USING ALL NETWORKS AROUND US

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

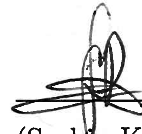
Kok-Kiong Yap
March 2013

© Copyright by Kok-Kiong Yap 2013
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Nick McKeown) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.



(Sachin Katti)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Guru Parulkar)

Approved for the University Committee on Graduate Studies

The very tool of fairness is sufficient for systematic unfairness.

Abstract

Our smartphones are increasingly becoming loaded with more applications and equipped with more network interfaces—3G, 4G, WiFi. The exponential increase in applications for mobile devices has powered unprecedented growth for mobile networks. Mobile operating system and mobile network infrastructure are struggling to cope with this growth.

In this thesis, I advocate that we *use all of the networks around us.* Our smartphones are already equipped with multiple radios that can connect us to multiple networks at the same time. By using those connections, we can open up tremendous capacity and coverage to better serve users, applications, and network operators alike. If done right, this will provide mobile users with seamless connectivity, faster connections, even lower charges and smaller energy footprints.

To do so, we must first overcome several technical challenges that I will be describing in this dissertation, along with their proposed solutions.

1. I design a client network stack that allows the applications to migrate flows from one network to another, aggregates bandwidth across multiple networks to achieve faster connections and provides applications with flexible control over interface choices. I also present a prototype of this client network stack—called Hercules—to demonstrate its viability and usefulness.

2. I develop the theoretical foundation for scheduling packets across multiple networks while taking into account that applications have interface preferences. Using this theoretical foundation, I design a packet scheduling algorithm—multiple interface Deficit Round Robin (miDRR)—that is formally shown to provide weighted max-min fair packet scheduling.

3. Finally, I outline a programmable open network architecture that supports users making use of multiple networks simultaneously. The architecture also enable the operators to continually innovate and provide better services. By deploying and operating this network on the Stanford campus, the design is validated and the architecture's benefits affirmed.

All in all, I show that making use of multiple networks at the same time is both technically possible and beneficial for users, applications, and network operators.

Acknowledgements

Nick McKeown is known to many as a brilliant researcher, a relentless entrepreneur, and a dedicated teacher. Having Nick as my advisor has been nothing short of good fortune. Over the years, Nick has taught me to have the spirit to try, the commitment to do my best, and the courage to confront my shortcomings. These years of invaluable experiences working with Nick have made me a better person not only professionally but also personally. For all of these reasons, I would like to thank Nick for his years of attention and guidance.

I have also greatly benefited from my interactions with Sachin Katti and Guru Parulkar. Sachin always offered a keen listening ear to any ideas, no matter how silly or crazy they were. This was often quickly followed by a sharp analysis of those ideas, which inevitably help shaped many of those found in this dissertation. Guru has always been a caring advisor who looks out for us students and makes sure everyone is taken care of. Guru has never failed to open his heart and provide his honest opinions.

I would also like to thank Mendel Rosenblum, John Ousterhout, and Balaji Prabhakar for serving as my defense committee and providing me with feedback that has undoubtably improve this thesis. Also, my interactions with Yinyu Ye, Monica Lam and Fouad Tobagi have been both inspiring and educational.

The time I spent with the McKeown Group has been fun. During this time, the group helped to develop and shape OpenFlow and software-defined networking. This period was marked by hard work; the brilliance of this group never fails to impress me. It has been an honor to be able to work with so many people who are smarter and more diligent than myself. This exposure and experience is an invaluable lesson. Among the many who deserves my thanks, I would especially like to express my gratitude to Te-Yuan Huang, who is the best collaborator; Masayoshi Kobayashi, who has been a great motivator[1]; Yiannis Yiakoumis, with whom I have argued so much and from whom I have learned just as much;

[1] Nothing is more motivating than to see Masa runs toward work.

Peyman Kazemian, who took the trouble to review and comment on this dissertation; Rob Sherwood, with whom I have discussed many research ideas; Martin Casado, who has always provided great advices and support despite being extremely busy; and Adam Covington, who has been an incredible officemate.

In the course of my work, I have developed and deployed many network services. None of this would have been possible without the supportive network administrators we have at Stanford. For this, I would like to thank Charlie Orgish, Joe Little, Johan van Reijendam, and Miles Davis for letting us muck with the network and bring it down to its knees so many times. I would also like to thanks Clearwire and NEC for providing the spectrum, equipment, and support that were necessary for completing this work.

I also have to thank my parents and sisters for bringing me up, educating me, and giving me the opportunity to pursue this degree so far away from home. I would especially like to dedicate this to my mother, who has always wanted me to become a "doctor."

Last, and definitely not least, I must thank my wife Serene and son Cayden for supporting me and encouraging me every step along the way, despite my working in the delivery room while Serene was giving birth to our son. Their unconditional love and care is what has made it possible for me to accomplish this goal.

Contents

Abstract **vii**

Acknowledgements **ix**

1 Introduction **1**

1.1 Background and Motivation 1

1.2 Problem Statements 3

1.2.1 A Client Cannot Exploit Multiple Networks 3

1.2.2 Policy-based Fair Scheduling onto Multiple Interfaces is Undefined . 4

1.2.3 Network in Support of Clients using Multiple Networks 5

1.3 Outline of Dissertation 5

2 Client Network Stack (Hercules) **7**

2.1 Problem Statement 7

2.2 Related Work 9

2.3 Hercules Client Network Stack Architecture 10

2.3.1 Implementation in Android 11

2.3.2 Evaluation: Overcoming Limitations to Exploit Multiple Networks . 13

2.3.3 Challenges with Current Networks and Devices 20

2.4 Buffer Optimization in the Hercules Network Stack 21

2.4.1 Late-Binding to Reduce Loss During Handover and Unnecessary Delay of Latency Sensitive Traffic 22

2.4.2 Implementation of Late-Binding in Kernel Bridge 23

2.4.3 Evaluation of Late-Binding 25

2.5 Summary 31

3 Multiple Interface Fair Queueing **33**

3.1 Problem Statement 33

3.2 Background and Related Work on Fair Queueing 36

3.3 Multiple Interface Fair Queueing 38

3.3.1 Max-min Fair Rate Allocation with Interface Preference 40

3.4 Performance Guarantees 41

3.4.1 Rate Guarantee 41

3.4.2 Leaky Bucket and Delay Guarantee 42

3.5 Rate Clustering Property 45

3.6 Summary 48

4 Multiple Interface Fair Queueing Algorithms **51**

4.1 Problem Statement 51

4.2 Background and Related Work 52

4.3 PGPS for Multiple Interfaces 53

4.3.1 Performance Bounds of PGPS for Multiple Interfaces 53

4.3.2 Non-causality of PGPS for Multiple Interfaces 60

4.4 Multiple Interface Deficit Round Robin (miDRR) 61

4.4.1 miDRR is max-min fair 64

4.4.2 Implementation 69

4.4.3 Evaluation 72

4.5 Summary 76

5 Programmable Open Network (OpenFlow Wireless) **79**

5.1 Problem Statement 79

5.2 Related Work 82

5.3 The OpenFlow Wireless Network Architecture 83

5.3.1 Supporting Radio Agnosticism 85

5.3.2 Slicing the Network 85

5.3.3 Software Friendly Network 87

5.4 Stanford Deployment of OpenFlow Wireless 89

5.5 Evaluation 92

5.5.1 Video Streaming with n-casting 92

5.5.2 Mobility Experiments 93

5.5.3 Network facilitated P2P Multicast 98
5.6 Summary 100

6 Conclusion **103**

Bibliography **105**

List of Tables

4.1 Pseudocode for DRR and miDRR, which is invoked when interface j is free to send another packet. The only difference between the two algorithms is that the highlighted line in Algorithm 4.4.1 is replaced by Algorithm 4.4.2 in miDRR. 62

5.1 Statistics on number of packet lost during handover. 95

5.2 TCP throughput (in Mb/s) observed during experiment. 98

List of Figures

1.1 Example of two applications and two interfaces for which fair scheduling at individual interfaces results in an unfair allocation of 0.5 and 1.5 Mbps respectively, because interface 1 allocates 0.5 Mbps to each flow and interface 2 only serves flow 2. Yet, fair allocation of 1 Mbps each is possible. 4

2.1 The Hercules architecture that presents the protocols and applications with a virtual interface for backward compatibility while multiplexing packets onto different networks using a switch. In turn, the packet processing switch is configured by a control plane that interacts with applications, other devices, and the networks. 10

2.2 Devices on which the Hercules network stack is run, allowing them to exploit multiple networks at the same time. 13

2.3 Hercules overhead evaluated on a Motorola Droid. 14

2.4 Diagram showing flow routes at each stage of the experiment. 15

2.5 Diagram showing address translation happening along the routes of each flow at each stage of the experiment, as illustrated in Figure 2.4. For example, A'< − >A1 implies the address A' is translated to A1 and vice versa. This translation can happen with either source or destination address. 16

2.6 Mobile's throughput during an experiment in which the flow is migrated from WiMAX to WiFi and then through a middlebox. 17

2.7 Stitching two networks: Steady state throughput of a laptop and phone with and without Hercules. 17

2.8 Stitching two networks: Throughput achieved when using Hercules to download a 100 MB file when WiFi is turned off from 20 s to 40 s, and WiMAX from 60 s to 80 s. 18

2.9 Connecting a laptop to ten wireless networks. The data rate increases as more networks are added (in the order listed in the figure). The arrows show when each interface is turned on. 19

2.10 Moving an ongoing flow from WiMAX to WiFi when a device stops moving as a away to demonstrate how feedback from other parts of the system can be used to improve the user experience. 19

2.11 Illustration of network stacks starting with the unmodified network stack in Linux, a non-optimized configuration of Hercules, and finally a buffer-optimized Hercules network stack. 23

2.12 Experiment setup for measuring buffer size of WiFi device by comparing PCI signals and antenna output. 25

2.13 PCI and WiFi outputs showing that the WiFi device can function with very little buffering in the hardware. 27

2.14 Left: the average number of retransmissions (in 0.3 s bins) for a TCP Cubic flow; interface 1 is disconnected after 6 s. The legend indicates the buffer size of the DMA buffer. Right: the expected number of retransmissions (error bars showing standard deviation) immediately after disconnecting interface 1. 28

2.15 Cumulative distribution function of the number of retransmissions in the second after the loss of the interface. The legend indicates the size of the DMA buffer. 28

2.16 Throughput of a flow when 50 (above) or 1 (below) packets are dropped after 10 s. Values are shown for 100 independent runs. 29

2.17 Sender congestion window and slow-start threshold of a single TCP Cubic flow with 50 packets dropped at 10 s. The wide (red) vertical bar indicates that the socket is in recovery phase, whereas the narrower (cyan) vertical bars indicate Cubic's disorder phase. 30

2.18 The effect of burst loss on TCP. 31

2.19 CDF of the time difference between the marked and prioritized packet. . . . 32

3.1 Examples of packet scheduling. An edge between a flow and an interface indicate the flow's willingness to use the interface. 35

3.2 Conceptual model for packet scheduling for multiple interfaces with interface preferences. Matrix Π encodes the flows willing to use each interface, and weight ϕ_i indicates flow i's rate preference. 39
3.3 Illustration of $A_i(\tau_1, t)$ and $S_i(\tau_1, t)$ and their respective upper and lower bounds. Observe that the horizontal distance between A_i and S_i characterizes the delay, while the vertical distance characterizes the backlog at time t. . . 43

4.1 Illustration of cumulative service under multiple interface fair queueing S_i and PGPS $\hat{S}_i$, with the relation of the service bound with respect to the delay bound. 59
4.2 Implementation of miDRR in Linux kernel to schedule outbound packets. . 70
4.3 Ideal implementation of miDRR to schedule both inbound and outbound packets. 71
4.4 Implementation of miDRR in a HTTP proxy to schedule inbound HTTP flows. 72
4.5 Simulation results for three flows over two interfaces. 73
4.6 CDF of scheduling time as a function of the number of flows for four interfaces. 75
4.7 CDF of scheduling time as a function of the number of interfaces for 32 flows. 75
4.8 TCP goodput of three inbound HTTP flows scheduled fairly using our HTTP proxy. 76
4.9 Clustering formed when our HTTP proxy schedules fairly across multiple interfaces. On the left is the clustering during the 11–18 s of the experiment and 29 s on. On the right is the clustering during 0–11 s and 18–29 s. . . . 76

5.1 Vision of future mobile network where the user can move freely between technologies, networks and providers. 80
5.2 The OpenFlow Wireless architecture where control is separate from the physical infrastructure. The control is then "sliced" using FlowVisor and SNMPVisor to provide fine-grained control to the services above. 84
5.3 Building a software-friendly network on top of OpenFlow Wireless by allowing the applications to talk directly to the controller via plugins. 88
5.4 Photographs of a WiFi AP and WiMAX base-station used in the Stanford deployment of OpenFlow Wireless. 90
5.5 Location of 30 WiFi APs deployed in Stanford's Gates building as part of an OpenFlow Wireless deployment. 91

5.6 Screenshots of UDP video with and without n-cast with 3% loss induced on each wireless link. The screenshots demonstrate how simple replication can benefit video quality. 93
5.7 Packet losses in vertical handover. 96
5.8 TCP throughput with handover 97
5.9 P2P chat via in-network mesh-casting (n-multicast). 99

Chapter 1

Introduction

The advent of smartphones brought about a rapid propagation of diverse applications operating over mobile networks. Mobile networks today are struggling to satisfy the different and often stringent requirements of these applications together with an unprecedented growth in mobile traffic.
My key proposal is that we should exploit all of the networks around us. Our smartphones are already equipped with several radios, allowing us to connect to multiple networks and giving us access to enormous capacity and coverage. Furthermore these networks have different characteristics that can be exploited to satisfy the different requirements of applications.
In this chapter, I outline recent developments in the mobile space to motivate my proposal of using all of the networks around us*. Following that, I will describe the key technical challenges that need be addressed.*

1.1 Background and Motivation

During the past couple of years, we have seen quite a change in the wireless industry. For example, handsets have become mobile computers running user-contributed applications on operating systems with open APIs. We are on a path toward a more open ecosystem, one that was previously closed and proprietary. The biggest winners are the users who now have more choices among competing innovative ideas.

The same cannot be said for the mobile networks serving these devices, which remains closed and (mostly) proprietary, and in which innovation is bogged down by a glacial standards process. The industry reports that demand is growing faster than wireless capacity, and the wireless crunch will continue for some time to come.

Yet, users expect to run increasingly rich and demanding applications on their smartphones, such as video streaming, anywhere-anytime access to their personal files, and online gaming; all of which depend on connectivity to the cloud over unpredictable wireless networks. Given the mismatch between user expectations and wireless networks development, users will continue to be frustrated with application performance on their mobile computing devices—on which connectivity comes and goes, throughput varies, latencies can be extremely unpredictable, and failures are frequent.

The problem is often attributed to a shortage of wireless capacity or spectrum; however, this claim cannot be entirely true. Today, if we stand in the middle of a city, we can likely "see" multiple cellular and WiFi networks. However, frustratingly, this capacity and infrastructure is not available to us. Our contracts with cellular companies restrict access to other networks; most private WiFi networks require authentication, effectively making them inaccessible to us. Even if the business reasons were eliminated, the technology employed in our mobile devices and today's network infrastructure still would not allow us to make use of multiple networks at the same time. Hence, although we are often surrounded by abundant wireless capacity, almost all of it is off-limits. Such inaccessibility is not good for us, and it is not good for network owners: Their network might have lots of idle capacity even though a paying customer is nearby.

Users should be able to travel in a rich field of wireless networks with access to all wireless infrastructure around them, leading to a competitive market-place with lower-cost connectivity and broader coverage. If a smart-phone can take advantage of multiple wireless networks at the same time, then the user can experience:

Seamless connectivity through the best current network, and having the ability to choose which network to connect to dynamically;

Faster connections by stitching together flows over multiple networks;

Lower usage charges by choosing to use the most cost-effective network that meets the application's needs;

Lower energy by using the network with the current lowest energy usage per byte.

In the extreme, if all barriers to fluidity are removed, users could connect to multiple networks at the same time, opening up enormous capacity and coverage.

1.2 Problem Statements

My goal is to *allow users to make use of multiple networks at the same time.* To achieve this vision, I will address the various technical challenges involved.

The good news is that smart phones are already armed with multiple radios capable of connecting to several networks at the same time. Today's phones commonly have four or five radios (e.g., GPRS, 3G UMTS, HSPA, LTE, WiFi). Shrinking geometries and energy-efficient circuit design will allow these mobile devices to have more radios in the future. In turn, more radios will allow a mobile device to talk to multiple networks at the same time for improved capacity and coverage, and seamless handover.

1.2.1 A Client Cannot Exploit Multiple Networks

This vision requires more than just multiple radios and multiple networks—it requires that the mobile client can take advantage of them. Today's clients are ill-equipped to do so, having grown up in an era of TCP connections bound to a single physical network connection, leads to several well-known shortcomings.

1. An ongoing connection-oriented flow—like TCP—cannot easily be handed over to a new interface, without re-establishing state.

2. If multiple network interfaces are available, an application cannot take advantage of them to get higher throughput; at best, it can use the fastest connection available.

3. A user cannot easily and dynamically choose interfaces at fine granularity to minimize loss, delay, power consumption, or usage charges.

These three limitations are not just the consequences of TCP. They are manifestations of the way the network stack is implemented in the operating system of the mobile device today. My goal is to understand how we can change today's mobile device to make use of multiple networks at the same time to overcome these limitations along the way.

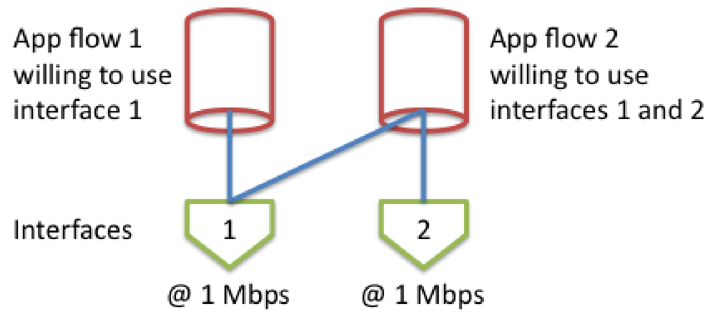


Figure 1.1: Example of two applications and two interfaces for which fair scheduling at individual interfaces results in an unfair allocation of 0.5 and 1.5 Mbps respectively, because interface 1 allocates 0.5 Mbps to each flow and interface 2 only serves flow 2. Yet, fair allocation of 1 Mbps each is possible.

1.2.2 Policy-based Fair Scheduling onto Multiple Interfaces is Undefined

When our mobile devices connect to one or more of the networks around us, we want to make the best use of these networks and exploit their heterogeneous characteristics. We might use 3G to gain wide area coverage and WiFi to minimize delay, and we might spread our traffic over several interfaces to maximize bandwidth. We might express a preference to save precious data rations, such as "do not use a 4G interface for streaming video," or require that secret VPN traffic only go over a trusted 4G network, and we might give precedence to one application over another, such as "if I'm playing a game, throttle my email and Dropbox traffic to 10% of the available link capacity, and devote all the remaining network bandwidth to the game."

Therefore, based on policies, we need a way to flexibly and efficiently control how the different network interfaces are used, how they are aggregated and pooled, and how traffic shares each individual interface.

Existing methods fall short on even simple examples. Fair scheduling algorithms—the basic building blocks for bandwidth and delay guarantees—assume a single interface. If we independently apply fair scheduling to each interface, the result is not fair, as illustrated by the example in Figure 1.1. Not only does classical single interface fair scheduling fail to apply, existing approaches such as TCP and MPTCP [52] do not address the problem. Of course, TCP is not equipped to handle multiple interfaces. MPTCP enables us to use multiple interfaces, but cannot accommodate heterogeneous application preferences. They also have no notion of policy constraints on interface usage.

My goal is to generalize the delivery of network traffic over multiple interfaces by developing a holistic scheduling framework that maps many applications to multiple interfaces while abiding by application preferences and user policies.

1.2.3 Network in Support of Clients using Multiple Networks

Not only is our mobile client ill-equipped to exploit multiple networks, our wireless networks today are also poorly positioned to support devices connected through multiple networks. As we look to the future, we want a network that supports a mobile computer moving freely and seamlessly from one network to another—regardless of who owns the network and the radio technology that it uses.

If users move freely among many networks, the service provider should be conceptually separated from the network owner. The service provider should handle the mobility, authentication, and billing for their users, regardless of the network to which they are connected. In today's network architecture, the technology and services are deeply integrated with the infrastructure, preventing the service provider from innovating and differentiating themselves to, for example, provide different mobility services. Hence, our future network architecture must support such a division in a manner that gives the service provider low-level control of the network infrastructure that in turn, provides them with the mechanism to innovate and differentiate.

On many occasions, applications can benefit from more direct interaction with the network: to observe more of the current network state and to obtain more control over the fate of their flows in the network. In turn, this interaction empowers the mobile client, allowing it to fulfill application preferences and user policies.

My goal is to design a simple network architecture that decouples the service providers from the network owners while providing applications with more direct interaction with the network.

1.3 Outline of Dissertation

In this first chapter, I outlined our motivation for making use of multiple networks and the three key technical problems that I will address in this dissertation.

The remainder of the dissertation is divided into three parts. In Chapter 2, I will describe a novel client network stack (Hercules) that allows the mobile device—its user

and applications—to exploit multiple networks. In Chapter 3, I will lay the theoretical foundations for generalizing the delivery of traffic over multiple interfaces before presenting practical algorithms for doing so in Chapter 4. In Chapter 5, I will present how we can design a network to support mobile clients and applications that uses multiple networks.

Chapter 2

Hercules: Client Network Stack for Devices with Multiple Interfaces

We want our mobile devices to be capable of efficiently making use of multiple networks. Namely, a mobile client should allow us to (1) aggregate bandwidth over multiple interfaces, (2) migrate flows from one network to another, and (3) dynamically choose interfaces at fine granularity to minimize loss, delay, power consumption, or usage charges. Today's client network stacks—designed for a past when only a single network interface was active at any point in time—are ill suited to do so.

In this chapter, I describe Hercules—a client network stack that allows us to make use of multiple networks. Using Hercules, I will demonstrate how we can exploit multiple networks without any changes to the network infrastructure. I will also show how we can reduce packet losses and delays by mapping packets to an interface at the last possible moment in Hercules.

2.1 Problem Statement

A key component of mobile computing is the mobile device—a smartphone, tablet, or laptop. Although the hardware for these devices has tremendously improved over the last few years, their operating systems were developed in the past when a single interface was the norm. Having grown up in an era of TCP connections bound to a single physical

network connection, it is unsurprising that the operating system of today's mobile devices are ill-equipped to exploit multiple networks. This situation creates several well-known shortcomings:

- An ongoing connection-oriented flow—like TCP—cannot easily be handed over to a new interface, without re-establishing state.
- If multiple network interfaces are available, an application cannot take advantage of them to gain higher throughput; at best, it can use the fastest connection available.
- A user cannot easily and dynamically choose interfaces at fine granularity to minimize loss, delay, power consumption, or usage charges.

We need to change the operating system to overcome these limitations to efficiently exploit multiple networks. Our ideal operating system should have the following properties.

1. The operating system should be able to handle multiple active network connections at the same time, unlike today's operating systems. For example, Android—a modern operating system for mobile devices—only allows one network interface to be active at a time. Android chooses the interface to use according to a preference order: If the device is connected to a WiFi network, Android automatically disconnects from WiMAX, which is clearly no good for us.

2. The operating system should be able to support the many network protocols available. For example, it should be able to allow the applications to exploit multiple networks, regardless of whether the application is using UDP, TCP, or some variant of TCP like MPTCP. By doing so, we decouple the operating system from the protocol stack, allowing novel protocols to be readily deployed as they are invented.

3. The operating system should provide a flexible mechanism for interacting with applications, operating systems of other devices, and even the networks to which it is connected. This flexibility will allow the operating system to coordinate with its peers and connected networks to best serve the applications and make the best use of the networks available.

4. The operating system should also be backward compatible. Specifically, it should (1) run on commercially available smartphone devices and laptops, (2) work with

unmodified existing applications, and (3) connect to existing production WiFi and cellular networks.

5. The operating system should handle dynamic changes in network connectivity. In contrast, today's end-host network stacks were designed for wired networks for which connectivity is static. As elaborated further in Section 2.4, this design results in unnecessary packet losses during handovers, and latency-sensitive traffic can be delayed by a competing flow. Ideally, the operating system should avoid these problems to allow the applications and users to easily migrate from one network to another.

In this chapter, I describe Hercules [57], a novel client network stack that satisfies the requirements. Using a prototype of Hercules based on Android (described in Section 2.3.1), I will present how this network stack overcomes the three limitations (in Section 2.3.2). Hercules also mitigates the packet losses and unnecessary delay as described in Section 2.4. In all, Hercules is a client network stack that allows us to efficiently make use of multiple networks at the same time.

2.2 Related Work

Many researchers have explored how to use multiple wireless interfaces at the same time [12, 14]. Some attempted to address how we should use multiple interfaces [50] or how we can deal with the issue that TCP is bound to a network address [34, 36]. Others proposed transport protocols that aggregate bandwidth across multiple interfaces [20, 26], support multi-streaming of independent byte streams [37], or provide the ability to hand over a TCP connection to a new physical path without breaking the application [47]. This work proposes a network stack that can incorporate these techniques and protocols and provides design guidelines for how these (and future) protocols can be implemented. Hence, this work is orthogonal and complementary to these proposals.

This work is also related to a number of recent optimizations to improve wireless network performance, some of which leverage sensors [29] whereas others exploit geolocation information [18] or leverage user-specified application policies [6]. Hercules compliments these techniques by providing the flexibility at the client to take advantage of these innovations.

Hercules mitigates packet losses during handover and unnecessary delay of latency-sensitive traffic. Consequently, Hercules augments efforts to reduce Web page load times,

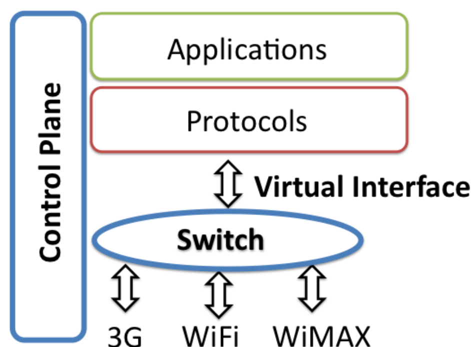


Figure 2.1: The Hercules architecture that presents the protocols and applications with a virtual interface for backward compatibility while multiplexing packets onto different networks using a switch. In turn, the packet processing switch is configured by a control plane that interacts with applications, other devices, and the networks.

particularly when there are competing flows [23]. This effort is related to recent work on "buffer-bloat" [1] that argues for reducing buffers (and therefore latency) in home routers. A similar buffer sizing proposal has been made for WAN and data-center networks [4, 5, 7, 39]. Unlike prior work, this work focuses on the mobile client network stack and the issues that arise in that context.

2.3 Hercules Client Network Stack Architecture

The Hercules network stack consists of three main components (illustrated in Figure 2.1): a switch to multiplex packets onto different networks; a virtual interface presented to the protocols and applications in a backward-compatible manner; and a control plane to coordinate the various ongoing activities.

In Hercules, traffic from an application needs to be spread over multiple interfaces. The application sends traffic using an arbitrary IP source address and the networking stack takes care of spreading the traffic over several interfaces, each with its own IP address. This traffic management is done using the virtual Ethernet interface to connect the application, with its local IP address, to a special gateway inside the Linux kernel. The gateway stitches together multiple interfaces, without the application knowing. Essentially, the gateway is a switch that rewrites the addresses in the packets before sending them to the appropriate interface. In this way, the packets in an application flow are decoupled from the IP addresses on each interface, which allows the set of interfaces to change dynamically as connectivity comes and

goes. This can be similarly done at the communicating peer and resolves the dilemma that we want to be compatible with existing applications and protocols that expect a single active network interface when supporting multiple active network interfaces. By multiplexing packets below the protocol stack, Hercules can support the many protocols implemented in the operating system, including those that are yet to be designed or implemented, thus allowing new protocols to be readily deployed.

Hercules controls how flows are routed onto their respective interfaces using a control plane that configures the switch through a well-defined protocol. Similarly, the control plane can communicate with applications to understand their intents and preferences, and with other control planes on other hosts, to negotiate how flows are spread across interfaces. The control plane might even negotiate with the networks to better serve the applications.[1] In principle, the control plane can be anywhere, for example, it can be implemented as a service in a mobile device, it can be run by the network operator, or it can be outsourced to a third-party provider. This programmable control plane allows Hercules to easily implement one or more mechanisms for interacting with applications, other devices, and the networks.

With this design, Hercules fulfills four of the five properties discussed in Section 2.1. To fulfill the last remaining property, Hercules must be implemented in a way to optimize the buffer management to avoid unnecessary losses and delays. A discussion of this concept is deferred to Section 2.4.

2.3.1 Implementation in Android

A prototype of Hercules is implemented using Android—a modern operating system for mobile devices—as its base. The following modifications are made.

Android/Linux The first problem to solve is that, by default, Android only allows one network interface to be active at a time—clearly no good for us. Android's Connectivity Service is modified to allow us to simultaneously use multiple interfaces. Android is based on a minimal Linux kernel that is missing several needed tools and kernel modules (e.g., the kernel module for virtual Ethernet interfaces). The modules are added and common utilities such as `ifconfig`, `route`, and `ip` are cross-compiled for Android.

[1] The discussion of how Hercules can negotiate with networks is deferred to Chapter 5 after a description is given of how the network can be modified to support a Hercules-enabled client.

Open vSwitch The switch (or gateway) is implemented using Open vSwitch (OVS).[2] Using the Android Native Development Kit (NDK) for the ARM or OMAP processors, OVS's kernel module and user-space control programs are cross-compiled for Android.[3] OVS replaces the bridging code in Linux, and lets us dynamically change how each flow is routed. OVS has an OpenFlow [30, 38] interface; therefore, we can use `<match,action>` flow-table entries to easily route, re-route, and handover existing connections.

Control Plane A small custom control plane is used to determine how flows are routed and re-routed in the prototype. The control plane runs as an Android background service, and applications can interact with the control plane using Android IPCs [2]. This control plane controls OVS using the OpenFlow protocol running over a TCP socket. It controls the network interfaces (and other local resources) through system calls (e.g., Android Runtime Process). The control plane can also communicate with control planes on other hosts using JSON messages, allowing it to negotiate how flows are spread across interfaces.

A similar prototype in Linux is used for laptops and servers used in the experiments. The prototype is run on four common mobile devices (three smartphones running Android, and a laptop running Linux), shown in Figure 2.2:

- Smartphone: Motorola Droid with TI OMAP processor (600 MHz) and 256 MB of RAM, CDMA with Verizon 3G data plan, running Android Gingerbread 2.3.3.
- Smartphone: Nexus One with Qualcomm ARM processor (1 GHz) and 512 MB of RAM, GSM, HSDPA with T- Mobile 3G data plan, running Android Gingerbread 2.3.3.
- Smartphone: Nexus S 4G with Cortex ARM processor (1 GHz) and 512 MB of RAM, CDMA, WiMAX with Sprint 3G/WiMAX data plan, running Android Gingerbread 2.3.5.
- Laptop: Dell with AMD Phenom II P920 quad-core processor (3.2 GHz) and 4 GB memory, installed with Ubuntu 10.04.

[2] OVS was recently upstreamed to Linux kernel 3.3 [15].
[3] The patches and instructions are publicly available at `http://goo.gl/MK5E8`.

(a) Android smartphones. (b) Laptop with ten interfaces.

Figure 2.2: Devices on which the Hercules network stack is run, allowing them to exploit multiple networks at the same time.

Where possible, experiments are performed using mobile phones, but sometimes it is infeasible (e.g., in one experiment, ten interfaces were needed, which is too many for current smart phones). Servers running Ubuntu 11.04 are also used as peer servers and middleboxes in the experiments.

2.3.2 Evaluation: Overcoming Limitations to Exploit Multiple Networks

Overhead of Switch

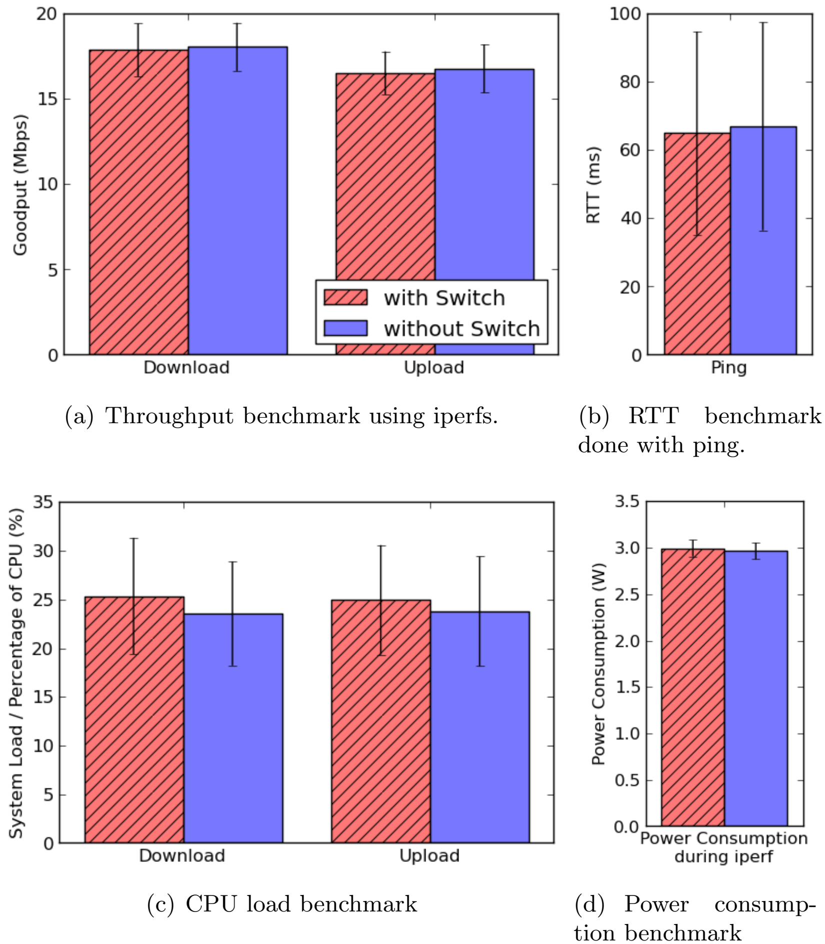
Hercules adds functionality to Android and inevitably consumes more power, more CPU cycles, and potentially reduces maximum throughput. The system is designed for minimal overhead, which our experiments confirmed.

Throughput Reduction The goodput for ten iperfs with and without OVS is measured. To maximize overhead cost, the least provisioned Android device we have (the Motorola Droid) is used. Figure 2.3(a) shows that the goodput is reduced by no more than 2%.

RTT Increase The delay incurred is profiled by sending 300 pings with and without OVS. Figure 2.3(a) shows no observable increase in round trip time.

CPU Load The CPU load is logged when running iperf on the Droid with and without OVS. Figure 2.3(c) shows that the CPU load increased by 1.8%.

Power Consumption To measure our prototype's impact on power consumption, the battery was removed and the device was powered through its USB port and a power monitor. Figure 2.3(d) shows negligible power increase with OVS.



(a) Throughput benchmark using iperfs.

(b) RTT benchmark done with ping.

(c) CPU load benchmark

(d) Power consumption benchmark

Figure 2.3: Hercules overhead evaluated on a Motorola Droid.

Flow Migration without Re-establishing State

Consider the first limitation: An ongoing connection-oriented flow cannot easily be handed over to a new interface without re-establishing state. This experiment shows how Hercules overcomes this limitation, i.e., how Hercules maintains a HTTP connection across a migration. The model of this experiment is a user arriving to work who wishes to migrate an ongoing video stream from a public WiMAX network to a corporate WiFi network.

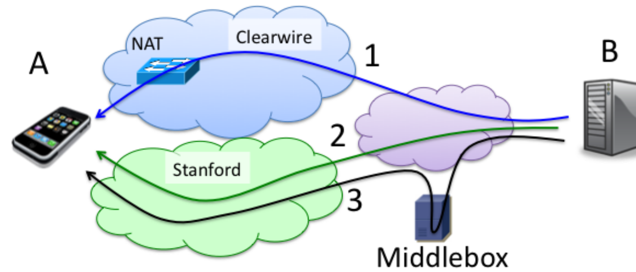


Figure 2.4: Diagram showing flow routes at each stage of the experiment.

In this experiment, both the client and the peer are running Hercules. During the migration, the client's IP address will change. This change has to be coordinated with the peer for a seamless migration through control packets between the control planes. The control packet signals the impending migration of an ongoing flow to the peer, which can be done without aid from the network. The peer then rewrites the addresses of the subsequently incoming packets such that flow migration is transparent to this application.

Several possibilities exists in this design space. In this implementation, the source address is rewritten to that of the initially established flow (as shown in Figure 2.5). At any point in time, the application in host A thinks that the communication is between addresses A' and B whereas the application in host B thinks that the communication is between addresses A1 and B'. The consistent views of the applications in the end hosts are maintained by the translations indicated in Figure 2.5. Another possible implementation is to always rewrite the address of the communicating peer to one that is arbitrarily picked at the onset of the flow.

Figure 2.6 shows the throughput of the session as the flow is migrated (as shown in Figure 2.4). Initially, the flow is routed through WiMAX; then, after 30 seconds it is migrated to WiFi. The control plane decides when to make the move and reconfigures OVS to change the addresses, rewrite packet headers, and switch packets to/from the new interface. This change is coordinated with the control plane of the peer. The result is an uninterrupted TCP flow that has been migrated from one network to another without re-establishing state.

To show the flexibility of our system, a very different migration mechanism (as described by Stoica et. al. in [48]) is also tested. The flow is routed through an off-path middlebox (or waypoint); each end communicates only with the middlebox. This mechanism could

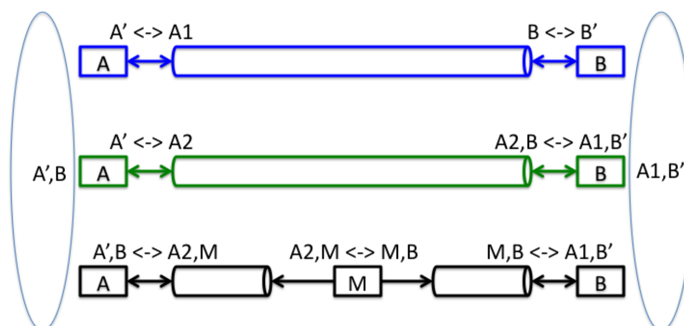


Figure 2.5: Diagram showing address translation happening along the routes of each flow at each stage of the experiment, as illustrated in Figure 2.4. For example, A’$< - >$A1 implies the address A’ is translated to A1 and vice versa. This translation can happen with either source or destination address.

be used, for example, to insert a firewall or DPI box in a corporate environment. In the experiment, the migration takes place at 50 seconds, with a brief drop in data rate when packets reach the middlebox.

The experiment shows that Hercules is quite powerful because both migrations were done without changing the network. Usually, migration and mobility are considered fixed functions of the infrastructure [42, 48].

Aggregating Bandwidth over Multiple Networks

Consider the second limitation: If multiple network interfaces are available, an application cannot take advantage of them to get higher throughput; at best it can use the fastest connection available. Hercules allows multiple networks to be used simultaneously.

To test how well this works, the number of interfaces is varied while data is being downloaded.

In this experiment, a 100 megabyte file is downloaded over five parallel TCP connections using `aria2c`. First, all five TCP connections ran over Stanford’s campus WiFi network; then, Clearwire’s WiMAX network was used. Finally, Hercules stitched both networks together. Each test was run ten times on two clients (the laptop and the Nexus S 4G smartphone), and the average throughput is reported. Figure 2.7 shows the average aggregate throughput with and without stitching. The laptop achieved 95% of the aggregate

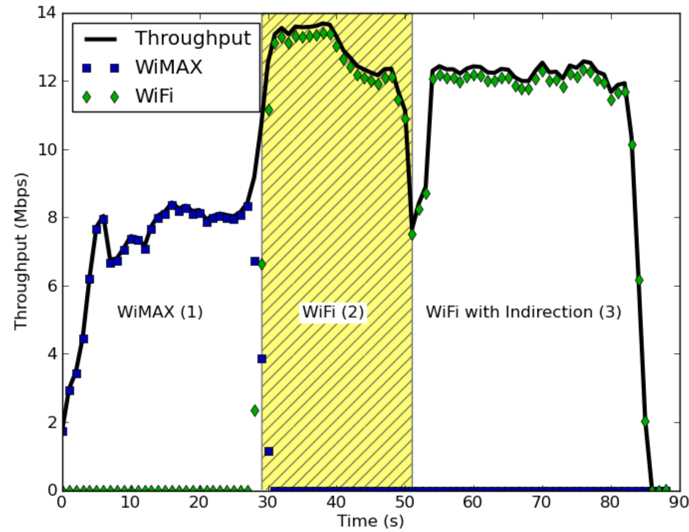


Figure 2.6: Mobile's throughput during an experiment in which the flow is migrated from WiMAX to WiFi and then through a middlebox.

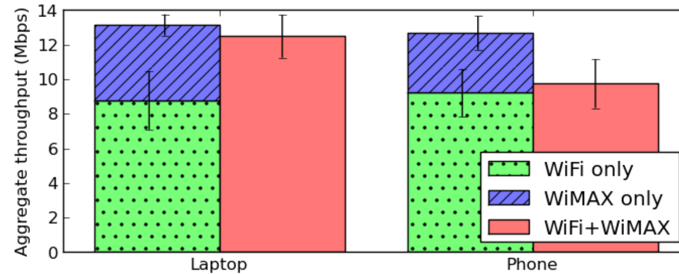


Figure 2.7: Stitching two networks: Steady state throughput of a laptop and phone with and without Hercules.

data rate, whereas the smartphone achieved 77%. Further investigation revealed that interference occurred between the WiFi and WiMAX interface in the mobile phone because the transceivers are close together. There is no fundamental reason why this issue cannot be resolved using better shielding—something we can expect if stitching becomes common.

Stitching interfaces together also helps maintain connectivity during times of packet loss or complete network outage, as Figure 2.8 shows. Each interface was turned off for 20 s during the experiment; connectivity was maintained because of the other interface.

Finally, to push the limits of stitching, ten network interfaces are stitched together (!). The ten networks are listed in Figure 2.9, and include four different wireless technologies:

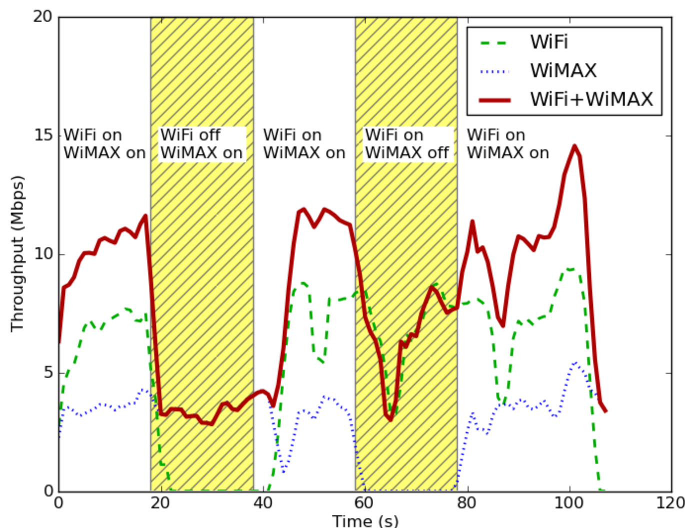


Figure 2.8: Stitching two networks: Throughput achieved when using Hercules to download a 100 MB file when WiFi is turned off from 20 s to 40 s, and WiMAX from 60 s to 80 s.

3G, WiMAX, WiFi 802.11a (5 GHz), and WiFi 802.11g (2.4 GHz) and include six different production networks. In doing so, the capacity available around us is being profiled. The laptop is used in this experiment because there was no way to attach so many interfaces to a smartphone. To measure the capacity brought by each successive interface, each interface is gradually brought up one at a time. Hercules then stitches it to the others to increase the data rate. Figure 2.9 shows the throughput rising as each interface is added, up to a maximum of 70 Mbps (more than three times the fastest interface).

Dynamic Choice of Network

Consider the third (and last) limitation: A user cannot easily and dynamically choose interfaces at fine granularity to minimize loss, delay, power consumption, or usage charges. This final experiment (inspired by [29]), shows how the user or application can choose the network to use. In this experiment, the phone's accelerometer is used to determine whether the device is moving. When the user is moving, WiMAX is chosen for greater coverage; when stationary, the free and faster WiFi network is selected (Figure 2.10).

Because the user (or client) makes the decision, faster innovations can to be designed and easily made available in the future, such as the methods described in [18, 29, 35].

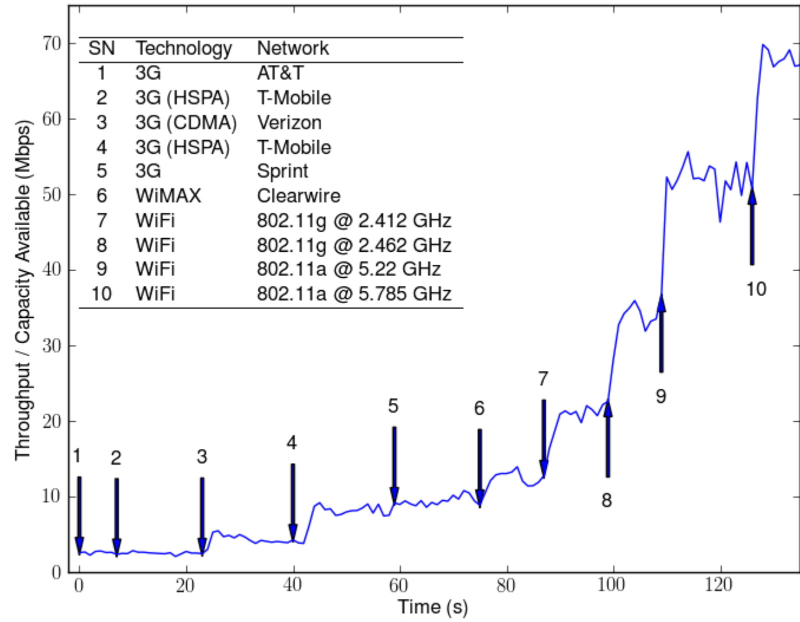


Figure 2.9: Connecting a laptop to ten wireless networks. The data rate increases as more networks are added (in the order listed in the figure). The arrows show when each interface is turned on.

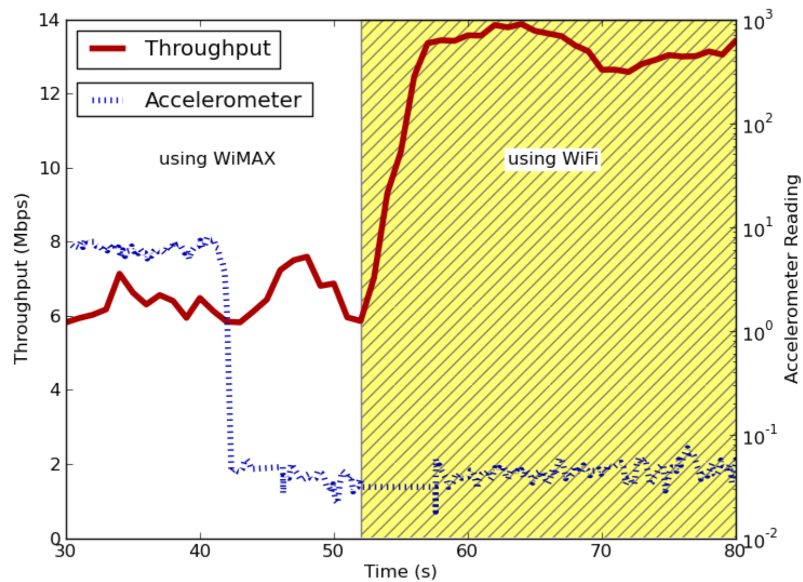


Figure 2.10: Moving an ongoing flow from WiMAX to WiFi when a device stops moving as a away to demonstrate how feedback from other parts of the system can be used to improve the user experience.

2.3.3 Challenges with Current Networks and Devices

The Hercules prototype using Android and Open vSwitch was able to overcome the limitations using only a refactored client network stack without modifying the fixed infrastructure. However, current networks and devices do not make it easy.

Address ambiguity A client might have two interfaces connected to different networks that use identical private address spaces. For exampl,e they might both use addresses starting from 192.168.0.0. Whereas packets can be sent via gateways on both networks, to reach hosts directly attached to either networks requires us to distinguish them by some means other than IP address, such as by forwarding packets based on the interface to which they are destined (if we know). Otherwise, one set of hosts will be unreachable, which is the case for this revision of Hercules. Solving this problem require more work.

Discovering connectivity Discovery protocols (e.g., DNS and DHCP) are typically tied to a particular network interface; therefore, if multiple networks are used, DNS and DHCP settings for each interface must be carefully tracked. To determine the networks available, hosts and routers on each interface have to be proactively ARPed.

Middleboxes Wireless networks—particularly cellular networks—are riddled with middleboxes [51] that interfere with flow migration. For example, a migrating flow might be blocked if the new network did not see a SYN packet that was observed during our experiments. This issue cannot be resolved without changing the network infrastructure. Hopefully, cellular providers will, in time, fix the middlebox problem.

Interfaces Sometimes, a single network requires different header formats depending on the physical device. For example, a 3G network requires Nexus One (using the Qualcomm MSM 3G chipset) to present a virtual Ethernet interface, whereas they are presented as IP interfaces on Google Nexus S and the Sierra 3G USB Dongle. Different interfaces also present different MTU to the network stack, e.g., 3G and Ethernet interfaces typically have MTU of 1,400 and 1,500 bytes respectively. The MTU in the prototype is set to the minimum of all interfaces to work around this problem. These are not limitations to the approach because rewriting the header format arbitrarily for each interface and fragmenting the packet accordingly is possible.

2.4 Buffer Optimization in the Hercules Network Stack

Achieving only the desired functionality is insufficient. Hercules has to be tuned to perform well in a dynamic mobile environment in which available network connectivity changes continuously—an aspect for which today's network stacks are poorly optimized.

Today's end-host network stacks were designed for wired networks in which connectivity is static. To send packets on these networks, applications create transport connections (e.g., TCP sockets) and write data to them. The network stack segments the data into packets, and adds the appropriate headers that include information on the source and destination IP addresses. Packets forming different TCP flows are then multiplexed into a buffer associated with the source IP address, which in turn pushes them into buffers associated with the physical network card from which packets are transmitted. The fate of a packet—in terms of the source IP address used, interface on which it is sent and in what order—is determined the moment the packet is pushed from the transport flow buffers into the lower layers, where appropriate headers are added and the packet is further enqueued in multiple buffer levels. In networks in which connectivity is static (i.e., the physical connection and associated parameters such as source IP are largely static), this design works quite well.

This multiplexed and buffered design of networking stacks performs poorly in a dynamic mobile environment in which available network connectivity changes continuously. If a device hands off to a new AP or switches to a different network interface, we lose all of the packets queued up in the buffers of the disconnected network interface. The interface buffers are often large (hundreds or thousands of packets), leading to large packet loss. If handoffs were rare, or if network conditions were constant, infrequent packet loss might be acceptable. However in a world with many wireless networks from which to choose and devices with multiple interfaces, flows will frequently be mapped to new interfaces; therefore, ways to eliminate (or reduce) such packet losses are needed.

The key problem is that packets are bound to an interface too early. Once an IP header has been added and a packet is placed in the per-interface queue, undoing the decision is very difficult, e.g., if the interface associates with a new AP or if we want to send the packet to a different interface (e.g., if the interface fails or if a preferred interface becomes available). The more packets buffered below the binding point, the greater the commitment and the greater the risk that the packets are lost if network conditions change. Even with

the best configuration, a typical mobile device today can lose up to 50 packets every time it hands off or changes interface.

Another common problem caused by binding too early is that urgent packets are unnecessarily delayed. Because many transport flows are multiplexed into a single per-interface FIFO, latency sensitive traffic is held up. The problem is worst when the network is congested and data backs up in the per-interface queue.

2.4.1 Late-Binding to Reduce Loss During Handover and Unnecessary Delay of Latency Sensitive Traffic

The two problems can be overcome if the network stack follows the late-binding principle, i.e., the decision on which packet to send on what interface is not made until the last possible instant. Late-binding is achievable by doing the following:

1. Minimize or eliminate packet buffering below the binding point. One consequence is that after the binding, the packet is almost immediately sent through the air.

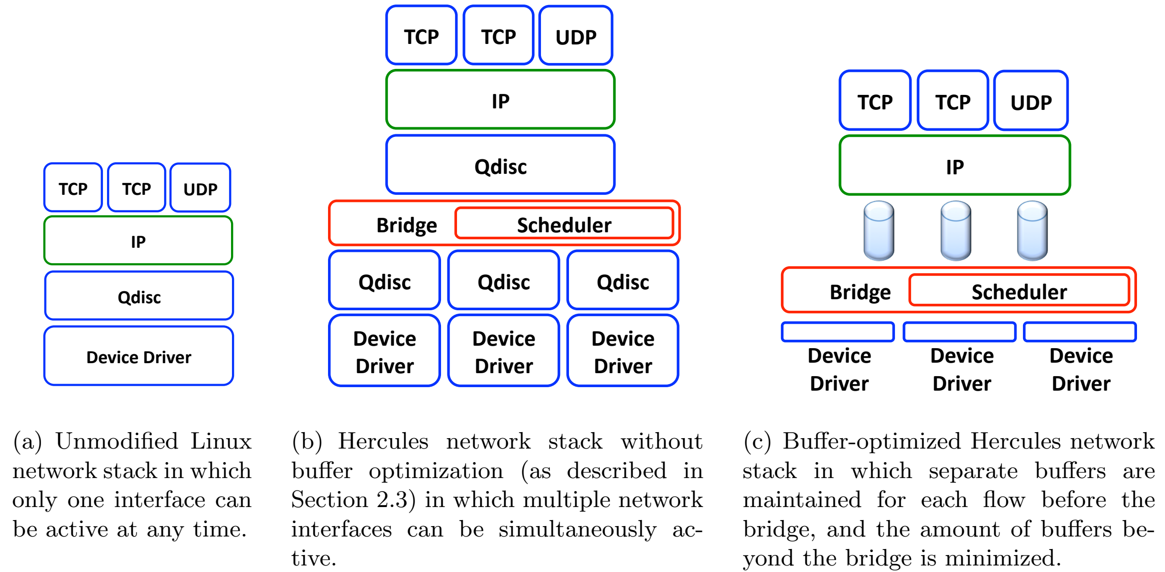
2. Keep flows in separate queues above the binding point to avoid latency-sensitive packets from being unnecessarily delayed. The queues need to be interface-independent to allow us to choose which packet to send on which interface.

To ease adoption, the design should also be hardware independent, allowing any network interface available to be used— precluding changes to the driver.

Recall in Section 2.3 that Hercules started with the default Linux network stack (illustrated in Figure 2.11(a)). Hercules then added a custom bridge that contains a packet-by-packet scheduler to decide which packet to send next and to which interface. This bridge is the point at which a packet is bound to its outgoing interface. Rather than modify the socket buffer to stop it binding packets too early, we allow it to bind packets to a virtual interface, and then remap it as it passes through the bridge. This process has the effect of leaving the socket API unchanged and making the application believe it is using a single interface, when, in fact, its packets may be spread over several interfaces. The resulting design is shown in Figure 2.11(b). Note that there are now qdisc[4] buffers above and below the bridge.

To keep flows separate above the binding point (the bridge), the default qdisc for the virtual interface is replaced with a custom queueing discipline that keeps a separate queue

[4] In Linux parlance, qdisc (for queueing discipline) is a per-interface FIFO queue.



(a) Unmodified Linux network stack in which only one interface can be active at any time.

(b) Hercules network stack without buffer optimization (as described in Section 2.3) in which multiple network interfaces can be simultaneously active.

(c) Buffer-optimized Hercules network stack in which separate buffers are maintained for each flow before the bridge, and the amount of buffers beyond the bridge is minimized.

Figure 2.11: Illustration of network stacks starting with the unmodified network stack in Linux, a non-optimized configuration of Hercules, and finally a buffer-optimized Hercules network stack.

for each socket. Fortunately, Linux was designed to make this process easy. To minimize buffering below the binding point, we make the qdisc bufferless and pass packets from the bridge—as soon as they have been scheduled—directly to the driver. The driver buffer is reduced to two packets, which as we will see is the minimum without disrupting the DMA process. This design pertains to the transmit path. The receive path is largely left unmodified except to forward all received packets onto the virtual interface. The final design is shown in Figure 2.11(c).

2.4.2 Implementation of Late-Binding in Kernel Bridge

The late-binding design is implemented in Linux 3.0.0-17 as a custom bridge in the form of a kernel module. The implementation operates on a Dell laptop running Ubuntu 10.04 LTS with an Intel Core Duo CPU P8400 with a 2.26 GHz processor, 2 GB RAM, and two WiFi interfaces. The two WiFi interfaces are an Intel PRO/Wireless 5100 AGN WiFi interface and an Atheros AR5001X+ wireless network adapter connected via PCMCIA. The implementation follows the design previously described and shown in Figure 2.11(c).

Some details of the implementation are explained here, starting from the top down.

Avoiding binding packets to a physical interface in the socket layer At the socket layer, the Linux virtual Ethernet (`veth`) interface is used to prevent the socket from binding the flow to a physical interface too early, while leaving the socket API (and the application) unchanged. The implementation requires careful handling of addresses (e.g., a WiFi interface will only send Ethernet packets with its own source address). The cost of modifying the header in the bridge is low because it simply requires rewriting header fields. Fortunately, the checksum is calculated later in the network hardware, or just before the DMA.

Sending and receiving packets to multiple interfaces The custom bridge kernel module makes use of the `netdev` frame hook made available since version 2.6.36. This hook allows the custom bridge to be modularly inserted into the Linux kernel.

Eliminating per-interface qdisc buffers below the bridge The `pfifo` or `mq` qdisc associated with the `net_device` of each interface has to be replaced in the bridge. In an unmodified kernel, the `dev_queue_xmit` function enqueues the packet (i.e., puts it in the `qdisc`). Subsequently, the packet is dequeued and delivered to the device driver using the `dev_hard_start_xmit` function. However, it is artificially difficult in the current network stack to enqueue into the qdisc and not drop the packet on failure attributable to the implementation of `dev_queue_xmit`. Therefore, the per-interface qdisc is completely bypassed by partially reimplementing `dev_queue_xmit` to directly invoke `dev_hard_start_xmit` and deliver the packet to the device driver. A full device driver buffer returns an error code that allows us to retry later. The consequence is that qdisc is bypassed without having to replace it.

Minimizing the driver buffer The Atheros WiFi chipset using the `ath5k` driver can be configured via `ethtool`, allowing the device buffer to shrink from 50 packets to two packets.

Avoiding unnecessary drops on the wireless interface When disconnected, the `ath5k` driver has the peculiar behavior—which should be considered a bug—of continuously accepting and then dropping, all packets from the bridge. Clearly, this behavior must be corrected if we want to reroute flows over a different interface. Therefore, the custom bridge checks that the WiFi is connected before sending a packet to the driver.

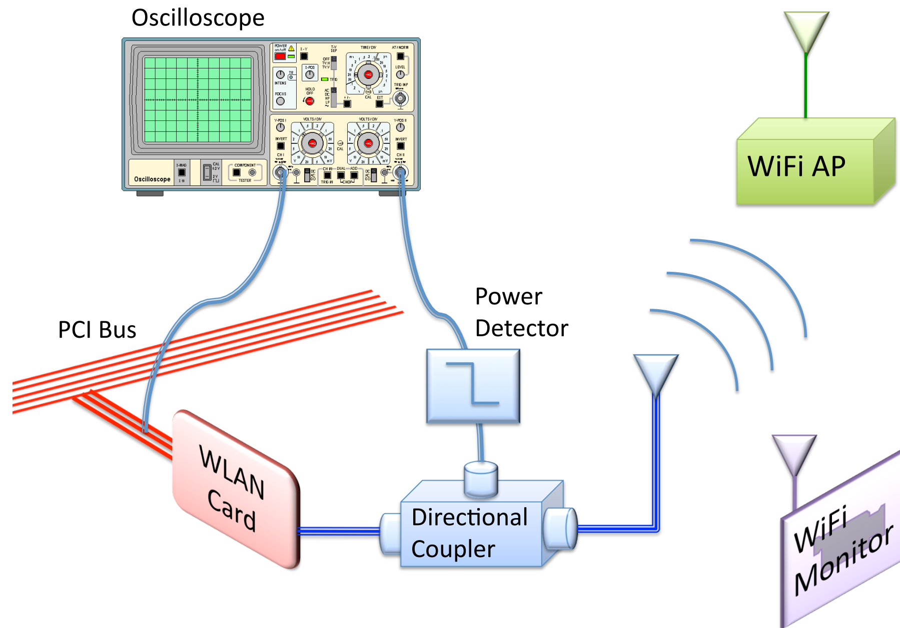


Figure 2.12: Experiment setup for measuring buffer size of WiFi device by comparing PCI signals and antenna output.

2.4.3 Evaluation of Late-Binding

Size of Device Buffer

Whereas qdisc and the driver DMA buffer can be controlled through software, the same cannot be said of the interface's hardware buffer. Late binding will be difficult if the hardware buffer itself is large and cannot be reduced; hence, an important question is the size of the hardware buffer on commodity wireless network interfaces.

Measuring this buffer size turns out to be surprisingly difficult, published datasheets of these cards do not include the number or any interfaces to configure them because they are considered proprietary. Hence, an experiment to reverse engineer the buffer size is performed. The experiment wrote a packet into the device (via DMA) and measured when the packet emerges from the device from its antenna. Our experimental setup shown in Figure 2.12 used a TP-Link TL-WN350GD card equipped with an Atheros AR2417/AR5007G 802.11 b/g chipset using the `ath5k` driver.

To measure the times of packets entering the card via DMA and leaving the card through the antenna, an oscilloscope is used to inspect the physical signals of the PCI bus where the `FRAME` pin of the bus [41] is measured during a DMA transfer. At the same time, the output from the wireless chip is measured using a directional coupler on its way to the antenna. The directional coupler taps only the transmitted signal, which is passed through a power detector to get a low frequency signal that can be observe on the same oscilloscope.

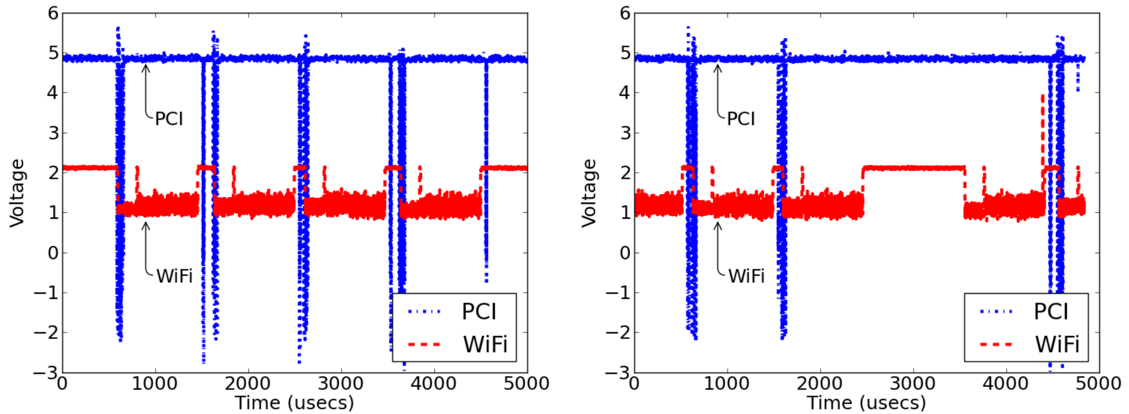
Figure 2.13(a) shows WiFi and PCI activity (both signals are active low, i.e., a lower voltage implies packet activity) when a burst of four 1,400 byte UDP packets is sent to a nearby AP at 18 Mbps. On the PCI bus, the packet is seen being transferred to the wireless chip, and a short status descriptor is sent back to the host after the transmission. On the antenna, a packet transmission consists of a CTS-to-self packet [3], followed by a SIFS (short inter-frame space), and then the actual packet data. Note that as soon as one packet finishes, the DMA transfer for the next packet is triggered. This transfer process is particularly clear in Figure 2.13(b), which shows the retransmission of a packet.[5] No PCI activity occurs during the contention and retransmission phase. Looking closely at the measurements, the interface starts sending the CTS-to-self [3], in preparation to send the next packet, even before the packet has completed its transfer across the PCI bus, indicating a highly pipelined, low-latency design.

The result is very encouraging. It suggests that minimizing the buffering after the binding point only requires changes to the kernel, and not to the wireless chipset—and this is for a chipset connected to the CPU through PCI. For more integrated solutions, such as the system-on-chip designs used in modern mobile handsets, the buffering inside the wireless device can be expected to be just as small, and if we can figure out how to bind a packet to the interface at the very last moment before the DMA, then the number of packets lost when the interfaces changes can be minimized.

Reduced Packet Loss by Late-Binding

The first goal of late-binding is to avoid losing packets unnecessarily when network connectivity changes and a flow is rerouted over to a new interface (using Hercules' custom bridge in the kernel). The test scenario is a Linux mobile device with two WiFi interfaces, each associated with a different access point. A TCP flow is established via interface 1; then, interface 1 is disconnected and the flow is rerouted to interface 2. Packet drops are

[5] A WiFi monitor is used to sniff the channel and verify that a retransmission occurred.



(a) PCI and WiFi outputs for a four packet burst on a WiFi card. At most one packet is inside the device at any time.

(b) PCI and WiFi outputs during a retransmission. The device does not fetch the next packet until the current packet has been transmitted.

Figure 2.13: PCI and WiFi outputs showing that the WiFi device can function with very little buffering in the hardware.

expected when interface 1 is disconnected; the number of drops is measured as a function of the amount of buffering below the binding point.[6] The effect of the retransmissions on the throughput of the TCP flow is also measured.

Figure 2.14 shows the number of packets retransmitted by the TCP flow over a 0.3 s interval. Interface 1 is disconnected after approximately six seconds, the experiment is repeated 100 times, and the results are averaged. The flow uses unmodified Linux TCP Cubic with a throughput of about 5 Mbps and an RTT of 100 ms. As expected, the graph clearly shows that after interface 1 is disconnected, the number of retransmissions increases proportionally with the size of the interface buffer. Although the knowledge that interface 1 is lost is available, these are the unrecoverable packets already committed to interface 1 and scheduled for DMA transfer. With the default buffer of 50 packets, an average of 26.3 packets is lost (and up to 50 packets can be lost as seen in Figure 2.15). If the buffer is reduced to just five packets, the loss is reduced to an average of 3.9 packets.

Next, the effect on TCP throughput when a flow is re-routed from interface 1 to interface 2 is evaluated. Ideally, no packets is dropped or delayed, and TCP throughput is unaffected. As previously observed, TCP reacts adversely to a long burst of packet losses.

[6] Recall that the amount of buffering beyond the bridge can be tuned using `ethtool` for the Atheros WiFi chipset using the `ath5k` driver.

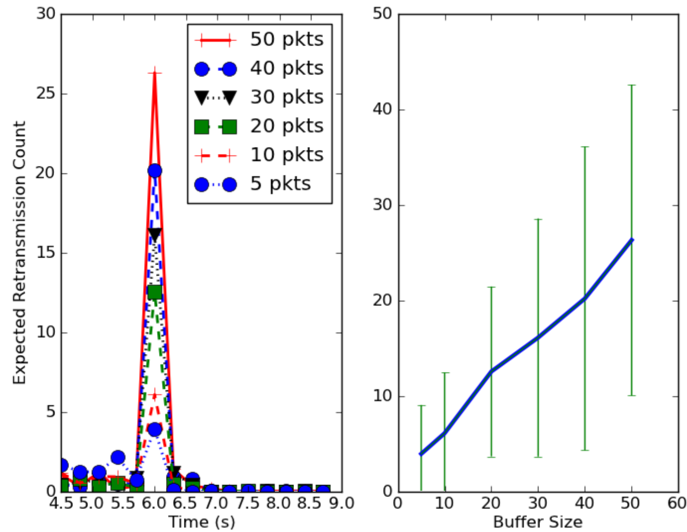


Figure 2.14: Left: the average number of retransmissions (in 0.3 s bins) for a TCP Cubic flow; interface 1 is disconnected after 6 s. The legend indicates the buffer size of the DMA buffer. Right: the expected number of retransmissions (error bars showing standard deviation) immediately after disconnecting interface 1.

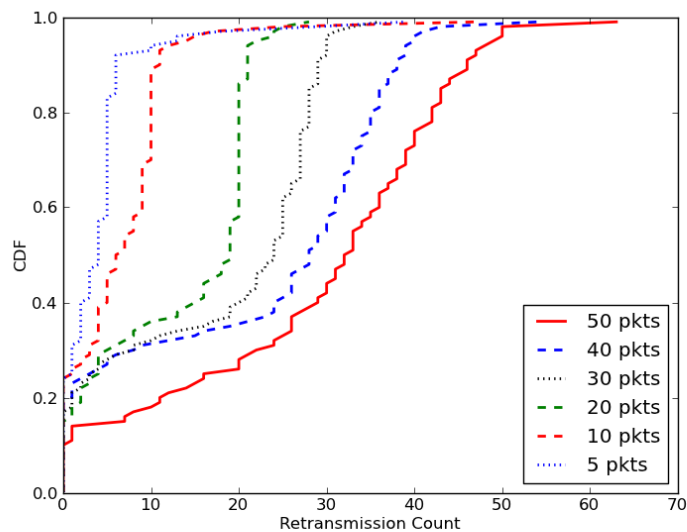


Figure 2.15: Cumulative distribution function of the number of retransmissions in the second after the loss of the interface. The legend indicates the size of the DMA buffer.

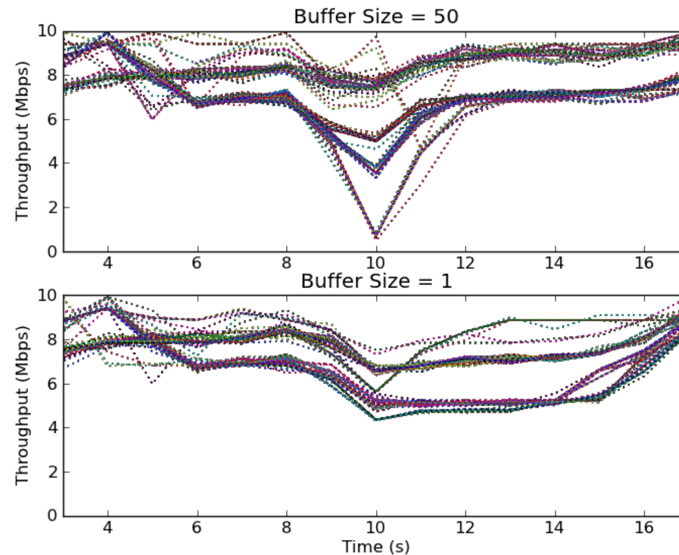


Figure 2.16: Throughput of a flow when 50 (above) or 1 (below) packets are dropped after 10 s. Values are shown for 100 independent runs.

Because the `ath5k` driver will not allow the driver buffer to be set to one packet, the effect of packet loss on throughput is measured using an emulation. In this experiment, the effect of packet loss during handover is emulated when the buffer size is down to just one packet. The bursty loss of packet(s) is emulated using a modified Dummynet [11] implementation. A single 10 Mbps TCP Cubic flow (RTT is 100 ms) is established through interface 1, and—to emulate disconnecting interface 1 after 10 s and rerouting through interface 2—one or a burst of 50 packets is dropped. The experiment was run 100 times and the throughput was measured using `tcpdump` (to reconstruct the flow), and plotted. Figure 2.16 shows that losing a burst of 50 packets (corresponding to a driver buffer of 50 packets) can significantly cause the throughput to decline. If the buffer is reduced to only one packet, throughput is relatively unaffected and no flows drop below 4 Mbps.

To better understand the effect of buffer size on TCP, the dynamics of TCP congestion window after the loss occurs is examined. TCPProbe [25, 55] is modified to tell us the congestion state of the socket and the sender congestion window `snd_cwnd`. The evolution of the state of the TCP flow when 50 packets is dropped is plotted in Figure 2.17, together with the slow start threshold `ssthresh`. The burst of drops causes TCP to enter the recovery phase for more than a second. The actual effect varies widely from run to run depending on the state of the TCP flow when the loss happens. This should come as no surprise as it has

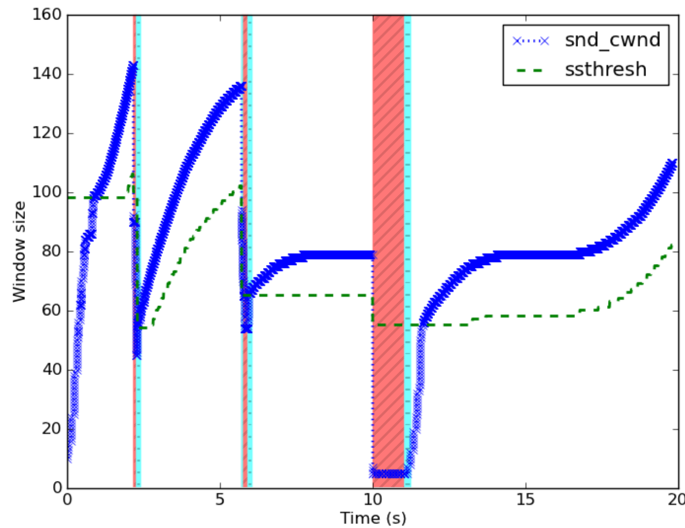


Figure 2.17: Sender congestion window and slow-start threshold of a single TCP Cubic flow with 50 packets dropped at 10 s. The wide (red) vertical bar indicates that the socket is in recovery phase, whereas the narrower (cyan) vertical bars indicate Cubic's disorder phase.

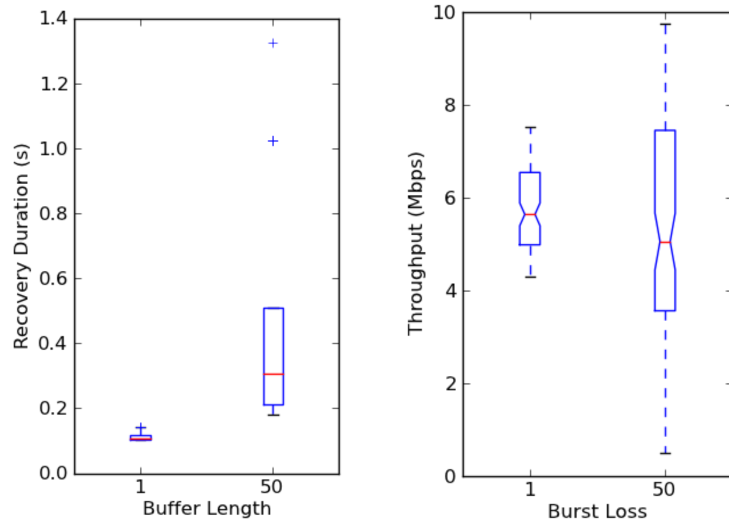
been observed many times that TCP throughput collapses under bursts of losses (e.g., [17]). The key point that this work notes is that packets are dropped unnecessarily in the sending host because of early binding.

As expected, TCP takes longer to recover from a burst loss of 50 packets than from just one packet loss. Figure 2.18(a) shows the distribution of the time it takes for a flow to exit the recovery phase after a burst loss. With a single packet drop, TCP recovers after 110 ms on average (1 RTT). With a burst of 50 packet drops, TCP recovers after 410 ms on average and 1.2 s in the worst measured case. Figure 2.18(b) shows how the packet loss reduces throughput in the second after the packet loss.

Reduced Delay for Latency-sensitive Traffic

A second goal of late-binding is to minimize the delay of latency-sensitive packets. How long a packet is delayed in the driver buffer, and how far we can reduce it, is evaluated in the following.

This experiment uses a single WiFi interface. A marker packet is sent, followed by a burst of 50 UDP packets, followed by a single urgent packet that is sent using a different



(a) Boxplot of the amount of time TCP flow stays in recovery phase after the burst loss.

(b) Boxplot of the throughput of TCP flow in the second after the burst loss.

Figure 2.18: The effect of burst loss on TCP.

port number.[7] `tcpdump` is used to measure the time from when the marker packet is received until the urgent packet is received. The experiment is repeated 50 times for each device buffer size. As before, the Atheros `ath5k` driver is used.

The results in Figure 2.19 show that a larger buffer size delays packets longer, which is not surprising; the driver only has one queue and can not distinguish packet priorities. With the default buffer of 50 packets, the median delay of the urgent packet is 135 ms. With only two packets in the driver, the median delay is only 7.4 ms, or 94% faster. Extrapolating to a driver buffer with only one packet, an urgent packet can be expected to be delayed by less than 5 ms.

2.5 Summary

One thing is clear: Wireless networks are here to stay, and over time, our applications and mobile devices will inevitably and increasingly exploit multiple interfaces simultaneously.

[7] In this implementation, each flow with a different port number is queued separately and treated fairly in a round-robin fashion. The queue with urgent packets can also be prioritized if so desired.

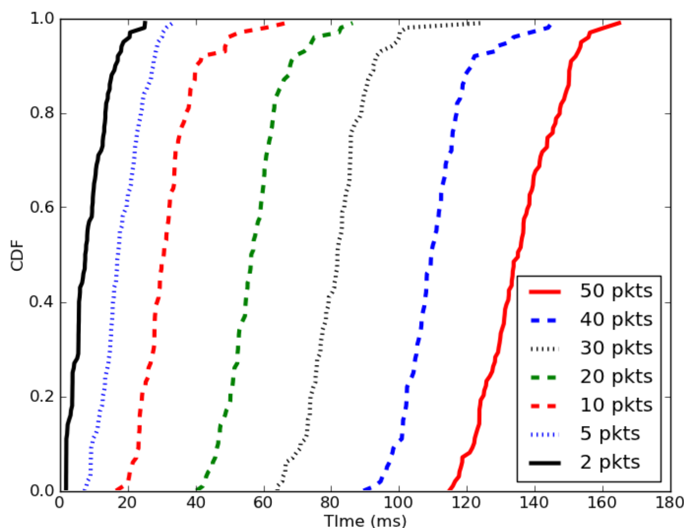


Figure 2.19: CDF of the time difference between the marked and prioritized packet.

It is time to update the client networking stack—originally designed with wired networks in mind—to support wireless connections that come and go, and are constantly changing.

Hercules achieves this by enabling the following: (1) handover an ongoing TCP connection without re-establishing state; (2) stitch multiple interfaces together for higher throughput; and (3) dynamically choose interfaces to minimize loss, delay, power consumption or usage charges. Therefore a refactored client network stack—like Hercules—can achieve our goal of exploiting multiple networks at the same time without modifying the fixed infrastructure.

Further, I introduced the principle of late-binding in which packets are mapped to an interface at the last possible moment, allowing the client network stack to perform in a dynamic mobile environment. Applying late-binding reduces the number of packets lost during a transition by three orders of magnitude, and better serves latency-sensitive flows because they are kept separate until as close to the moment of transmission as possible.

This chapter showed how the proposed Hercules network architecture effectively updates our mobile network stacks to better serve users and applications by exploiting multiple networks at the same time. Interestingly, Hercules can be practically implemented in a way that is backward compatible with existing applications, hosts, and network infrastructures.

Chapter 3

Multiple Interface Fair Queueing

Now that our smartphones have multiple interfaces (WiFi, 3G, 4G, etc.), we are beginning to have preferences for which interfaces an application may use. We may prefer to stream video over WiFi because it is fast, yet stream VoIP over 3G because it provides continued connectivity. We also have relative preferences, such as giving Netflix twice as much capacity as Dropbox. Our mobile devices need to schedule packets in keeping with our preferences while making use of all of the capacity available. This is the natural domain of fair queuing. In this chapter, I show that traditional fair queueing schedulers cannot take into account a user's preferences for some interfaces over others. I then extend fair queueing to the domain of multiple interfaces with user preferences, which guides the design of a packet scheduler in the next chapter.

3.1 Problem Statement

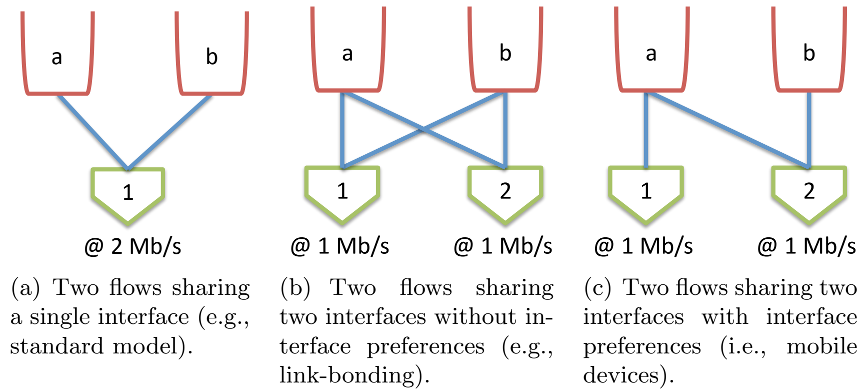
Nowadays, we connect to the Internet from our personal devices via a variety of networks, and often we can connect to multiple networks simultaneously. For example, our phones have 3G, 4G, and WiFi interfaces, and having two or more interfaces active simultaneously is becoming increasingly common among users. At the same time, we are learning that we have preferences on how to use different networks. For example, because 3G/4G connectivity is often capped, we might prefer to download music and stream videos over free WiFi connections. If we are making a VoIP call through Skype, we might prefer to use WiFi

because the latency of 3G networks is higher. However if 4G is available, we may use it because LTE latencies are much smaller than those of 3G. If we are on the move and streaming music from Pandora, we may prefer to use cellular to ensure we are persistently connected. Also, if we are accessing a secure website, we may prefer to use cellular because the connection is encrypted. In the future, we may have preferences that are not currently supported; for example, we may want to use all of the interfaces simultaneously to give all of the available bandwidth to a single application—on a mobile device equipped with Hercules, described in Chapter 2. A large number of such preferences exist, and at their heart, they indicate the conditions under which we want an application to use a given interface.

Preferences are not new, and mobile operating systems now offer coarse preferences through a variety of ad-hoc mechanisms. For example, Android allows us to specify that updating applications should only happen over WiFi, or that Netflix should only use WiFi, and so on. Similarly, Windows Phone has a feature called DataSense whose goal is to ensure that application and user preferences to use WiFi and/or cellular are applied for certain applications. Some applications come with the ability to select whether to only use WiFi. We ourselves sometimes find creative ways to implement these preferences; we might switch off cellular data when we want to force applications to use WiFi or when we are close to our monthly data cap. However, no systematic way exists to ensure that our applications follow our preferences when a mobile device has multiple interfaces.

My goal is to invent a systematic framework and algorithm for using and sharing multiple interfaces while respecting user preferences regarding how they should be used and by which applications. I aim to support binary *interface preferences*, such as ones that disallow particular applications from using certain interfaces, as well as *rate preferences*, where users might want to guarantee preferential treatment to select applications (e.g., allocate at least half of the bandwidth of the WiFi interface to Netflix). At the same time, I wish to maximize the utilization of the available capacity.

To my surprise, I found no prior work that addresses this problem. One might guess that the large amount of work on fair queueing for multiple interfaces might apply. However, prior work does not include the notion of interface preferences: All applications are allowed to use all interfaces in these frameworks. The prior work focuses only on rate preferences, using techniques such as weighted fair queueing (WFQ) [16] to provide weighted fairness. Such work can be found in several contexts, ranging from multihoming to wireless mesh networks.



(a) Two flows sharing a single interface (e.g., standard model).

(b) Two flows sharing two interfaces without interface preferences (e.g., link-bonding).

(c) Two flows sharing two interfaces with interface preferences (i.e., mobile devices).

Figure 3.1: Examples of packet scheduling. An edge between a flow and an interface indicate the flow's willingness to use the interface.

Interface preferences significantly complicate the problem and render prior work inapplicable. To see why, consider the simple toy example shown in Figure 3.1, where two applications share the available interfaces, and no rate preferences exist (i.e. each flow is given the same weight). As prior work suggests, assume we apply WFQ independently on every interface. If only a single interface is present (Figure 3.1(a)), WFQ will provide an equal fair allocation of 1 Mb/s for each flow. Suppose now we have two 1 Mb/s interfaces and the same total capacity of 2 Mb/s. If the flows have no interface preferences and are willing to use both interfaces (Figure 3.1(b)), the fair allocation remains 1 Mb/s for each flow, which can be achieved by implementing WFQ on each interface. However, if we introduce the interface preference that flow a can use both interfaces and flow b can only use interface 2 (Figure 3.1(c)), implementing WFQ on each interface fails to provide a fair allocation: Flow a will get 1.5 Mb/s, while flow b only gets 0.5 Mb/s. This arises because interface 1 gives flow a all of its capacity, as it is the only flow willing to use the interface, and WFQ on interface 2 divides its capacity equally between the two flows.

My goal is to provide an allocation that meets the rate preference (in this case, an unweighted fair share) while respecting interface preferences. In our toy example this means giving each flow 1 Mb/s by giving flow a all of the capacity of interface 1 and flow b all of the capacity of interface 2.

Note that this notion of fairness is a conscious choice for our system. An alternative choice would be to penalize flow b because it is unwilling, or is not allowed, to use one of the interfaces. Instead, wherever possible, we give each flow its weighted fair share of capacity

(defined by the rate preference) without ever violating the interface preference and without ever unnecessarily wasting capacity (i.e., remain work-conserving on all interfaces).

In some cases, the interface preference (which is considered sacrosanct) stands in the way of meeting the rate preference. Going back to our example, if the user declared a rate preference that flow b should have twice the rate of flow a, we have a problem. If no interface preference existed, flow a would receive 0.67 Mb/s, and flow b would receive 1.33 Mb/s. However, because flow b can only use interface 2, we can give it at most 1 Mb/s. Should we give flow a only 0.5 Mb/s to honor the rate preference? Our design decision is no. We never want to waste capacity, so we want to give flow a all of the remaining capacity. We believe this is a reasonable prioritization of goals: While relative flow preferences from users are typically suggestive, interface usage and efficient capacity utilization are prescriptive in nature.

My goal is to invent a practical and efficient scheduling algorithm that schedules packets to meet the rate preference wherever possible, while respecting interface preferences and never wasting capacity. In this chapter, I prove that this is achieved via a classical max-min fair allocation, weighted to give relative rate preference between flows. I will rigorously prove that this provides bandwidth and delay guarantees that are analogous to that for the single interface case, and I will show that such a solution exhibits the *rate clustering property*—which provides the insights needed to design a simple and efficient fair queueing algorithm. The description and analysis of the algorithm is deferred to Chapter 4.

3.2 Background and Related Work on Fair Queueing

Scheduling of flows (or tasks) onto interfaces (or servers) is an important problem that has been studied rigorously. A packet scheduler answers the question

> *When an interface is available, which packet should be sent?*

An ideal packet scheduler answers this question in a way that fulfills several desirable properties:

1. **Work-conserving/Pareto efficient.** To be Pareto efficient in this context means giving rates to flows such that *it is not possible to increase the rate of one flow without decreasing the rate of another flow.* In other words, the total number of packets scheduled is maximized, and no capacity is wasted.

2. **Meet rate preferences.** A packet scheduler should implement the relative priorities of flows encoded by weights ϕ. For example, if flow a has double the weight of flow b, i.e., $\phi_a = 2\phi_b$, we would expect flow a to be allocated twice the rate of flow b, i.e., $r_a = 2r_b$.

For a *single* interface, weighted fair queueing, through algorithms like Packetized Generalized Processor Sharing (PGPS) [40], fulfills all of the above properties by providing each flow with its weighted fair share rate, r_i/ϕ_i. PGPS is known to be max-min fair for a single interface.

Definition 1. *Max-min fair rate allocation is a rate allocation where no flow can get a higher rate without decreasing the rate of another flow that has a lower or equal allocation.*

Because max-min fair is a special case of Pareto efficiency, PGPS is Pareto efficient. The PGPS algorithm meets all of our goals for a single interface by assigning a finishing time to each packet when it arrives and then uses the simple strategy of sending the packet with the *earliest finishing time*. With one interface, PGPS is work-conserving and can faithfully provide the user's weighted preference between flows. If the user asserts a preference that "flow a should receive twice the rate of flow b", then we simply set the weight of a to be twice the weight of b, and PGPS will provide the right allocation.

Fair queueing has also been extended to the case of multiple interfaces within the context of link-bonding in [8], where no notion of interface preferences exists. This has subsequently been analyzed using queueing theory [43], and simple efficient DRR-like algorithms have also been proposed [53, 54]. Our result generalizes these prior works, allowing us not only to compute the max-min fair rate as discussed in [31, 32] but also to provide us with insights to build a practical packet scheduler.

Independently, recent work extended fair queueing along a different dimension. DRF [21, 22] tells us how to schedule fairly over multiple resources. This work differs mainly in that it considers homogeneous resources (e.g., bandwidth on different interfaces), while DRF considers heterogeneous resources (e.g., CPU and bandwidth). A generalization of both works would further improve our understanding of fair queueing but is beyond the scope of this dissertation.

Megiddo showed that the max-min fair allocation is the lexicographic maximum allocation [31]. This insight can be used to derive the max-min allocation via the well-known water-filling technique. Using this, Moser et. al. devised a water-filling-like algorithm to

compute the rate allocation in multiple interface fair queueing in [32, 33]. This work differs in that it is focused on designing a provably optimal packet scheduler, which automatically figures out the max-min fair allocation without explicitly computing the rate allocation.

3.3 Multiple Interface Fair Queueing

Our specific scheduling problem is captured by the abstract model in Figure 3.2, with three flows served by two output interfaces. In this model, each flow i has a weight ϕ_i to indicate its relative priority (its rate preference). For example, if $\phi_1 = 2\phi_2$, application 1 should receive double the bandwidth of application 2 when both are backlogged. Each flow also has interface preferences to indicate the subset of interfaces it is willing to use. If flow a is willing to use interface 1, this is denoted $\pi_{a1} = 1$. The flows' interface preferences are captured by connectivity matrix $\Pi = [\pi_{ij}]$. The matrix represents a bipartite graph (as shown in Figure 3.2), where an edge exists between flow a and interface 1 if and only if flow a is willing to use interface 1, i.e., $\pi_{a1} = 1$. It is critical to note that the bipartite graph is often incomplete—meaning Π is not all-ones, i.e., not all flows are willing to use all of the interfaces. As such, we cannot aggregate interfaces to reduce to the classical single interface case. The combination of rate preferences encoded by weights ϕ and interface preferences encoded by matrix Π indicating absolute restrictions on interface usage allows us to describe a rich variety of interface usage policies.

My goal is to design an efficient packet scheduler that accounts for the interface preferences captured in this model and meets the rate preferences. An ideal packet scheduler answers the question of *when an interface is available, which packet should be sent?* As in the single-interface case, the scheduler answers this question in a way that fulfills several desirable properties. With the introduction of the multiple interfaces and interface preferences, two more properties are added. The properties are listed below in roughly descending order of importance.

1. **Meet interface preferences.** We want a packet scheduler that will only send a packet to an interface it is willing to use. In other words, the packet scheduler must faithfully implement Π.

2. **Work-conserving/Pareto efficient.** As previously described, to be Pareto efficient means giving rates to flows such that *it is not possible to increase the rate of one flow*

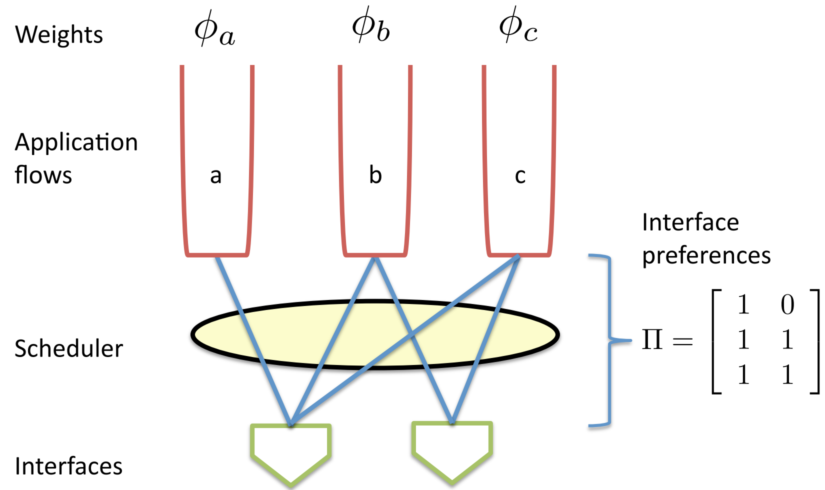


Figure 3.2: Conceptual model for packet scheduling for multiple interfaces with interface preferences. Matrix Π encodes the flows willing to use each interface, and weight ϕ_i indicates flow i's rate preference.

without decreasing the rate of another flow. In other words, we want to maximize the total number of packets scheduled and not waste any capacity. Because max-min fair is a special case of Pareto efficiency, this property will be trivially satisfied by any max-min fair solution.

3. **Meet rate preferences, where possible.** We want a packet scheduler that implements the relative priorities of flows encoded by weights ϕ.

   As pointed out by the example in Section 3.1, interface preferences can make rate preferences infeasible without violating work-conservation. In the case where the rate preferences are feasible, we want the scheduler to always faithfully follow them. In the case where they are not feasible, we first meet the rate preferences subject to the interface preferences and then use up any leftover capacity serving flows that can use it. This means some flows will receive a higher rate than they would if we capped them at their rate preference. However, the key is that no flow will be made worse off; it will only benefit from extra capacity made available to it because other flows were unwilling to use all of the interfaces.[1]

[1] Formally, we will find the lexicographical maximum rate allocation vector, which is as "fair" a rate allocation as we can possibly get without violating the constraints.

4. **Use new capacity.** If we add an interface, we should use it to increase capacity for all flows willing to use it. When a flow ends, other flows sharing its set of interfaces should benefit from the freed-up capacity.

3.3.1 Max-min Fair Rate Allocation with Interface Preference

Consistent with the single-interface case, this work takes the approach of max-min fair queueing. Specifically, we want the rate allocation for each flow $r = [r_i]$ to be max-min fair, where $r_i = \sum_j r_{ij}$, and r_{ij} is the rate at which interface j is serving flow i. Such a rate allocation is subjected to the following constraints:

1. The rate allocated to flow i (denoted as $r_i = \sum_j r_{ij}$) is less than or equal to its demands D_i, i.e., $r_i \leq D_i$.
2. The rate expected of each interface must not exceed its capacity C_j, i.e., $\sum_i r_{ij} \leq C_j$.
3. The routing constraint Π is satisfied, i.e., $\pi_{ij} = 0 \implies r_{ij} = 0$.

The task at hand is how to compute the rate allocation. This computation is needed to understand the properties of such an allocation, and set the weights for each flow. As shown in [31], the weighted fair rate allocated to the flows $[r_i/\phi_i]$ is the lexicographically maximum allocation. Using this, Moser et. al. proposed an algorithm to compute the weighted max-min fair rate [32].

This work presents an alternative method that uses convex optimization. This relies on the proportional fair [28] allocation being max-min fair, as proved in Theorem 1.

Theorem 1. *The proportional fair allocation r is also max-min fair.*

Proof. Assume allocation r is proportional fair but not max-min fair. Since r is not max-min fair, flows a and b exists where $r_a > r_b$ and it is possible to transfer allocation from application a to b. The magnitude of gain for flow b and the magnitude of the loss by flow a are equal, denoted as ϵ. The resulting max-min fair allocation is denoted as $s = [s_i]$.

Because r is proportional fair, then by definition,

$$\begin{aligned}
\textstyle\sum_i (s_i - r_i)/r_i &\leq 0 \\
(s_a - r_a)/r_a + (s_b - r_b)/r_b &\leq 0 \\
-\epsilon/r_a + \epsilon/r_b &\leq 0 \\
r_a &\leq r_b
\end{aligned}$$

This results in a contradiction. Hence, r must be max-min fair. □

Therefore, we can find the proportional fair allocation instead by maximizing the *sum of strictly concave utilities*, such as in the following convex problem:

$$\max \sum_i \log\left(\textstyle\sum_j r_{ij}\right)$$

$$\begin{aligned} \text{subjected to} \quad \textstyle\sum_j r_{ij} &\leq D_i \\ \textstyle\sum_i r_{ij} &\leq C_j \\ r_{ij} &= 0 \quad ,\forall i,j,\pi_{ij}=0 \\ r_{ij} &\geq 0 \quad ,\forall i,j \end{aligned}$$

3.4 Performance Guarantees

3.4.1 Rate Guarantee

In WFQ with a single interface of capacity C, the rate flow i receives, $r_i(t) \geq \frac{\phi_i}{\sum_j \phi_j} C$. By picking ϕ_i appropriately, a minimum rate at which each flow will be served can be guaranteed. For example, if flow 1 is to receive at least 10% of the link rate, then we simply set $\phi_1 = 0.1$ and make sure $\sum_i \phi_i \leq 1$.

Now that we have the weighted max-min fair allocation for the *multiple interface* case, it is worth asking if we can still give a rate guarantee for each flow in the system. Theorem 2 tells us that a flow will indeed receive an equivalent rate in multiple interface fair queueing.

Theorem 2. *Under the weighted max-min fair allocation, the rate flow i receives is at least its weighted fair share among all of the flows willing to share one or more interfaces with i,*

$$r_i(t) \geq \frac{\phi_i}{\sum_{j|\exists k,\pi_{ik}=1,\pi_{jk}=1} \phi_j} \sum_{j,\pi_{ij}=1} C_j(t), \tag{3.1}$$

where t denotes a particular time.

Proof. Imagine that all of the flows willing to share one or more interfaces with i use exactly the same set of interfaces. Then, the equation above is an equality because r_i is the weighted

max-min fair allocation. If any flow uses less than this weighted fair share (because the flow has no more packets to send, or because the flow uses an interface that i is unwilling to use, or because the flow is unwilling to use an interface that i uses), then it would increase service rate allocation to the remaining flows, including flow i. Hence, the inequality holds. □

It follows that under the weighted max-min allocation, flow i will receive at least its weighted fair share of the interfaces it is willing to use,

$$r_i(t) \geq \frac{\phi_i}{\sum_j \phi_j} \sum_{j,\pi_{ij}=1} C_j(t).$$

because this is smaller than the right-hand side of Equation 3.1. For simplicity of notation, the guaranteed rate for flow i is denoted as g_i, where $r_i(t) \geq g_i,\ \forall t$.[2] Hence, if in Figure 3.2 we want flow a to receive at least 20% of C_1, then it is sufficient to set $\phi_1 = 0.2$ and $\phi_a + \phi_b \leq 1$ because only flow b shares interfaces with flow a. When we run the algorithm to set the weighted max-min fair allocation, flow a will receive at least the requested service rate.

3.4.2 Leaky Bucket and Delay Guarantee

Another well-known property of single-interface WFQ is that it allows us to bound the delay of a packet through the system if the arrival process is constrained. The usual approach is to assume that arrivals are *leaky-bucket constrained*. If $A_i(t_1, t_2)$ is the number of arriving packets for flow i in time interval $(t_1, t_2]$, then we say A_i conforms to (σ_i, ρ_i) (denoted $A_i \sim (\sigma_i, \rho_i)$) if

$$A(t_1, t_2) \leq \sigma_i + \rho_i(t_2 - t_1) \quad , \forall t_2 \geq t_1 \geq 0. \tag{3.2}$$

The burstiness of the arrival process is bounded by σ_i, while its sustainable average rate is bounded by ρ_i.

In the classic single-interface WFQ proof, it can be shown that the delay of a packet (the interval between when its last bit arrives to when its last bit is serviced) in flow i is no more than σ_i/ρ_i. Admission control is very simple: If $\sum_i \rho_i < r$, and $\sum_i \sigma_i \leq B$, where B

[2] The service rate $r_i(t)$ can be bounded more tightly by calculating the weighted max-min fair rate for each flow, assuming they are all backlogged. Let the result be $\mathcal{R}^* = [r^*_{ij}]$ and $g_i = \sum_j r^*_{ij}$. It can be proved that $r_i(t) \geq g_i \quad , \forall t$. However, this does not yield a closed-form solution. The proof is fairly simple and is omitted here.

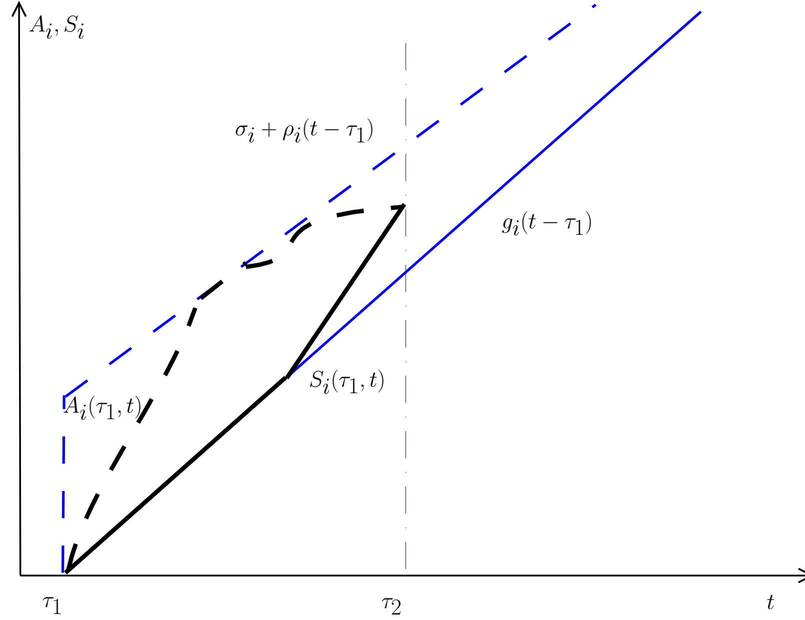


Figure 3.3: Illustration of $A_i(\tau_1, t)$ and $S_i(\tau_1, t)$ and their respective upper and lower bounds. Observe that the horizontal distance between A_i and S_i characterizes the delay, while the vertical distance characterizes the backlog at time t.

is the size of the packet buffer, then flow i can be admitted into the system, and the delay guarantee can be met.

Multiple interface fair queueing has the same property, and the delay of a packet in flow i is no more than σ_i/ρ_i (Theorem 3). However, the process of deciding whether a new flow can be admitted is more complicated than for the single interface case. We have to know which interfaces the flow is willing to use and whether the requested service rate ρ_i can be met. This means the system has to pick values for $\phi_j, \forall j$ such that the rate is guaranteed by Equation 3.1, $r_i(t) > \rho_i$. If this condition can be met, then the delay guarantee is accomplished, and the departure process will also be (σ_i, ρ_i)-constrained.

Theorem 3. *Imagine we wish to admit (σ_i, ρ_i)-constrained flow i into a multiple interface fair queueing system, and the flow is willing to use a subset of the interfaces. If $\sum_j \sigma_j \leq B$, and if we can find values of $\phi_j, \forall j$ such that $r_i(t) > \rho_i$, then the delay of any packet in flow i is upper bounded by σ_i/ρ_i.*

Proof. Let $S_i(t_1, t_2)$ be the service received by flow i in time interval $(t_1, t_2]$. Consider flow i that arrives at τ_1 (i.e., becomes backlogged) and finishes at τ_2 (i.e., becomes non-backlogged). We observe that $A_i(\tau_1, t)$ is upper bounded by (3.2) and $S_i(\tau_1, t)$ is lower bounded by

$$S_i(\tau_1, t) \geq g_i(t - \tau_1) \quad , \forall \tau_1 \leq t \leq \tau_2,$$

where $r_i(t) \geq g_i$, as illustrated in Figure 3.3. Because the delay is the length of horizontal line between A_i and S_i, we can find its maxima through simple calculus.

We begin by deriving the inverse functions of the bounds,

$$y = \sigma_i + \rho_i(t_a - \tau_1) \quad \rightarrow \quad t_a = \frac{y - \sigma_i}{\rho_i} + \tau_1$$
$$y = g_i(t_s - \tau_1) \quad \rightarrow \quad t_s = \frac{y}{g_i} + \tau_1.$$

The delay of packets must then be upper bounded by

$$D = t_s - t_a = \frac{y}{g_i} - \frac{y - \sigma_i}{\rho_i}.$$

Observe that D is an affine function with gradient

$$\frac{dD}{dy} = \frac{1}{g_i} - \frac{1}{\rho_i},$$

which is a strictly negative constant because $\rho_i < g_i$. This means D is monotonically decreasing. In this analysis, we are interested in the domain of $t \geq \tau_1$. Hence D is maximized at $y = \sigma_i$. This, in turn, implies that packet delay

$$D \leq \frac{\sigma_i}{g_i} < \frac{\sigma_i}{\rho_i}$$

because $\rho_i < g_i$. This maximum delay occurs for the last bit that arrives at time τ_1 for the last bit that arrives in the initial burst.[3] □

[3] It can further be shown that $\sigma_i / \sum_i a^*_{ij}$ is a tight upper bound of packet delay. This delay occurs in the worst case scenario, where $\sigma_j \gg \sigma_i$ and $\rho_j = g_j$ for all $j \neq i$ and flow i experiences the worst possible arrival process $A_i(\tau_1, t) = \sigma_i + \rho_i(\tau_1, t)$.

3.5 Rate Clustering Property

For single-interface WFQ, all active flows are served at the same weighted rate r_i/ϕ_i, which is necessary and sufficient conditions for weighted max-min fairness. This property does not hold true for multiple interface fair queueing. However, a scheduler that implements multiple interface fair queueing exhibits an analogous property—the *rate clustering property*, as defined in Definition 2.

Definition 2 (Rate Clustering Property). *A scheduler satisfies the* rate clustering property *if*

1. *It splits the union set of flows and interfaces into disjoint clusters, where each flow and each interface can only belong to a single cluster.*

2. *Within a cluster $\mathbb{C}_i$, all flows are served at the same rate*[4] *(by the interfaces also in $\mathbb{C}_i$). i.e,*

$$a, b \in \mathbb{C}_i \implies r_a = r_b.$$

3. *Among the clusters containing an interface that flow a is willing to use, flow a will* only *belong to the cluster with the highest rate, i.e.,*

$$a \in \operatorname*{arg\,max}_{C_i, \exists j \in C_i, \pi_{aj}=1} r(\mathbb{C}_i),$$

where $r(\mathbb{C}_i)$ is the rate at which cluster $\mathbb{C}_i$ serves its flows.

Any scheduler satisfying the rate clustering property is max-min fair. Intuitively, from the perspective of an arbitrary flow a, the rate clustering property divides flows and interfaces into three distinct sets of clusters. The first set comprises clusters that do not have any interface that flow a is willing to use. The second is the cluster where flow a belongs. The third set comprises clusters to which flow a can possibly belong to but does not.

Recall from Definition 1 that in a max-min fair allocation, no flow can get a higher rate without decreasing the rate of another flow that has a lower or equal allocation. Let us consider how flow a could get a higher rate. Clearly, it cannot get a higher rate by using any interface in the first cluster. If it increases its rate by getting more from its own

[4] With the introduction of multiple interfaces, the reader has to differentiate between the rate at which an interface serves a flow versus the aggregate rate at which the flow is being served by all the interfaces. In this case, we are referring to the latter.

cluster, the rate clustering property tells us that flows from the same cluster all have the same rate as flow a, and therefore, increasing its rate will decrease the rate of a flow of equal allocation. Similarly, since flow a belongs to the cluster with the highest rate, flows belonging to the third set of clusters have a lower (or equal) rate than flow a does. If flow a gets any rate from the third set, it will decrease one of a lower or equal allocation. Hence, the rate clustering property ensures that the allocation is max-min fair.

It turns out that the rate clustering property is not only sufficient but also necessary for a max-min fair scheduler. This is formally proven in Theorem 4.

Theorem 4. *A work-conserving system is max-min fair* if and only if *the following conditions are satisfied.*

1. *If flows i and j are actively serviced by a common interface (i.e., in the same cluster), their allocated rate is the same, i.e.,*

$$\exists k, \quad i, j \in \mathcal{U}_k \implies r_i = r_j,$$

   where $\mathcal{U}_k = \{i, r_{ik} > 0\}$.

2. *If both flows i and j are willing to use interface k, but only flow i is actively using it (meaning the flows are in different clusters), the rate allocated to flow j must be greater than or equal to that of flow i, i.e.,*

$$\exists k, \quad i \in \mathcal{U}_k, j \in \mathcal{F}_k \implies r_j \geq r_i,$$

   where $\mathcal{F}_k = \{i, \pi_{ik} = 1\}$.

To prove the theorem, we begin with a (self-evident) lemma on the Pareto efficiency of a work-conserving system.

Lemma 1. *In a work-conserving system, no flow can increase its allocation without decreasing another's allocation, i.e.,*

$$\delta_i > 0 \quad \implies \quad \exists \delta_j < 0,$$

where δ_i is the change in flow i's allocation.

In other words, if an allocation is *not* max-min fair, there must exist flows i and j where decreasing the flow with larger allocation will increase the other flow's allocation. This leads to our next lemma on the sufficient conditions for max-min fairness.

Lemma 2 (sufficient condition)**.**
In a work-conserving system, the following conditions (as listed in Theorem 4) imply that the allocation is max-min fair.

1. $\exists k, \quad i, j \in \mathcal{U}_k \implies r_i = r_j$.
2. $\exists k, \quad i \in \mathcal{U}_k, j \in \mathcal{F}_k \implies r_j \geq r_i$.

Proof. Assume the opposite; i.e., both conditions are always true, but the system is not max-min fair. Because the system is work-conserving, there is no idle capacity if any flow is backlogged. From Lemma 1, for the system to not be max-min fair, there must exist flows i and j such that j can increase its allocation by decreasing i's while $r_i > r_j$.

This exchange of allocation can happen in two ways:

1. The exchange occurs on interface k, i.e., r_{jk} is increased while r_{ik} is decreased. This means $r_{ik} > 0$ and $a \in \mathcal{U}_k$. If $j \in \mathcal{U}_k$, then $r_i = r_j$ by the first condition. Else, j must at least be in $\mathcal{F}_k$ for r_{jk} to be increased to a non-zero amount. This means $r_j \geq r_i$ by the second condition. In either case, it contradicts the requirement that $r_i > r_j$.
2. The allocation could be exchanged through a series of intermediary flows. Denote the n intermediary flows involved as flow $1, 2, \cdots, n$ where $n > 0$. This means flow i would exchange allocation with flow 1, which, in turn, passes the allocation to flow 2, and so on. Flow i must share a common interface with flow 1, and the above arguments must hold. This means $r_1 \geq r_i$, $r_2 \geq r_1$, and so on. Putting this together, we see $r_i \leq r_1 \leq r_2 \leq \cdots \leq r_n \leq r_j$, which contradicts $r_i > r_j$.

Hence, the allocation must be max-min fair if the conditions are true. □

This conditions are also necessary as shown in Lemma 3.

Lemma 3 (necessary condition)**.**
In a work-conserving system that is max-min fair, the following conditions must be true:

1. $\exists k, \quad i, j \in \mathcal{U}_k \implies r_i = r_j$.
2. $\exists k, \quad i \in \mathcal{U}_k, j \in \mathcal{F}_k \implies r_j \geq r_i$.

Proof. Consider a work-conserving system that is max-min fair. Let us evaluate the two conditions in this scenario:

1. If both flows i and j are actively serviced by a common interface k, flow i must not have an allocation greater than flow j. Else, we can increase r_j by decreasing r_i. The converse similarly applies. Because $r_i \not> r_j$ and $r_j \not> r_i$, $r_i = r_j$.

2. If both flows i and j are willing to use interface k, but only flow i is actively using it, then flow i must not have a greater allocation than flow j. Else, r_{jk} can be increased by decreasing r_{ik}. Hence, $r_i \leq r_j$.

Thus, both conditions are necessarily true for a work-conserving max-min fair system. □

Proof of Theorem 4. Putting the lemmas together, we have Theorem 4. This tells us that all a packet scheduler needs to do is to maintain the rate clustering property, and it will lead to a max-min fair allocation. □

3.6 Summary

In this chapter, I have set out to design a packet scheduler that satisfies several important properties, and I seek to do so by achieving weighted max-min fairness over the rate allocation $r = [r_i]$. These properties are indeed satisfied by multiple interface fair queueing.

1. **Meet interface preferences.** Clearly, multiple interface fair queueing meets this property by design. The user sets values π_{ij} in the algorithm to represent interface preferences, and a flow is never scheduled on an interface for which $\pi_{ij} = 0$.

2. **Pareto efficient.** This follows directly because the packet scheduler gives a weighted max-min rate allocation.

3. **Meet rate preferences, where possible.** In multiple interface fair queueing, by picking the appropriate weights ϕ_i, flow i will receive a guaranteed share of the outgoing line, as shown in Theorem 2. Further, if $\phi_a = 2\phi_b$ for flows a, b and both flows are in the same cluster, then $r_b = 2r_a$, which meets the designated rate preference.

4. **Use new capacity.** If new capacity becomes available, then we want to know that multiple interface fair queueing will use it. There are three reasons more capacity becomes available: A new interface comes online, the rate of an interface increases, or a flow ends freeing up capacity. In each case, if extra capacity becomes available on an

> interface, this capacity can be used by the flows that are willing to use the interface. In turn, these flows might free up yet more capacity on other interfaces, that, in turn, can be taken up by other flows, and so on. A max-min fair solution will maximize the minimum rate in the allocation, so although some flows may receive a higher rate, no flow will receive a lower rate.

Having shown that multiple interface fair queueing fits our bill, the task remaining is to design an algorithm that achieves multiple interface fair queueing—which is the topic of Chapter 4 in this dissertation.

Chapter 4

Multiple Interface Fair Queueing Algorithms

In Chapter 3, I showed how interface preference renders prior work in WFQ unusable. In this chapter, I will present a novel packet scheduler called miDRR that meets our needs by generalizing DRR for multiple interfaces. I demonstrate a prototype running in Linux on a mobile device and show that it works correctly and can easily run at the speeds we need.

4.1 Problem Statement

The desire to use several network interfaces at one time on our mobile devices led us to the problem of packet scheduling with interface and rate preferences. The introduction of interface preferences gives the classical problem of packet scheduling a new twist.

Not only do interface preferences render WFQ algorithms unusable, they challenge our understanding of packet scheduling. Prior packet scheduling algorithms have mostly been defined using service fairness, i.e., by assuring that a pair of flows send equal numbers of bits over time. This means we can track how much service each flow receives over time and just make sure that they are equal. With interface preferences, one flow can receive a higher rate than another in a max-min fair allocation. Therefore, it is possible for the flow to accumulate more service over the other, resulting in an unequal number of bits being sent over time. This changes the way we think about fairness in rate and in service.

Hence, it is crucial that we understand the implications of such preferences and how they change the solution space. In Chapter 3, I presented how a classical max-min fair allocation can be extended to meet the need for multiple interfaces.

In this chapter, I will use the resulting insight to design a simple and efficient algorithm: multiple interface deficit round robin (miDRR) that schedules packets to meet the rate preference wherever possible while respecting interface preferences and never wasting capacity. Our algorithm is practical to implement, yet we can formally prove its correctness. At first blush, designing an algorithm appears too daunting because each interface needs to know how fast flows are being scheduled by other interfaces, leading to a communication explosion in the packet scheduler. The key intuition behind our algorithm—that makes it practical without sacrificing correctness—is that if an interface maintains a single state bit per flow, then a generalized version of DRR meets our needs.

4.2 Background and Related Work

In this section, two classical fair queueing algorithms are reviewed. These algorithms serve as the basis for designing algorithms for multiple interface fair scheduling.

In [40], Packet-by-packet GPS (PGPS) is proposed as a practical approximation of WFQ—where the latter is a fluid model where flows are sent bit by bit. PGPS employs a simple strategy: When the interface is available, send the packet that finishes the earliest under GPS. This simple strategy is shown to yield the weighted max-min fair allocation, and it provides bounded delay as compared to WFQ.

Despite its simplicity, PGPS was considered too complex for hardware implementation. In response, Shreedhar and Varghese [46] proposed Deficit Round Robin (DRR) as a simpler alternative to PGPS. DRR operates in a simple manner. The algorithm serves each flow the same number of times by serving them in a round robin fashion. To provide relative preferences, e.g., serve flow a twice as fast as flow b, the number of bits served during each turn—known as the quantum—is adjusted accordingly. To ensure fairness over time while sending packets integrally, DRR also maintains a per-flow deficit counter, which is how much service this flow has earned but has not been provided over time. DRR—while simple—is provably fair. Furthermore, the algorithm requires $O(1)$ work for each decision, as compared with the $O(\log(n))$ for PGPS, where n is the number of flows. DRR is widely deployed in switches and routers today.

DRR is also commonly used as the basis for many other derivative algorithms. For example, MS-DRR and MS-URR [53, 54] are modeled after DRR for the link-bonding application described in [9]. As previously noted, such algorithms do not fit the bill for multiple interface fair scheduling because they do not address the interface preferences.

4.3 Packet-by-packet Generalized Processor Sharing (PGPS) for Multiple Interfaces

For a *single* interface, PGPS [40] provides each flow with its weighted fair share rate, r_i/ϕ_i. The PGPS algorithm meets all of our goals for a single interface by assigning a finishing time to each packet when it arrives, and then it uses the simple strategy of sending the packet with the *earliest finishing time*. With one interface, PGPS is work-conserving and can faithfully provide the user's weighted preference between flows. If the user asserts a preference that "flow a should receive twice the rate of flow b," then we simply set the weight of a to be twice the weight of b, and PGPS will provide the right allocation. We might wonder if we can define a finishing time across *multiple* interfaces, taking into consideration usage preferences, and then for each interface schedule the packet with the earliest finishing time. We now prove that this strategy would provide us with bounded rate and delay guarantees when applied to multiple interfaces with interface preferences.

4.3.1 Performance Bounds of PGPS for Multiple Interfaces

Delay Guarantee

Single-interface PGPS is shown to have additional delay of no more than L_{max}/C, as compared with single-interface GPS, where L_{max} is the maximum length of a packet and C is the capacity of the interface. A similar bound can be provided for PGPS applied to multiple interface fair queueing, allowing the delay guarantee for multiple interface fair queueing (in Theorem 3) to be naturally extended to the algorithm. The approach adopted here is to consider the cases where the packetized nature of PGPS imposes more delays than multiple interface fair queueing does, and to bound these delays.

Consider a sequence of packets serviced by interface j under multiple interface fair queueing with PGPS. Let packet p_k be the kth packet in this sequence. If p_k is serviced by

interface j under multiple interface fair scheduling, it can suffer additional delays in PGPS for the following reasons:

1. p_k did not arrive in time to be scheduled for service; hence, other packets are scheduled for service by interface j, while p_k has to wait due to the packet constraint. This mis-ordering delay is denoted as d_l.

2. Interface j can also fall behind because it has to service a full packet, although under multiple interface fair scheduling, it did not service all of the bits in the packet because the packet can be split and serviced over multiple interfaces. Further, the policy of serving the packet with the earliest finishing time first can induce interface j to serve a packet that it did not under the idealized service discipline. This can also result in interface j falling behind. This delay due to mis-serviced packets is denoted as d_w.

3. Finally, p_k might be serviced at a faster rate across multiple interfaces under multiple interface fair scheduling than what an individual interface can offer in PGPS. Hence, an extra delay can be incurred if p_k is serviced at a lower rate by interface j than what it receives under the idealized discipline. This delay due to the different service rate is denoted as d_r.

Each of these delays is analyzed in the following.

Delay due to mis-ordering, d_l: Bacuse mis-ordering is due to a packet arriving too late to be scheduled in time, it is bounded by a maximum size packet being sent on the slowest interface. Within this time, the blocking packet must have been served, and this late packet can be served next. Formally, this is captured in Lemma 4.

Lemma 4.

$$d_l \leq \frac{L_{max}}{C_{min}},$$

where L_{max} is the maximum size of a packet, and $C_{min} = \min_i C_i$ is the minimum capacity among all interfaces.

Proof. Assume interface j only services packets that it also serviced under GPS. Let the length of p_k be L_k, its arrival time be a_k, and its departure time under multiple interface fair queueing and PGPS be u_k, t_k respectively. Consider packet p_m where m is the largest index for which $0 \leq m < k$ and $u_m > u_k$.

For $m > 0$, p_m begins transmission at $t_m - (L_m/C_j)$, and packets indexed $m+1$ to k must arrive after this time, i.e,

$$a_i > t_m - \frac{L_m}{C_j}, \quad \forall m < i \leq k.$$

Because the packets indexed from $m+1$ to $k-1$ arrive after $t_m - (L_m/C_j)$ and depart before p_k under multiple interface fair queueing,

$$\begin{aligned} u_k &\geq \frac{\sum_{i=m+1}^{k} L_i}{C_j} + t_m - \frac{L_m}{C_j} \\ \therefore u_k &\geq t_k - \frac{L_m}{C_j} \end{aligned}$$

Therefore,

$$d_l = t_k - u_k \leq \frac{L_{max}}{C_j} \leq \frac{L_{max}}{C_{min}}.$$

□

Delay due to mis-serviced packets, d_w: Consider the following example with interfaces i, j having rates of C_i, C_j, respectively, where $C_i \ll C_j$. If flows $a, b \in \mathcal{F}_i$ and $b \in \mathcal{F}_j$ (where flow $a \in \mathcal{F}_i$ if and only if it is willing to use interface i), we can show that under multiple interface fair queueing service, interface i would only serve flow a and interface j would only serve flow b.

However under PGPS, interface i would service flow b because of the following: The next packet in the queue for interface j has finishing time $t_{j+1} = t_j + (L_{j+1}/C_j)$, where t_j is the finishing time of the current packet being serviced and L_x is the length of the xth packet. Similarly, the next packet in the queue for interface i has finishing time $t_{i+1} = t_i + (L_{i+1}/C_i)$. Because $C_i \ll C_j$, t_{j+1} can be smaller than t_{i+1} and $b \in \mathcal{F}_i$, resulting in interface i choosing to service flow b because interface i will serve the packet with the earliest finishing time. This results in an additional delay d_w being added to the packets of flow a that are being serviced by interface i.

In the worst case, flow a might have a maximum size packet, while flow b has a lot of small packets. In that case, interface i will send the packets in flow b until their finishing time is L_{max}/C_i. Given that interface i is slower, this delay is dilated by a factor of C_j/C_i. Hence, flow a can be delayed by up to $L_{max}C_j/C_i^2$, and this scenario

can be repeated for the same flow at a maximum of $\lceil \log_2 n \rceil$ times. This means the delay due to mis-serviced packets is bounded by $\lceil \log_2 n \rceil \frac{L_{max} C_{max}}{C_{min}^2}$, as shown in the following lemma.

Lemma 5.

$$d_w < \lceil \log_2 n \rceil \frac{L_{max} C_{max}}{C_{min}^2},$$

where n is the number of interfaces, L_{max} is the maximum size of a packet, and $C_{min} = \min_i C_i, C_{max} = \max_i C_i$ are the minimum and maximum capacities among all interfaces, respectively.

Proof. Consider the above example. Interface i will continue to choose to service flow b until the next packet queued for service in interface j has a finishing time greater than t_{i+1}. While the finishing time of packets for interface j increases by a maximum of $t_{i+1} < L_{max}/C_i$, they are sent at a lower rate on interface k, delaying interface i up to $L_{max}C_j/C_i^2$.

Thus, interface j can delay interface i by at most $L_{max}C_j/C_i^2$. However, we have n interfaces in system. Can the other systems affect interface i similarly?

Once interface i has been delayed, it can only be further delayed by another interface that is similarly delayed. Therefore, this can only happen to interface i for $\lceil \log_2 n \rceil$ times, where n is the number of interfaces in the system.

$$d_w < \lceil \log_2 n \rceil \frac{L_{max} C_{max}}{C_{min}^2}.$$

□

Under multiple interface fair queueing, a packet in flow j can also be split and serviced over multiple interfaces. Under PGPS, it is necessary that interface k service the entire packet, meaning it has to service the bits in that packet that it does not serve under GPS. We can think of the above as PGPS servicing a partial packet it did not serve under GPS, which can be considered a special scenario of the above lemma.

Delay due to different service rate, d_r: Under multiple interface fair queueing, p_k of flow i would be serviced at its weighted max-min fair rate $a_i(t)$. However, under PGPS, a packet must be serviced as an entity, i.e., by a single interface at the capacity or

service rate of that interface, which is unlikely to be exactly $a_i(t)$. This can incur an extra delay d_r, which is upper bounded by how long it takes a maximum size packet to be served by the slowest interface L_{max}/C_{min}, as shown in Lemma 6.

Lemma 6.

$$d_r < \frac{L_{max}}{C_{min}},$$

where L_{max} is the maximum length of a packet, and C_{min} is the minimum capacity for a interface.

Proof. Consider p_k in flow i of length L_k serviced at $a_i(t)$ under multiple interface fair queueing and C_j under PGPS. The difference in transmission time is

$$\begin{aligned} d &= \frac{L_k}{C_j} - \frac{L_k}{a_i(t)} \\ &\leq L_k \left(\frac{1}{C_{min}} - \frac{1}{\max_t a_i(t)} \right) \\ \therefore d_r &< \frac{L_{max}}{C_{min}}, \end{aligned}$$

as required by our lemma. □

This result is intuitive: A packet cannot be delayed more than the time it takes for it to be serviced under PGPS.

Equipped with the above lemmas, we can now upper bound the delay of a packet under PGPS for multiple interfaces.

Theorem 5.

$$F_P - F_G < L_{max} \left(\frac{\lceil \log_2 n \rceil C_{max}}{C_{min}^2} + \frac{2}{C_{min}} \right),$$

where F_P, F_G are the finishing times of packet under PGPS and multiple interface fair queueing, respectively.

Proof. Consider that

$$\begin{aligned}
& F_P - F_G \\
\leq \; & d_l + d_w + d_r \\
< \; & \frac{L_{max}}{C_{min}} + \lceil \log_2 n \rceil \frac{L_{max} C_{max}}{C_{min}^2} + \frac{L_{max}}{C_{min}} \\
< \; & L_{max} \left(\frac{\lceil \log_2 n \rceil C_{max}}{C_{min}^2} + \frac{2}{C_{min}} \right),
\end{aligned}$$

as we can see from Lemmas 4, 5 and 6. □

Similar to single-interface PGPS, the additional delay incurred by multi-interface PGPS is a function of L_{max}, the capacities of the interfaces C, and the number of interfaces n. These quantities are readily available in the process of flow admission, allowing the delay bound (in Theorem 3) to be easily extended for multi-interface PGPS.

Service Bound

The difference in cumulative service under single-interface GPS and single-interface PGPS is bounded by the length of the largest packet L_{max}. We can provide a similar bound based on the delay bounds we have just derived in Theorem 5. This enables us to upper bound the cumulative service that flow i receives under multi-interface PGPS, compared to multiple interface fair queueing.

Theorem 6. *The difference in cumulative service between PGPS and GPS is*

$$\begin{aligned}
& S_i(0,t) - \hat{S}_i(0,t) \\
< \; & L_{max} \left(\frac{\lceil \log_2 n \rceil C_{max}}{C_{min}^2} + \frac{2}{C_{min}} \right) \sum_{j, \pi_{ij}=1} C_j.
\end{aligned}$$

Proof. Consider the cumulative service of flow i under multiple interface fair queueing and PGPS, denoted as $S_i(0,t)$ and $\hat{S}_i(0,t)$, respectively. At any point in time, the delay is bounded by Theorem 5 (as illustrated by Fig. 4.1). Because the service rate of flow i is

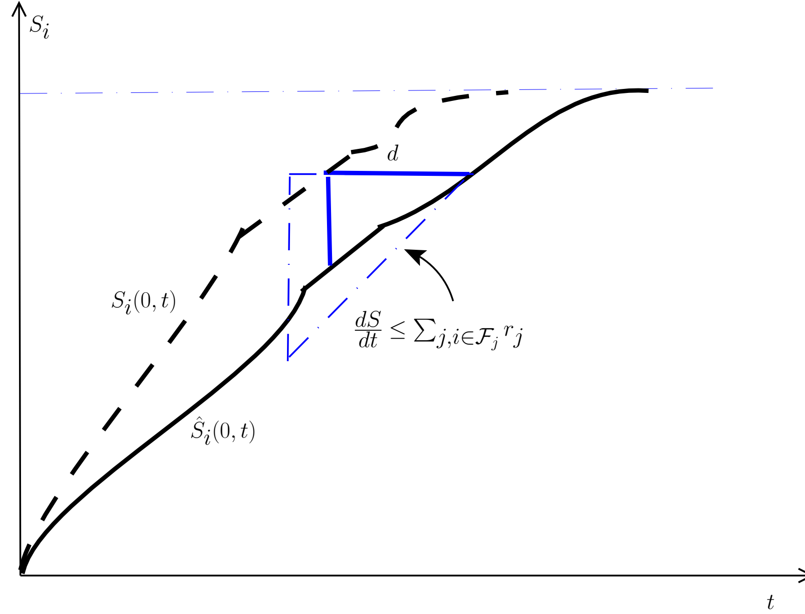


Figure 4.1: Illustration of cumulative service under multiple interface fair queueing S_i and PGPS $\hat{S}_i$, with the relation of the service bound with respect to the delay bound.

upper bounded by $\sum_{j,\pi_{ij}=1} C_j$, we can deduce that

$$\begin{aligned} & S_i(0,t) - \hat{S}_i(0,t) \\ < \;& \frac{dS}{dt}(F_P - F_G) \\ < \;& L_{max} \left(\frac{\lceil \log_2 n \rceil C_{max}}{C_{min}^2} + \frac{2}{C_{min}} \right) \sum_{j,\pi_{ij}=1} C_j . \end{aligned}$$

□

Theorem 6, in turn, bounds the maximum backlog possible under multi-interface PGPS. This allows us to check if we have sufficient packet buffers to accommodate the additional backlog. Again, the service difference is a function of L_{max}, the capacities of the interfaces C, and the number of interfaces n, which is readily available during flow admission.

Let us consider a system of 12 interfaces with $L_{max} = 1500$ bytes, $C_{max} = 1$ Gbps, and $C_{min} = 100$ Mbps. The extra delay incurred by multi-interface PGPS is 5.04 ms as compared to 0.012–0.12 ms in the single interface case. This also implies a maximum service

difference of 3.125 MB for a flow with aggregate bandwidth of 5 Gbps, which is significantly more than the 1.5 KB incurred by single-interface PGPS.

4.3.2 Non-causality of PGPS for Multiple Interfaces

Although intuitively simple, PGPS is not possible with a causal algorithm for multiple interfaces with interface preferences. Unlike the single interface case, the algorithm must know about future packet arrivals.

Our proof is by counter-example. Consider the example illustrated in Figure 3.1(c), where flows a and b share interface 2, while only flow a can use interface 1. Let head-of-line packets of the flows at $t = 0$ be p_a and p_b, respectively (with lengths L and $L/2$ bits, respectively) and all flows have equal priority. Both interfaces run at 1 Mb/s. At $t = 0$ when interface 2 becomes available, it must decide if p_a or p_b would finish first under PGPS. Consider the following two scenarios:

1. If no new flows arrive after $t = 0$, each flow would get rate 1, so the finishing times of p_a and p_b would be $f_a = L$ and $f_b = L/2$ respectively and p_b will finish first.

2. Assume three new flows arrive shortly after $t = 0$ and they are only willing to use interface 2. Rather than compete for interface 2, flow a will continue to use interface 1, and its rate will remain at 1 Mb/s. Meanwhile, flow b's rate reduces to 1/4 Mb/s. In this case, p_a would finish first.

Because the finishing order of the packets in these two scenarios is different and the packet scheduler cannot causally determine which scenario would occur, it cannot determine the relative finishing order of the packets. This leads us to Theorem 7.

Theorem 7. *In the presence of interface preferences, a packet scheduler cannot always causally determine the relative order of the packet finishing time.*

Now, consider the same example, but without interface preferences, i.e., both flows a and b are willing to use all interfaces available, as illustrated in Figure 3.1(b). In this case, packet p_b will always finish first. Even if three new flows arrive shortly after $t = 0$, both flows a and b would be both slowed down to a rate of 2/5 Mb/s because the new flows are also willing to use both interfaces. The proportional change in rate in flows a and b, i.e., *fate-sharing* among the flows, allows us to know the relative finishing order of the packets at the time of their arrival.

The key difference in scheduling packets with and without interface preferences—and the reason prior work fails to help—is that fate-sharing is no longer true with interface preferences. Without interface preferences, if the number of active flows changes, or if the capacity of an interface changes, all flows are equally affected and share the same fate. With interface preferences, changes affect flows using one interface more than others.

4.4 Multiple Interface Deficit Round Robin (miDRR)

Knowing that we cannot use a scheduler that calculates finishing time in advance, could we turn to reactive scheduling mechanisms such as Deficit Round Robin (DRR) [46]? The idea behind DRR, summarized in Algorithm 4.4.1, is to serve each flow the same number of times by serving them in a round robin fashion. To provide rate preferences, the number of bits served during each turn—known as the quantum—is adjusted accordingly. To ensure fairness over time while sending packets integrally, DRR maintains a per-flow deficit counter, which is essentially a metric for how much service this flow has earned but has not been provided over time. This seems to avoid the problems of causality, as we make no assumptions about the interface rates and start deficit counting only after flows arrive.

However, a naive implementation of DRR on each interface does not work either. In our simple example in Figure 3.1(c), DRR would give the same rate allocation as WFQ (flows a and b would get 0.5 Mb/s and 1.5 Mb/s, respectively), whereas we know there is a feasible max-min allocation of 1 Mb/s per flow.

The underlying problem is that if a flow is willing to use more than one interface, when an interface schedules a packet, it has no way of knowing what rates the flows are getting from *other* interfaces. Having this information is crucial to ensure max-min fairness. An obvious solution is for interfaces to exchange information about the rates that flows are receiving from every interface. This seems to require the algorithm to keep track of the rates provided to each flow, and when it is making its own packet scheduling decision it would decide whether or not servicing that flow leads to a max-min fair solution. As the reader can guess, this scheme would require an impractical amount of state information to be maintained and exchanged as well as interfaces to know their own instantaneous rates. Both of these are problematic requirements, especially on mobile devices. Therefore, how might one achieve a max-min fair rate allocation that can be calculated independently

Symbol	Description
BL_i	Backlog of flow i
$Size_i$	Size of flow i's head-of-line packet
Q_i	Quantum for flow i
DC_i	Deficit counter for flow i
$\mathcal{F}_j$	Set of flows willing to use interface j
$\mathcal{C}_j$	Current flow interface j is serving
$\mathcal{B}$	Set of backlogged flows
SF_{ij}	Interface j's service flag for flow i (Service flags for new flows are initiated at zero.)

Algorithm 4.4.1: DRR(j)

if $\mathcal{F}_j \cap \mathcal{B} = \emptyset$
  then return
$i = \mathcal{C}_j$
if $Size_i \leq DC_i$
  then $\begin{cases} \text{Send } Size_i \text{ bytes} \\ DC_i = DC_i - Size_i \end{cases}$
if $BL_i = 0$
  then $\begin{cases} DC_i = 0 \\ \text{Remove } i \text{ from } \mathcal{B} \end{cases}$
if $BL_i = 0$ **or** $Size_i > DC_i$
  then $\begin{cases} i = C_j = \text{Next backlogged flow for } j \\ DC_i = DC_i + Q_i \end{cases}$

Algorithm 4.4.2: MIDRR-CHECK-NEXT(i, j)

$C_j =$ Next backlogged flow for j
while $SF_{ij} \neq 0$
  do $\begin{cases} SF_{ij} = 0 \\ C_j = \text{Next backlogged flow for } j \end{cases}$
$SF_{ik} = 1 \quad, \forall k \neq j$
return (i)

Table 4.1: Pseudocode for DRR and miDRR, which is invoked when interface j is free to send another packet. The only difference between the two algorithms is that the highlighted line in Algorithm 4.4.1 is replaced by Algorithm 4.4.2 in miDRR.

on each interface without ever having to calculate and exchange the actual achieved rates among all of the interfaces?

The key contribution of miDRR is achieving max-min fairness over multiple interfaces with interface preferences *while requiring almost no coordination among the interfaces*. Specifically, it requires no rate computations and, at most, one bit of coordination signaling from each interface for every flow. The one bit is a boolean service flag, and there is one flag at an interface for every flow. The flag indicates whether a flow has been serviced recently by another interface. When an interface considers servicing a flow, it skips it if the service flag is set. We show that this simple mechanism and minimal book-keeping achieves a max-min fair allocation when we have interface preferences. By obviating the need to exactly track service rates and—as we shall see—implicitly enforcing the relative rates between any pair of flows, the service flag allows miDRR to be scalable and highly-distributed by minimizing the overhead of communication among interfaces.

The bareness of the mechanism is surprising. How can a single flag be sufficient to let us achieve a max-min fair rate when we do not even know the rates of each interface, nor do we know the rates achieved by the flows themselves? The insight is that to ensure max-min fairness, it is sufficient to know only the relative rates achieved between flows; the absolute value is not needed. Further, each interface only needs to know the relative rates achieved among flows that it is allowed to service according to the interface preferences. Finally, we do not even need to know a precise value of the relative rate. The packet scheduler only needs to check if a particular flow's rate is higher than at least one other flow it is servicing on the same interface. If so, the scheduling decision is simple: It should not service the flow with the relatively higher rate. If it iteratively applies the above condition to all its flows, it will eventually service the flow that will push it toward a max-min fair rate allocation overall, as we show formally in the next section. Below, we describe how the algorithm operates.

Maintaining the boolean service flag requires two tasks:

1. Interface j maintains one service flag SF_{ij} for each flow i that it serves. The flag is for other interfaces to indicate to interface j that flow i has been serviced recently.

2. When interface k serves flow i, it <u>sets</u> service flags $SF_{ij} \forall j \neq k$ to tell the other interfaces that flow i has been served.

3. When interface j considers flow i for service, it <u>resets</u> service flag SF_{ij}.

Race conditions on the service flag can easily be handled by a standard mutex. More importantly, maintaining the service flag therefore does not require keeping track of the rate at which each interface is serving each flow. Nor do we have to know how much another interface have served a flow.

In essence, the algorithm entails each interface implementing DRR independently with the slight modification of checking the service flag before serving the flow. The algorithm is summarized in the pseudocode described in Table 4.1. Just like DRR, miDRR captures relative rate preferences between flows through the quanta assigned to the flows.

To better understand how the algorithm works, we return to our example of two interfaces and two flows, as shown in Figure 3.1(c). Say interface 2 is free and is now moving on to serve flow b. It would find its service flag with flow b to not be set because no other interface is serving flow b, and it would therefore proceed to serve flow b. As prescribed by DRR, the deficit counter of flow b (DC_b) would be incremented by its quantum Q_b, and potentially one or more flow b packets would be sent. When a packet is sent, its length is deducted from the deficit counter. Flow b would be served until its deficit counter is insufficient for the next packet. At that point, interface 2 will move on to flow a. Because interface 1 is serving flow a at the same rate, interface 2 will find its service flag with flow a set by interface 1. Given that the service flag is set, interface 2 will not serve flow a. Instead it would move back to flow b after resetting its service flag for flow a. As this algorithm continues, interface 1 will only serve flow a, and interface 2 will only serve b, yielding the desired result.

The takeaway is that each interface can do its own DRR packet scheduling while checking and communicating through the service flags.

4.4.1 miDRR is max-min fair

We now prove that miDRR leads to a (weighted) max-min fair allocation. The proof consists of showing that miDRR fulfills the rate clustering property (explained in Definition 2) and therefore is max-min fair (by Theorem 4). To understand this, we take the perspective of an interface j running miDRR. Among the flows willing to use interface j, interface j will serve a subset of these flows at the same rate (which is what DRR does for a single interface). The flows served by interface j are in the same cluster.

Say flow a—among the flows willing to use interface j—is not served by interface j. Flow a must have its service flag with interface j set at least once every τ_j, where τ_j is

the time difference between successive visits by the interface j scheduler to schedule flow a. This means flow a is being served as often or more often than a flow served by interface j, which is served once every τ_j. Flow a is therefore in a different cluster that is at a higher rate.

Hence, with miDRR, interfaces serve the flows in their clusters at the same rate, and flows being passed over by interfaces in a cluster are served by another cluster at a higher or equal rate. Therefore, miDRR fulfills the rate clustering property.

We will now formally prove this in Theorem 8.

Theorem 8. *miDRR provides a (weighted) max-min fair allocation.*

We begin by extending the definition of fairness metric proposed in [46].

Definition 3. *Let directional fairness metric from flow i to flow j be*

$$FM_{i\to j}(t_1,t_2) = S_i(t_1,t_2)/\phi_i - S_j(t_1,t_2)/\phi_j,$$

where $S_i(t_1,t_2)$ is the number of bytes sent by flow i in the time interval $(t_1,t_2]$, and ϕ_i is the priority of flow i. Alternatively, we say flow i has received service of $S_i(t_1,t_2)$ in time interval $(t_1,t_2]$.

Consider the system in steady state, i.e., (1) all flows are continuously backlogged in $(t_1,t_2]$, (2) no new flows arrive in this interval, and (3) the rate of the interfaces does not change. The conditions for weighted max-min fairness in Theorem 4 can be rewritten in terms of the directional fairness metric FM as:

1. If flows i,j are actively serviced by a common interface k, the directional fairness metric from i to j and vice versa are zero, i.e.,

$$\exists k, \quad i,j\in\mathcal{U}_k \implies FM_{i\to j} = FM_{j\to i} = 0.$$

2. If flow i is actively serviced by some interface k, while flow j is willing to use interface k but is not actively using it, then the directional fairness metric from j to i is greater or equal to zero, i.e.,

$$\exists k, \quad i\in\mathcal{U}_k, j\in\mathcal{F}_k \implies FM_{j\to i}\geq 0.$$

This means that if a packet scheduler maintains these conditions on FM at all times, it will be max-min fair. We show this is the case by finding upper bounds on FM, which, in turn, tells us the flow service allocations will be fair, amortized over time.

The following Lemma 7 tells us that the value of the deficit counter in miDRR is bounded.

Lemma 7. *At the end of each service turn for a flow, its deficit counter is greater or equal to zero and less that the maximum packet size, i.e.,*

$$0 \leq DC_i < MaxSize'_i,$$

where $MaxSize'_i$ is the maximum packet size of flow i.

Proof. From the algorithm, a flow finishing its service turn can fall into the following cases:

1. Its backlog is cleared; hence, DC_i is reset to zero.
2. Its backlog is not cleared, for DC_i must be less than the maximum packet size. Hence, $DC_i < MaxSize'_i$.

Further DC_i is always greater or equal to zero. Hence, the lemma must be true. □

From the bound on the deficit counter, we can now bound the amount of service received by a flow:

Lemma 8. *Consider a time interval $(t_1, t_2]$ where flow i is continuously backlogged. Let m be the number of service turns it receives in this interval. The service received by flow i can be bounded by*

$$mQ_i - MaxSize'_i < S_i(t_1, t_2) < mQ_i + MaxSize'_i.$$

Proof. Let $DC_i(j)$ be the value of DC_i at the end of the jth service turn, and $S_i(j)$ be the number of bytes sent by the flow in that service turn (i.e., the amount of service it received). In the algorithm, the flow starts with $DC_i(j-1)$, receives Q_i in credit, sends $S_i(j)$ bytes, and is left with $DC_i(j)$. It then follows that

$$S_i(j) = Q_i + DC_i(j-1) - DC_i(j)$$

because the flow is always backlogged in $(t_1, t_2]$.

By summing over the m service turns, we have

$$S_i(t_1, t_2) = \sum_{j=1}^{m} S_i(j) = mQ_i + DC_i(0) - DC_i(m).$$

By substituting the bound presented in Lemma 7,

$$mQ_i - MaxSize'_i < S_i(t_1, t_2) < mQ_i + MaxSize'_i,$$

as per our lemma. □

Lemma 7 bounds the deficit counter (and hence, the amount of service) that a flow can carry over from one service turn to another (as seen in Lemma 8). This ensures that a flow cannot accumulate unfair service from one turn to another, which lays the foundation for the following two lemmas.

Lemma 9. *Consider the pair of flows i, j where flow i is serviced at higher rate than flow j and flow j is serviced by interface k. It can be shown that*

$$FM_{i \to j} > -2MaxSize',$$

where $MaxSize'$ is length of the maximum sized packet.

Proof. Consider a service turn on interface k where flow j is serviced by the interface but not flow i. It means flow i has been served once or more between the time it is considered by interface k, i.e., $SF_{ik} = 1$. Hence, the number of service turns that flows i, j receive in $(t_1, t_2]$ is related by

$$m_i(t_1, t_2) \geq m_j(t_1, t_2).$$

Using Lemma 8, we know that

$$\begin{aligned} FM_{i \to j} &= S_i(t_1, t_2)/\phi_i - S_j(t_1, t_2)/\phi_j \\ &> \left(m_i(t_1, t_2)Q_i + MaxSize'_i\right)/\phi_i \\ &\quad - \left(m_j(t_1, t_2)Q_j + MaxSize'_j\right)/\phi_j \\ &> \left(m_i(t_1, t_2) - m_j(t_1, t_2)\right)Q' \\ &\quad - MaxSize'_i - MaxSize'_j \\ &> -2MaxSize', \end{aligned}$$

where $Q' = Q_i/\phi_i$ and we can choose $\min_i \phi_i \geq 1$ without loss of generality. □

Lemma 10. *Consider the pair of flows i, j where both flows are always serviced by interface k. It can be shown that*

$$|FM_{i\to j}| < Q' + 2MaxSize',$$

where $MaxSize'$ is the length of the maximum size packet, and $Q' = Q_i/\phi_i$.

Proof. Consider a service turn of flow j. From the algorithm, $SF_{ik} = 0$ for flow i to be also serviced. Hence,

$$|m_i(t_1, t_2) - m_j(t_1, t_2)| = 1.$$

From Lemma 8,

$$S_i(t_1, t_2)/\phi_i < m_i(t_1, t_2)Q' + MaxSize'_i/\phi_i$$

and

$$S_j(t_1, t_2)/\phi_i < m_j(t_1, t_2)Q' + MaxSize'_j/\phi_j.$$

This means

$$\begin{aligned}
|FM_{i\to j}| = |FM_{j\to i}| & \\
= \; & |S_i(t_1, t_2)/\phi_i - S_j(t_1, t_2)/\phi_i| \\
< \; & |m_i(t_1, t_2) - m_j(t_1, t_2)|Q' \\
& + MaxSize'_i/\phi_i + MaxSize'_j/\phi_j \\
< \; & Q' + 2MaxSize'
\end{aligned}$$

because we can choose $\min_i \phi_i \geq 1$ without loss of generality. □

Proof of Theorem 8. We are now ready to prove Theorem 8—that miDRR gives a max-min fair allocation. Lemma 9 shows that the service lag of a faster flow compared to a slow flow is strictly bounded by two maximum size packets. Lemma 10 shows that the service difference between two flows that are supposed to be served at the same rate is strictly bounded by Q' plus two maximum size packets. Both lemmas prevent accumulation of unfairness over time, showing us that miDRR indeed provides fair queueing for multiple interfaces in the presence of interface preferences. □

In summary, our observation that the earliest finishing packet cannot be causally determined in a multi-interface setting due to a lack of fate-sharing led to the notion that we must track the conditions under which flows experience different relative service rates. The formation of clusters leads to sufficient conditions for max-min fair packet scheduling that describe exactly when flows experience the same or unequal rates. miDRR achieves a max-min fair allocation by serving flows at rates that adhere to the conditions in Theorem 4 by means of the service flag.

Why is a 1-bit service flag sufficient?

While Theorem 4 formalizes the notion that the rate clustering property serves as sufficient conditions for max-min fair packet scheduling, it is worth understanding intuitively why this is so. It all boils down to the fact that the service flag implicitly captures the relative service rate between a pair of flows.

At an intuitive level, consider that flow a served only by interface j. If Q_a is the DRR quantum of flow a in bits, and τ_j is the time between successive visits by the interface j scheduler to schedule flow a, then flow a is served at rate Q_a/τ_j. Assume for a moment that all of the flows using interface j have the same weight (i.e. the same ϕ values), then all of them have the same service rate. Now, consider a second flow b that is willing to use interface j but is not actively being scheduled by j because it is receiving enough service elsewhere. This is accomplished if flow b has its service flag SC_{bj} set at least once every τ_j so that interface j will skip serving it every time. This will happen if (and only if) flow b is already being served as often or more often than τ_j by other interfaces. This means it needs no service at interface j.

4.4.2 Implementation

Can we use miDRR to implement user preferences in a practical mobile device? Specifically, we want to use miDRR to schedule both incoming and outgoing packets over the multiple interfaces that modern mobile devices have.

It is relatively simple to implement miDRR to schedule outgoing packets from a mobile device. Our current implementation uses a custom Linux kernel bridge, which is written using 1,010 lines of C code. Our implementation, in Linux 3.0.0-17, follows the architecture illustrated in Figure 4.2. This custom bridge allows us to steer individual packets to whichever interface the scheduling algorithm chooses while being transparent for the

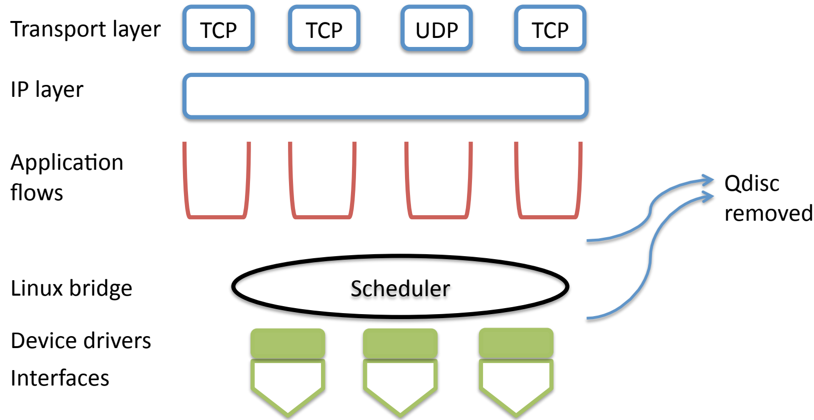


Figure 4.2: Implementation of miDRR in Linux kernel to schedule outbound packets.

applications. This is done by presenting the applications with a virtual interface with an arbitrarily chosen address and then rewriting the packet headers appropriately before transmission. We find that the overhead of this bridge and packet header rewriting operations are negligible when compared to the relatively moderate speeds found in wireless interfaces.

The twin problem of scheduling *incoming* packets over multiple interfaces is harder. The ideal implementation would be a proxy in the Internet that collects all of the flows headed toward a mobile device at a location close to the last-mile wireless connections used by the device, as in Figure 4.3. This can be accomplished by a service provider such as AT&T if the device is connected to the provider's HSPA, LTE, and WiFi networks at the same time. The scheduler at this proxy could then use miDRR to schedule all of the flows on the different paths that lead to the different interfaces on the mobile device while respecting preferences. Clearly, this has deployability and performance challenges because it will require us to aggregate all traffic at one point before scheduling it.

In this work, to ease deployment concerns, we use an HTTP-based scheduling technique that can be implemented on the mobile device itself and that still allows us to come close to ideal packet scheduling for incoming packets. Specifically, we implemented miDRR to schedule inbound HTTP traffic in an HTTP proxy, as depicted in Figure 4.4. Our HTTP proxy is written in 512 lines of Python code. Because the majority of packets to and from mobile devices are HTTP transfers [19], we feel that this is a reasonable compromise. Furthermore, such a fully in-client HTTP scheduler is easily introduced into today's mobile device.

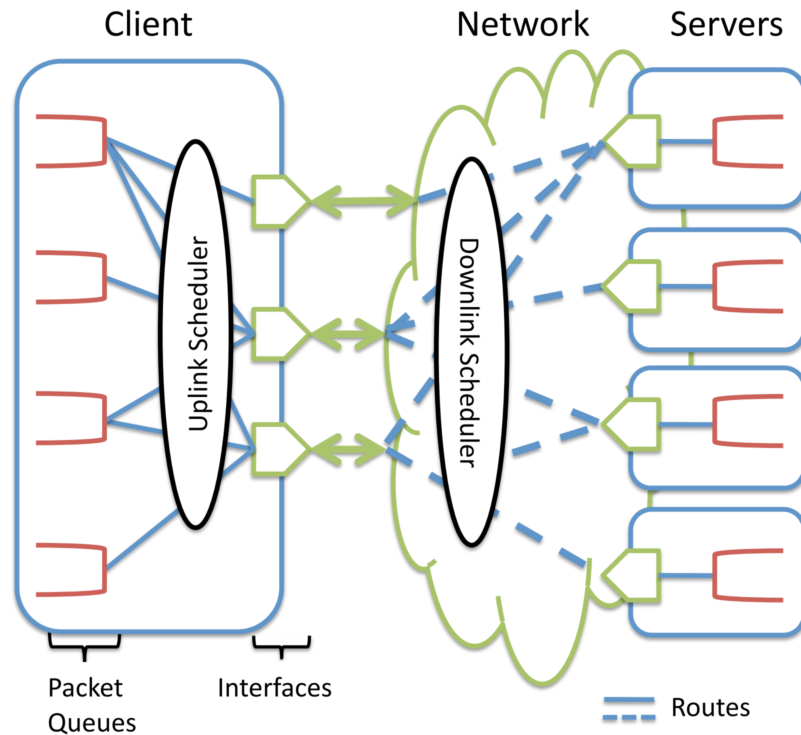


Figure 4.3: Ideal implementation of miDRR to schedule both inbound and outbound packets.

In this implementation, we make use of the byte-range option available in HTTP 1.1 to divide a single GET request into multiple requests and to decide an interface on which to send each request. This allows us to divide an inbound transfer into multiple parts, each of which can arrive over different interfaces at the same time. The responses are then collected, spliced together, and returned to the application. By choosing the interface on which a request is placed, we also select the interface over which the corresponding inbound data will arrive. When combined with request pipelining, we can always have some pending requests on each interface, making sure that all of the available capacity is utilized. To provide fairness between the HTTP flows, we implemented miDRR to regulate the inbound HTTP traffic. For example, the proxy can choose to send flow a's requests while withholding flow b's to make sure that each flow gets its fair share of the bandwidth.

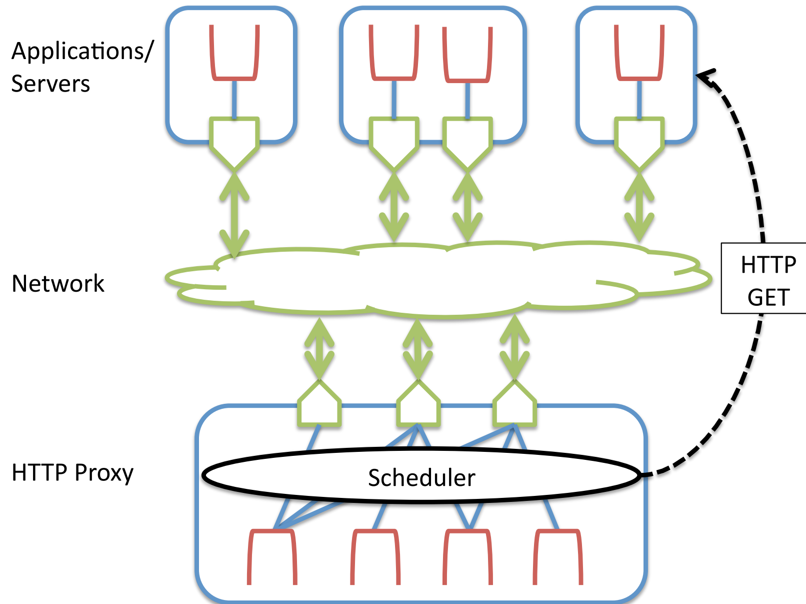


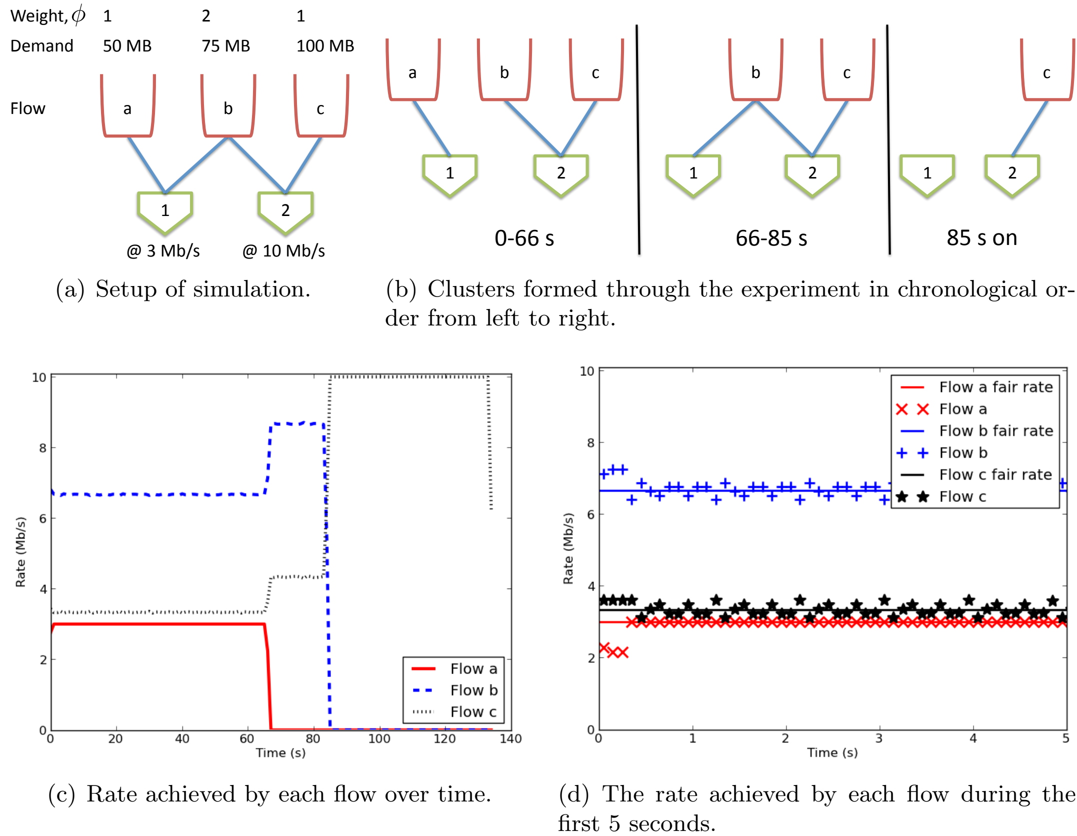
Figure 4.4: Implementation of miDRR in a HTTP proxy to schedule inbound HTTP flows.

4.4.3 Evaluation

We evaluate miDRR using our prototype implementation on a Dell laptop running Ubuntu 10.04 LTS with an Intel Core Duo CPU P8400 at 2.26 GHz processor and 2 GB RAM. We aim to evaluate whether miDRR can provide a max-min fair allocation on both the uplink and the downlink with varying network conditions and traffic workloads. In our experiments, we use between two and 16 WiFi interfaces to test miDRR. The interface NICs are either the inbuilt Intel PRO/Wireless 5100 AGN WiFi chip or an Atheros AR5001X+ wireless network adapter connected via PCMCIA. We use the Atheros `ath5k` driver for the Atheros wireless adapter.

Fair packet scheduling with miDRR

As shown in Theorems 9 and 10, miDRR provides weighted max-min fair queueing, but it can deviate from an ideal bit-by-bit max-min fair scheduler. To test how far it can deviate, we check the performance of miDRR in a simulation that allows us to avoid complications such as time-varying network conditions. We run the simulation using an example of three flows served by two interfaces, as illustrated in Figure 4.5(a).



(a) Setup of simulation.

(b) Clusters formed through the experiment in chronological order from left to right.

(c) Rate achieved by each flow over time.

(d) The rate achieved by each flow during the first 5 seconds.

Figure 4.5: Simulation results for three flows over two interfaces.

The result is shown in Figure 4.5(c). As we can see, flow a achieved a rate of 3 Mb/s using interface 1, while flows b and c shared the 10 Mb/s on interface 2 at a ratio of their weights 2:1. This is the weighted fair allocation we expect. When flow a completed after 66 s, flow b immediately increased its rate to 8.67 Mb/s using interfaces 1 and 2 simultaneously, i.e., aggregating bandwidth across both interfaces. Similarly, flow c's rate increased to 4.33 Mb/s, which further increased to 10 Mb/s when flow b completed after 85 s. This shows that miDRR does indeed provide weighted fair queueing.

Throughout the experiment, the rate clustering property was upheld. We illustrate the clusters formed in Figure 4.5(b). As load changes (flows a and b finish), clusters change as well. Zooming at the first phase (0–66 s), flow a's cluster has rate 3 Mb/s and flow c's cluster a rate of 3.33 Mb/s. Flow b gets twice this rate (6.66 Mb/s) because its weight is twice that of flow c in the same cluster.

However like many practical approximations, miDRR is imperfect. Figure 4.5(d) zooms in on the first 5 s. We can see that flow a initially only receives about 2 Mb/s (instead of 3 Mb/s) while interface 1 serves flow b. However the algorithm quickly corrects this, and a weighted fair rate is achieved. Also, the rate achieved by flows a and b fluctuates around the ideal fair rate due to the atomic nature of packets and the size of the quanta used.

Overhead of miDRR

We designed miDRR to be lightweight and simple, with each interface performing packet scheduling in a fairly independent manner with minimum communication overhead. However, what exactly is the overhead of miDRR in our implementation? To answer this question, we profiled our Linux kernel bridge—measuring the time it takes to make a scheduling decision. For each experiment, we present the bridge with 1,000 packets spread and queued across all of the flows and record the time it takes to make a scheduling decision for each packet.

Figure 4.6 shows that the scheduling time is independent of the number of flows because the algorithm does not need to go through every flow to make a scheduling decision. The decision is made as soon as the scheduler finds the next flow it should serve. However, the scheduling overhead increases with the number of flows an interface has to consider before finding one that it should serve. This number is proportional to the number of flows that have their service flags set, which is correlated with the number of interfaces. With more interfaces in the system, the chances of finding a flow with its service flags set would increase. This extra search time is shown in Figure 4.7.

Even with 16 network interfaces, miDRR can make a scheduling decision in less than 2.5 μs. Put differently, the algorithm can support a traffic rate more than 3 Gb/s for 1,000-byte packets. Therefore, we believe that it is quite feasible to implement miDRR in practice, even for a high-speed packet scheduler in the kernel.

HTTP Fair Scheduling

The HTTP-proxy based implementation of miDRR on the downlink cannot support fine-grained packet scheduling, but it does allow us to evaluate the algorithm over an operational network. We check if even under such scenarios our scheduler can provide fairness. We evaluate this by serving three HTTP flows over two interfaces. The setup is similar to

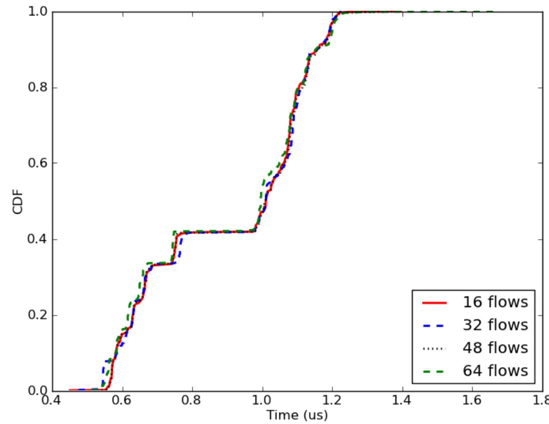


Figure 4.6: CDF of scheduling time as a function of the number of flows for four interfaces.

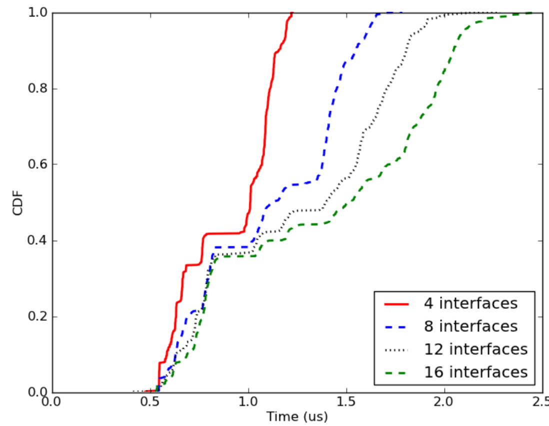


Figure 4.7: CDF of scheduling time as a function of the number of interfaces for 32 flows.

Figure 4.5(a) except that all flows have the same weight, and the interface speed varies as the experiment runs.

We plot the goodput achieved by each flow over time in Figure 4.8. In this setup, flows a and c will achieve whatever interfaces 1 and 2, respectively, can provide at the current speed of the interfaces. However, flow b is willing to use both interfaces, so it should always achieve the same rate as the faster flow. This is the correct max-min fair allocation. We indeed observe this behavior in Figure 4.8, i.e., the rate of flow b always tracks the faster flow. We are observing the rate clustering property because flow b will share the faster interface equally with the faster flow, forming a cluster with it, as illustrated in Figure 4.9.

Despite having only very coarse-grained control over the inbound traffic, we find surprisingly that the HTTP scheduler is able to provide fair scheduling over multiple interfaces

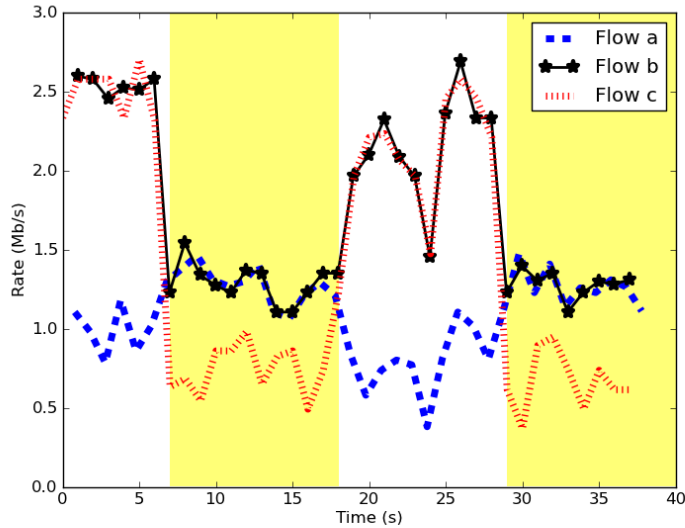


Figure 4.8: TCP goodput of three inbound HTTP flows scheduled fairly using our HTTP proxy.

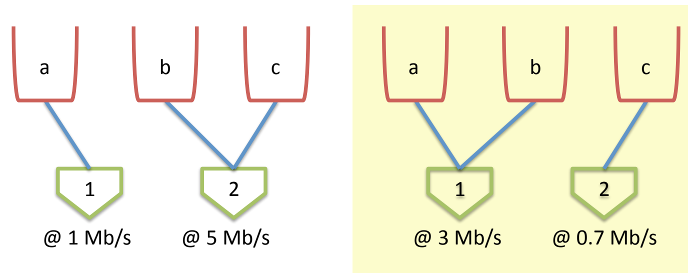


Figure 4.9: Clustering formed when our HTTP proxy schedules fairly across multiple interfaces. On the left is the clustering during the 11–18 s of the experiment and 29 s on. On the right is the clustering during 0–11 s and 18–29 s.

while conforming to interface preferences. This performance holds even while the scheduler is reacting to fluctuating link capacities. Given that a large fraction of the traffic on mobile devices is HTTP, this suggests that an HTTP layer scheduler is sufficient to build a full system that allows users to leverage all of their interfaces while respecting preferences.

4.5 Summary

The desire to use several network interfaces at a time on our mobile devices led us to the problem of packet scheduling with interface and rate preferences. The introduction of interface preferences gives the classical problem of packet scheduling a new twist. Not only

do interface preferences render prior algorithms unusable but also it changes the way we think about fairness in rate and in service.

Hence, it is crucial that we understand the implications of such preferences and how they change the solution space. This chapter presents a simple and efficient algorithm (miDRR) and empirical measurements of the algorithm running in practice. By achieving max-min fair allocation, we know that miDRR fulfills the following properties: (1) meets interface preferences, (2) is Pareto efficient, (3) meets rate preferences, where possible, and (4) uses new capacity.

We expect this understanding, in general, and our algorithm, in particular, to be useful in many applications beyond the mobile application we described. Allocating tasks to machines in a data center poses a similar scheduling problem, where certain tasks might prefer to use only more powerful machines. We could also use the algorithm to assign compute tasks to CPU cores in a system such as NVIDIA Tegra 3 4-plus-1 architecture, where 4 powerful cores are packaged with a less powerful one. A computation-intensive task such as graphics rendering might prefer to use only the more powerful cores. As we continue to build large systems by pooling smaller systems together, we expect an increasing number of situations where our results will prove useful.

Chapter 5

OpenFlow Wireless: Programmable Open Wireless Network Architecture

As users make use of all of the networks around them, it is imperative that the network infrastructure makes that easier, both by design and by policy. This approach does not just benefit the users. It also presents major advantages to the network operators.
In this chapter, I explore the design of a network to support mobile clients making use of multiple networks at the same time. My blueprint for such a network—OpenFlow Wireless—decouples the network architecture from its underlying wireless technologies, and virtualizes the physical infrastructure though "slicing." Further, OpenFlow Wireless provides direct support to the applications. To validate this design, I deployed and operated a test network at Stanford, which provided us with anecdotal evidence that such a programmable open wireless network architecture is indeed viable and desirable.

5.1 Problem Statement

If we really want to let users *make use of the networks around us*, why do we not make it easier—in design and in policy—for a mobile client to move freely between spectrum and

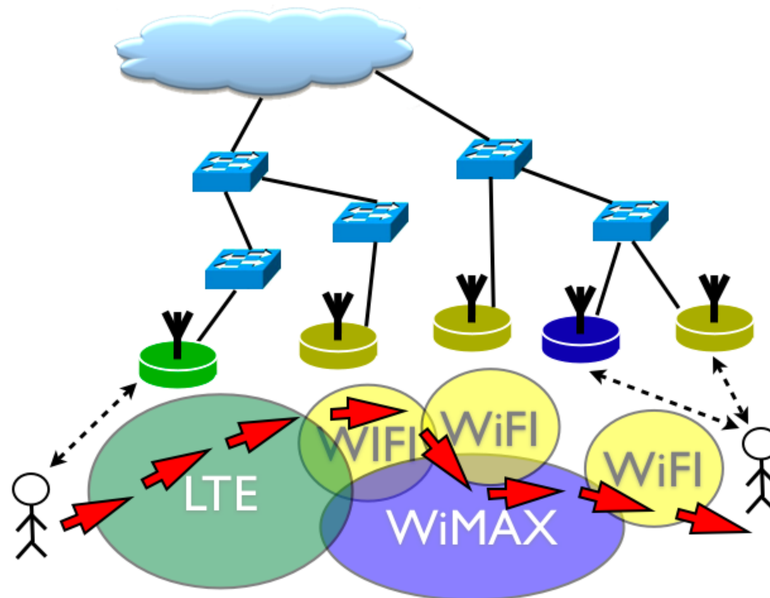


Figure 5.1: Vision of future mobile network where the user can move freely between technologies, networks and providers.

networks owned by different cellular and WiFi providers, as shown in Figure 5.1? While this approach is clearly counter to current business practices and would require cellular providers to exchange access to their networks more freely than they do today, we believe it is worth exploring because of the much greater efficiency and capacity it could bring to end users. Interestingly, a several-fold increase in capacity could be made available for little to no additional infrastructure cost.

If done right, this presents major advantages for the network operators:

Increased capacity through more efficient statistical sharing. Cellular network operators tend to heavily overprovision their networks in order to handle peak load and congestion. Most of the time, the network is lightly loaded. If, instead, they were able to hand off traffic to one another or move it from cellular to WiFi networks, then their traffic loads would be smoother and their networks more efficient. For example, what if AT&T could re-route traffic from its iPhone users to T-Mobile during system overload? Or what if T-Mobile could re-route its customers flows to a nearby WiFi hotspot?

Exploit differences in technologies and frequency bands. Mobile technologies such as EVDO and HSPA provide wide area coverage with consistent bandwidth guarantees, while technologies such as WiFi provide high bandwidth and low latency. Lower frequencies provides better coverage and penetration; higher frequencies provides better spatial reuse. Being able to use the most appropriate technology for the application at hand would make best use of available capacity. For example, a backup where intermittent connectivity is tolerable can be done via WiFi, where higher throughput is possible.

Open up new sources of capacity. The ability to move between networks also open up new sources of capacity. For example, one can now use a network such as that of fon.com to supplement one's main network, without having to deploy an extensive WiFi network. Such crowd-sourcing can be a powerful tool to cover dead spots and relieve congestion.

To support and achieve this vision, this work outlines a programmable network that supports heterogeneous wireless technologies, allowing us to "stitch together" a multitude of wireless networks available today. Not only does this network supports a user making use of multiple networks at the same time, but it also allows the operators to continually innovate and provide better services to users.

As our cellular networks transition to IP, this is an opportune time to change the way our wireless networks are organized. IP has been tremendously successful in bringing choices and innovation to the end user. Arguably, its greatest feat has been enabling innovation at the edges. IP is simple, standardized, and provides universal connectivity. However, as-is, IP is not the right choice for the future mobile Internet. It is ill-suited to support mobility and security, and it is hard to manage. Its architecture is fixed, allowing little room to add new capabilities. Today, cellular providers feel the pain from poor support for mobility, security, and innovations in general. If we tweak IP to solve these problems, we will find new limitations. We need a network that allows continued innovation for services we cannot yet imagine while permitting existing applications to operate unchanged.

In this chapter, I present the OpenFlow Wireless (or OpenRoads) network architecture—a blueprint for an open programmable mobile network. The goals of OpenFlow Wireless are as follows:

1. The architecture should decouple the overarching network architecture from the underlying wireless technology. This allows for new wireless technologies to be readily integrated and deployed. This also allows the same backhaul network to be used for multiple "networks," which potentially could lead to a reduction in capital and operating costs for the operators.

2. The network should allow different service providers to share a common physical infrastructure. If users are to move freely among many networks, the service providers need to be separate from the network owner. Service providers should handle the mobility, authentication, and billing for their users, regardless of the networks to which they are connected.

3. The architecture should allow network operators to continually innovate. Operators should be able to safely and incrementally roll out new services to customers, instead of relying on a standards-driven process moving at a glacial pace. This means we should be able to readily extend the network to support users, mobile devices, and mobile applications.

OpenFlow Wireless not only allows users to make use of multiple networks at the same time but also provides a mobile wireless network platform that enables experimental research and realistic deployments of networks and services. The research community has a big part to play in bringing this new open architecture to fruition. Much like operators trying out new features in their operational networks, OpenFlow Wireless allows researchers to research and deploy their experimental services with the production networks of their campuses, providing a platform to realistically evaluate research ideas for mobile services.

In this chapter, I will describe OpenFlow Wireless and present how it achieves its stated goals. I then discuss an actual deployment of OpenFlow Wireless in the School of Engineering at Stanford University, and the lessons learned. Finally, I will present selected evaluations and demonstrations.

5.2 Related Work

OpenFlow Wireless is based on the ideas of OpenFlow [30]; hence, it shares many of the architectural ideas proposed by OpenFlow and its predecessor Ethane [13].

OpenFlow is a feature added to switches and routers, allowing these datapath devices to be controlled through an external, standardized API. OpenFlow exploits the fact that almost all datapath devices already contain a flow-table (originally put there to hold firewall ACLs), although current switches and routers do not have a common external interface. In OpenFlow Wireless, OpenFlow is added to WiFi access points (APs) and WiMAX base-stations as well by modifying their software, and in principle, the same thing could be done for LTE and other cellular technologies.

In OpenFlow—and therefore in OpenFlow Wireless—the network datapath is controlled by one or more remote controllers that run on a PC. The controller manages the flow-table in all of the datapath elements and decides how packets are routed. In this manner, the datapath and its control are separated, and the controller has complete control over the datapath operations. The controller can define the granularity of a flow. For example, a flow can consist of a single TCP session or any combination of packet headers (Layer 1-4) that allows for aggregation.

5.3 The OpenFlow Wireless Network Architecture

Figure 5.2 provides an overview of OpenFlow Wireless architecture. At the high level, OpenFlow Wireless uses (1) OpenFlow to separate control from the datapath through an open API; (2) FlowVisor [45] to create network slices and isolate them, and (3) SNMPVisor to mediate device configuration access among services or experiments. These components virtualize the underlying infrastructure directly relate to my vision for future wireless Internet design in terms of decoupling mobility from physical networks (OpenFlow), and allowing multiple service providers to concurrently control (FlowVisor) and configure (SNMPVisor) the underlying infrastructure.

For this network, I used the freely available open-source controller NOX [24], but any controller is possible as long as it speaks the OpenFlow protocol. NOX provides network-wide visibility of the current topology, link-state, flow-state, and all other network events. As a network OS, NOX hosts applications or plug-ins that can observe and control the network's state—for example, to implement a new routing protocol, or in this case, to implement new mobility managers. The mobility manager can choose to be made aware of every new application flow in the network and can pick the route each takes. When the user moves, the mobility manager is notified, and can decide to re-route the flow. Because

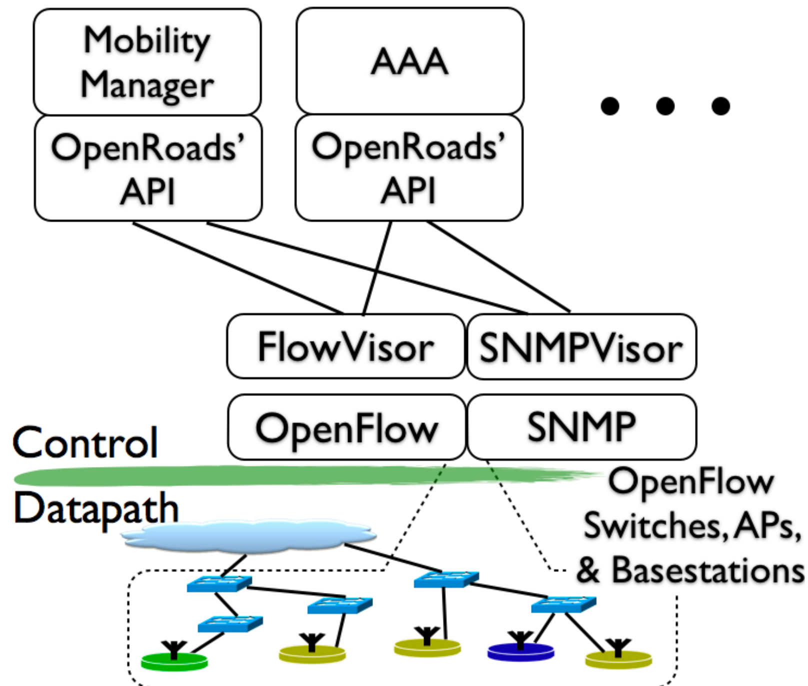


Figure 5.2: The OpenFlow Wireless architecture where control is separate from the physical infrastructure. The control is then "sliced" using FlowVisor and SNMPVisor to provide fine-grained control to the services above.

OpenFlow is independent of the physical layer (i.e., it does not matter whether the wireless termination point is running WiFi or WiMAX), vertical handoff between different radio networks is transparent and simple.

The openness of the controller makes it easy to add or change the functionality of the network. For example, a researcher can create a new mobility manager (e.g., one that provides faster or lossless handoff) by simply modifying an existing one. In our prototype deployment (discussed later in this chapter), this happened many times, as researchers and students exchanged code and built on one another's work. In this way, rapid innovation is possible. Further, by separating the datapath and its control, OpenFlow Wireless reaps the many benefits of centralized control. Anecdotally, network administrators are receptive to a centrally managed network that is easily monitored.

Taken to the extreme, an application could be an entire mobility service, akin to the cellular services we buy from companies like AT&T, Vodafone, and Orange. An application can be written to implement AAA, billing, routing, directory services, and so on—all

running as programs on a controller. And because the controller itself is simply a program running on a server, it can be placed anywhere in the network—even in a remote data center.

5.3.1 Supporting Radio Agnosticism

Many handover mechanisms today are specific to wireless technologies. For example, WiMAX forum recommends how handover could be achieved in a mobile WiMAX network where GRE tunneling is commonly employed. These mechanisms often make assumptions about specific wireless technologies that are not directly applicable to other wireless technologies. A key feature of OpenFlow Wireless is its radio agnosticism, i.e., its ability to connect to a mobile device through any wireless technology. This allows for mobility across networks that use a multitude of wireless technologies, e.g., to accomplish handover from WiFi to WiMAX and vice versa.

To reconcile the differences among these networks, we reduce handover in OpenFlow Wireless to the lowest common denominator for popular wireless technologies, i.e., re-routing flows. Advocating flow-based management to the mobile industry is preaching to the choir. The concept of managing the network at the flow or terminal granularity is well-established. However, Ethernet-IP based networks tend to manage with granularity of packets. To introduce the idea of flows to these networks, we exploit OpenFlow. OpenFlow brings the concept of a flow to switches, routers, and WiFi APs, which can then manage packets identified to be a flow from headers spanning from Ethernet addresses to TCP/UDP ports. This allows a flexible definition of flows, which in turn provides a powerful way to manage the network.

While not a requirement for radio agnostic handover, OpenFlow Wireless advocates the use of simple dumb base-stations for other wireless technologies, akin to what LWAPP advocated for WiFi. This allows an uniform way to accomplish handover from one wireless technology to another, reducing the influence of wireless technologies on the design of the backhaul and bringing us closer to a network that is radio agnostic.

5.3.2 Slicing the Network

Although we have explained how we can run a new experimental service in the OpenFlow Wireless network, the question of how we can have multiple competing services running

at the same time in the same network remains. How could one service allow its users to roam freely across multiple physical networks? The trick here is to slice, or virtualize the network, allowing multiple controllers to co-exist, each controlling a different slice of the network. A slice may consist of one user or many users, one network or many networks, one subset of traffic or all traffic. OpenFlow Wireless uses the FlowVisor, an open-source application created specifically to slice OpenFlow networks.

FlowVisor slices a network by delegating control of different flows to different controllers. As shown in Figure 5.2, FlowVisor is an additional layer added between the datapath and controllers. Because the FlowVisor speaks the OpenFlow protocol to the datapaths, the datapaths believe they are controlled by a single controller (the FlowVisor), and because the FlowVisor speaks OpenFlow to the controllers, the controllers think they each control their own private network of switches (meaning a virtual network). In other words, FlowVisor is a transparent proxy for OpenFlow. The trick is to correctly isolate the flows according to a policy, and hence create one slice with its own private "flowspace" (a range of header values) per experiment. FlowVisor works by deciding which OpenFlow messages belong to each slice and passing them to the controller for that slice. If, for example, Controller A is responsible for all of Alice's traffic, then FlowVisor passes all control messages relevant to Alice to Controller A. Therefore, FlowVisor separates slices according to a policy, defined by the network manager, by enforcing strict communication isolation between slices.

A direct consequence of slicing the network is that slicing/virtualization allows "versioning" in the production network, meaning new features can gradually be incorporated into production. Different slices can be dedicated to different versions, some more stable than others, as new features are carefully rolled out in stages. In this way, new features can be deployed and tested quickly, then gradually made available network-wide, or even shared among network operators. Such an ecosystem allows for the survival of the fittest, bringing the best to users. Also, legacy clients can be supported on a separate legacy slice, and the network can now evolve without being held back by backward compatibility.

Slicing also allows delegation. Network administrators can cascade FlowVisors to further delegate (or slice) the flow space allocated to them. Repeated delegation makes sense in networks with a hierarchy of control; for example, in a campus network, the network manager delegates a slice of the network to the individual building network administrators, and that can in turn be sliced (using another FlowVisor) to provide new slices for researchers.

Such delegation means researchers can safely run experiments in a production network. Be default, FlowVisor allocates flowspace to the production network, which can be routed using legacy protocols. Each experiment is assigned its own slice, defined by the flowspace and topology, and implemented with the FlowVisor. Because real users are already connected to the production network, this process makes opt-ins relatively simple. If the network is sufficiently large, then experiments can be run at the same scale as, say, a campus wireless network. They could even be run over multiple networks on multiple campuses.

While OpenFlow provides a means to control the OpenFlow Wireless datapath, it does not provide a way to configure the datapath elements: e.g., setting power levels, allocating channels, enabling and disabling interfaces. This job is normally left to a command line interface, SNMP or NetConf. Although simple in principle, configuration is tricky in a sliced network, as we want to configure each slice independently. For example, we might wish to disable a certain network interface in one slice, without disabling the same physical interface that is shared by another slice. OpenFlow Wireless slice datapath configuration using "SNMPVisor," which runs alongside the FlowVisor, to allow an experimenter to configure his or her individual slice. FlowVisor slices the datapath, and SNMPVisor slices the configuration by watching SNMP control messages, and sending them to the correct datapath elements (and possibly modifying them). Similar to FlowVisor, SNMPVisor acts as a transparent SNMP proxy between the datapaths and controllers, providing the same features of versioning and delegation.

Sometimes it is difficult to slice the configuration, if not impossible. For example, in setting power levels for different slices on a WiFi AP, if slices share a channel, then different transmission power levels should be set for the flows in each slice—something that is not possible with existing APs. We follow the general mantra of slicing where we can and exposing non-sliceable configuration parameters to users via feedback and error messages.

5.3.3 Software Friendly Network

With slicing, OpenFlow allows "versioning," which ultimately provides operators the opportunity to continually innovate. This allows innovations in operational services in a mobile network to be decoupled from the glacial standardization process. Operators can differentiate themselves by extending support to users, mobile devices, and mobile applications. Many applications would indeed benefits from a more direct interaction with the network.

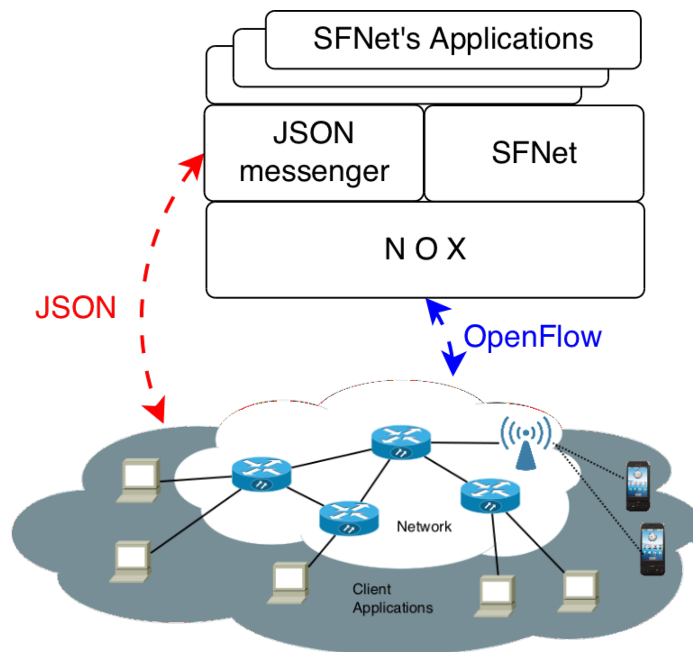


Figure 5.3: Building a software-friendly network on top of OpenFlow Wireless by allowing the applications to talk directly to the controller via plugins.

Today, there is usually a clean separation between networks and the applications that use them. Applications send packets over a simple socket API; the network delivers them. Part of the success of the Internet undoubtedly comes from this simple and consistent interface between applications and the network.

However, many applications can benefit from a richer interface to the network with more visibility of its state, and more control over its behavior. Past efforts to increase the richness of the APIs, such as RSVP [10] and Active Networking [49], have not been very successful. OpenFlow Wireless—which has a software defined control plane—presents a new opportunity to provide such interaction between applications and networks, i.e., to move towards a more "software-friendly" network.

To explore a possible path, a plugin for an OpenFlow Wireless control plane was created to allow applications to query the network state and issue network service requests directly. This plugin—called SFNet—is illustrated in Figure 5.3. This proposal is distinctive in that it allows applications to communicate with the network directly.[1] The key role

[1] This work focuses on how an application communicate with the network, and does not discuss how networks along a route can coordinate among one another to fulfill a request.

of a software-friendly network is to bridge the semantic gap between applications and the network. While exposing network services to application writers can potentially improve application performance, the low-level operations required, such as route calculation or discovering network topology, are forbidding for most programmers. By presenting high-level APIs to the program and hiding the implementation details, OpenFlow Wireless reduces the barrier to entry and increases the uptake of network services. This leads to three possible scenarios:

1. One scenario is for every application to provide its own “plugin” to the network OS to view and control the network, and to also define its own application-specific communication protocol to the plugin. For example, a plugin optimized for Skype might interface directly with the network OS to set up paths, reserve bandwidth, and create access control rules.

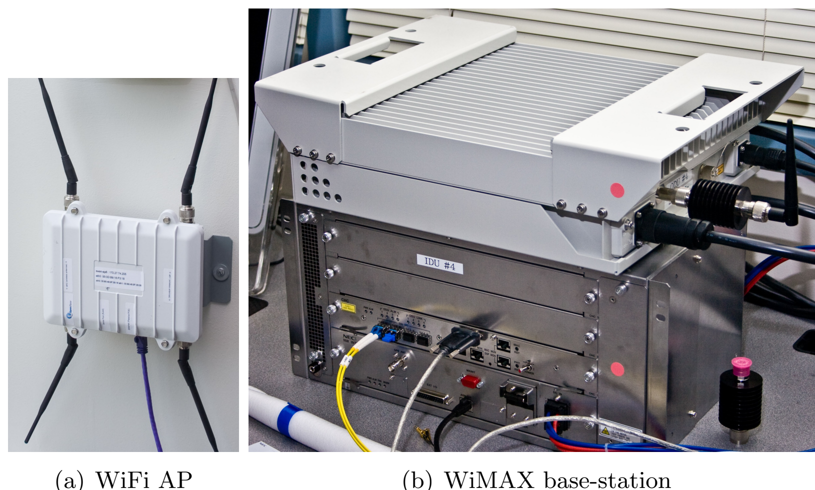
2. Alternatively, over time, a relatively small number of “de facto standard” plugins might emerge for common tasks (e.g., a plugin for multicast, another for multipath routing, and yet another for bandwidth reservations).

3. A third scenario is where plugins emerge to suit certain classes of applications (e.g., a plugin for chat applications, another for real-time video, and a third for low- latency applications).

Of course, all three models can co-exist. Many applications may choose to use common feature plugins, whereas others can create their own. OpenFlow Wireless does not propose or mandate any particular model, it merely makes all three possible so each application can choose its own path. The “winning features” are picked by adoption, rather than by standards bodies.

5.4 Stanford Deployment of OpenFlow Wireless

OpenFlow Wireless was deployed in the Stanford’s School of Engineering to help us understand what would be needed to build and deploy such a network. This deployment uses more than 10 1-GB OpenFlow Ethernet switches, more than 90 WiFi APs, and two NEC WiMAX base-stations. This includes switches from NEC (IP8800) and HP (ProCurve 5406ZL); both are OpenFlow-enabled through a prototype firmware upgrade. The WiMAX



(a) WiFi AP (b) WiMAX base-station

Figure 5.4: Photographs of a WiFi AP and WiMAX base-station used in the Stanford deployment of OpenFlow Wireless.

base-station was built by NEC and runs a firmware jointly developed with Rutgers University. One base-station is also deployed on the roof of the Packard building, operating at 5 W of power and using 6 MHz of spectrum provided by Clearwire. The WiFi APs are based on the ALIX PCEngine boxes with dual 802.11g interfaces. The APs run the Linux-based software reference switch and later Open vSwitch, and are powered by passive Power-over-Ethernet to reduce the cabling needed. Figure 5.4 shows photographs of the WiFi APs and WiMAX base-stations deployed. Figure 5.5 shows the location of these APs throughout the Gates Computer Science Building.

For this deployment, we wanted to allow an end-user to opt-in to (one or more) experiments. This can be done by assigning a different SSID to each experiment, which requires each AP to support multiple SSIDs. An experiment runs inside its own "slice" of resources, a combination of multiple SSIDs and virtual interfaces. When a slice is created, a virtual WiFi interface is created on all of the APs in the slice's topology, and assigned a unique SSID. Since each experiment can be assigned a distinct SSID, users may agree to opt-in to an experiment simply by choosing an SSID. Using virtual interfaces is easy on these APs because they run Linux. Although more expensive than the cheapest commodity APs, they still cost less than a typical enterprise AP. The same idea could be applied to a low-cost AP running Openwrt. Using a separate SSID for each experiment also means that each SSID can use different encryption and authentication settings. However, all of the virtual

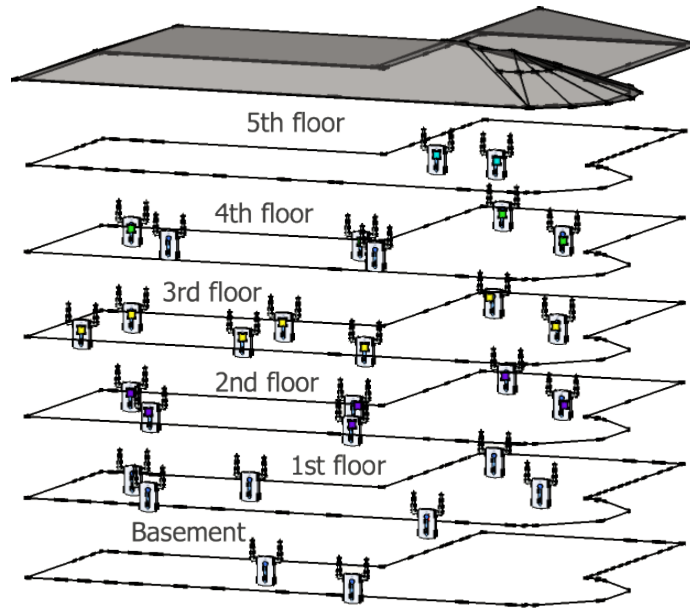


Figure 5.5: Location of 30 WiFi APs deployed in Stanford's Gates building as part of an OpenFlow Wireless deployment.

interfaces are limited to using the same wireless channel and power settings. Each SSID (i.e., slice) is part of a different experiment and therefore attached to a different controller created by the experimenter. FlowVisor is responsible for connecting each slice to its own controller.

To aid the deployment, several tools for monitoring and visualization have also been developed. These tools are detailed in [60]. Over the last four years, this deployment has been expanded and operated as a production network for many students and faculty in the computer science department. The network has also been used as our guest network many times over the years. Further, OpenFlow Wireless has also been deployed in several homes [64]. All these provide strong anecdotal evidence that the proposed OpenFlow Wireless architecture is viable for actual deployment and production use.

Also, to help with the exploration of software-friendly network designs, a prototype called SFNet was created on top of OpenFlow Wireless. SFNet allows applications to directly interact with the network using a high-level API. By exploiting the global view provided by NOX, SFNet easily supports high-level primitives, such as network status requests and resource reservations. Data exchanges between applications and SFNet are represented

in JSON (JavaScript Object Notation), which is a simple and concise data format supported by most modern programming languages.

As an example, let us describe how an application can discover the location of SFNet's controller, pre-requisite to using SFNet itself. The application first sends a discovery request using a UDP packet addressed to a predefined IP address and port (e.g., 224.0.0.3:2209 in this case), and the response is returned directly using another JSON message. This avoids any broadcasting in the discovery process. Using the response, the application can set up a TCP socket with the controller, which forms the communication channel for subsequent JSON messages.

5.5 Evaluation

To verify the functionalities of OpenFlow Wireless, I will now present several selected experiments or demonstrations. More evaluation can be found in my publications.

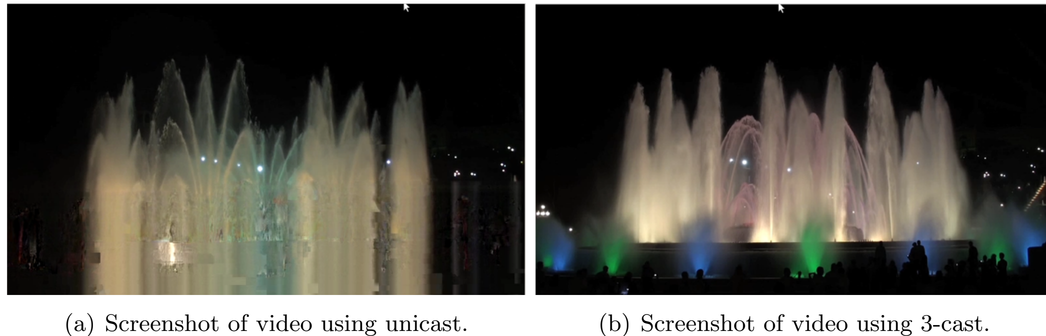
5.5.1 Video Streaming with n-casting

A simple and naive way to use multiple networks at the same time is to duplicate the packets across n distinct paths. The mobile device will then receive multiple copies of each packet over different paths and radios. This can be viewed as a generalized variant of macro-diversity described in the WiMAX standard.

In a demonstration presented at Mobicom 2009, we showed how a video stream can be 3-casted to provide a "high-reliability" service. The video stream is then received by the mobile client using multiple wireless channels, each with 3% packet loss. By using n-casting (with two streams over WiFi and a third stream over WiMAX), we demonstrated how replication can improve video quality, as visually captured in Figure 5.6.

The goal of this demonstration is not to advocate n-casting, but to show how OpenFlow Wireless enables application-specific network optimization. In this demonstration, only the UDP video stream is replicated in the network when sent to the mobile client; the rest of the traffic is sent to a selected interface. This showcases OpenFlow Wireless' capability of doing per-flow traffic engineering. Written in just 227 lines of C/C++, n-casting also demonstrates the ease of developing mobility services on top of OpenFlow Wireless.

A variation of this demonstration was also shown as part of the plenary of GENI Engineering Conference 9, where a video stream from a moving golf-cart was n-casted across the



(a) Screenshot of video using unicast. (b) Screenshot of video using 3-cast.

Figure 5.6: Screenshots of UDP video with and without n-cast with 3% loss induced on each wireless link. The screenshots demonstrate how simple replication can benefit video quality.

country to Washington, D.C. The demonstration was given in front of a live audience that experienced the difference in video quality first hand. A video capture of the demonstration is available at `http://goo.gl/OwzI1`.

5.5.2 Mobility Experiments

As a first foray into creating experiments with OpenFlow Wireless, students in a 12-week project-based class in Fall 2008 were charged with designing their own novel mobility manager, then deploying them into the network. Some interesting designs resulted. OpenFlow Wireless' design meant all mobility managers were immediately able to accomplish handover between WiFi and WiMAX, resulting in insights about handovers in such a heterogeneous environment. Another group used network state information from NOX to predict which channel they should use during a handover to minimize the hunt time. In each project, the students demonstrated the manager working in the actual production network, running simultaneously in its own slice and evaluated the results as such. Examples of these mobility managers are as follows:

1. One group designed a mobility manager (Hoolock) to perform lossless handoff that receives packets in-order. The handover exploits the fact that if a device can communicate through different wireless technologies, it must also have multiple radios.

   We will illustrate the working of Hoolock using an example. Imagine a host handing over from AP a to AP b. Since it has two interfaces, the host associates with AP b with its second interface. The routing in the network is then updated. However,

delay along the route before and after rerouting can be different; packets could be in transit along both routes for a while. Exploiting the fact that packets entering each interface on the host are in order, we buffer the packets in the interface connected to AP b while waiting for the packets from AP a to flush. After some time, the host will dissociate with AP a and switch completely to AP b, to complete the handover.

2. Another group created a handover that uses n-casting.[2] A typical n-cast handover between AP a and AP b occurs as follows. The mobile host starts with a single connection to AP a. While on the move, the host uses its idle interface to scan for and associate with AP b. Once associated, the controller begins n-casting or bi-casting to both AP a and b. The bi-casting continues until the mobile device dissociates from AP a and sends a notification to the controller to resume unicast. During the n-cast, the host is likely to receive multiple copies of the same packet. Also, differences in path latencies and loss rates could cause reordering among the two packet streams. To mitigate the effects of duplicates and out-of-order packets on TCP, a custom client-side Netfilter module was employed on the receiving network stack, to perform some level of reordering. It buffers a small amount of incoming out-of-order packets from both interfaces to remove duplicates and eliminate reordering.

The above-mentioned mobility managers exploit devices with multiple interfaces. The control software is written in such a way that this is assumed. Virtualization in OpenFlow Wireless ensured that all devices controlled by these slices indeed have multiple interfaces. Creating and testing this in conventional wireless networks would be difficult, if not impossible.

These mobility managers are evaluated against conventional hard handover in terms of packet loss during handover and TCP throughput. The evaluation network setup is simple: Two WiFi APs and a WiMAX base station are connected to a single OpenFlow switch. A server is also connected to the OpenFlow switch to generate traffic for the experiment. In handover experiments, a mobile client moves between two WiFi APs. For vertical handover, a mobile with a WiFi and a WiMAX interface moves between a WiFi AP and the WiMAX base station.

[2] Unlike the n-casting demonstration that replicates traffic all the time, this scheme only engaged n-casting during the handover.

Table 5.1: Statistics on number of packet lost during handover.

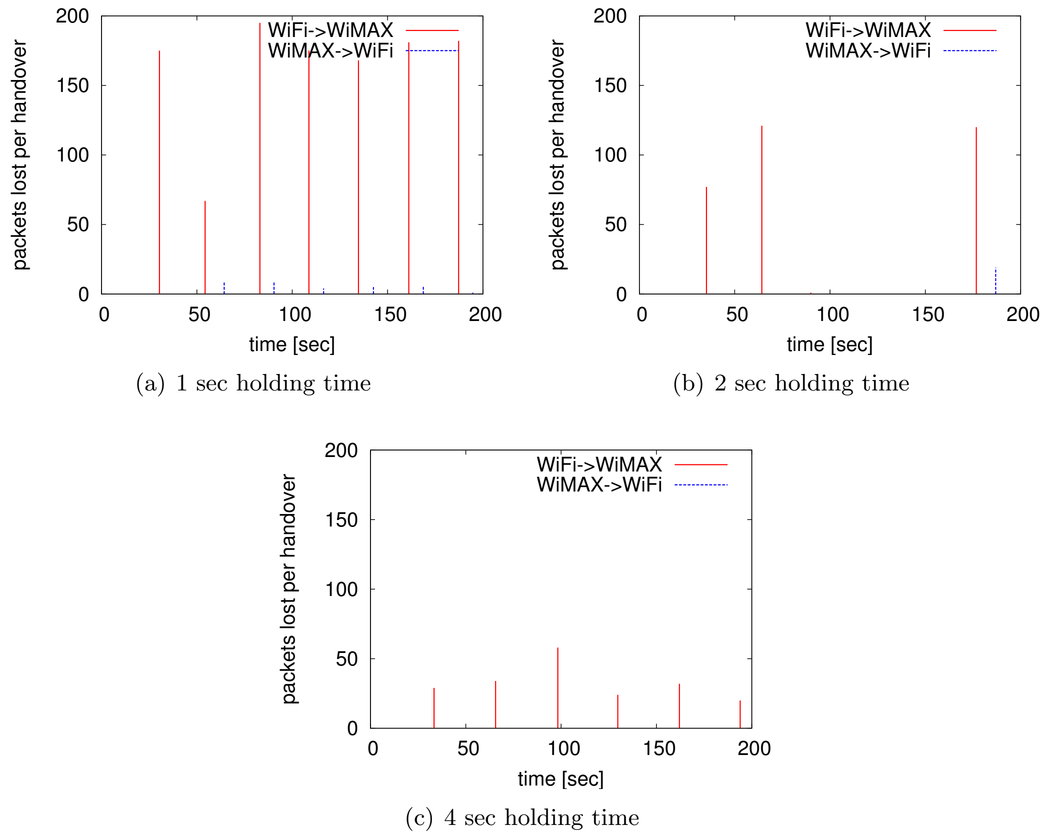
Handover Scheme (holding time)		Average	Standard Deviation	Minimum	Maximum
Hard		37.72	22.10	1	98
Hoolock (1 s)		7.1	12.1	0	39
Hoolock (2 s)		0.034	0.18	0	1
Bicasting (1 s)		0	0	0	0
Bicasting (2 s)		0	0	0	0
Vertical (1 s)	to WiMAX	161.25	32.5	67	195
	to WiFi	6.1	9.43	0	37
Vertical (2 s)	to WiMAX	87.2	50.11	0	135
	to WiFi	1.3	4.33	0	19
Vertical (4 s)	to WiMAX	39.9	19.5	0	80
	to WiFi	0.0	0.0	0	0

Packet Loss during Handovers

In each experiment, a client alternated between two WiFi APs (or between WiMAX base station and WiFi APs in the case of vertical handover) every 10 seconds for 20 times. ICMP requests were sent from the server to the client, and the number of packets lost during handover was measured. The interval between ICMP messages was 20 ms.

Table 5.1 shows the packet loss statistics. Hard handover loses the largest number of packets. Losses in hard handover occurs in spikes corresponding to the handover timings. In the case of bi-casting handover with one-second holding time, no packet loss was observed in all 20 handovers. For Hoolock, with holding time of two seconds, one packet loss was observed in one out of 20 handovers. With Hoolock, increasing the holding time reduced packet loss.

Figure 5.7 shows the performance of vertical handovers for holding times of 1, 2, and 4 seconds. With vertical handover, two handover types—WiFi to WiMAX and WiMAX to WiFi—show very different results. Handover from WiMAX to WiFi creates much smaller number of packet losses than that from WiFi to WiMAX. This is because network entry to WiMAX takes several seconds. If we release the WiFi connection before the WiMAX connection is established, then packets will be dropped. Increasing the holding time reduces such packet loss.



(a) 1 sec holding time

(b) 2 sec holding time

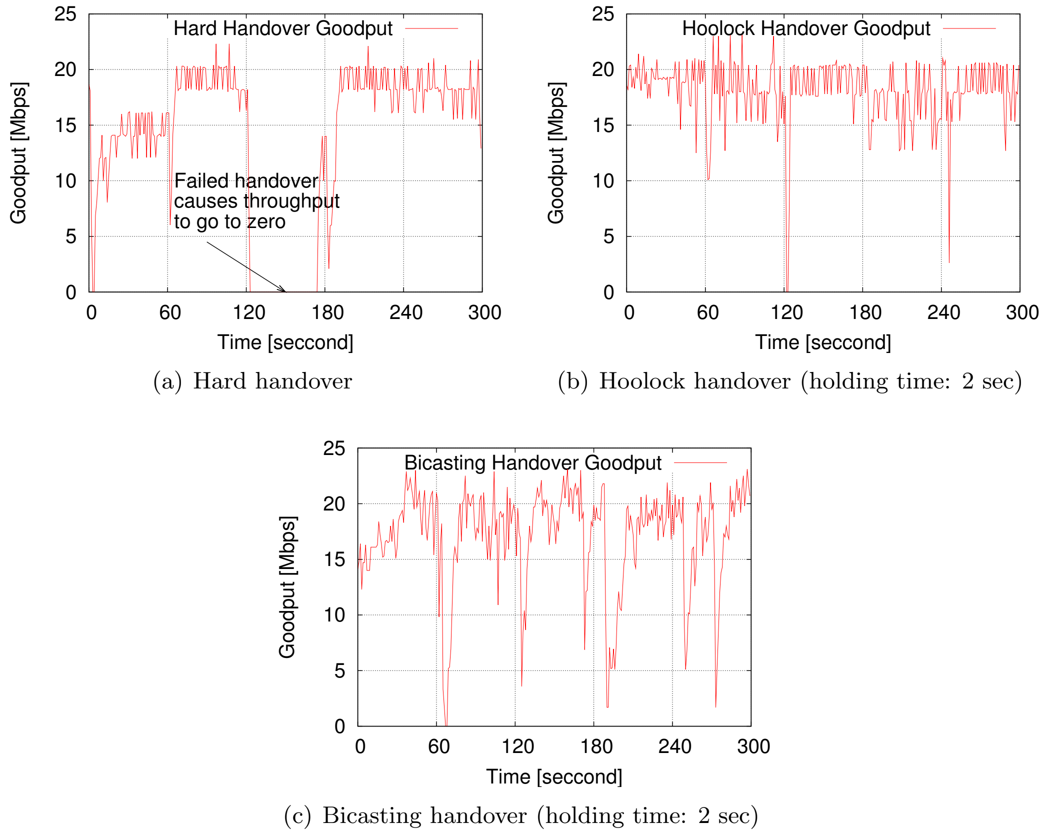
(c) 4 sec holding time

Figure 5.7: Packet losses in vertical handover.

TCP Throughput during Handovers

The TCP throughput during handovers was also measured using iperf. Here, the mobile host is the receiver of TCP data transfer. During the 300 s of data transfer, the mobile host switches WiFi APs every 60 seconds. Table 5.2 shows the aggregate results of the experiment. Figure 5.8 show the evolution of goodput for hard handover, Hoolock, and bi-casting (both with holding time of 2 s).

Hoolock and bi-casting both improve on the simple hard handover scheme. In hard handover, we observe a long period of zero goodput. During the handover, large amounts of packet loss causes the sender to throttle its window size. On the other hand, Hoolock employs two interfaces to achieve nearly zero packet loss during handover and thus a much more stable goodput. The goodput of bi-casting is less stable for two reasons. First, the OpenFlow switch performing bi-casting is a software-based switch. Since bi-casting requires



(a) Hard handover

(b) Hoolock handover (holding time: 2 sec)

(c) Bicasting handover (holding time: 2 sec)

Figure 5.8: TCP throughput with handover

replicating packets to two output interfaces, the relatively slow software switch incurs delay on the packet delivery. Second, the receiver maintains a buffer of out-of-order packets that is periodically flushed. This period determines the delay and mis-sequencing of packets during handover. A small period would cause more frequent packet reordering, while a large period would incur longer delays. The combined effects of delay and packet reordering are evident in the unstable goodput that appears shortly after handovers as TCP attempts to converge to a new equilibrium. We note that choosing a suitable operating point for bi-casting is an interesting candidate for future work.

The point here is not that the mobility managers were radically new; it is that each one was written by non-experts in less than four weeks by building on top of an open-source codebase. It is surprising to find that each mobility manager could be written in approximately 200 lines of C++. These experiences serve as anecdotal validation that a

Table 5.2: TCP throughput (in Mb/s) observed during experiment.

Handover Scheme	Average	Standard Deviation	Minimum	Maximum
Hard	13.7	7.19	0	22.3
Hoolock (2 s)	18.0	2.87	0	23.1
Bi-casting (2 s)	17.1	4.58	0	23.1

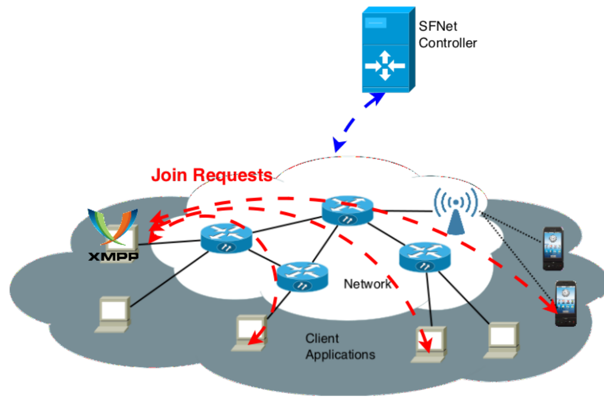
system like OpenFlow Wireles can indeed be useful to the research community by easing the innovation process in wireless networks.

5.5.3 Network facilitated P2P Multicast

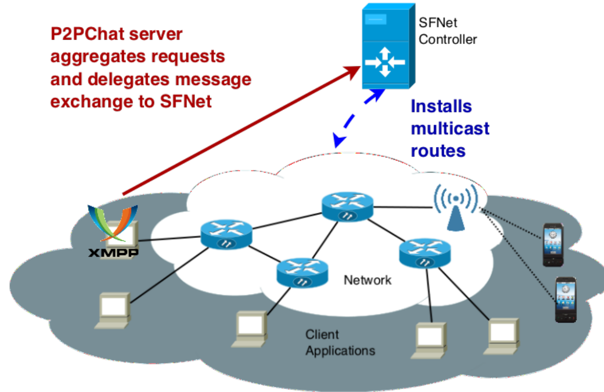
To demonstrate how a network can be built in support of the applications, SFNet was built on top of OpenFlow Wireless. In this experiment, SFNet is used to support a multicast session for an application. Here, the application gives SFNet a set of IP addresses participating in the multicast with a selected multicast IP address; SFNet then returns a response (success or failure). Subsequently, messages sent to that multicast address will be delivered to the participants. SFNet uses OpenFlow Wireless' API to find the shortest network paths among the participants. Each message sent to the multicast IP address is duplicated in-network where necessary. To deliver packets among n participants, SFNet installs n multicast trees—from a host to the other $n-1$ hosts. Each multicast message is carried only once on each link of the multicast tree. This efficiency is critical for increasingly important telepresence applications such as high-definition multi-user video conferencing.

We next describe how we can use this multicasting in SFNet to improve a chat service that uses XMPP as a rendezvous point to set up chat sessions, hereafter referred to as P2PChat. Figure 5.9 describes a use scenario of P2PChat. To join the service, each user submits a join request to P2PChat using the XMPP protocol (Figure 5.9(a)). P2PChat aggregates requests for a chat session and submits the IP addresses of the participants in a request to SFNet, which installs the appropriate multicast routes for the session using OpenFlow Wireless' API (Figure 5.9(b)). The participants can then communicate with one another by sending messages to the multicast IP address. The chat messages are forwarded directly in-network, instead of through the server (Figure 5.9(c)).

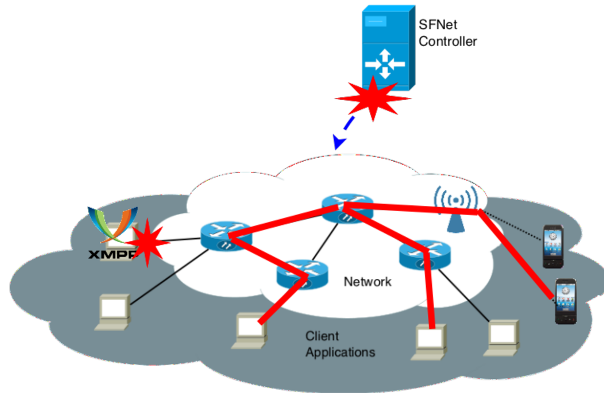
Once a chat session is set up, the participants can communicate directly using multicast sessions. This translates to reduced traffic as well as lower latency. Our results show that by having participants communicate directly, we can reduce delay from 21 ms to 3 ms



(a) Chat clients issues join requests to P2PChat server. This is how the chat will be carried out today, i.e., via the server at all times.



(b) P2PChat server delegates message exchange to the network through SFNet, which uses OpenFlow Wireless' API to install n-multicast routes.



(c) Clients switch to P2P chat, where the malfunction of the XMPP server and/or SFNet controller will not affect the chat session.

Figure 5.9: P2P chat via in-network mesh-casting (n-multicast).

compared to communicating through the XMPP server residing in the same LAN. In the meantime, relieved of its message routing duty, the P2PChat server can scale to serve more users. Moreover, chat server failure will not affect any of the ongoing chat sessions. This opens up interesting possibilities, such as using a transient client in the network as the chat server.

5.6 Summary

The architecture of wireless networks is going to change significantly in the coming years, with a slow convergence of cellular and WiFi networks—with cellular operators are moving towards an all-IP network. Users will want to move across spectrum and networks seamlessly while making use of one or more networks at the same time. Networks have to change in response to these developments. Without change, the industry will stay closed and based on proprietary equipment. The goal of this work is to help open up the infrastructure allowing multiple ideas to co-exist in the same physical network—and therefore enable innovation to happen more freely and more quickly. Opening a closed infrastructure might seem like a naive pipe dream, but recall the change that Linux brought to the computer industry through a dedicated community of open-source developers. The best place to start opening the wireless infrastructure might well be on our own college campuses, where we replace our wireless networks with a more open and backwardly compatible alternative, that supports virtualization. The blueprint presented here—OpenFlow Wireless—is a starting point.

OpenFlow Wireless achieves three main goals stated at the start of this chapter.

1. OpenFlow Wireless decouples the network architecture from its underlying wireless technologies through radio agnosticism. This also allows new wireless technologies, e.g., 802.11ac or LTE Advance, to be readily integrated into the production network once they become available.

2. OpenFlow Wireless allows different service providers to operate over the same physical network infrastructure by slicing the network both in flowspace and in configuration space. While keeping each of them isolated, this also allows each service provider to innovate and differentiate itself, creating a more vibrant ecosystem for wireless network infrastructure.

3. Finally, OpenFlow Wireless allows operators to continually innovate in their networks, by supporting "versioning" that allows them to deploy new services, e.g., creating a software-friendly network that provides direct network support to users and applications.

As such, OpenFlow Wireless—when brought to fruition—will provide users with better network services while giving operators a better network that is more programmable and more open.

Chapter 6

Conclusion

Just a few years ago, a mobile phone had only one radio and no third-party applications. This has changed dramatically. Today, it is common for a mobile device to have four or five radios, and a burgeoning army of third-party developers are creating hundreds of thousands of applications, games, and content for mobile devices.

The diverse demands of these applications are straining the network stack of the mobile operating system. For example, Skype and Dropbox both have stringent and yet very different demands. Skype wants to have a stable guaranteed data rate to stream audio smoothly with minimal jitter, while Dropbox only cares to transfer data as cheaply as possible. With all of these applications, we are also beginning to have preferences for which interfaces an application may use. For instance, we may prefer to use Dropbox over WiFi because it is cheap or free, but Skype over 3G because it gives continued connectivity.

To satisfy these demands and preferences, we have to make use of the many networks connected via the multiple radios available on the mobile device. However, today's mobile operating systems are ill equipped to exploit the multiple radios available. This is unsurprising considering that they were designed in an era when network stacks only needed to handle a single physical network connection at a time. This means that even if multiple networks are available, the operating system does not know how to aggregate the available bandwidth to provide a higher data rate, nor does it know how to migrate an ongoing TCP flow from one interface to another to provide seamless flow migration. To allow users and applications to make use of all the networks around them, we have to rethink the design of the network stacks in our mobile operating systems.

In this thesis, I introduce Hercules, a novel client network stack, that allows the user and application to make use of multiple networks at the same time. Through careful design, Hercules allows applications to flexibly choose which interfaces to use, aggregate bandwidth across multiple interfaces, and handover flows from one network to another. All these can be accomplished with minimal or no assistance from the network infrastructure.

Further, I developed an efficient fair queueing algorithm—multiple interface Deficit Round Robin (miDRR)—to augment the functionalities of Hercules. This algorithm builds on the theoretical foundations of weighted max-min fair queueing for multiple interfaces with interface preferences. This foundation is developed in this dissertation, and miDRR can be formally shown to yield the correct allocations. By incorporating miDRR as its packet scheduling algorithm, Hercules can provide users and applications with fine-grained control over routing and provisioning of each flow.

While many of our desired functionalities can be achieved simply by changing the client network stack, the network infrastructure will continue to play a significant role in the mobile ecosystem. If the network infrastructure is redesigned to allow users to make use of multiple interfaces simultaneously and move seamlessly among networks, both users and network operators would benefit. The users will gain access to greater capacity, wider coverage, and ultimately better services. The operators can operate a more efficient network, one that is less over provisioned and one that exploits the diversity of technologies and frequency bands to better serve its customers.

In this dissertation, I presented OpenFlow Wireless as a blueprint for such a network. Through radio agnosticism and network slicing, OpenFlow Wireless can virtualize the network and allows multiple service providers to operate using the same infrastructure. Using a preliminary deployment at Stanford, I was also able to show how OpenFlow Wireless can become a network that provides more direct support to applications—a software-friendly network.

Throughout this dissertation, I have explored how we can make use of all the networks around us, covering the practical aspects of client network stack design and architecture of the network infrastructure, and also the theoretical aspects of packet scheduling over multiple interfaces with interface preferences. The result is encouraging. At each step, I overcame many of the current limitations in mobile networks to provide users and applications with better network service. Hopefully, this thought marks not only the end of my dissertation, but also the beginning of a better mobile device and network infrastructure.

Bibliography

[1] Bufferbloat project website. `http://www.bufferbloat.net/`.

[2] Intents and Intent Filters. `http://developer.android.com/guide/topics/intents/intents-filters.html`.

[3] Unapproved draft standard for information technology- telecommunications and information exchange between systems- local and metropolitan area network- specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std P802.11-REVma/D8.0*, 2006.

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proc. ACM SIGCOMM '10*, Aug. 2010.

[5] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. *SIGCOMM Comput. Commun. Rev.*, 34(4):281–292, August 2004.

[6] Aruna Balasubramanian, Ratul Mahajan, and Arun Venkataramani. Augmenting mobile 3G using WiFi. In *Proc. ACM MobiSys '10*, Jun. 2010.

[7] Neda Beheshti, Yashar Ganjali, Ramesh Rajaduray, Daniel Blumenthal, and Nick McKeown. Buffer sizing in all-optical packet switches. In *Optical Fiber Communication Conference and Exposition and The National Fiber Optic Engineers Conference*, page OThF8. Optical Society of America, 2006.

[8] Josep M. Blanquer and Banu Özden. Fair queuing for aggregated multiple links. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 189–197, New York, NY, USA, 2001. ACM.

[9] Josep M. Blanquer and Banu Özden. Fair queuing for aggregated multiple links. *SIGCOMM Comput. Commun. Rev.*, 31:189–197, August 2001.

[10] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205 (Proposed Standard), September 1997. Updated by RFCs 2750, 3936, 4495, 5946.

[11] Marta Carbone and Luigi Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev.*, 40(2):12–20, April 2010.

[12] Casey Carter, Robin Kravets, and Jean Tourrilhes. Contact networking: a localized mobility system. In *Proc. ACM MobiSys '03*, May 2003.

[13] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, August 2007.

[14] Ranveer Chandra. Multinet: Connecting to multiple ieee 802.11 networks using a single wireless card. In *in IEEE INFOCOM, Hong Kong*, 2004.

[15] Jonathan Corbet. Routing open vswitch into the mainline. `https://lwn.net/Articles/469775/`.

[16] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, August 1989.

[17] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. Proportional rate reduction for tcp. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 155–170, New York, NY, USA, 2011. ACM.

[18] Jakob Eriksson, Hari Balakrishnan, and Samuel Madden. Cabernet: Vehicular Content Delivery Using WiFi. In *Proc. ACM MOBICOM*, Sep. 2008.

[19] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. A first look at traffic on smartphones. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 281–287, New York, NY, USA, 2010. ACM.

[20] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. RFC 6182 (Informational).

[21] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 1–12, New York, NY, USA, 2012. ACM.

[22] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 24–24, Berkeley, CA, USA, 2011. USENIX Association.

[23] Google. Spdy: An experimental protocol for a faster web. `http://dev.chromium.org/spdy/spdy-whitepaper`.

[24] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Comput. Commun. Rev.*, 38:105–110, Jul. 2008.

[25] Stephen Hemminger. tcp_probe.c, 2004.

[26] Hung-Yun Hsieh and R. Sivakumar. pTCP: an end-to-end transport layer protocol for striped connections. In *Proc. IEEE ICNP 2002*, Nov. 2002.

[27] Te-Yuan Huang, Kok-Kiong Yap, Ben Dodson, Monica S. Lam, and Nick McKeown. PhoneNet: a phone-to-phone network for group communication within an administrative domain. In *ACM MobiHeld '10*, pages 27–32, Aug. 2010.

[28] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *The Journal of the Operational Research Society*, 49(3):pp. 237–252, 1998.

[29] Hari Balakrishnan Lenin Ravindranath, Calvin Newport and Samuel Madden. Improving wireless network performance using sensor hints. In *Proc. USENIX NSDI '11*, 2011.

[30] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation

in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, Apr. 2008.

[31] Nimrod Megiddo. Optimal flows in networks with multiple sources and sinks. *Mathematical Programming*, 1974.

[32] S. Moser, J. Martin, J.M. Westall, and B.C. Dean. The role of max-min fairness in docsis 3.0 downstream channel bonding. In *Sarnoff Symposium, 2010 IEEE*, pages 1 –5, april 2010.

[33] Scott Moser. *Downstream Resource Allocation in DOCSIS 3.0 Channel Bonded Networks*. PhD thesis, Clemson University, Aug 2011.

[34] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. RFC 4423 (Informational), May 2006.

[35] Anthony J. Nicholson and Brian D. Noble. BreadCrumbs: forecasting mobile connectivity. In *Proc. ACM MobiCom '08*, Sep. 2008.

[36] Erik Nordstrom, David Shue, Prem Gopalan, Rob Kiefer, Matvey Arye, Steven Ko, Jennifer Rexford, and Michael J. Freedman. Serval: An end-host stack for service-centric networking. In *Proc. 9th Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA, April 2012.

[37] L. Ong and J. Yoakum. RFC 3286 (Informational).

[38] The OpenFlow Switch Consortium. `http://www.openflow.org`.

[39] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, July 2011.

[40] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1:344–357, 1993.

[41] PCI Local Bus. *PCI SIG*, rev 3.0 edition, February 2004.

[42] C. Perkins. RFC 5944 (Proposed Standard).

[43] David Raz, Benjamin Avi-Itzhak, and Hanoch Levy. Fair operation of multi-server and multi-queue systems. *SIGMETRICS Perform. Eval. Rev.*, 33(1):382–383, June 2005.

[44] Rob Sherwood, Michael Chan, Adam Covington, Glen Gibb, Mario Flajslik, Nikhil Handigol, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, Jad Naous, Srinivasan Seetharaman, David Underhill, Tatsuya Yabe, Kok-Kiong Yap, Yiannis Yiakoumis, Hongyi Zeng, Guido Appenzeller, Ramesh Johari, Nick McKeown, and Guru Parulkar. Carving research slices out of your production networks with OpenFlow. *ACM SIGCOMM Comput. Commun. Rev.*, 40(1):129–130, 2010. *(Best Demonstration at SIGCOMM 2009)*.

[45] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[46] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. *SIGCOMM Comput. Commun. Rev.*, 25(4):231–242, October 1995.

[47] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *ACM MobiCom '00*, 2000.

[48] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proc. ACM SIGCOMM '02*, Aug. 2002.

[49] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(5):81–94, October 2007.

[50] Cheng-Lin Tsao and Raghupathy Sivakumar. On effectively exploiting multiple wireless interfaces in mobile hosts. In *Proc. ACM CoNEXT '09*, Dec. 2009.

[51] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. An untold story of middleboxes in cellular networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):374–385, August 2011.

[52] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the*

8th USENIX conference on Networked systems design and implementation, NSDI'11, pages 8–8, Berkeley, CA, USA, 2011. USENIX Association.

[53] Haiming Xiao. Analysis of multi-server round robin service disciplines, 2005. Master Thesis at National University of Singapore.

[54] Haiming Xiao and Yuming Jiang. Analysis of multi-server round robin scheduling disciplines. *IEICE TRANS.*, 2002.

[55] Kok-Kiong Yap. Tcp probe++. `https://bitbucket.org/yapkke/tcpprobe`.

[56] Kok-Kiong Yap, Te-Yuan Huang, Ben Dodson, Monica S. Lam, and Nick McKeown. Towards software-friendly networks. In *ACM APSys '10*, 2010.

[57] Kok-Kiong Yap, Te-Yuan Huang, Masayoshi Kobayashi, Yiannis Yiakoumis, Nick McKeown, Sachin Katti, and Guru Parulkar. Making use of all the networks around us: a case study in android. *SIGCOMM Comput. Commun. Rev.*, 42(4):455–460, 2012.

[58] Kok-Kiong Yap, Sachin Katti, Guru Parulkar, and Nick McKeown. Delivering capacity for the mobile internet by stitching together networks. In *Proc. 2010 ACM workshop on Wireless of the students, by the students, for the students*, pages 41–44, Sep. 2010.

[59] Kok-Kiong Yap, Masayoshi Kobayashi, Rob Sherwood, Te-Yuan Huang, Michael Chan, Nikhil Handigol, and Nick McKeown. OpenRoads: empowering research in mobile networks. *ACM SIGCOMM Comput. Commun. Rev.*, 40(1):125–126, 2010. *(Best Poster at SIGCOMM 2009)*.

[60] Kok-Kiong Yap, Masayoshi Kobayashi, David Underhill, Srinivasan Seetharaman, Peyman Kazemian, and Nick McKeown. The stanford openroads deployment. In *WINTECH '09: Proceedings of the 4th ACM international workshop on Experimental evaluation and characterization*, pages 59–66, New York, NY, USA, 2009. ACM.

[61] Kok-Kiong Yap, Nick McKeown, and Sachin Katti. Multi-server generalized processor sharing. In *Teletraffic Congress (ITC 24), 2012 24th International*, pages 1 –8, sept. 2012.

[62] Kok-Kiong Yap, Rob Sherwood, Masayoshi Kobayashi, Te-Yuan Huang, Michael Chan, Nikhil Handigol, Nick McKeown, and Guru Parulkar. Blueprint for introducing innovation into wireless mobile networks. In *Proc. ACM VISA '10*, pages 25–32, 2010.

[63] Yiannis Yiakoumis, Sachin Katti, Te-Yuan Huang, Nick McKeown, Kok-Kiong Yap, and Ramesh Johari. Putting home users in charge of their network. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 1114–1119, New York, NY, USA, 2012. ACM.

[64] Yiannis Yiakoumis, Kok-Kiong Yap, Sachin Katti, Guru Parulkar, and Nick McKeown. Slicing home networks. In *Proceedings of the 2nd ACM SIGCOMM workshop on Home networks*, HomeNets '11, pages 1–6, New York, NY, USA, 2011. ACM.