figure 6.17(c)).



(a) Representative Execution



(b) Round-trip Time



(c) Packet Loss

Figure 6.17: `KwikrAdapt` in the case of mobility leading to link fluctuations. Figure (a) shows a representative execution demonstrating how `KwikrAdapt` responds more smoothly to link fluctuations (during the shaded interval). Figures (b) and (c) show how the round-trip time and packet loss rate are improved by the more stable bandwidth estimate.

6.7.3 Handoffs

Next, we look at the issue of handoff. We forced a handoff to take place in the middle of the call, moving from a low bandwidth cell to a higher bandwidth cell. We compare a regular Skype client with a `KwikrAdapt`-enabled Skype client, running each 24 times. In Figure 6.18(a), we observe a representative execution. Notice that after the handoff, the `KwikrAdapt`-enabled client reevaluates the available bandwidth, significantly increasing the data rate; by contrast, the regular Skype client maintains a low data rate. In Figure 6.18(b), we give the CDF for the data rate across all experiments.

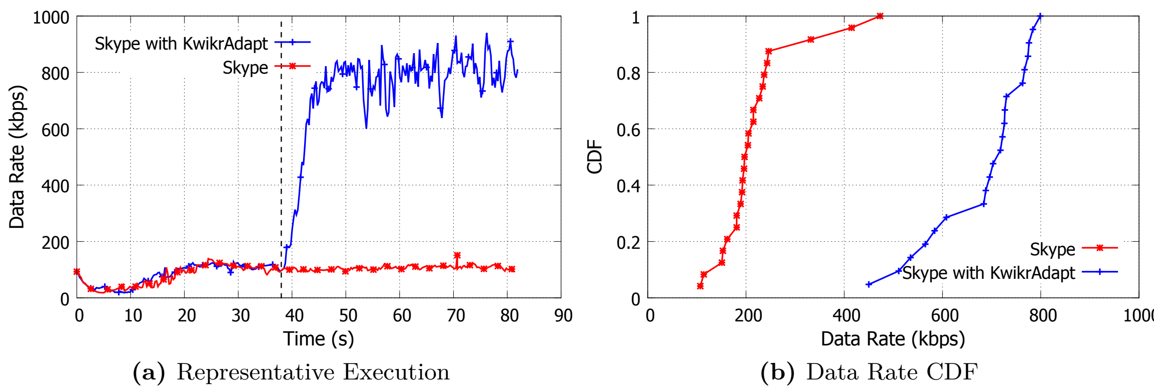


Figure 6.18: `KwikrAdapt` in the case of handoffs. Figure (a) shows a representative execution demonstrating how `KwikrAdapt` responds to a handoff by updating the bandwidth estimate. The vertical dashed line shows the point in time the handoff occurred. Figure (b) show a CDF of the data rate, showing much improved performance.

6.8 Chapter Summary

In this chapter, we showed that when real-time streaming applications perform end-to-end bandwidth adaptation in the presence of Wi-Fi bottleneck conditions, the quality of the streams suffer. We presented a user-level module named `KwikrAdapt` that real-time streaming applications can use to take Wi-Fi-aware bandwidth adaptation decisions. Through empirical studies, we showed that `KwikrAdapt` helps real-time flows know when they can be less conservative in backing off, which helps increase the quality of the streams while incurring no additional cost. It knows when the client is mobile so that flows can proactively adapt to changing Wi-Fi channel conditions rather than reacting when problems occur in the streams. And finally, it ramps up faster after a handoff rather than going through the slow bandwidth probing phase that makes ramp-ups quite slow. Through empirical studies using TFRC and Skype, we showed how `KwikrAdapt` can allow real-time flows achieve much higher data rates while keeping delays and losses at bay during congestion periods, reduce the worst loss percentage and RTT spikes during link fluctuations, and handle handoffs much gracefully to provide higher quality streams to the user.

Chapter 7

Bottleneck Alleviation via Link Diversity

7.1 Introduction

Wi-Fi networks have been patterned after Ethernet that preceded them. Just as a host plugs into an Ethernet port, a Wi-Fi client "plugs into" a single access point in its vicinity through the process of association, and remains associated with it until the link is broken, say due to mobility.

We argue that this approach, inspired by the wired world, is suboptimal and needs to be revisited. At any point, the client typically has a choice of multiple Wi-Fi links — not merely Wi-Fi access points within range but in fact ones that the client has the credentials to connect to. This is especially so in settings such as enterprises, schools, airports, hotels, malls, etc. where there is often a single entity that has deployed a large number of access points.

The quality of a wireless link varies frequently, causing sporadic bursts of packet loss, which can severely impact real-time streaming due to its sensitivity to packet loss, delay, and jitter. Therefore, rather than performing *selection*, i.e., picking one "best" link to connect over, we argue that the client would be better off *hedging*, i.e., connecting over multiple links simultaneously. If the client's traffic were then replicated across these links, the client would then enjoy the performance offered by the best link, even as which link is the best changes frequently.

However, replicating traffic over multiple links would mean a substantial overhead. To address this challenge, we present `DiversiFi`, which leverages *network-side buffering*, whether at existing Wi-Fi access points or at a separate “middlebox”, to provide much of the benefit of diversity through replication but with little of its overhead. *This combination of benefit without the overhead is a key novel contribution.*

Specifically, to make the case for `DiversiFi`, we focus on *real-time, interactive streaming* (e.g., voice-over-IP (VoIP)) [1], which places stringent demands on the network in terms of end-to-end latency, jitter, and packet loss rate, much more so than on-demand streaming (e.g., YouTube), which has the luxury of a large playout delay. For instance, the round-trip latency must be under 300 ms for VoIP [12] and under 100 ms for interactive games [65], for otherwise the user would notice a lag. Also, the packet loss rate for VoIP should ideally be under 1% [91], although burst losses can cause noticeable artifacts even at lower loss rates.

In Section 3.2, we presented experimental data to show that the Wi-Fi last hop is a significant factor in poor streaming quality. Our analysis drew on both a year’s data from a large VoIP provider that serves hundreds of millions of users, and also our own experiments with the participation of 274 users across 22 countries and involving 9224 simulated VoIP calls over a 2-month period. From our experiments, we showed that the overall poor call rate (PCR) is over 10%. Further, we showed that for both the VoIP provider’s dataset and ours, the *relative* difference in PCR between Wi-Fi and wired clients is 40-50%.

Therefore, we focus specifically on the Wi-Fi last hop and make the case for leveraging link diversity to improve real-time streaming performance. We argue that clients often have more than one access point within range that they could connect to. Furthermore, the performance of the links to these multiple access points is only weakly correlated, say in terms of the pattern of packet loss. Thus, replicating real-time traffic over multiple links — we focus on doing so over two links, a *primary* and a *secondary* — yields significant benefits. We term this approach *cross-link replication.* We show that the diversity benefit of such cross-link replication dominates both fine-grained link selection and also temporal replication (i.e., retransmission with an offset) over a single link. It also provides significant benefits over and above PHY-layer spatial diversity in MIMO systems such as 802.11ac. Our experiments with a client equipped with two Wi-Fi NICs show that cross-link diversity helps cut down

[1]As noted in Section 7.3, we focus here on the downlink.

the poor call rate (PCR) for VoIP by 2.24x relative to single-link transmission.

Having established the benefits of cross-link replication, we turn to how this could be realized in practice, *without* requiring two NICs, without incurring the overhead of duplicated traffic or, importantly, without *impacting other, non-realtime applications.* Our general approach involves confining cross-link replication to real-time flows, with non-real-time traffic continuing to flow over a single, default link, as it would have in the absence of `DiversiFi`. Further, the replicated packets are held in network-side buffers, close to the client but yet not being transmitted over the air unless the need arises (i.e., a packet is lost on the primary link). We show how a simple change to make the access point's queuing policy *head-drop* instead of the usual tail-drop, coupled with a shortening of the queue length, enables an efficient realization of `DiversiFi`. We also present an alternate approach, which leaves the access points unmodified and moves the buffering into a network middlebox. The introduction of the middlebox also helps avoid the overhead of duplicating the stream over the WAN.

Our experimental results confirm that `DiversiFi` provides the benefit of replication for real-time flows (e.g., a reduction in PCR from 4.9% down to 0% over 61 runs), with minimal overhead (only 0.62% of the packets being duplicated wastefully compared to nearly 100% with naive duplication). Importantly, a competing TCP flow, which is oblivious to `DiversiFi` being used for the real-time flows, only suffers a throughput degradation of 2.5%.

Thus, `DiversiFi` *imposes a minimal duplication overhead and ensures coexistence with competing, non-real-time applications, which remain oblivious to the use of* `DiversiFi` *for improving the performance of real-time flows.*

In summary, our contributions in this work include:

- An empirical analysis to show the significant benefits of cross-link replication.
- An architecture and implementation to gain the benefits of cross-link diversity without its overhead to improve the performance of real-time streams.

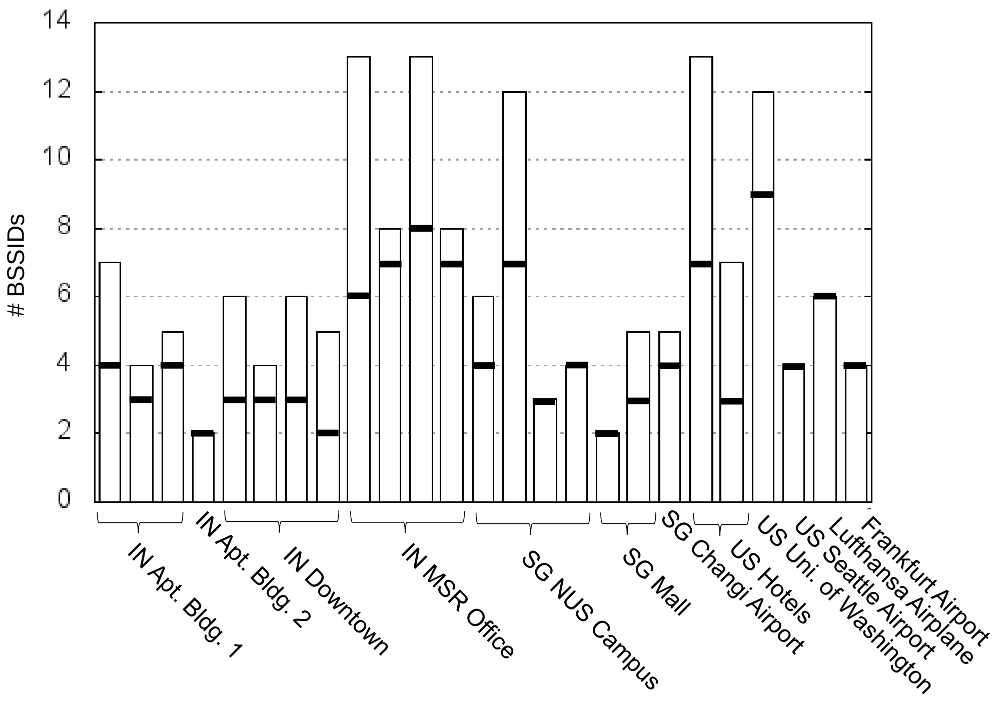


Figure 7.1: The number of BSSIDs (bars) and distinct channels (thick black dashes within each bar) the client could connect to at various locations. Across all locations, the client could connect to at least 2 different BSSIDs.

7.2 Leveraging Link Diversity

In this section, we discuss the merits of cross-link packet replication across multiple Wi-Fi links in improving the reliability of real-time streams.

7.2.1 Availability of Multiple Wi-Fi Links

As the key idea in `DiversiFi` is to leverage link diversity to improve the quality of real-time streams, we look at availability of multiple Wi-Fi access points that a user could potentially connect to. We went to various enterprise and public locations, such as offices, campuses, serviced apartments, hotels, malls, etc., across 3 cities — Bengaluru, Seattle, and Singapore — in different countries and determined how many BSSIDs were within range on the networks that our client could connect to. Figure 7.1 presents the results. We see that the number of BSSIDs available was 6 at the median, with at least 2 across all locations and up to 13 in some locations. Even on an in-flight Wi-Fi network the client had a choice of 6 BSSIDs!

Even if we only count the number of distinct channels corresponding to the BSSIDs seen (marked by the bold dashes in Figure 7.1), to discount the possibility of virtual access points, the median count is 4, with the range spanning 2 to 9. We conclude, therefore, that there is ample scope for `DiversiFi` to operate in such locations.

The situation, however, is less rosy in residential locations. Indeed, in our NetTest dataset discussed earlier (Section 3.2.2), which is skewed towards residential locations, we found that in only 30% of the cases did the client device have more than one BSSID that it could connect to. Nevertheless, we note that even in such locations, the incipient trend towards multi-band access points would make multiple Wi-Fi links available to clients.

Although the availability of multiple Wi-Fi links does not by itself indicate the performance of these links, the results presented in Section 7.2.2 show that even a weak secondary link could be very beneficial.

7.2.2 Benefits of Cross-Link Replication

To make the case for leveraging link diversity through replication in `DiversiFi`, we consider several questions:

1. How does *replication* for diversity compare with fine-grained link *selection* proposed previously (e.g., [69])?
2. How does the diversity arising from *cross-link replication* (in particular, the duplication of a stream concurrently over two links) compare with that from *temporal replication* over a single link?
3. Does *cross-link replication* provide benefits over PHY-layer spatial diversity in *MIMO*?

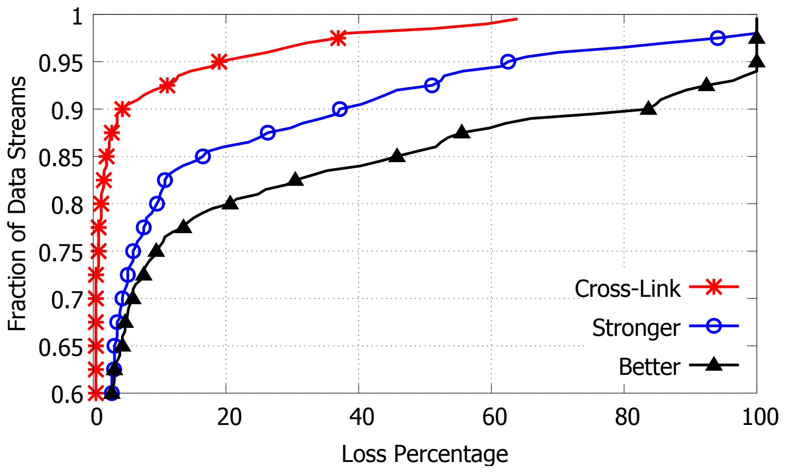
Our analysis in this section focuses solely on the benefits of diversity through replication. *Later, in Section 7.3, we explain how this benefit can be had without the overhead of duplicating traffic or requiring clients to have two NICs.*

Unless stated otherwise, the data presented in the sub-sections that follow is based on experiments conducted with G.711-like UDP data streams, with a data rate of 64 kbps, 160-byte packets, and a 20 ms inter-packet spacing. We used a Lenovo W520 ThinkPad laptop as the client, with an Intel Centrino Ultimate-N 6300 AGN internal Wi-Fi NIC and a TP-Link TL-WDN3200 dual-band ABGN USB adapter. Each experiment comprised a 2-minute simulated call during which a copy of the G.711-like stream was sent to each NIC of the client. With this setup, we gathered data for 458 such simulated calls, at a variety of locations, including offices, serviced apartments, downtown areas, and a conference setting in Bengaluru and Singapore. The dataset includes traces corresponding to various challenging situation such as a weak link, client mobility, external interference from a microwave oven, and network congestion. The experiments were conducted on both 2.4 GHz and 5 GHz networks depending on what were available in the locations we conducted experiments at.

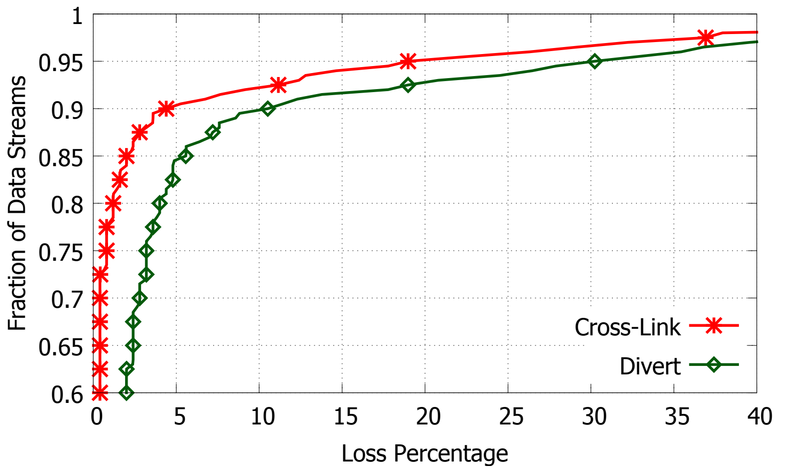
In terms of the metrics, we consider both network-level ones such as the packet loss rate and the one-way delay jitter, and also a voice call metric, the poor call rate (PCR). For the network-level metrics, we divide the simulated call into 5-second periods and focus on the worst such period, for there is evidence that worst degradation in a (short) call is a significant determinant of the overall user-perceived call quality [98]. For voice call metrics, we run the network trace through the G.711 codec, and use the degree of interpolation and extrapolation of voice samples to estimate PCR, in accordance with well-established models [17, 18]. Note that the trace captures all network-related impairments that a stream may suffer, including packet delay, jitter, and loss.

Cross-Link Replication vs. Link Selection

In the presence of multiple Wi-Fi access points in the vicinity, *selection*, i.e., picking the one "best" link out of many choices, is what OSes typically do today. The question is how *replication* compares with this simple strategy of selection. To evaluate this question, we have the two client NICs connect to the two strongest access points, and then implement two selection strategies: `stronger`, which picks the stronger of the two links based on the RSSI (as is typically done in OSes), and `better`, which samples both links for the first 5 seconds and settles on the one that performs better during this trial period for the rest of the call. We compare these strategies to `cross-link`, which replicates the stream on both links.



(a)



(b)

Figure 7.2: CDF of loss rate over worst 5-second period, comparing `cross-link` replication to (a) link selection based on `stronger` and `better`, and (b) fine-grained link selection (`Divert`). Compared to all link selection strategies, `cross-link` replication helps to reduce the loss percentage significantly.

Figure 7.2(a) shows the CDF of the packet loss rate during the worst 5-second period for the three strategies noted above. The figure shows that `cross-link` dominates both selection strategies, especially in the tail. For example, while the 90^{th}%tile packet loss rate is 37% and 84%, respectively, for the `stronger` and `better` selection strategies, it is only 4.4% for the `cross-link` replication strategies.

There has also been work on *fine-grained* link selection, wherein a fresh selection is made and the link switched, frequently. Divert [69] is an example of such a strategy. We evaluate

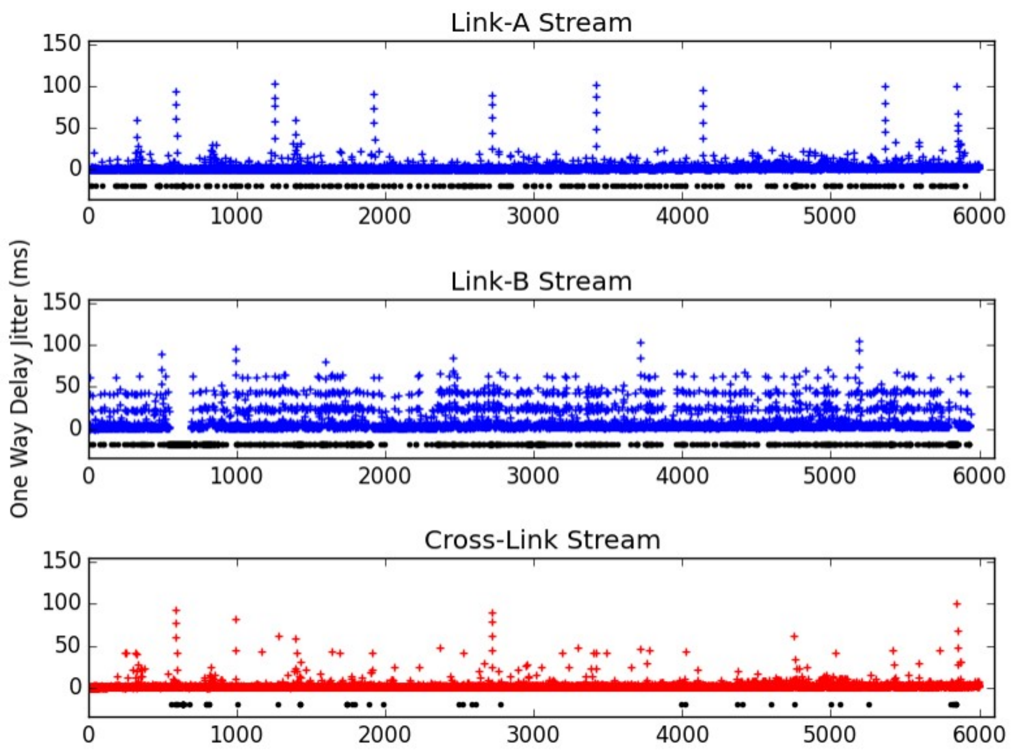


Figure 7.3: An example trace showing how cross-link replication over a weak link A (loss rate: 4.3%) and an even weaker link B (loss rate: 15.4%) yields a much lower loss rate of 0.88% (losses are shown as black dots along the bottom of each plot) and also a lower delay jitter (plotted on the y axis).

this strategy, `Divert`, where a link switch is triggered if the number of frames lost exceeds a threshold T within a window H frames. We use a setting of $H = 1$ and $T = 1$, as in [69]. Figure 7.2(b) shows the CDF of packet loss rate for the worst 5-second period. We see that `Divert` performs worse than `cross-link`, with the 90^{th}%tile packet loss rate being 10.5% for `Divert` compared to 4.4% for `cross-link`. The key observation is that while the switching of links in `Divert` is triggered by one or more packet losses, the switching is only in the hope of improved likelihood of reception for *future* packets. The packets that were lost before the switch are themselves not recovered. On the other hand, in `cross-link`, the receiver has the benefit of reception on *both* links, so packets that are lost on one link will likely be received on the other.

As a final illustration of how cross-link replication is qualitatively different from link selection, we show, in Figure 7.3, the one-way delay and packet loss characteristics of two of our

G.711-like UDP streams in the case of two weak links: link A with an overall packet loss rate (i.e., for the entire 2-minute simulated call, not just the worst 5-second period) of 4.3% and link B with an overall packet loss rate of 15.4%. With traditional link selection, link A would clearly dominate B, so the client would be better off just sticking to A. However, with cross-link replication across A and B, the overall packet loss rate drops to 0.88%. In other words, *the better of the two links (link A) benefits from replication over a significantly worse link (link B), showing the benefit of diversity.*

Cross-Link vs. Temporal Replication

If replication is the key to improving performance, a natural question is why do *cross-link* replication. Would temporal replication, i.e., sending multiple copies of each packet spaced over time, on the same link offer the same benefits? Note that any such replication would be over and above link-layer retransmissions that already happen in the 802.11 MAC layer. However, unlike the latter, which would tend to happen on a very fine timescale (tens of μs), the temporal replication (`temporal`) would happen over a much longer timescale. The benefit of a larger temporal spacing is the greater diversity in the channel conditions across the multiple transmissions. However, the temporal spacing would be constrained by the deadline for the real-time stream.

We consider `temporal` with two copies of each packet being transmitted, with a spacing Δ between them of up to 100 ms. (Given the 300 ms limit the end-to-end round-trip for VoIP [12], 100 ms one-way on just the Wi-Fi hop would seem to be a reasonable limit.) Figure 7.4 shows the CDF of the packet loss rate for the worst 5-second period for `temporal`, with Δ ranging from 0 ms (i.e., two copies of each packet being transmitted back-to-back) to 100 ms (i.e., the copies being spaced apart by 100 ms). We find that temporal replication does improve the packet loss rate compared to the baseline (i.e., no replication), with the improvement being greater the larger the spacing Δ. For instance, at the 90^{th}%tile, `temporal` with $\Delta = 100$ms has a packet loss rate of 23.7% compared to 37.2% for the baseline. However, `temporal` still underperforms `cross-link`, which has a 90^{th}%tile packet loss rate of 4.4%.

Thus, it is clear that `cross-link`, with its spatial and channel diversity (i.e., links to

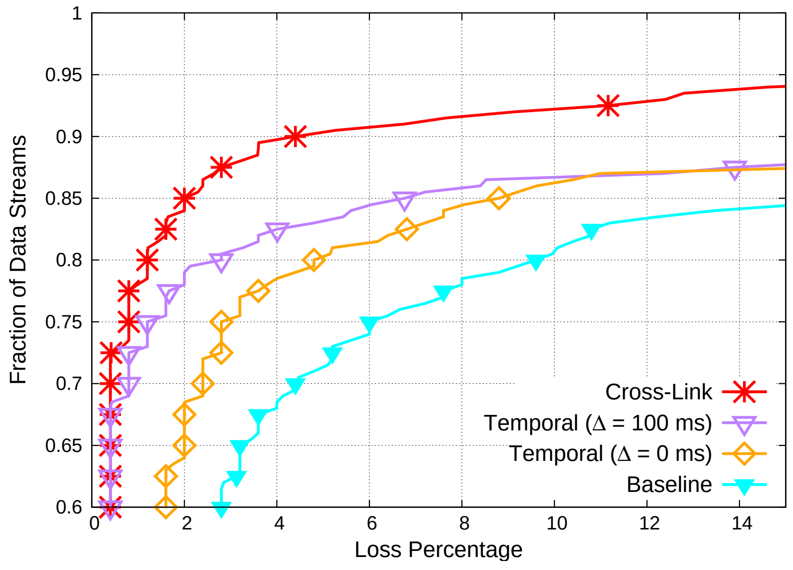
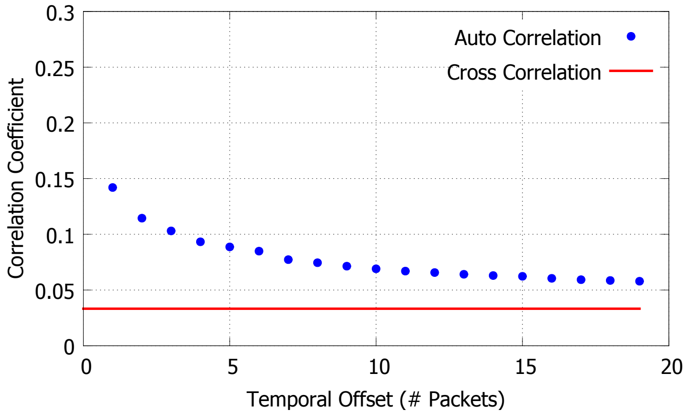


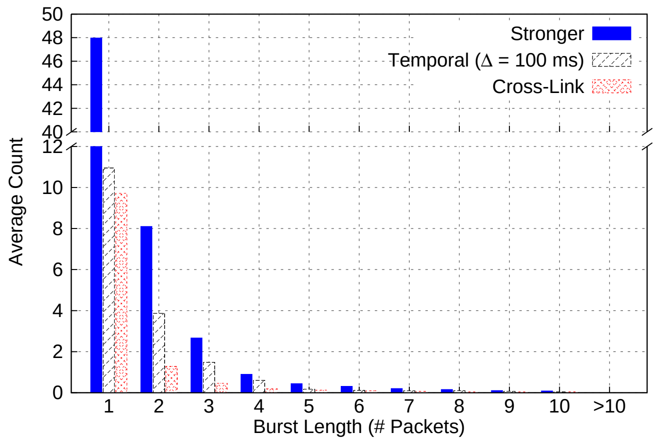
Figure 7.4: CDF of loss rate over worst 5-second period, comparing `cross-link` replication to `temporal` replication. `Cross-link` replication overperforms `temporal` replication for all the considered Δ values.

access points at different locations and on different channels, possibly even different bands), dominates `temporal`. The latter suffers from the bursty nature of impairments on a link. We illustrate this in two ways. First, we compute the auto-correlation of the packet loss time series on a link and compare that with the cross-correlation of the corresponding series across two links. Figure 7.5(a) plots the results. We see that even with a lag of 20 packets (or 400 ms, given the 20 ms inter-packet spacing), the auto-correlation is greater than the cross-correlation. Second, since packet loss bursts are particularly problematic for real-time streams such as VoIP, we compare `cross-link` and `temporal` in this regard. Figure 7.5(b) plots the distribution of packet loss burst lengths for the baseline (which is `stronger`, without any replication), `temporal`, and `cross-link`. We see that not only does `cross-link` have a lower packet loss rate than `temporal`, its losses also tend to be less bursty. For instance, out of the 6000 packets in a 2-minute simulated call, 25.6 on average were lost in `cross-link`, out of which only 15.9 were in bursts of 2 or more consecutive packets. In comparison, an average of 61.9 packets were lost in `temporal`, with a much larger proportion, 51.0, occurring in bursts.[2]

[2]These loss statistics are for the entire 2-min simulated calls while much of the preceding analysis was for



(a)



(b)

Figure 7.5: (a) Auto-correlation vs. cross-correlation of the packet loss process within a link and across two links, respectively. (b) Distribution of packet loss burst lengths for `stronger` link selection, `temporal` replication, and `cross-link` replication. `Cross-link` replication overperforms `temporal` replication because cross-correlation of packet losses across links is quite low. This also alows `cross-link` replication to make packet losses less bursty.

Benefits over and above 802.11ac with MIMO

With the advent of MIMO, Wi-Fi provides spatial diversity at the PHY layer. For example, 802.11n supports up to 4 spatial streams whereas 802.11ac supports up to 8. The question is whether cross-link replication (`cross-link`) provides any *additional* benefit when there already is PHY-layer diversity through MIMO, the gains due to which have been studied

the worst 5-sec period.

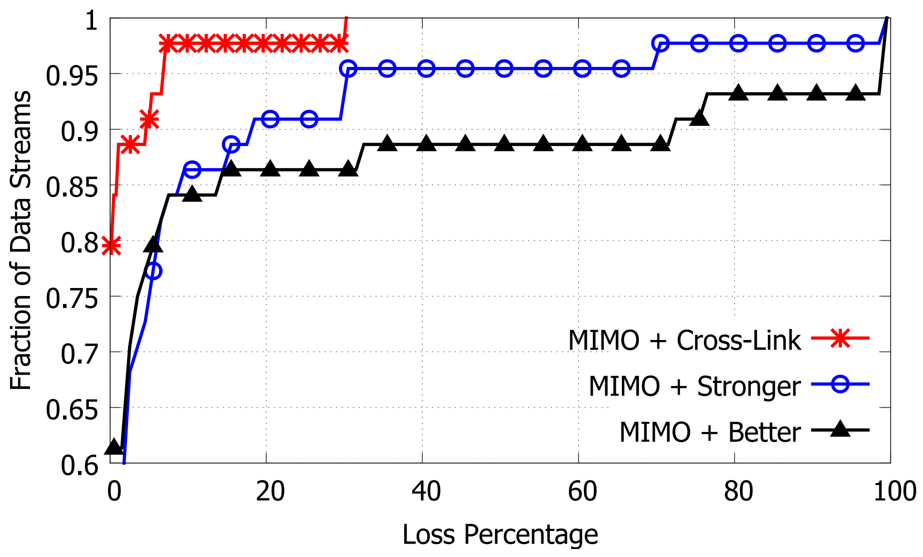


Figure 7.6: CDF of loss rate over worst 5-second period, comparing `cross-link` replication to 802.11ac with `MIMO`. `Cross-link` replication provides benefits over and above `MIMO` by reducing the loss percentage significantly.

previously [85]. Since we have no control over the access points in our experiments in the wild, we conducted experiments in the lab with 6-antenna D-Link DIR-850L dual-band 802.11ac access points and 2-antenna Linksys WUSB6300 dual-band 802.11ac NICs on the client. We gathered data for 44 simulated calls and, as before, plot the CDF of the packet loss rate for the worst 5-second period in Figure 7.6. We see that `cross-link` has a lower loss rate than `MIMO` alone.

Thus, `cross-link` provides benefits over and above `MIMO`. Why is this? The PHY-layer diversity in `MIMO` arises because the separation, on the order of the carrier wavelength, of multiple transmit (or receive) antennas, results in the multipath fading of the spatial streams being largely independent [66]. However, there are other link impairments that such PHY-layer diversity does not address. For example, with shadowing and external interference, all co-channel spatial streams could be affected simultaneously.

Improvement in VoIP Quality

So far we have focused on packet-level metrics such as the loss rate. However, the question remains as to how cross-link replication would impact VoIP quality. Figure 7.7 plots the

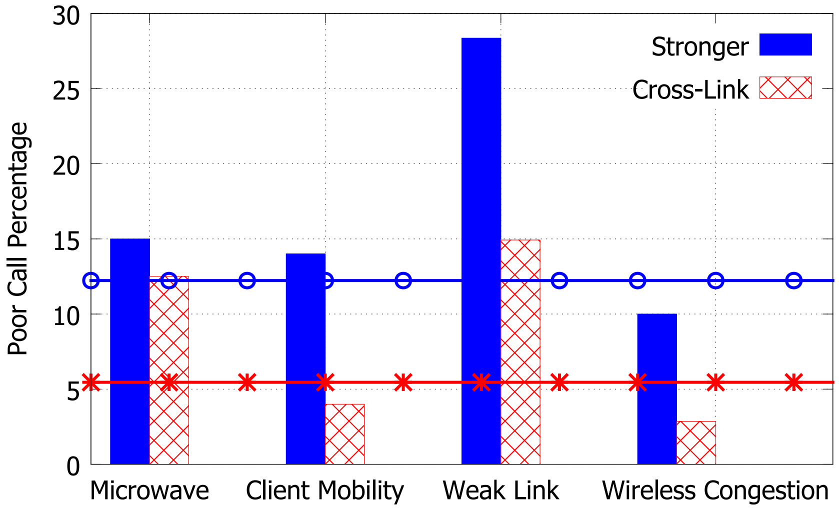


Figure 7.7: The poor call rate (PCR) with `stronger` link selection and `cross-link` replication, broken down by various impairments. `Cross-link` replication helps to reduce the PCR by 2.24× with respect to the `stronger` link selection strategy.

estimated poor call rate (PCR) for the set of simulated calls. We see that `cross-link` helps cut down the PCR by 2.24× relative to `stronger`, from 12.23% to 5.45%. While the relative improvement is even greater (3.5×) in the cases of client mobility and wireless congestion, it is much less (1.2×) in the case of microwave interference. This is an artifact of our experimental setting wherein a majority of the links available in the vicinity of the microwave were impacted by the interference (there were no 5 GHz links available). In general, greater diversity could be had from cross-technology replication (e.g., across Wi-Fi and 3G/4G), but keeping the duplication overhead manageable would be more challenging and we defer investigation of this alternative to future work.

Benefit for High Bandwidth Streams

We also experimented with real-time streams of a much higher data rate than VoIP, as might be typical of video or gaming. We performed 80 runs, each lasting 2 minutes, with a 5 Mbps stream, comprising 1000-byte packets, with a 1.6 ms inter-packet spacing. Figure 7.8 plots the CDF of the packet loss rate for the worst 5-second period. Again, we see a significant benefit from `cross-link` replication, with the 90th%tile packet loss rate improving to 1.7%

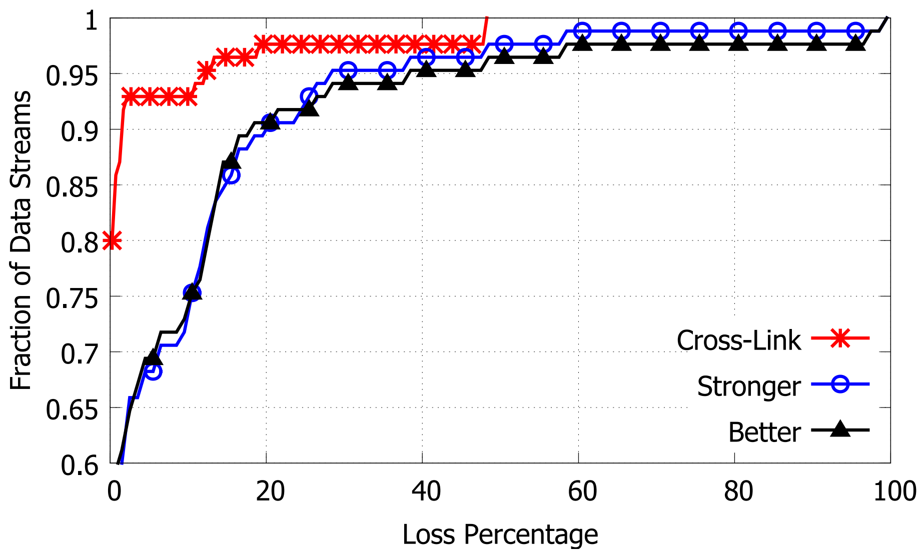


Figure 7.8: CDF of loss rate over worst 5-second period, comparing `cross-link` replication to link selection based on `stronger` and `better` for high-rate 5 Mbps streams. `Cross-link` replication is capable of reducing the loss percentage of higher data rate flows.

compared to 20.5% for link selection based on `stronger`. This finding, coupled with the design we present in Section 7.3 to avoid the overhead of unnecessary duplication of packets on the air, means that `cross-link` replication can be applied to high-rate streams as well as to the low-rate ones.

7.2.3 Moving from Analysis to Design

The analysis presented above has made the case for cross-link replication. But how do we realize this in practice? In our experiments thus far, we have used a client with two Wi-Fi NICs to receive a copy of the stream on each. This is a challenge for deployment since few clients would have two Wi-Fi NICs. Also, cross-link replication done naively would mean the duplication of all packets, including the many that are received successfully on both links. This is undesirable not only because of the overhead on the recipient but also because of the negative impact it could have on other clients that share the medium, especially if the duplicated stream is a high-bandwidth one (e.g., a video or game stream).

In the next section (Section 7.3), we present our design and implementation of `DiversiFi` to achieve virtually all of the benefits of full cross-link replication but with little of its overhead.

7.3 Design and Implementation

We now present our design of `DiversiFi`, which centers on cross-link stream replication for diversity, but with network-side buffering and implicit or explicit packet selection, to limit the actual duplication of packets in the air to just (or close to) what is necessary. While we focus here only on streaming in the *downlink direction*, which is arguably the more challenging direction because of the lack of control over the access points, we believe our design would apply equally in the uplink direction and would likely be easier to implement because the client would have direct control over what packets are sent over which link and when.

7.3.1 Design Requirements

For reasons of *deployability*, in view of the large installed based of Wi-Fi, we seek a solution that does not require any changes to Wi-Fi at the hardware layer.

For reasons of *generality*, we would like our solution to be transparent to the application, be it audio/video conferencing [5, 13], cloud-based gaming [10, 14], or app streaming [8]. We would not like to depend on any application support other than characterizing the real-time stream in terms of the rate, deadlines, etc. (which the application would do anyway when using protocols such as RTP [84, 83]).

For reasons of *coexistence*, any steps taken to improve the quality of real-time streaming for a client should not be to the detriment of other traffic flows, including non-real-time ones, at that client or other clients.

7.3.2 Design Elements

Initialization

When an application initiates a real-time stream, we need to know the stream rate, packet size, and the packet deadlines, so that the network stack, whether at the local host or the remote peer, can take the appropriate actions with respect to replication, buffering, and selection, as noted below. Since real-time streaming applications are typically based on the Real-Time Protocol (RTP) [84], we can use the *payload type* field to look up the corresponding *profile* [83], without the need for modifying applications.

Multi-Link Association

To enable a client with a single Wi-Fi NIC to associate with multiple access points, we use past work on multi-link association (e.g., [32]). In a nutshell, the client creates multiple virtual adapters, each with a different MAC address, and a separate access point association. It then switches between the links, changing channels, if necessary, and using the 802.11 power save mode (PSM) to keep its association on a link alive even when it has switched away to a different link.

In `DiversiFi`, the client creates separate virtual adapters and links for the real-time stream. For instance, in the setups shown in Figures 7.9, 7.10, and 7.11, the client has two virtual adapters, labeled as *primary* and *secondary*, for the real-time stream, each associated with a different access point. The primary associates with an access point chosen based on the standard OS policy (e.g., strongest signal) and the secondary with the next-best access point. In addition, it has a default virtual adapter, DEF, used for all of the other (non-real-time) traffic (not shown in the figure).

At any point in time, the primary real-time virtual adapter would be associated with the same access point and on the same channel as DEF. So switching between the primary adapter used for a real-time flow and DEF used for a non-real-time flow would *not* incur any overhead.

However, switching between links on different channels, such as the primary and secondary links for a real-time flow, can take a non-negligible amount of time, e.g., 2.3 ms in our measurements reported in Section 7.4. So minimizing the frequency of switching between the primary and secondary links would be desirable, to minimize a negative impact on other flows on DEF, in line with our goal of coexistence.

Stream Replication

Cross-link stream replication lies at the heart of `DiversiFi` and the benefits reported in Section 7.2. The question is where and how the packet stream is replicated.

One option is to replicate at the source, i.e., the remote peer. However, this would not only require modification of the source, it would also entail the overhead of duplicating traffic over the entire end-to-end WAN path.

An alternative would be to replicate the stream close to the receiver, for instance, at an SDN-capable switch on an enterprise LAN. While this would avoid the overhead of duplication on the WAN, it would require the presence of an SDN-capable switch and also the ability for the receiver to configure the switch by installing suitable match-action rules [16], say using APIs such as [54].

Network-Side Buffering

While cross-link stream replication helps improve the quality of a real-time stream, it does so at the cost of wastefully duplicating (on the secondary link) the overwhelming majority of packets that are delivered successfully on the primary link itself. Packet buffering, along with packet selection discussed next in Section 7.3.2, is key to mitigating this overhead. The replicated packets are held in a buffer in the network, close to the receiving client (to help minimize the additional latency, should these need to be delivered), yet not delivered over the air (on the secondary link) unless needed.

Network-side buffering could be performed at the access points themselves, taking advantage of the power save mode (PSM), whereby the access point would start buffering packets destined to the client when the client sends a sleep message (Null frame with the Power Management bit set) and would deliver the buffered packets when the client sends a wakeup message (Null frame with the Power Management bit cleared). However, this would require slight modifications to the access point to ensure efficiency. We discuss this in detail later in Section 7.3.3 (Access Point with Minimal Modification),

An alternative, which would leave the access point unmodified, is to perform buffering at a separate middlebox. We defer a detailed discussion on this to Section 7.3.3 (Unmodified Access Points with Middlebox).

Packet Selection

While buffering close to the client allows the possibility of packets being delivered with a low (additional) latency, we need a way for the client to identify the packets of interest and have these delivered selectively. The method used for selection would depend on where buffering happens. If it is in the access point as part of the standard PSM processing, the selection is done implicitly. If it is in the middlebox, selection could be done explicitly, using the RTP sequence number and timestamp for identification. However, our current implementation uses a simpler alternative, which we will discuss below in Section 7.3.3 (Unmodified Access Points with Middlebox).

7.3.3 Design of DiversiFi

An "End-to-End" approach to realizing `DiversiFi` would be to work with an unmodified network infrastructure, in particular, unmodified access points and no middleboxes. This is illustrated in Figure 7.9. As noted in Section 7.3.2, the PSM mechanism could be used to perform network-side buffering on the secondary link. However, this can be quite inefficient.

The per-client buffer at the access point is typically managed as a tail-drop queue and moreover can grow to a large size (e.g., the default size is 64 packets in OpenWRT, and 50-500 packets in Aruba [1]). When the client, upon missing a packet on the primary link, switches to the secondary link and sends a wakeup message, the access point would only

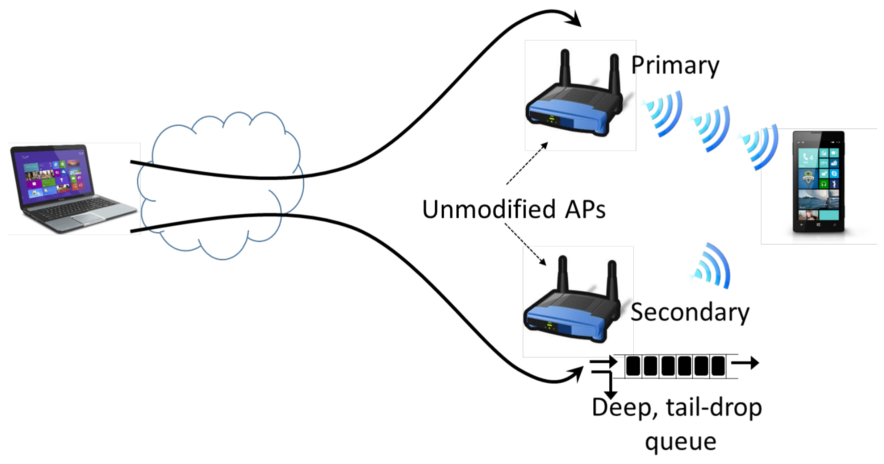


Figure 7.9: "End-to-End" architectural alternative for `DiversiFi`.

deliver the missing packet after it has delivered the possibly many packets ahead of it in the queue. This would result in a significant overhead due to unnecessary duplication of packets over the air. Further, if the tail-drop queue at the secondary access point fills up, new packets will be dropped, making these unavailable for recovery should these be lost on the primary link.

Access Point with Minimal Modification

The key source of inefficiency above is that the secondary access point's buffer is maintained as a tail-drop queue that can grow to a large size. What we need instead is a buffer that holds a *small number* of the *most recent* packets. Accordingly, we introduce two changes in how the access point's buffer is managed: *head-drop* behavior instead of tail-drop, and a *settable maximum queue size*, as illustrated in Figure 7.10. *Note that these changes only apply to the real-time links; the default link, DEF, would remain unaffected.*

The maximum queue size should be set based on the characteristics of the stream. For example, for a VoIP stream, if we have a budget of 100 ms for the Wi-Fi hop (as noted in Section 7.2.2) and the inter-packet spacing is 20 ms, the maximum queue size should be set to 5 packets. In general, the client could signal the desired maximum queue size to the

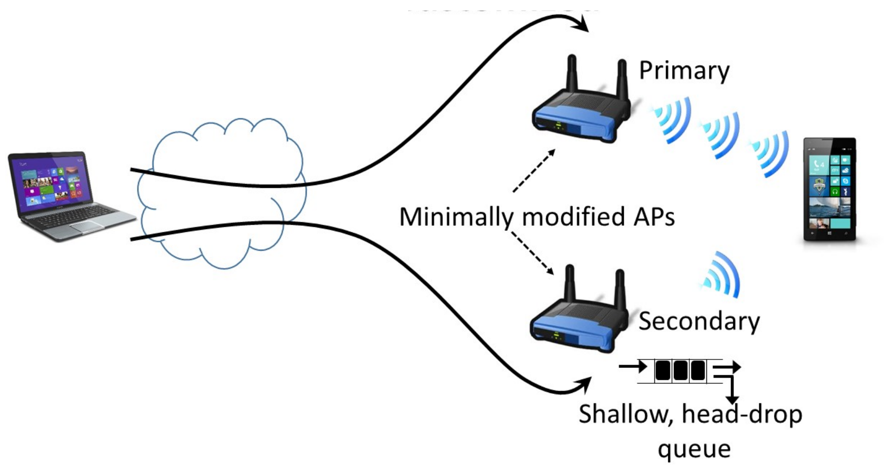


Figure 7.10: "Customized AP" architectural alternative for `DiversiFi`.

access point on a per virtual interface basis, using an unused information element in the 802.11 association request frame.

With this "Customized AP" approach (the client logic for which is shown in Algorithm 7.1), head-drop queuing with a small queue size would ensure that the queue is purged of all but a small number of the most recent packets. Therefore, when the client switches to the secondary link to recover a packet it has missed on the primary link, it would at most receive the small number of packets that are in the secondary's queue. Thus, much of the inefficiency of the "End-to-End" approach is avoided.

As a further optimization, the client could perform implicit packet selection (Section 7.3.2), by switching to the secondary link just a little before the desired (i.e., missing) packet reaches the head of the queue. Then, in principle, the client should be able to receive the desired packet, and immediately switch back to the primary link, thereby avoiding any duplication. However, in practice we find that the access point could also transmit additional queued packets, when all of these are handed down to the hardware queue in one go. We quantify the consequent overhead in our experiments presented in Section 7.4.

Note that although there would be additional delay incurred in retrieving a missing packet over the secondary link, this would not impact the user-perceived quality so long as it is less

Algorithm 7.1: Client logic with “Customized AP”. Upon missing a packet, the client switches to secondary just in time for the missing packet to reach the head of the queue. The client also switches to the secondary periodically, just to keep the association alive.

```
1  Procedure LinkSwitch(IPS, MTD, LSL)
2  // InterPacketSpacing = 20 ms, MaxTolerableDelay = 100 ms,
   LinkSwitchingLatency = 2.8 ms
4      SRT ← 40 ms // SecondaryResidencyTime
6      PLT ← 2 * IPS (= 40 ms) // PacketLossTimeout
8      AKT ← 30 s // AssociationKeepaliveTimeout
10     APQL ← $\frac{MTD}{IPS}$ (= 100/5 = 5) // APQueueLen
12     ETTRH ← IPS * APQL - LSL // ExpectedTimeToReachHead
13     while True that do
15         ReceivePackets
16         if PacketNotReceived then
18             SetPacketAsLostOnPrimary
20             ScheduleSwitchToSecondaryAfter (ETTRH)
22             ScheduleSwitchBackToPrimaryAfter (ETTRH+PLT)
23         end
24         if LostPacketReceivedOnSecondary then
26             SwitchBackToPrimaryImmediately
27         end
28         if SecondaryInactiveFor(AKT) then
30             SwitchToSecondary
32             ScheduleSwitchBackToPrimaryAfter(SRT)
33         end
34     end
```

than the playout delay. The *MaxTolerableDelay* parameter in Algorithm 7.1 is set accordingly, and any packet not retrieved within this period is deemed as lost. Hence, in our evaluation we focus on the residual packet loss even after the secondary link has been tapped rather than on the additional delay.

It is important to note that the above queue modifications we do apply to only the per-device

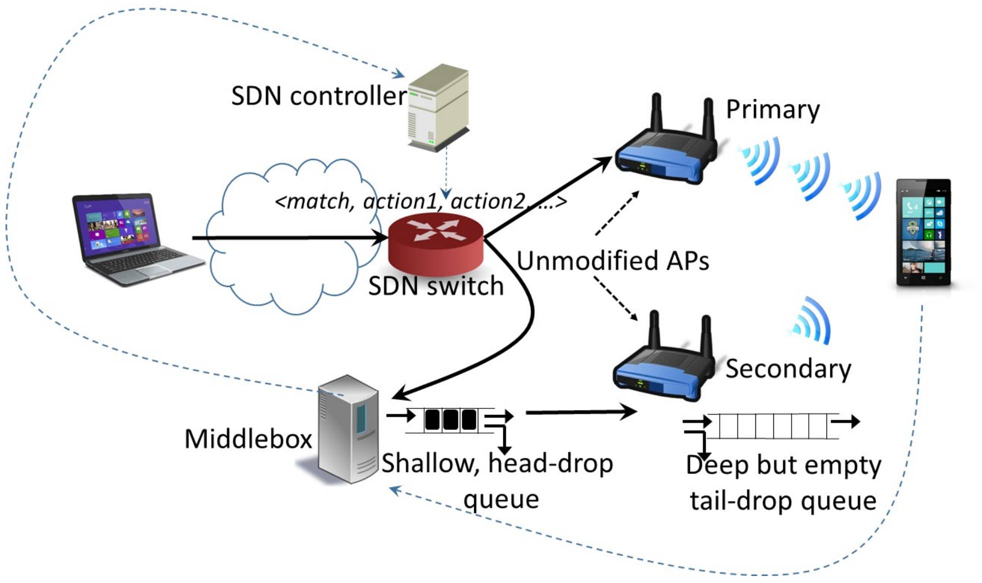


Figure 7.11: "Middlebox" architectural alternative for `DiversiFi`. Solid/dashed lines show the data/control flow.

queues that the access point maintains for clients that have gone into Power Save Mode. Therefore, the extra cross-link packets that the secondary access point has to buffer will not impact other clients that are connected to it.

While the "Customized AP" approach calls for changes to the typical access point behavior (albeit no hardware changes), we note that head-drop queuing is gaining broader support for other reasons (e.g., the CoDel proposal [2] to address the bufferbloat problem) and is already supported in certain access point implementations (e.g., OpenWRT). Ubuntu, which is used as an embedded OS, supports both head-drop and a settable queue size [15]. Further, we believe it would be easy for access point vendors to implement these.

Unmodified Access Points with Middlebox

An alternative that would leave the access point unmodified is to move the functionality of network-side buffering into a separate middlebox. An SDN-capable switch would replicate the stream, sending one copy directly to the client via its primary link and the second copy to the middlebox (which would otherwise not be on the data path). The middlebox would perform network-side buffering in lieu of buffering at the secondary access point. Figure 7.11

illustrates this architecture.

When the client misses a packet on the primary link and wishes to retrieve it over the secondary link (to benefit from link diversity, per Section 7.2), the client switches to the secondary link and sends a request to the middlebox for the specific packet(s) that it desires. After receiving the packets, it switches back to the primary link. Depending on the packet deadlines, this process can be deferred a little to allow the client to retrieve more than one missing packet in one go. In any case, the secondary access point merely acts as a conduit for the packets forwarded to it by the middlebox and does not itself perform any buffer management.

Since the middlebox is under our control and the client performs explicit packet selection, this approach could, in principle, avoid duplicating any packets. However, our current implementation is based on a simple start-stop protocol, wherein the middlebox starts delivering packets upon receipt of start message and stops upon receiving a stop message. So there could be some duplication of packets.

Further, replication at an SDN-capable switch on the local network would mean that the remote peer remains uninvolved. Such a deployment, with a middlebox and an SDN-capable switch that the client has the credentials to install rules on, would likely be feasible in settings such as enterprises, campuses, etc., where the network is under the control of a single entity. We evaluate middlebox performance in Section 7.4.4.

7.3.4 Selecting a Design Alternative

The three designs we proposed above have their own pros and cons. While the End-to-End approach is the easiest in terms of realization, it has the cons of 100% packet duplication in the WAN path and sub-optimal packet buffering at the secondary access point. The Customized AP approach optimizes packet buffering at the secondary access point, but still requires packets to be replicated in the WAN path. The Middlebox approach removes the need for packet duplication in the WAN path and optimizes packet buffering, but requires additional infrastructure to be installed. Therefore, selecting the right alternative depends on where `DiversiFi` is being used. For example, organizations might find it advantageous to deploy the Middlebox to improve VoIP performance, whereas home users can use either the End-to-End or Customized AP approaches.

7.3.5 Client-side Implementation

We have implemented the client-side piece of `DiversiFi` as an user-level library that delivers real-time streams to applications. Multi-link association is implemented using the stock Ath9k driver on the Linux 2.6.33 kernel, which already supports multiple instances of virtual station mode needed to maintain multiple associations. The stock implementation had minor bugs, which we fixed. For instance, one corner case related to the the power-save message not being successfully received by the access point. To address this, we added 5 driver-level retry attempts to increase reliability, and also ensured that the channel switch operation is only invoked after the power-save message has been successfully delivered.

With regard to deployability on other platforms, both Windows and Android Wi-Fi drivers also include support for multiple virtual NICs, for supporting Wi-Fi-Direct mode in conjunction with station mode. We believe that multi-station mode support can be enabled through software-only changes, in the Wi-Fi driver, similar to the existing implementation in the Ath9k Linux driver.

7.4 Evaluation

The goal of our evaluation is to see if our implementation of `DiversiFi` over a single NIC and carefully managed network-side buffering, yields a diversity gain along the lines of what the two NIC experiments showed. We are also interested in characterizing the overhead and coexistence issues, if any.

7.4.1 Experimental Setup

Our experiment setup consisted of a Linux laptop equipped with a single NIC — an Ath9k based DLink 802.11bgn PCMCIA card — which serves as the client. We implemented

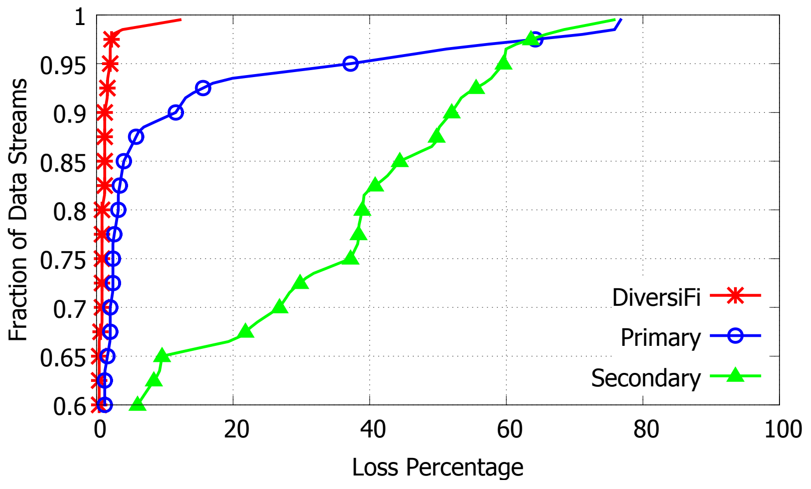


Figure 7.12: CDF of loss rates over the worst 5-second periods comparing `DiversiFi` with primary and secondary links. `DiversiFi` outperforms the single-link case by providing large gains in the form of reduced packet losses.

minimal access point customization (head-drop queues with a maximum size of 5 packets) from Section 7.3.3 on two Netgear WNDR3800 802.11n access points with 2x2 MIMO running OpenWrt. We setup the access points on two different 2.4 GHz channels (1 and 11) at the diagonal ends of a rectangular office space (30 m X 15 m area) with cubicles and walls. At most locations within the office, the client could maintain concurrent associations with both access points. The stronger of the two links is set as the Primary, and the other one as the Secondary.

At each location, we simulated VoIP calls from a wired client on the LAN to the Wi-Fi-client. The Wi-Fi-client then requests the sender-side `DiversiFi` user library to replicate the stream to the IP address of the secondary link in addition to the primary.

Separately, we also ran experiments with a middlebox, the details of which appear in Section 7.4.4 below.

7.4.2 Loss Recovery

We conducted 61 sets of runs that interleaved single link experiments and `DiversiFi` at different locations in the office building. We calculated the packet loss rate in the worst

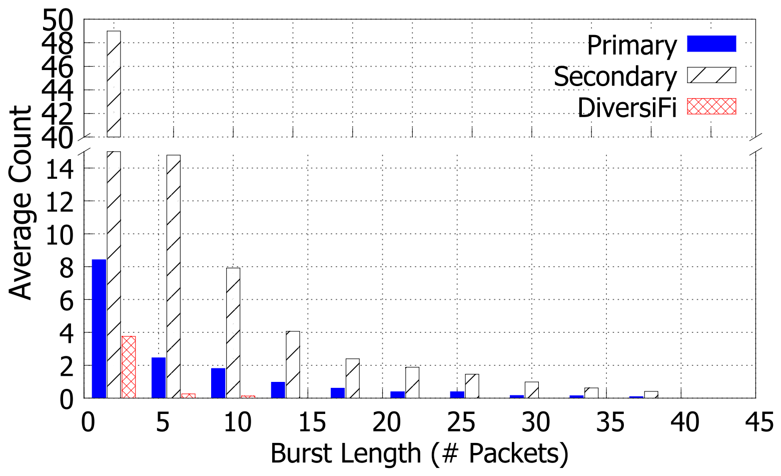


Figure 7.13: Distribution of packet loss burst lengths. `DiversiFi` helps cut down burst losses significantly, when compared to the single-link case.

5-second window during each call. Figure 7.12 plots the CDF of these loss rates, both for the single link case (where the client connects over the stronger, primary link), and for `DiversiFi` (where the client connects over both primary and secondary links). We find that `DiversiFi` significantly outperforms the single-link case. For instance, while the 90^{th}%tile loss rate (computed by considering the worst 5-second period for each of the 61 calls) for the primary link was 11.6% and for the secondary link was 52%, that for `DiversiFi` was only 1.2%. The large gain with `DiversiFi` arises because it is quite unlikely that the worst 5-second period experienced on the secondary would coincide with that on the primary.

`DiversiFi` also helps cut down the incidence of burst losses, as shown in Figure 7.13. When just the primary link is used, each 2-minute call suffers a loss of 44.3 packets on average, 35.9 of those in bursts of two or more packets. In comparison, with `DiversiFi` only 0.9 out of the 2.7 packets lost on average per call was in bursts (which implies that only a fraction of the calls suffered any burst losses at all).

This performance improvement also translates into an improvement in the poor call rate (PCR), which drops from 4.9% and 26.2%, respectively, when the primary link or the secondary link alone is used, to 0% with `DiversiFi`.

7.4.3 Duplication Costs, Overheads and Fairness

While DiversiFi yields significant gains, the question is how much overhead is imposed by replication and what impact it has on fairness to other, non-real-time traffic. We focus here on the duplication overhead over Wi-Fi; the overhead over the WAN would either be 100% if the replication is done at the source or 0% if it is done at a local middlebox.

For the set of 61 calls, the average packet loss rate on the primary link alone is 1.97%. (This is computed over the entire length of the calls, not just the worst 5-second period considered above.) With DiversiFi, the average packet loss rate drops to 0.05% because an overwhelming majority of the primary link losses are recovered on the secondary link. In addition to these *useful* transmissions over the secondary link, 0.62% of the packets were transmitted *unnecessarily* over the secondary link.

This (wasteful) duplication overhead of about 1 packet (0.62%) for every 3 lost on the primary (1.97%), arises because of two reasons: first, the access point inserts multiple packets into its hardware queue and transmits all of them even when the packet that the client is looking for is at the head of the queue, and second, the client has to receive packets from the secondary access point at some minimal periodicity, irrespective of whether there is packet loss, just to keep its (secondary) association alive.

To evaluate the impact of such wasteful duplication on other traffic, we performed iperf-based TCP measurements concurrently with a VoIP flow, with DiversiFi turned on or turned off alternately. Note that even when DiversiFi is turned on, the TCP traffic is confined to the DEF link (Section 7.3.2). However, when DiversiFi is turned on, the NIC would be switched between channels, so we need to examine the possibility of a performance impact on traffic flowing on DEF.

The results from 26 sets of runs is shown in Figure 7.14, which plots the CDF of the difference in TCP throughput between staying continuously on the primary link (DiversiFi is turned off) and switching back-and-forth between the primary and secondary links (DiversiFi is turned on). As these runs were interleaved in time *i.e.,* a primary-only run followed by a DiversiFi run, in some cases the primary-only run had a lower throughput than the

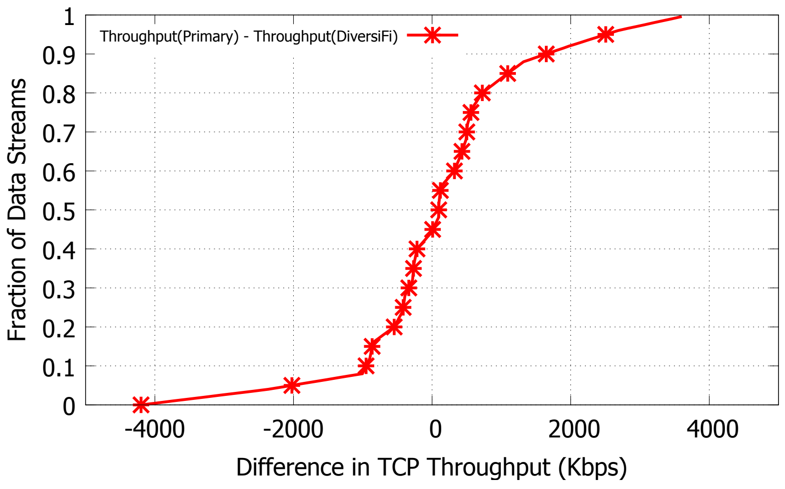


Figure 7.14: Difference in throughput of a TCP flow, with and without `DiversiFi`. `DiversiFi` has minimal impact on non-real-time streams.

`DiversiFi` run because of varying channel conditions. We find the points are distributed almost evenly about the zero difference line, indicating that `DiversiFi` has little impact on the throughput of the TCP flow. Indeed, over the 26 runs, the average TCP throughput with `DiversiFi` turned on is 3.9 Mbps compared to 4.0 with it turned off, a difference of only 2.5%.

Another point of concern is whether `DiversiFi` can potentially harm a flow when the primary link is very good because of the periodic switches to the secondary link to keep the secondary association alive. While it is true that these switches will not provide any benefits, the incurred overhead is minimal. The switching latency to switch to the secondary link and back to the primary link is 4.6 ms (2.3 ms $\times$ 2). This is non-negligible when compared to typical VoIP packet spacings, which are usually in 10s of ms.

Thus, we conclude that `DiversiFi` *provides significant gains for real-time VoIP flows while imposing little overhead in terms of channel switching or unnecessary duplication, thereby ensuring coexistence with competing, non-real-time traffic.*

Table 7.1: Delay in milliseconds to collect a buffered packet on the Secondary link for two schemes. The additional delay introduced by the middlebox is acceptable for real-time streaming applications.

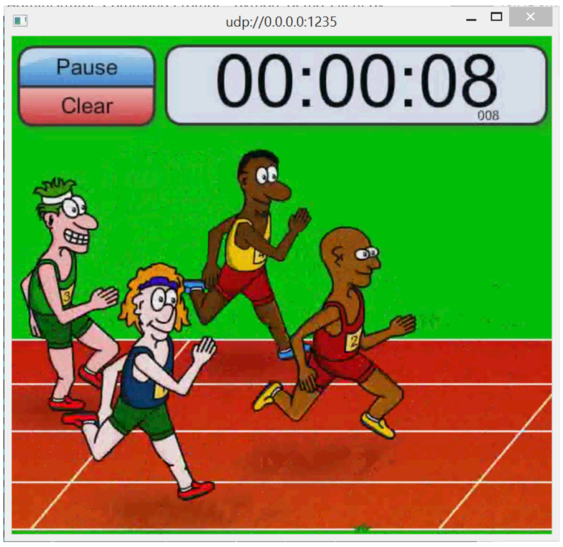
	Total	Switching	Network	Queuing
Middlebox	5.2	2.3	2	0.9
Access Point	2.8	2.3	0.5	-

7.4.4 Middlebox Performance

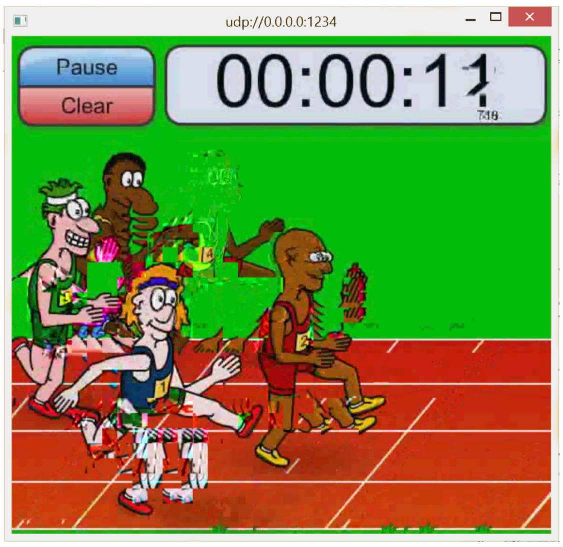
We also evaluate the performance of network-side buffering at a middlebox compared to buffering at the access point. We implemented the middlebox using MIT Click router software V2.1 running on a quad-core i7 16GB RAM machine, and also setup an SDN switch using Open vSwitch V2.0.2, and Floodlight controller V1.0 software. The `DiversiFi` client library requests the middlebox to start replicating a specific real-time flow destined to the same client, which triggers the middlebox to have a match-action rule for replication installed in the SDN switch. The replicated packets are buffered in a shallow head-drop queue and retrieved by the client whenever packet loss is detected.

One question is how much additional latency is entailed in the client contacting the middlebox for missing packets as compared to retrieving these from the secondary access point. We measured the delay between the reception of last packet on the primary link and the first packet received on the secondary link across 100 runs of primary-to-secondary switches Table 7.1 reports the total delay as well as the break up. Channel switching operations, which includes sending power-save and wakeup messages, takes up an average latency of 2.3 plus 0.5 = 2.8 ms. An additional 5.2 minus 2.8 = 2.4 ms delay is introduced by the middlebox, which is likely to be acceptable for real-time streaming applications.

To evaluate the scalability of the middlebox, we initiated concurrent real-time streams numbering between 0 to 1000, where each stream gets replicated and sent to the middlebox. We then measured the delay experienced by a Wi-Fi-client to retrieve a lost packet from the middlebox. We find that the delay increases very gradually with an increasing load of flows. At 1000 streams, the total delay, including switching overhead, increased by only 1.1 ms



(a)



(b)

Figure 7.15: Screenshots of the ffplay instances that uses (a) `DiversiFi`, and (b) only the Primary connection. `DiversiFi` significantly improves the user-perceived quality of the video.

compared to zero load, suggesting that a single middlebox can easily serve a large Wi-Fi deployment.

7.4.5 Video Streaming Performance

To study how `DiversiFi` helps improve real-time video streaming, we implemented a video streaming server using FFmpeg [3]. The server reads a video file stored on the server, encodes it using the H.264 codec, and then transmits it as a UDP packet stream. We used a Windows Surface Pro 2 laptop having an inbuilt 802.11abgn Wi-Fi NIC and an external TP-Link TL-WDN3200 dual-band ABGN USB adapter. It used ffplay [4] to play the video streams sent by the server.

The server duplicates each packet and sends one copy of each packet to the two Wi-Fi connections of the client. The client runs two ffplay instances. One of them uses the default setup where it plays the stream using the packets received only on the primary interface. The other uses the `DiversiFi` setup and plays packets received on either of the Wi-Fi interfaces and discards any packets that have already been played.

While the client was playing the video stream, we started walking around the office floor and Figure 7.15(a) shows a screenshot of the primary-only player at a point in time when the primary interface was getting a weak signal from the access point it was connected to. Figure 7.15(b) shows a screenshot of the `DiversiFi` player at the same time instant. Comparing the two, it is evident that `DiversiFi` allows users enjoy a significantly better streaming experience as it helps the client "repair" the stream by using the packets received on the secondary interface.

7.5 Chapter Summary

Wi-Fi links are often a significant cause of poor performance for real-time interactive streaming such as VoIP. `DiversiFi` helps improve performance for such streaming significantly using cross-link replication of the stream over the multiple Wi-Fi links that are often available to a client. This helps cut down the poor call rate for VoIP by 2.24×. We show how network-side buffering allows `DiversiFi` to provide the benefits of replication to real-time flows, with little duplication overhead and while ensuring coexistence with non-real-time applications.

Part IV

Conclusion and Future Work

Chapter 8

Conclusion

8.1 Summary and Contributions

Wireless networks are increasingly becoming the norm of the mode of Internet access of the majority of users thanks to the exponential proliferation of hand-held devices such as smartphones and tablet PCs. Therefore, network service providers have to improve the quality of the last-mile link so that users enjoy greater performance in their applications.

When users experience performance issues, they typically think their wireless network, either Wi-Fi or cellular, is the root cause. This belief is anecdotal, and to verify this we conducted wide-scale measurement studies to analyze to what extent the last-mile wireless link becomes the bottleneck in real-world networks. As these studies highlighted, though wireless networks do impose quality problems on applications, they are not the culprits all the time. Therefore, we designed, implemented, and evaluated novel tools that can detect bottlenecks and pathological conditions in both Wi-Fi and cellular networks.

A challenging class of applications that places stringent demands on the network is real-time streaming applications. Our measurement studies showed that they exhibit poor performance more often on wireless networks than on wired networks. To this end, we presented two systems to alleviate issues imposed on real-time streams by Wi-Fi networks. First, we introduced a novel bandwidth adaptation mechanism that improves the quality of streams by adapting with accordance to the underlying Wi-Fi link conditions. Next, we presented a novel system that increases the reliability of streams by leveraging cross-link packet replication across multiple Wi-Fi connections.

We summarize the contributions made in this dissertation as follows.

Studying the Performance of Wireless Networks in the Wild

We first presented empirical data to show to what extent the last-mile wireless links can affect real-time data streams, such as VoIP calls in the wild. The global measurement studies we conducted showed that both cellular and Wi-Fi links can be the bottleneck source and degrade the quality of data streams significantly. We presented an analysis of a large dataset we obtained from one of the world's largest VoIP service providers to reinforce this finding further. For example, it showed the relative difference of poor quality calls placed on Wi-Fi networks and LAN networks is about 46%.

However, the studies we conducted also showed that the last-mile wireless link is not the source of quality degradations all the time. In fact, in our cellular measurement study, we found that in all the bottlenecked network paths we studied, only 68.9% and 25.7% were bottlenecked by 3G and LTE networks, respectively, thus highlighting the need for a nuanced approach in determining the location of bottlenecks in networks.

Detecting Bottlenecks in Cellular Networks

As the last-mile link cannot always be blamed for quality degradations, we presented novel tools that are capable of detecting bottlenecks and pathological conditions in wireless networks. First, we designed, implemented and evaluated a technique named `QProbe` that can detect bottlenecks in cellular networks. `QProbe` exploits the unique characteristics of PF schedulers to perform its detection. Through simulations, controlled experiments and a large measurement study comprising over 600 users across 33 countries, we showed that `QProbe` can locate bottlenecks with $> 85\%$ accuracy.

Detecting Bottlenecks in Wi-Fi Networks

We designed, implemented and evaluated a suite of detectors named `Kwikr` that are can detect various pathological conditions in Wi-Fi networks such as congestion, mobility, handoffs, and weak-links, which cause the Wi-Fi link to be bottlenecked. We introduced the novel Ping-Pair technique that detects the onset and the conclusion of congestion periods in the Wi-Fi network by pinging the access point using carefully crafted ICMP ping packets. We used passively observable and simple to obtain signals like RSSI, BSSID to detect the other pathological conditions. Our detectors can be deployed on off-the-shelf devices and requires no modifications in either the device or the access point, and hence boasts practicality that is non-existent in the state-of-the-art. Through experimental studies, we showed that `Kwikr` can detect the aforementioned bottleneck conditions with $> 90\%$ accuracy.

Bandwidth Adaptation Using Wi-Fi Hints

Bandwidth adaptation is a critical component in real-time streaming applications that help them deliver streams at the highest quality that the network can support. However, as these techniques operate end-to-end, they are oblivious to bottleneck conditions that might be present in Wi-Fi networks. This causes their adaptation decisions to be suboptimal.

We presented a novel user-level module named `KwikrAdapt` that continuously observes the ongoing situation in a Wi-Fi link via the Wi-Fi hints provided by `Kwikr` and uses them to do better bandwidth adaptation of streams. Through empirical studies conducted in different Wi-Fi networks and settings, we showed that by using `KwikrAdapt`, real-time streaming applications can operate at relatively higher data rates when the network is congested, reduce the worst packet loss percentage by $\sim 70\%$ during mobile periods, and reduce the time required to ramp up after a handoff by $3.1\times$.

Robust Real-Time Streaming via Cross-Link Packet Replication

Finally, we presented a system named `DiversiFi` that provides a significant quality improvement of real-time data streams by replicating packets across multiple Wi-Fi connections. We showed that this approach provides better performance in real-time streams rather than selecting the best link to connect to, which is the default process of establishing a Wi-Fi connection. We presented several architectural alternatives that can reap the benefits of cross-link packet replication across two simultaneous Wi-Fi connections with less overhead.

Through experiments conducted on an end-to-end system implementation, we demonstrated that `DiversiFi` is capable of reducing poor quality data streams by 2.24×, while limiting the wasteful packet duplication to just 0.62%. We also showed that `DiversiFi` coexists well with non-real-time data streams, such as TCP flows.

8.2 Future Work

The following are some of the possible extensions and future work of the work presented in this dissertation.

On-Demand Duplication in DiversiFi. The DiversiFi implementation we presented always duplicates packets in the WAN path. It switches to the secondary Wi-Fi link only if a packet does not arrive on the primary Wi-Fi link (and to keep the association alive), which helps cut down the over-the-air duplicate transmissions significantly. It would be beneficial if the over-the-WAN duplication can be reduced by implementing a switching mechanism through which the sender can either switch on or off duplication on-demand. It would be challenging to design such switching policies because the receiver has to determine when it needs duplicates and when it doesn't. For example, if the process is reactive, the receiver will not be able to recover any lost packets using the secondary Wi-Fi connection. Therefore, the receiver should predict the occurrence of a loss event, and request duplicates in advance. Developing such techniques will be an interesting research problem.

DiversiFi with Cellular Connections. Prior work has looked at using TCP multi-path solutions over simultaneous cellular and Wi-Fi connections to increase the throughput of a flow. It is interesting to study whether we can follow a similar technique to increase the robustness of real-time streams. In `DiversiFi`, we used two Wi-Fi connections in parallel, which imposes deployability constraints as such a scheme is hard to realize without the modification of devices or networks. However, using a mix of cellular and Wi-Fi connections is already prevalent in modern devices, and hence targeting such a scheme would be beneficial in terms of practicality. It is interesting to study whether we can achieve the same robustness of streams by using cross-link packet replication across Wi-Fi and cellular connections and what the overhead will be in doing so.

Kwikr Enhancements. The `Kwikr` detector suite we presented in this dissertation is currently capable of detecting four pathological conditions: congestion, mobility, handoffs, and weak-links. It can be enhanced by adding more detectors to detect other conditions like hidden terminals, non-Wi-Fi interference (*e.g.,* caused by microwave ovens, cordless phones, *etc.*) and co-channel interference. This is not as straightforward as one might think if the practicality of the proposed solutions are important. State-of-the-art detectors in the literature either use special hardware or run the Wi-Fi interface in monitor mode to detect these pathologies.

Skype Integrations. We are collaborating with the Microsoft Skype team to integrate some of our work in the Skype application. In particular, we are interested in studying the benefits of integrating `KwikrAdapt` in Skype's bandwidth adaptation algorithm. We are also conducting a trial of `DiversiFi` in the Microsoft Redmond and Bangalore campuses to study its benefits, with the end-goal of integrating it in Skype such that it would come into effect on devices that meet `DiversiFi`'s hardware requirements.

Combining KwikrAdapt and DiversiFi. As `Kwikr` and `DiversiFi` are two orthogonal approaches that were designed to alleviate wireless bottlenecks to improve the reliability and robustness of real-time streams, it is interesting to study the benefits that can be yielded by combing the two systems. There are multiple advantages in doing so. For example, in the presence of congestion, real-time streaming applications can use `KwikrAdapt` to achieve a higher data rate on the primary Wi-Fi link while retrieving lost packets through the secondary Wi-Fi link via `DiversiFi`. Similarly, `DiversiFi` can be used to obtain packets that are lost during a handoff on the primary Wi-Fi link, while using `KwikrAdapt` to ramp up the data rate faster on the primary when the handoff completes.

References

[1] Buffer Size of AP for Clients in Power Save Mode. http://community.arubanetworks.com/t5/Controller-Based-WLANs/Buffer-size-of-AP-for-clients-in-power-save-mode/ta-p/176888.

[2] Controlled Delay (CoDel). http://www.bufferbloat.net/projects/codel.

[3] FFmpeg. https://ffmpeg.org.

[4] ffolay. https://ffmpeg.org/ffplay.html.

[5] Google Hangouts. http://www.google.com/+/learnmore/hangouts/.

[6] IEEE 802.11e Standard, 2005. http://standards.ieee.org/findstds/standard/802.11e-2005.html.

[7] Industry First: Smartphones Pass PCs in Sales. http://fortune.com/2011/02/07/industry-first-smartphones-pass-pcs-in-sales/.

[8] Microsoft Azure RemoteApp. https://www.remoteapp.windowsazure.com/.

[9] Number of Smartphones Sold to End Users Worldwide from 2007 to 2014. http://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/.

[10] OnLive. https://www.onlive.com/.

[11] Qualcomm QxDM Professional. https://www.qualcomm.com/documents/qxdm-professional-qualcomm-extensible-diagnostic-monitor.

[12] Quality of Service for Voice Over IP. http://www.cisco.com/c/en/us/td/docs/ios/solutions_docs/qos_solutions/QoSVoIP/QoSVoIP.pdf.

[13] Skype. http://www.skype.com/.

[14] Sony PlayStation Now. `http://www.playstation.com/en-us/explore/psnow/`.

[15] Ubuntu Stochastic Fairness Queueing. `http://manpages.ubuntu.com/manpages/trusty/man8/tc-sfq.8.html`.

[16] UDP Packet Replication Using Open vSwitch. `http://blog.sflow.com/2013/11/udp-packet-replication-using-open.html`.

[17] P.862 : Perceptual Evaluation of Speech Quality (PESQ): An Objective Method for End-To-End Speech Quality Assessment of Narrow-Band Telephone Networks and Speech Codecs, February 2001. ITU-T Recommendation.

[18] P.862.1 : Mapping Function for Transforming P.862 Raw Result Scores to MOS-LQO, November 2003. ITU-T Recommendation.

[19] A. Adya, P. Bahl, R. Chandra, and L. Qiu. Architecture and Techniques for Diagnosing Faults in IEEE 802.11 Infrastructure Networks. In *MobiCom*, 2004.

[20] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. Padmanabhan, and G. Voelker. NetPrints: Diagnosing Home Network Misconfigurations using Shared Knowledge. In *NSDI*, 2009.

[21] G. Ananthanarayanan, V. N. Padmanabhan, L. Ravindranath, and C. A. Thekkath. COMBINE: Leveraging the Power of Wireless Peers through Collaborative Downloading. In *MobiSys*, 2008.

[22] Android WifiInfo API. `http://developer.android.com/reference/android/net/wifi/WifiInfo.html`.

[23] J. Apostolopoulos and M. Trott. Path Diversity for Enhanced Media Streaming. *IEEE Communications Magazine*, 2004.

[24] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. In *ICDCS*, 1995.

[25] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *SIGCOMM*, 1996.

[26] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP Performance over Wireless Networks. In *MobiCom*, 1995.

[27] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting Mobile 3G Using WiFi. In *MobiSys*, 2010.

[28] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services, 1998. RFC 2475.

[29] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, 1994.

[30] R. L. Carter and M. E. Crovella. Measuring Bottleneck Link Speed in Packet-switched Networks. *Performance Evaluation*, 1996.

[31] A. Chakraborty, V. Navda, V. N. Padmanabhan, and R. Ramjee. Coordinating Cellular Background Transfers Using Loadsense. In *MobiCom*, 2013.

[32] R. Chandra, V. Bahl, and P. Bahl. MultiNet: Connecting to Multiple IEEE 802.11 Networks Using a Single Wireless Card. In *INFOCOMM*, 2004.

[33] A. Downey. Clink: a Tool for Estimating Internet Link Characteristics, 1999. `http://allendowney.com/research/clink/`.

[34] A. Croitoru, D. Niculescu, and C. Raiciu. Towards Wifi Mobility Without Fast Handover. In *NSDI*, 2015.

[35] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. WiFi, LTE, or Both? Measuring Multi-Homed Wireless Internet Performance. In *IMC*, 2014.

[36] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI*, 2015.

[37] C. Dovrolis, P. Ramanathan, and D. Moore. What Do Packet Dispersion Techniques Measure? In *INFOCOM*, 2001.

[38] C. Dovrolis, P. Ramanathan, and D. Moore. Packet-Dispersion Techniques and a Capacity-Estimation Methodology. *IEEE/ACM Transactions in Networking*, 2004.

[39] A. B. Downey. Using Pathchar to Estimate Internet Link Characteristics. In *SIGMETRICS*, 1999.

[40] A. B. Downey. Using pathchar to Estimate Link Characteristics. In *SIGCOMM*, 1999.

[41] DSCP - RFC 2474. `https://tools.ietf.org/html/rfc2474`.

[42] DSCP Common Values - RFC 2475. `https://tools.ietf.org/html/rfc2475`.

[43] C. P. Fu and S. C. Liew. TCP Veno: TCP Enhancement for Transmission Over Wireless Access Networks. *IEEE Journal on Selected Areas in Communications*, 2003.

[44] L. Golubchik, J. Lui, T. Tung, A. Chow, A. Lee, G. Franceschinis, and C. Anglano. Multi-path Continuous Media Streaming: What are the Benefits? *Performance Evaluation*, 2002.

[45] R. Gummadi, D. Wetherall, B. Greenstein, and S. Seshan. Understanding and Mitigating the Impact of RF Interference on 802.11 Networks. In *SIGCOMM*, 2007.

[46] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Operating Systems Review*, 2008.

[47] K. Harfoush, A. Bestavros, and J. Byers. Measuring Bottleneck Bandwidth of Targeted Path Segments. In *INFOCOM*, 2003.

[48] M. Heusse, F. Rousseau, G. Berger-Sabbatel, and A. Duda. Performance Anomaly of 802.11b. In *INFOCOM*, 2003.

[49] M. Heusse, F. Rousseau, R. Guillier, and A. Duda. Idle Sense: An Optimal Access Method for High Throughput and Fairness in Rate Diverse Wireless LANs. In *SIGCOMM*, 2005.

[50] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang. Locating Internet Bottlenecks: Algorithms, Measurements, and Implications. In *SIGCOMM*, 2004.

[51] N. Hu and P. Steenkiste. Evaluation and Characterization of Available Bandwidth Probing Techniques. *IEEE Journal on Selected Areas in Communications*, 2003.

[52] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[53] M. Jain and C. Dovrolis. End-To-End Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput. In *SIGCOMM*, 2002.

[54] Jamie Stark. Lync and Software-Defined Networking, December 2013. `http://blogs.technet.com/b/lync/archive/2013/12/17/lync-and-software-defined-networking.aspx`.

[55] S. J. Julier and J. K. Uhlmann. A New Extension of the Kalman Filter to Nonlinear Systems. In *AeroSense*, 1997.

[56] S. Kandula, K. C.-J. Lin, T. Badirkhanli, and D. Katabi. FatVAP: Aggregating AP Backhaul Capacity to Maximize Throughput. In *NSDI*, 2008.

[57] P. Kanuparthy, C. Dovrolis, K. Papagiannaki, S. Seshan, and P. Steenkiste. Can User-Level probing Detect and Diagnose Common Home-WLAN Pathologies. *ACM SIGCOMM Computer Communication Review*, 2012.

[58] K. Kashibuchi, Y. Nemoto, and N. Kato. Mitigating Performance Anomaly of TFRC in Multi-Rate IEEE 802.11 Wireless LANs. In *GLOBECOM*, 2009.

[59] L. W. Kay. A Rate-based TCP Congestion Control Framework for Cellular Data Networks, 2005. PhD thesis, National University of Singapore.

[60] S. Keshav. A Control-theoretic Approach to Flow Control. In *SIGCOMM*, 1991.

[61] H. Kushner and P. Whiting. Convergence of Proportional-Fair Sharing Algorithms Under General Conditions. *IEEE Transactions on Wireless Communications*, 2004.

[62] H. N. Kyung-Hwa Kim and H. Schulzrinne. WiSlow: A Wi-Fi Network Performance Troubleshooting Tool for End Users. In *INFOCOM*, 2014.

[63] K. Lai and M. Baker. Measuring Link Bandwidths Using a Deterministic Model of Packet Delay. In *SIGCOMM*, 2000.

[64] K. Lakshminarayanan, V. N. Padmanabhan, and J. Padhye. Bandwidth Estimation in Broadband Access Networks. In *IMC*, 2004.

[65] K. Lee, D. Chu, E. Cuervo, Y. Degtyarev, S. Grizan, J. Kopf, A. Wolman, and J. Flinn. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Cloud Gaming. In *MobiSys*, 2015.

[66] A. Lozano and N. Jindal. Transmit Diversity vs. Spatial Multiplexing in Modern MIMO Systems. *IEEE Transactions on Wireless Communications*, 2010.

[67] B. Melander, M. Bjorkman, and P. Gunningberg. A New End-To-End Probing and Analysis Method for Estimating Bandwidth Bottlenecks. In *GLOBECOM*, 2000.

[68] A. K. Miu, H. Balakrishnan, and C. E. Koksal. Improving Loss Resilience with Multi-Radio Diversity in Wireless Networks. In *MobiCom*, 2005.

[69] A. K. Miu, G. Tan, H. Balakrishnan, and J. Apostolopoulos. Divert: Fine-grained Path Selection for Wireless LANs. In *MobiSys*, 2004.

[70] Native Wi-Fi API. `https://msdn.microsoft.com/en-us/library/ms705969.aspx`.

[71] B. Mah. Pchar: A Tool for Measuring Internet Path Charateristics. `http://www.kitchenlab.org/www/bmah/Software/pchar/`.

[72] QProbe Project Page. `http://www.comp.nus.edu.sg/~nimantha/qprobe.html`.

[73] QProbe-Cellular Link Diagnostic Tool. `https://itunes.apple.com/sg/app/qprobe-cellular-link-diagnostic/id988472779`.

[74] H. Rahul, H. Hassanieh, and D. Katabi. SourceSync: A Distributed Wireless Architecture for Exploiting Sender Diversity. In *SIGCOMM*, 2010.

[75] 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*, 2012.

[76] L. Ravindranath, C. Newport, H. Balakrishnan, and S. Madden. Improving Wireless Network Performance Using Sensor Hints. In *NSDI*, 2011.

[77] S. Rayanchu, A. Mishra, D. Agrawal, S. Saha, and S. Banerjee. Diagnosing Wireless Packet Losses in 802.11: Separating Collision from Weak Signal. In *INFOCOM*, 2008.

[78] RFC 6817 - LEDBAT. `https://tools.ietf.org/html/rfc6817`.

[79] V. Ribeiro, M. Coates, R. Riedi, S. Sarvotham, B. Hendricks, and R. Baraniuk. Multifractal Cross-Traffic Estimation. In *ITC*, 2000.

[80] V. Ribeiro, R. H. Riedi, and R. G. Baraniuk. Spatio-Temporal Available Bandwidth Estimation for High-Speed Networks. In *BEst Workshop*, 2003.

[81] C. Rodbro, S. Andersen, and K. Vos. Systems and methods for controlling packet transmission from a transmitter to a receiver via a channel that employs packet queuing when overloaded, September 2012. US Patent Number 8259570.

[82] M. Schiavone, P. Romirer-Maierhofer, F. Ricciato, and A. Baiocchi. Towards Bottleneck Identification in Cellular Networks via Passive TCP Monitoring. In *ADHOC-NOW*, 2014.

[83] H. Schulzrinne and S. Casner. RTP Profile for Audio and Video Conferences with Minimal Control, 2003. RFC 3551.

[84] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, 2003. RFC 3550.

[85] V. Shrivastava, S. Rayanchu, J. Yoon, and S. Banerjee. 802.11n Under the Microscope. In *IMC*, 2008.

[86] Skype Bandwidth Requirements. `https://support.skype.com/en/faq/FA1417/how-much-bandwidth-does-skype-need`.

[87] J. Strauss, D. Katabi, and F. Kaashoek. A Measurement Study of Available Bandwidth Estimation Tools. In *IMC*, 2003.

[88] L. Sun, S. Sen, and D. Koutsonikolas. Bringing Mobility-Awareness to WLANs Using PHY Layer Information. In *CoNEXT*, 2014.

[89] S. Sundaresan, N. Feamster, and R. Teixeira. Measuring the Performance of User Traffic in Home Wireless Networks. In *PAM*, 2015.

[90] S. Sundaresan, Y. Grunenberger, N. Feamster, D. Papagiannaki, D. Levin, and R. Teixeira. WTF? Locating Performance Problems in Home Networks, 2013. Georgia Institute of Technology SCS Technical Report GT-CS-13-03.

[91] T. Szigeti and C. Hattingh. *End-to-End QoS Network Design: Quality of Service in LANs, WANs, and VPNs.* Cisco Press. `http://www.ciscopress.com/store/end-to-end-qos-network-design-quality-of-service-for-9781587143694`.

[92] RFC 5348 - TFRC Protocol. `https://tools.ietf.org/html/rfc5348`.

[93] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *OSDI*, 2002.

[94] E. Vergetis, R. Guerin, and S. Sarkar. Packet-Level Diversity - From Theory to Practice: An 802.11-based Experimental Investigation. In *ITC*, 2005.

[95] E. Vergetis, E. Pierce, M. Blanco, and R. Guerin. Packet-Level Diversity - From Theory to Practice: An 802.11-based Experimental Investigation. In *MobiCom*, 2006.

[96] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low Latency via Redundancy. In *CoNEXT*, 2013.

[97] E. Wan and R. V. D. Merwe. The Unscented Kalman Filter for Nonlinear Estimation. In *AS-SPCC*, 2000.

[98] B. Weiss, S. Moller, B. Belmudez, and B. Lewcio. Analysis of Call-Quality Prediction Performance for Speech-only and Audio-Visual Telephony. In *QoMEX*, 2014.

[99] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, 2013.

[100] Wireshark. `https://www.wireshark.org/`.

[101] D. Wisely. *IP for 4G.* John Wiley & Sons, 2009.

[102] Q. Xu, S. Mehrotra, Z. Mao, and J. Li. PROTEUS: Network Performance Forecast for Real-time, Interactive Mobile Applications. In *MobiSys*, 2013.

[103] Y. Xu, Z. Wang, W. Leong, and B. Leong. An End-to-End Measurement Study of Modern Cellular Data Networks. In *PAM*, 2014.

[104] S.-h. Yoo, J.-H. Choi, J.-H. Hwang, and C. Yoo. Eliminating the Performance Anomaly of 802.11b. In *ICN*, 2005.

Appendices

Appendix A

Finding the Hop Count of a Network Path

The design of the `QProbe` algorithm involves finding the number of hops between the server and the client, so that the TTL of the load packets can be set to expire before they enter the cellular network (Section 4.3). To achieve this, the client cannot simply do a traceroute to the server because of several reasons. First, the client's traceroute will find the number of hops in the uplink direction whereas we want to find the number of hops in the downlink direction (the uplink and downlink paths can be asymmetric). To overcome this, the server can do the traceroute, but as some subnets in the Internet completely block ICMP messages, traceroute might not be usable. Therefore, instead of using traceroute, we design a simple but elegant algorithm that the client can run to find the number of hops in the downlink direction from the server to the client.

Algorithms A.1 and A.2 summarize the code we run at the server and the client, respectively. The client first sends a "start" message to the server. The server then sends packets with TTL values from 1 to 64, in powers of 2. As the IP header is stripped off by the time a packet reaches the application layer, the server sets the TTL of the IP header in plain text in the packet payload. The client then reads through the received packets' TTL values and sends the minimum TTL value it received to the server. The correct number of hops is then between this minimum value, say min_ttl, and $min\_ttl \div 2$. Therefore, the server sends packets having TTL values starting from $min\_ttl \div 2$ to min_ttl in increments of 1 to the client. The client again reads through the received packets and finds the minimum value that it received. This value is the number of hops between the server and the client in the downlink direction, provided that packet losses did not occur.

Algorithm A.1: Find_Hops: The Server-Side Algorithm.

```
Open UDP socket $s$ on port $p$
while $true$ do
    $message, client\_addr \leftarrow s$.receive()
    if $message =$ “$start$” then
        $ttl \leftarrow 1$
        while $ttl < 65$ do
            $s$.setTtl($ttl$)
            $payload \leftarrow ttl$.toString()
            $s$.sendTo($client\_addr$, $payload$)
            $ttl \leftarrow ttl \times 2$
        end
    end
    else
        $previous\_ttl \leftarrow message$.toInt()
        $ttl \leftarrow previous\_ttl \div 2$
        while $ttl < previous\_ttl$ do
            $s$.setTtl($ttl$)
            $payload \leftarrow ttl$.toString()
            $s$.sendTo($client\_addr$, $payload$)
            $ttl \leftarrow ttl + 1$
        end
    end
end
```

Algorithm A.2: Find_Hops: Client-Side Algorithm.

```
Output: Number of hops between the server and the client
Open UDP socket $s$
$s$.setTimeout(5)                                    // Times out after 5 seconds
$server\_addr \leftarrow (server\_ip,\ p)$
$payload \leftarrow$ "$start$"
$s$.sendTo($server\_addr$, $payload$)
$minimum\_ttl \leftarrow 64$
while $s$ does not timeout do
    $message, addr \leftarrow s$.receive()
    $ttl \leftarrow message$.toInt()
    if $ttl < minimum\_ttl$ then
        $minimum\_ttl = ttl$
    end
end
$payload \leftarrow minimum\_ttl$.toString()
$s$.sendTo($server\_addr$, $payload$)
while $s$ does not timeout do
    $message, addr \leftarrow s$.receive()
    $ttl \leftarrow message$.toInt()
    if $ttl < minimum\_ttl$ then
        $minimum\_ttl \leftarrow ttl$
    end
end
return $minimum\_ttl$
```