SYSTEMS-LEVEL SUPPORT FOR MOBILE DEVICE CONNECTIVITY

by

Anthony J. Nicholson

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

Doctoral Committee:

Associate Professor Brian D. Noble, Chair
Associate Professor Mingyan Liu
Assistant Professor Jason Nelson Flinn
Assistant Professor Zhuoqing Morley Mao

To Cory.

ACKNOWLEDGEMENTS

While the responsibility for all errors, misstatements, and outlandish claims contained herein is soley mine, credit for the positive aspects of this dissertation lies with many people.

First, Brian Noble, my advisor. You were an excellent mentor, not just in the ways of computer science. Most importantly, you helped me make the transition from a software engineer who expected micromanagement to a more autonomous researcher and thinker. Thanks for helping undo the damage of several years spent as a code monkey before graduate school.

Besides Brian, I would also like to thank my other co-authors and collaborators: Mark Corner, Scott Wolchok, David Wetherall, Ian Smith, Mike Chen, Yatin Chawathe, Jeff Hughes, Junghee Han and David Watson. Each of you helped shape the document to follow, whether you like it or not.

The other members of my committee—Mingyan Liu, Jason Flinn, and Morley Mao. Your feedback and suggestions were invaluable, and my work certainly benefited from fresh viewpoints outside of our research group.

My undergraduate advisor at Kansas, Arvin Agah. I never seriously considered graduate school until I did my senior thesis with you. Thanks also for suggesting I work for a few years first—I would have certainly burned out otherwise.

Thanks also to the Michigan Mobility Group, past and present: Mark Corner, Landon Cox, Minkyong Kim, James Mickens, Sam Shah, along with the students in Jason Flinn's

and Pete Chen's groups. We created, discussed, and put out of their misery many different ideas together.

I would not have made it this far without the support of my parents, Nick and Terri, and my sisters Sarah and Blake. Thanks for acting interested when I bored you with details of my research.

To little Andre. By the time you read this—if ever—I'll be even older and more boring than I am today. Thanks for your help with the final revisions of this document. You always knew where to mash some keys with your fist to emphasize a point.

Most importantly, I thank my wife, Cory. You have been endlessly supportive over the five years that we have been together. I owe you big for dropping everything and camping out in Ann Arbor for the past two years while I finally finished up. At least you got a baby out of the deal!

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1

INTRODUCTION

The recent past has seen the advent of mobile computing devices. As compared to more conventional computers such as laptops or desktops, handhelds are resource-constrained with regard to CPU, storage, user interface, and battery life. Ameliorating these various shortcomings has been the focus of a great deal of research. As a result, the increasing capabilities of these devices let users run the same applications as on more powerful machines.

However, mobile data access has lagged behind because the combination of device mobility and dependence on fleeting wireless network connections invalidates some basic assumptions often made by operating systems designers. Specifically, much systems research on support for mobile networking in fact addresses only nomadic usage cases, with adverse consequences for the user experience.

Nomadic computing refers to using a device primarily while stationary at one location, but subsequently deactivating the device, moving to a new location, and resuming work. *Mobile computing* refers to continual use of a device while its user moves around their daily environment. For example, laptop computers are primarily used nomadically, while PDAs and mobile phones are also suited for mobile usage.

Nomadic devices and mobile devices do share common problems. The bandwidth and latency provided by wireless connections is often far inferior to that of a wired connection. Such connectivity is sporadically available, depending on the link technology (e.g. WiFi, GPRS), the geographic layout of access points, and the quality of APs' back-end connections. Also, there are often multiple access points to choose from at a location.

Improving network access for nomadic devices, however, is essentially just a series of optimizations for local conditions. Consider a typical laptop computer user. During the course of his day, he may use his laptop at home, at work, at a coffee shop after work, and again at home in the evening. But whenever he is traveling between locations, his laptop remains in his backpack, deactivated. Only when he is stationary at a location does he open the lid, at which point the operating system attempts to connect to a wireless network.

The situation is somewhat more complex for truly mobile devices. Consider a different user who carries a smartphone with her throughout the day. This device features both a WiFi and 3G cellular data connection, along with modest flash storage and battery life. Her phone remains active the entire day, tucked away in her pocket. Even though she rarely interacts with the screen and keys, whenever she does open the phone she wants her email inbox and RSS feeds already to be up-to-date. Other network applications also run in the background, performing their tasks on her behalf whenever sufficient network bandwidth exists.

The fundamental difference between these two scenarios lies in the stability of the wireless network connection. The nomadic user's day is a set of unrelated sessions at a sequence of fixed locations. Once the best available network at each discrete location has been discovered, the network endpoint remains stable and static until the user closes his laptop and leaves. The mobile user's day, however, is one continuous computing session with a network connection that is constantly in flux. At times the mobile user finds herself within range of several high-quality WiFi APs. At other points, her device relies on the lower-bandwidth (but more ubiquitious) 3G connection, or finds itself entirely disconnected. Mobile devices must therefore acknowledge that fluctuating network connections are a fact of life, and plan accordingly.

These observations are not entirely new. A great deal of systems research has focused on solving the problems introduced by device mobility. Despite their varied benefits, none sufficiently handles the most fundamental issue of mobile computing—the *movement* of devices and their users. For example, if a device is currently in an area of low connectivity, but the user is about to turn a corner and encounter plentiful, high-quality WiFi, the device ideally would delay non-critical traffic briefly. But applications of location

prediction to mobile computing have by and large been centralized solutions, intended to allow pre-provisioning of resources in mobile phone [3, 4, 13, 49, 58, 77] or VoIP [72] networks. This centralization is problematic, given privacy concerns and the fact that such information has significant value when out of range of network infrastructure.

Furthermore, although public spaces are increasingly awash in wireless connectivity, these opportunities are not currently exploited to their fullest potential. Prior work has recognized that at one physical location more than one usable access point is often present, and lets devices connect to many networks simultaneously using just one physical radio. This is a step in the right direction, because devices can now exploit all available connectivity in their vicinity, but data flows must manually be bound to a specific AP, however. User mobility complicates this task even further, because the connection qualities of available APs are constantly changing as the device associates with different access points. This forces every application to experimentally measure important connection qualities such as bandwidth, latency, or port restrictions, before requesting a flow be bound to a given AP.

This dissertation advances the argument that networking support in current operating systems fails to handle device mobility sufficiently. Although the issues noted above have been partially addressed in the literature, a more holistic approach to networking support in operating systems is still needed. Of primary concern to mobile devices is how the wireless connectivity available to the device changes, as a result of user mobility and the uneven deployment of public access points. In other words, we should be concerned with the *derivative of connectivity*—how it changes over time.

The remainder of this document presents an architecture for system-wide networking support that considers both the opportunities and the challenges raised by mobile devices.

1.1 Thesis Statement

It is my thesis that:

By considering how network coverage changes over time and exploiting available connectivity to the fullest, operating systems can greatly improve network quality and availability for mobile devices without requiring explicit management by users or application designers.

1.2 Challenges

Designing this new architecture meant first confronting the following challenges inherent to mobile computing:

- **Non-uniform physical deployment by unrelated administrators.** Apart from certain exceptions such as campuses and office parks, wireless connectivity available in public spaces is deployed without concern for overall availability. WiFi access points are located wherever their owner's home or office happens to be, not where they are most needed or in a uniform layout. Even when privately-owned APs are deliberately made available for use—by commercial providers such as T-Mobile or WayPort, or as part of grassroots wireless collectives [11, 57, 68]—locating APs requires human intervention to study deployment maps or databases.

- **Non-uniform connection quality.** Publicly-available access points offer significantly varying qualities of service. Clients accessing remote Internet hosts through wireless access points see a large variance in the bandwidth and latency of such connections. Different wireless network providers set their own policies with regard to blocking, redirecting, or allowing transport-layer traffic on a per-port basis. Many access points do not allow anonymous clients to use their infrastructure at all.

- **Multiple, overlapping radio technologies of differing capabilities.** Mobile devices increasingly feature multiple wireless radios that support different link-layer technologies. Each link-layer technology was designed with a certain usage model

in mind. For instance, GPRS provides broad coverage over long distances, but the connections it provides typically have far higher latency and lower bandwidth than connecting directly to the Internet via a WiFi access point. Point-to-point technologies like Bluetooth and ZigBee connect pairs of devices in close physical proximity, rather than devices and wired network infrastructure. WiMax, a long-range version of WiFi, mixes qualities of WiFi and of GPRS. All of these access technologies have varying monetary costs as well. Furthermore, in the case of WiFi multiple APs of differing quality may be present at one location.

- **Energy constraints of mobile devices.** In contrast with traditional stationary computing, the energy supply of a mobile device must be a first-class concern. Wireless radios are one of the leading energy consumers on these devices. Judicious use of wireless interfaces can therefore benefit the overall user experience immensely.

1.3 Overview of the Dissertation

This thesis was validated through the design and implementation of a comprehensive set of changes to the systems-level software typically found on mobile devices. This project had several distinct parts.

Chapter 2 examines the problem of selecting the best wireless access point from among many possible choices. The strategy most prevalent in contemporary operating systems—selecting the unencrypted AP with the strongest link signal—often fails to select the AP with the best application-level quality of connection to the Internet. These results motivate the design of a new AP selection daemon that quickly connects to each available access point and determines its suitability for use. Evaluation results show a 22–100% increase in the percentage of scans that successfully found a usable AP, as compared to selecting based on signal strength alone.

Chapter 3 augments this technique with a user-centric mobility model that tracks not only user movement but also the quality of APs seen at different locations. Applications query this service to obtain *connectivity forecasts*—estimates of the quality of network connectivity that will be available to the device at a certain point in the future. A prototype

implementation was deployed for several weeks of real usage, and the resultant data used to evaluate the efficacy of these connectivity forecasts to several applications of interest to mobile device users.

Chapter 4 presents Juggler, a virtual link layer that allows a mobile device to connect simultaneously to many wireless access networks through just one physical WiFi radio. Because mobile devices often encounter multiple usable WiFi APs at once, choosing only one results in forgoing the full potential for network access that that location. Given the varied quality of such wireless links, one can ill afford such under-exploitation if a consistent, high-quality user experience is to be maintained. Using a deployed prototype, this chapter shows how Juggler enables nearly instantaneous WiFi handoff, striping of data flows across multiple low-quality links in parallel, and maintaining simultaneous foreground Internet connectivity and a low-bandwidth side channel. This side channel is appropriate for communication to mesh networks, the user's personal area network, or other ad-hoc groups.

The dissertation concludes with discussion of related work and a summary of the conclusions to be drawn from this work.

CHAPTER 2

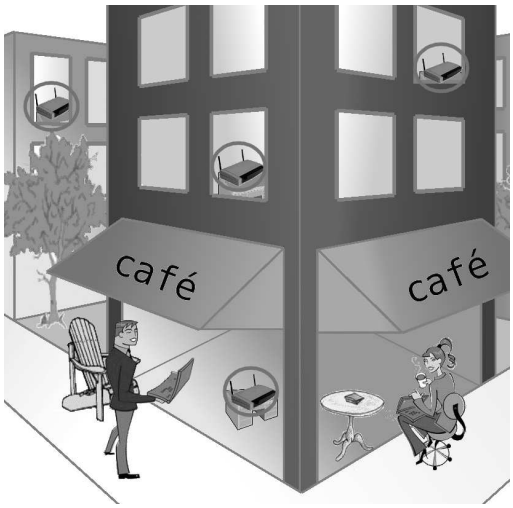
DISCOVERING NETWORK CONNECTIVITY

Mobile users have come to expect nearly constant connectivity, provided in part by the ever-increasing density of wireless access points. 802.11 wireless LAN access points (APs) are increasingly widespread in urban areas, with users commonly finding multiple APs on each scan.

Access point selection is still a critical problem, however. Consider the scenario illustrated in Figure 2.1. Customers at a sidewalk cafe encounter four access points—one from the coffee house, two from residents on the floors above (who may not even allow strangers to use their access points), and another next door, part of the city's free WiFi deployment. Which AP will provide the best quality of service?

Unfortunately, these APs are under decentralized control, and are managed by a varied set of residents and businesses. Consequently, many APs reject or restrict foreign users in a variety of ways. Since there is no common administrative control, there is also no centralized database to guide users' selection policies in favor of the APs providing the best service. While many searchable databases of "wardriving" maps exist [39, 74], these maps become outdated quickly and provide no information about access points apart from the basic information broadcast in the beacon signal.

Worse, AP selection is driven by the physical layer. The selection policy currently used by most operating systems for automatically selecting an access point simply scans for APs and then chooses the unencrypted one with the highest signal strength. However, this policy, which is known as *strongest signal strength*, or *SSS*, ignores other factors that matter to the end user. For example, the AP with the strongest signal might belong to a



Users increasingly must choose the "best" access point from many (circled in the illustration).

Figure 2.1: A sea of bandwidth.

pay service, to which the user does not subscribe. APs that appear open may use MAC address filtering to block traffic from foreign users. The bandwidth and latency of an AP's Internet connection depends on the type of ISP to which its owner subscribes—a cable modem, DSL, or dial-up. The signal strength of the access point is orthogonal to such considerations.

Currently this problem can be solved by hand-tuning connection preference tables, to enumerate the APs and networks to which one's device should connect. This is only feasible for the most common locations a user visits. At other locations, users might have to try several available networks before finding a usable connection. This is an onerous task at best that should be automatic.

Furthermore, users' computing devices are increasingly always-on, pervasive devices with wireless radios. Such devices continually need to find the best wireless connection at new locations without any user input as their owners move through their daily routine. For this usage paradigm to be possible, one needs to reduce the friction mobile devices currently encounter when trying to easily find the best available wireless connection.

To determine the scope of the problem, I conducted a small field study, examining the efficacy of strongest-signal selection. The results showed that the SSS algorithm often

chose an unusable AP when a usable AP existed, and in fact performed no better than choosing an AP at random. Usable in this case means an AP which both grants an IP address to unknown clients via DHCP, and allows Internet traffic through at least one port. This suggested that signal strength is an insufficient predictor of AP quality.

Ideally, wireless clients should quickly examine all available connection points, and *automatically* select the one appropriate for current needs that provides the best quality of service. In this chapter, I present Virgil[1], an improved access point selection system. Virgil quickly associates to each new AP found in a scan set and runs a battery of simple tests designed to probe the AP's suitability for use. Virgil uses a small set of *reference servers* spread throughout the Internet to estimate expected bandwidth and round-trip-time to remote servers.

Users also want seamless mobility from an application-level perspective, but different access points may allow or deny traffic on different network ports. Virgil therefore connects to reference servers on a wide range of TCP and UDP ports to check for port traffic blocked or redirected by each prospective AP. Based on the test results, Virgil chooses the best access point available, rather than guessing based on metrics like signal strength.

Evaluation results from five neighborhoods in three different cities show Virgil found a usable access point from 22% to 100% more often than selecting based on signal strength. I also show that maintaining a database of application-level AP test results, not just the link layer information one might find in wardriving databases, boosts these success rates even higher for neighborhoods a user visits often. Finally, analysis reveals that Virgil's overhead, while not negligible, is still reasonable enough to be useful to users. Compared with selecting access points manually, Virgil is faster and fully automatic, removing an unnecessary burden from users. Furthermore, overheads in revisited neighborhoods are indistinguishable from that required by strongest-signal-strength policies.

In the course of my field study and subsequent evaluation of the prototype implementation of Virgil, I encountered and tested nearly 4000 access points. The trace logs and AP databases are now freely available to the community through the CRAWDAD[2] archive.

[1] In *The Divine Comedy*, Virgil was Dante's guide through the underworld.

[2] `http://crawdad.cs.dartmouth.edu/`

2.1 Contributions

This chapter makes the following contributions:

- First, I show that the AP selection algorithm most frequently in current use (strongest-signal-strength) often performs no better than selecting access points at random.
- Second, I illustrate the benefit of quickly associating to each available AP and testing the suitability of each for use.
- Third, I present detailed data on the properties of over 4000 real-world access points, including not just beacon frame information but also application-level test results.

2.2 Definitions

In the remainder of this chapter, I repeatedly refer to several properties of access points, and will therefore define them here.

A given TCP port is **open** via an access point if a client can receive data from a remote server over a TCP connection on that port number.

A port is **closed** for a given AP if it is not possible to establish such a TCP connection on that port number.

A port is **redirected** for a given AP if it is possible to establish a TCP connection on that port number to a remote host, but the connection is in fact redirected to a different end host than expected. This is common for pay access point that require "splash screen" logins.

An access point is deemed **usable** if it both grants a DHCP address to anonymous clients and at least once TCP port is open to the remote reference server. For example, a public hotspot that blocks all TCP traffic except port 80 (`HTTP`) would still be considered usable.

2.3 Legal and Security Issues

Previous "war-driving" studies passively scanned the air for beacon signals that access points willingly broadcast. What I am proposing—actively connecting to each open ac-

cess point and transmitting a small amount of data to estimate that AP's connectivity to the Internet—arguably raises the question of whether it is legal to connect to any open (but possibly private) wireless network. While most jurisdictions worldwide prohibit unauthorized access to computer systems, it is not clear how these laws apply to using someone's wireless connection [43].

Such concerns are not trivial. However, it is also true that many individuals (and enterprises) are completely willing to allow strangers to connect to the Internet via their wireless networks. Many coffee shops offer free wireless connectivity. Most major cities have one or more "grassroots" wireless collectives, such as the Bay Area Wireless Users Group [11], Seattle Wireless [68], and NYCWireless [57]. Many local governments are deploying free APs in public spaces as well. For example, we will see that the field study often detected APs belonging to the city's infrastructure and to a grassroots organization in the same location. In the presence of such truly open networks, I argue that my technique is still useful. If it were possible to modify the 802.11 broadcast beacons to include an "open" flag, one could leverage it to restrict the search to only open networks.

A second problem with scanning and using relatively unknown wireless networks is caused by the rise of "evil twin" or "pharming" attacks on public access points [12]. In such attacks, a criminal uses his laptop to masquerade as a wireless AP. When other users connect to his "AP", he interposes on all their data traffic before forwarding it on to a valid AP (or simply dropping it altogether). Thus, even if users negotiate encryption keys—by using `HTTPS`, for example—the attacker can interfere with key establishment and steal all subsequent credit card numbers, bank data, or passwords protected by such session keys.

I argue that if users cannot trust their network access points, end-to-end encryption is the only reliable way to protect sensitive data. My prior work [55, 56] focused on solving this problem of establishing end-to-end trust when neither party trusts any of the intervening network hops completely—not even their network access points. This and other related work [18, 52, 65] are complementary to the main focus of this chapter—improving the wireless access point discovery process.

```
scan for all available APs
log AP beacon information for all APs
for each unencrypted AP do
   try to get IP address by DHCP
   if DHCP successful then
      (1) estimate round-trip-time to reference server
      (2) test open ports
      (3) estimate bandwidth to reference server
```

Figure 2.2: Field study script.

2.4 Field Study

Many popular operating systems (including Windows XP, Mac OS, and Linux) use essentially the same policy to guide wireless access point selection when more than one AP is available. If the system finds one of the APs in a list of "preferred networks" explicitly saved by the user, it chooses that AP. Otherwise, it simply scans for all available APs and chooses the unencrypted AP with the strongest signal strength. I will call this algorithm *strongest-signal-strength* or SSS.

The problem of AP selection first drew my interest because I suspected selecting APs based on signal strength is often the wrong thing to do. Specifically, I designed a field study to answer the following questions:

1. Do users commonly see multiple access points each time they scan for a new AP?

2. Does strongest-signal-strength selection often choose an unusable access point when a different, usable AP was available?

3. Do usable access points vary significantly with regard to the quality of Internet connection they provide?

2.4.1 Methodology

For the field study setting, I chose Chicago, the third-largest city in the United States (population: 2.8 million [15]). Since all cities have different neighborhoods of varying density, I studied three representative neighborhoods:

(a) All Encountered Access Points

	Downtown	**Residential**	**Suburban**
APs found	797	464	256
APs per scan	2.4	2.0	1.8
APs granted IP address	78 (9.8%)	81 (17.5%)	43 (16.8%)
APs using encryption	445 (55.8%)	287 (61.9%)	148 (57.8%)

(b) Open Access Points

	Downtown	**Residential**	**Suburban**
APs granted IP address	78	81	43
Usable APs	42 (53.9%)	81 (100%)	42 (97.7%)
APs redirect port 80	38 (48.7%)	1 (1.2%)	1 (2.3%)
APs with open port 80	37 (47.4%)	75 (92.6%)	39 (90.7%)
APs with closed port 80	3 (3.8%)	5 (6.2%)	3 (7.0%)

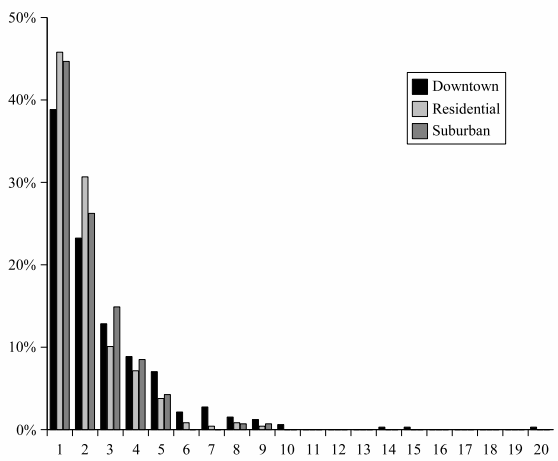
I walked an approximately 1.3 km^2 area in each of three Chicago neighborhoods. *Usable APs* were APs that granted an IP address to our handheld device and allowed port traffic through at least one TCP port.

Table 2.1: Field Study, AP Statistics.

- The Loop (Downtown): Chicago's central business district. Workday population density: 235,000/km^2 [15].
- Wicker Park (Residential): A middle-class, high-density urban neighborhood. Residential population density: 7400/km^2 [15].
- Evanston (Suburban): An upper-middle-class suburb and college town, north of the city limits. Residential population density: 3700/km^2 [15].

In all three neighborhoods, I walked a 1/2 mi^2 (1.3 km^2) grid of streets with a PDA containing a WiFi card. I chose to "warwalk" rather than "wardrive" so the script would have time to associate with APs and run tests, rather than just log 802.11 beacon information. The PDA ran Familiar Linux, a distribution targeted for handheld devices [35].

Note that these results are not intended to represent any realistic mobility model. I quite literally walked up and down streets in these neighborhoods in a grid fashion. The results in aggregate, however, are useful for drawing conclusions about the quantity, quality and frequency of wireless connectivity available in the target neighborhoods.



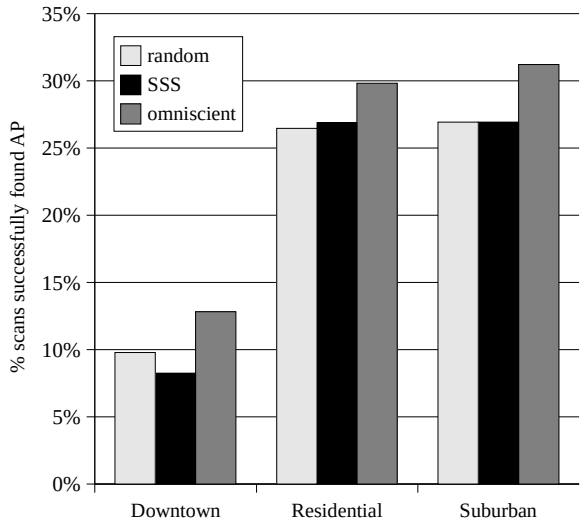
Percentage of scans for each neighborhood that found a given number of APs.

Figure 2.3: Field Study, Histogram, APs encountered per scan.

I used a Compaq iPAQ handheld with an 802.11b wireless LAN card to collect data on the density and properties of different access points in an urban environment. Figure 2.2 summarizes the field study script in pseudocode.

The reference server (RS) was a dedicated machine at the University of Michigan, directly connected to the Internet. To estimate round-trip-time, the script simply pinged the RS twice, and used the second result to avoid transient ping timeouts often seen on the first attempt. The RS also ran a simple daemon which listened on 37 common TCP ports, including SSH (22), SMTP (25), HTTP (80), Windows DCOM (135) and Samba (445). To test for open ports, the field study script sent a random integer `nonce` to the RS on each TCP port number and the RS returned (`nonce + 1`). I performed this nonce exchange rather than simply testing for establishment of TCP connections in order to verify that the access point was not redirecting traffic on that port to its own server.[3] If the script received anything other than the expected `nonce + 1` value, the port was marked *redirected*. If the connect to the reference server failed, the port was marked *closed*. To estimate bandwidth to the RS, the script connected to a special reserved port on the RS. The RS then transmitted random data at full speed over the TCP connection. The script received data for one second and then broke the connection. To avoid the effects of TCP

[3]Many commercial access points redirect all traffic to a special sign-on page (a splash screen).



Random algorithm chooses an unencrypted AP at random. *SSS* chooses the unencrypted AP with the strongest signal strength. *Omniscient* simulates an algorithm which uses the results of AP probes to choose the AP with the best bandwidth.

Figure 2.4: Field Study, Simulated percentage of scans that found a usable AP.

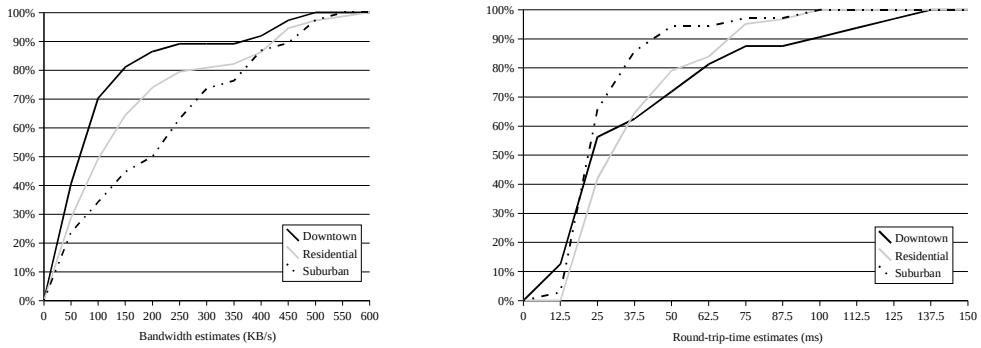
slow start, I discarded the first 500 ms of data and calculated the bandwidth estimate from the remainder. Note that an unavoidable consequence of this strategy is that some APs may be tested when the client is at the very edge of the AP's usable range, resulting in an overly pessimistic data point.

2.4.2 Access Point Statistics

I encountered a total of 1517 unique access points in all three neighborhoods. However, as Table 2.1 shows, few of these granted a DHCP address to the iPAQ handheld. It is interesting to observe that well over half of all APs had WEP or WPA encryption enabled. This suggests that a majority of users were proactive enough about their security to manually enable a feature that typically defaults to off on most consumer-grade APs.

Figure 2.3 shows the histogram of APs found per scan in each neighborhood. Note that, for a substantial percentage of scans, multiple APs were available. This is encouraging since, when foraging for bandwidth, only one access point out of many need be usable at a given location for a user to be satisfied.

Table 2.1 also gives statistics for the subset of access points that granted a DHCP address to the client. Around half of the open APs in the central business district block



(a) Bandwidth

(b) Round-trip-time

Cumulative distribution functions. Note the variance in RTT and bandwidth per AP within each neighborhood.

Figure 2.5: Field Study, RTT and bandwidth.

all TCP port traffic, and redirect port 80. This corresponds to the expected use of "splash-screen" logins for commercial hotspots. As I will see, such APs are a major source of error for the strongest-signal-strength AP selection policy, since what appears to be an open AP with a strong signal is in fact useless unless the user has an account with the service provider.

2.4.3 Missed Connectivity Opportunities

For each of the three neighborhoods studied, I took a 60-minute trace segment and applied a sliding window to generate several 30-minute "walks". Each walk represents a sequence of scans with a different set of seen APs.

For each walk, I first simulated a "random" algorithm, which simply chooses a random unencrypted AP. Next, I simulated the strongest-signal-strength selection algorithm. This generated a sequence consisting of the strongest signal strength APs from each scan in the walk. Lastly, I applied an omniscient algorithm, which used the results of the tests to choose the "best" AP from each scan set. For this experiment, the best AP was the one that granted a DHCP address and had at least one port open. If more than one AP qualified, the simulator picked the one that had the best bandwidth estimate to the reference server.

For each generated AP sequence, the evaluation metric was the percentage of scans that would have found a usable AP. The difference between the performance of the omniscient algorithm and the SSS algorithm represents the time during which a client using SSS would have been disconnected even though there was a usable AP within range. I averaged the results of all the different 30 minute walks for each algorithm (random, SSS and omniscient), and graphed the results above.

As Figure 2.4 shows, in comparison to SSS, the omniscient algorithm found a usable AP 56%, 11%, and 16% more often in the downtown, residential and suburban neighborhoods, respectively. This represents significant missed connectivity opportunities for users. As noted above, this is partly due to hotspots with "splash screen" logins. But since such hotspots were almost entirely confined to downtown, the connectivity gap in the residential neighborhoods cannot be accounted for solely by SSS choosing commercial hotspot APs. This suggests SSS is often passing on usable APs because of their signal strength, when the APs with stronger signals are in fact unusable.

Most strikingly, the simulations show that simply choosing an AP at random outperforms SSS for downtown, and is roughly the same in the other two neighborhoods. I interpreted this result as yet another validation of my belief that an AP's signal strength is a poor predictor of its suitability for use.

Furthermore, across all three neighborhoods, only 10.8% of access points were usable, but 22.6% of scan sets had a usable AP. This further reinforced my belief that choosing the best access point out of all the ones that can be seen at any given point is critical.

2.4.4 All APs Are Not Created Equal

Lastly, the field study sought to examine how access points vary in the quality of service they provide. If different APs all provide basically the same quality connection, then when multiple usable APs were present at one spot an AP selection algorithm might as well just choose one at random.

However, the results showed access points are heterogeneous. Figure 2.5 shows the cumulative distribution functions for both the round-trip-time and bandwidth estimates, for all APs encountered during the field study. None of the CDFs converge rapidly to

		Downtown			Residential			Suburban		
port #	**service**	**open**	**redir**	**closed**	**open**	**redir**	**closed**	**open**	**redir**	**closed**
135	DCOM	36	8	34	30	1	50	0	3	40
445	SMB	29	8	41	27	1	53	0	4	39
25	SMTP	28	9	41	31	0	18	36	0	7
21	FTP	29	16	33	63	1	17	37	0	6
22	SSH	37	8	33	69	2	10	37	0	6
23	telnet	39	10	29	73	1	7	37	0	6
79	finger	40	9	29	70	2	9	40	0	3
80	HTTP	37	38	3	75	1	5	39	1	3

DCOM (Microsoft's RPC) and Samba (Windows file sharing) were the most filtered.

Table 2.2: Field Study, Ports of interest.

100%, indicating a large variance in the results. This further bolstered my belief that, when multiple usable APs are present at one location, an AP selection algorithm should use the results of tests like those conducted for this field study to guide its choice.

I also found that access points vary widely with regard to what TCP port traffic they allow, block, or redirect. Table 2.2 illustrates the results of port scans for the eight most-blocked services. While most APs allowed the majority of port traffic, these were some notable exceptions. Ports 135 (DCOM) and 445 (Samba) are used by Microsoft Windows for remote procedure calls and file sharing. These are common points of entry for hackers, and are often blocked at the ISP for that reason. Port 25 handles the Simple Mail Transfer Protocol (SMTP). ISPs often block this port to prevent spammers from using the AP as a broadcast point. Finally, note that in downtown, HTTP traffic is often redirected (for splash-screen logins), but such hotspots are rarely seen in the residential neighborhoods.

These port results suggest it is useful for AP selection algorithms to know what port traffic a given AP allows. For example, suppose a user has configured her e-mail program to send e-mail by connecting to her own ISP's SMTP server over port 25. When she moves to a new location and opens her laptop, several usable networks are available. All things being equal, she would prefer that her computer uses an access point that allows her to connect directly to her ISP's SMTP server without blocking or redirecting port 25 traffic. Otherwise, she must close Thunderbird and switch to a webmail interface which may or may not be available from her mail server.

2.5 Virgil

The results of the field study motivated my belief that selecting access points based on signal strength results in a significant waste of potential network connectivity. Given that SSS performed no better than random selection in the field study simulations, I concluded that signal strength is an insufficient criterion to consistently predict AP usability.

Armed with these lessons, I designed a new AP selection system, named Virgil. Virgil scans for available APs, then quickly and cheaply probes each for suitability of use. Virgil's algorithm for selecting a new access point is as follows:

1. Scan for all available APs
2. Test each unencrypted AP in the scan set
    - Get AP properties (SSID, MAC address, signal strength, et cetera)
    - Try to get DHCP address from AP
    - If successful, probe the AP and store test results in a local database
3. Select the "best" AP, based on test results

In addition to open access points, the user may have authorization to use certain non-public APs. For example, she may encrypt her home wireless AP, and/or buy service from a hotspot provider. Virgil therefore allows users to manually enumerate "closed" APs (or SSIDs, for pay services such as T-Mobile) that should be considered for use when seen. The user must obviously enter either the WEP/WPA encryption key for encrypted access points, or her username/password credentials for APs with "splash-screen" logins.

Since Virgil stores tests results in a local database, it improves performance by not rescanning often-seen APs. Our design consists of a user-level AP selection daemon running with root-level privileges. This process scans for new APs in the background, building this history database. Virgil chooses a new access point when (1) the device first boots, or returns from hibernation or suspend, and (2) the current AP is no longer usable. The selection daemon uses a simple heartbeat to a reference server to determine when the current AP is no longer usable.

2.5.1 Probing an AP

The goal of the AP selection daemon is to always choose the "best quality" access point out of all the APs available at a given physical location. "Quality" is highly subjective, but Virgil considers the following to be important criteria:

- Bandwidth from the Internet to the client via this AP
- Port traffic that this AP blocks or redirects
- Round-trip-time from the client to remote servers

Since AP selection must be quick to be beneficial to users, Virgil performs all of these tests in parallel by spawning a thread to handle each port test and the RTT and bandwidth tests.

To aid in AP testing, Virgil uses a set of *reference servers* that is diverse in terms of both geography and network topology. Reference servers simply listen for TCP and UDP connections on a wide range of port numbers (e.g., 1–65535) and respond to port and bandwidth probe requests. At the start of each scan set, Virgil randomly chooses a reference server to use for that round of testing. This mitigates false negatives in the case where an AP is fine but the Internet route to a certain reference server is broken. Another option would be to use multiple reference servers simultaneously, and average the results. I chose not to do this because of the additional network traffic required, in order to be conservative of the iPAQ's battery.

As in the field study, Virgil tests the status of a hand-compiled list of 45 common port numbers. Additionally, however, this base set is augmented at runtime by other TCP and UDP ports that are currently in use by the client, or have recently been used. For example, suppose the user is currently connected to her office through a virtual private network (VPN), on a port number not in the base set. When migrating to a new access point, she would obviously prefer an AP which permits traffic on that port number, so her VPN services remain available.

For UDP ports, Virgil simply sends the nonce (since there is no concept of "connect" for UDP). If it receives `(nonce+1)`, the port is "open". If it receives something different, the port is "redirected". If it receives nothing before a timeout expires, the port is "closed".

Round-trip-time and downstream bandwidth from the reference server were calculated in the same fashion as for the field study above. Virgil focuses on estimating downstream bandwidth rather than upstream bandwidth because applications such as web traffic, streaming media, email, and newsgroup reading are overwhelmingly unidirectional. A recent study of wireless traffic on a campus WLAN [36], however, showed that upstream traffic comprised a significant portion of not just peer-to-peer but also web traffic. It is unclear if such workloads comprise as large a fraction of network traffic for the general population (as opposed to college students). If so, it would be useful to revise the design to estimate bandwidth in both directions.

2.5.2 Leveraging History

Each time Virgil scans an AP, it saves the AP's information in a local database. Each database record includes the following information for all APs:

- ESSID, MAC address, channel number
- Encryption status
- Signal strength, noise level, transmit power
- DHCP success (did the AP grant an IP configuration?)
- Timestamp of last scan, number of time seen since last scan

For APs that granted an IP configuration via DHCP, some additional information is recorded:

- Round-trip-time estimate (milliseconds)
- Bandwidth estimate (bytes/second)
- Port status for each scanned port (open, closed, redirected)

Virgil keeps this database to improve performance, by not repeatedly rescanning access points that the user frequently encounters. Therefore, when Virgil examines the APs seen in a scan set, it only tests APs that do not already have a database record.

Naïvely, this would forever "blacklist" any APs which performed poorly the first time they were seen. The quality of service provided by an access point can change over time (as network conditions change, customers switch to different ISPs, or the access point load fluctuates).

Virgil therefore forces periodic re-scans of access points. Each time Virgil sees an access point that is already in the database, it doesn't re-scan it but updates its timestamp field, and increments the number of times it has been seen since it was last scanned. The user configures two thresholds—the maximum time that should pass between forced re-scans, and the maximum number of times seen. Once either threshold has been exceeded, the next time the AP is seen in a scan set, Virgil forces a rescan and resets the "times seen" counter to zero.

To ensure freshness, Virgil periodically re-scans the AP currently being used by the user. Since her device is already associated with that AP, there is no interruption of service apart from the minimal network load imposed by the AP tests. By default, Virgil freshens the current AP's database record every 30 minutes, but this is a configurable value.

2.5.3 Choosing the "best" AP

Once Virgil is finished with a scan set, each AP in the set has a record in the database. Based on these test results, Virgil chooses an AP, associates with it, and retreats into the background until needed to choose an AP again.

As the obvious first step, Virgil creates a candidate set consisting of only those APs which were open. By "open", I mean that both (1) the AP granted a DHCP address to the client, and (2) at least one port was found to be open. To this set one adds APs the user has manually configured, such as pay hotspots to which she subscribes and/or encrypted APs for which she holds a key. This eliminates other pay hotspots (which block and/or redirect all traffic until users subscribe) and APs that selectively block traffic based on (for example) MAC addresses.

This may often leave more than one candidate. The user specifies how they would like such ties to be broken. If the user is primarily browsing the web or transferring large amounts of data, an obvious choice is to use bandwidth as a tie-breaker. On the other hand, if the user is dealing with latency-sensitive applications (for example, ssh), he may choose RTT. Finally, if a certain critical application needs an open port, the user may prefer APs which allow such traffic. Automatically negotiating the optimal tradeoff between these considerations is a focus of future work.

2.5.4 User Feedback

Most of the AP selection mechanism described thus far happens automatically without any intervention or attention from the user. Once Virgil settles on a new AP, it notifies the user by raising an alert (similar to the alert balloons used by Windows XP) in the corner of the screen.

Some users may want more information about the ramifications of this choice. Such users click on the notice, loading a status screen. This status screen summarizes the test results of the AP that was just chosen. Most importantly, this summary indicates which applications currently in use may stop working as a result of using this new AP. Recall the earlier example of a user who uses SMTP mail. If she moved to an access point that blocks port 25, this summary screen would inform her that her email program will not be able to send email at this location. Virgil infers that Thunderbird is an email program from the well-known port number of the connections it has established in the past. The user is *not* merely told that port 25 is blocked, since that information is meaningless to the vast majority of users.

If the user decides that using her mail reader is critically important, she indicates this, causing Virgil to review the most recent scan set and see if another open AP is available that doesn't block the port in question. If so, it associates to this new access point and informs the user. If no other AP is available, the user is informed so she can decide to walk to another location, for instance.

2.6 Prototype

I implemented a Virgil prototype on Linux, and have cross-compiled it for both x86 laptops and ARM-based handhelds (specifically, the Compaq iPAQ). The prototype implements all aspects of the design outlined above, with three exceptions. First, Virgil does not constantly scan for new access points in the background. Ideally, Virgil would leverage a system such as Juggler (see Chapter 4) to continuously scan for new access points, without having to disassociate from its current AP, but the results presented below do not reflect this. Instead, AP scanning happens when the current AP becomes unusable. Second, I have not implemented the user feedback component described in Section 2.5.4. Third, rather than using multiple reference servers, Virgil uses the same, single reference server as for the field study.

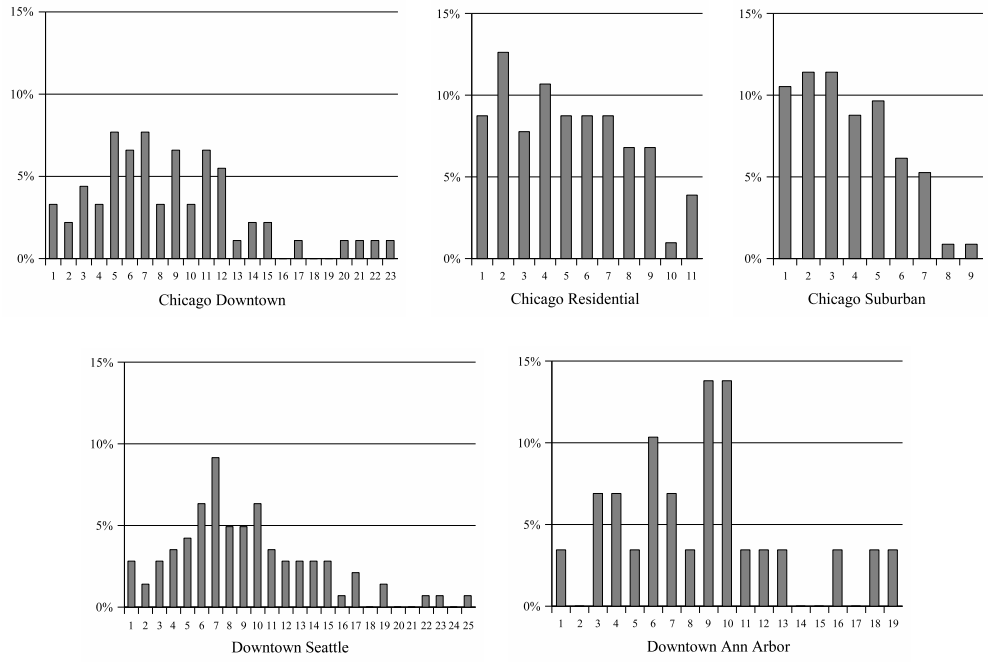
When multiple usable APs were available, the prototype used bandwidth to the reference server as the tiebreaker.

2.6.1 Active Scanning

When scanning for new access points, the primary challenge is the tension between delay and false negatives. If one cuts off testing too early, Virgil may not find all usable access points. On the other hand, if the delay Virgil imposes is burdensome to users, they will abandon the system.

As a result, the prototype has some built-in timeouts. Specifically, DHCP address acquisition times out and fails after 5 seconds. Similarly, port scans fail and return "closed" if the TCP connect takes longer than 5 seconds, or if Virgil has not received a response to the nonce in 5 seconds. I experimentally chose the value of 5 seconds by successively lowering the timeout value until I started to notice false negatives. That is, DHCP attempts and port scans were failing simply because there was not enough time for the process to complete. This also caps the average time to scan an unusable AP at around 5 seconds.

The prototype leverages the Linux wireless extensions toolkit. It uses the output of `iwlist scan` to generate a scan set at the start of AP discovery, and uses `iwconfig` to record each AP's MAC address, channel number, et cetera, in the local database.



Percentage of scans for each neighborhood that found a given number of APs.

Figure 2.6: Evaluation: Histogram, APs per scan.

All of the tests on a given AP (port probes, RTT, and bandwidth estimates) occur in parallel for maximum efficiency. Virgil uses *pthreads* to spawn a thread for each of the tests, and the main thread performs a `pthread_join` on each thread to wait until all tests have finished before proceeding.

2.6.2 Tracking open connections

Our prototype uses the Linux utility `netstat` to track open TCP and UDP connections. A thread wakes every 60 seconds, runs `netstat`, and updates an in-memory database. Since the report generated by `netstat` buffers all used ports for the last 60 seconds, this lets Virgil capture even the briefest port activity.

Each record in this database corresponds to one port number and type (UDP or TCP). For TCP connections, it notes if the connection was inbound (listening), or outbound. Finally, a timestamp notes the last time that the port was seen to be in use. By sorting this database in order by timestamp, Virgil easily determines which ports have been in most

	Downtown Chicago	Residential Chicago	Suburban Chicago	Seattle	Ann Arbor
APs seen	559	438	273	870	225
Scan sets	91	103	114	142	29
APs per scan set	6.1	4.3	2.4	6.1	7.8
DHCP success	23 (4.1%)	61 (13.9%)	41 (15.0%)	54 (6.2%)	25 (11.1%)
Encrypted APs	292 (52.2%)	261 (59.6%)	128 (46.9%)	475 (54.6%)	151 (67.1)

Percentages in parentheses are percent of total number of APs seen.

Table 2.3: Evaluation: AP statistics.

recent use, and are therefore most important to ensure remain open when migrating to a new AP.

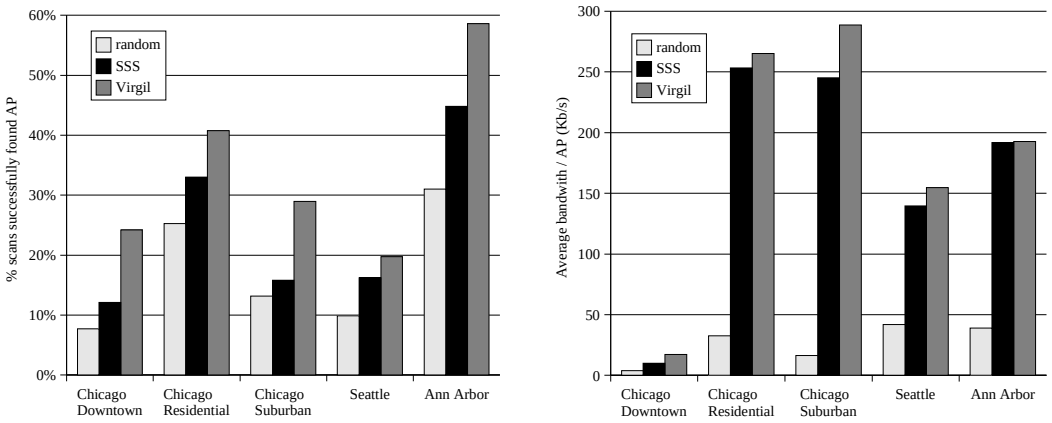
2.7 Evaluation

In evaluating the prototype implementation, I sought answers to the following questions:

- Compared to selecting on signal strength, how much more successful is Virgil in finding a usable network connection?
- How much better are the connections that Virgil finds?
- How beneficial is tracking AP history?
- Is Virgil's overhead reasonable such that it is useful to users?

I used a similar methodology to that of the earlier field study. Along with the three neighborhoods in Chicago previously studied, I also tested Virgil in Seattle, Washington (city population: 573,000, metropolitan area: 3.8 million) and Ann Arbor, Michigan (population: 114,000) [15]. These two cities gave us data points for medium- and small-sized cities, respectively.

Figure 2.6 charts the histogram of APs encountered per scan, for each of the five neighborhoods. I found more APs per scan on average when evaluating Virgil than during the field study. This may partly result from the fact that, while I re-walked the same three Chicago neighborhoods, I did not retrace my steps exactly. I also used different hardware



(a) Percentage of Scans Successful

(b) Average Bandwidth of Open APs (KB/s)

Virgil finds usable APs more frequently than selecting based on signal strength—from 22% to 100% more often. The quality of the APs Virgil finds is also better (based on bandwidth to the network).

Figure 2.7: Evaluation, Improvement of Virgil over SSS.

(a different iPAQ) for the evaluation runs than for the field study, due to equipment failure. As Table 2.3 shows, Virgil encountered 2365 different APs. As stated above, the evaluation log data and the logs from the field study are freely available via the CRAWDAD archive.

2.7.1 Connection Time and Quality

I returned to Chicago with the completed Virgil prototype on the same iPAQ handheld. In Seattle and Ann Arbor, I walked a similarly-sized portion ($\sim$1.3 km^2) of each city's downtown area.

As I walked each neighborhood, Virgil ran in the background, handling AP selection for the Linux operating system on the iPAQ. Virgil was configured to log information on all APs seen on each scan, the test results of all APs that were probed, and the final choice of AP for each scan set.

This log data let me reconstruct, after the fact, the sequence of access points that the strongest-signal-strength algorithm would have chosen. Based on the test results of probed APs, I calculated two metrics. The first was the percentage of scans which would have found a usable AP, given the selection algorithm (random, SSS or Virgil). The second was

the estimated average bandwidth, in KB/s, of the APs that each algorithm selected. The results are shown in Figure 2.7.

For all five neighborhoods, Virgil found a usable AP significantly more often than did SSS or random selection. The improvement over SSS ranged from 22% (in Seattle) to 100% (in downtown Chicago). While Virgil's connectivity percentages (ranging from 19.7% to 58.6%) are still insufficient for applications requiring seamless connectivity, it represents a significant improvement over the current state of the art.

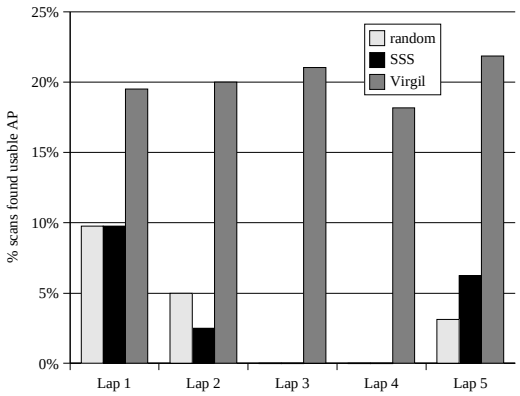
Figure 2.7(b) illustrates the average bandwidth estimates of the open APs chosen by each algorithm. Virgil still outperforms SSS, but by a much smaller margin than for connectivity. The reason for this is that SSS may only find a handful of APs in a neighborhood, but the ones it finds may happen to have high bandwidth to the reference server. On the other hand, Virgil finds more APs and therefore must average across a wide range of bandwidth connections.

2.7.2 History

Next, I sought to quantify the benefit of storing AP test results in the local database. A rough estimate of the space overhead imposed by this database can be derived from the test results above. Roughly two hours of constant scanning and walking in each neighborhood generated databases on the order of 20-30 KB in size. These are unoptimized, text-file databases. Clearly, though, if the results showed storing this history leads to little performance benefit, then they could be discarded.

I walked a 1.5 km loop from downtown Ann Arbor to campus and back five times, logging Virgil's output as before. The walk was meant to simulate the daily mobility of a hypothetical student who lives downtown, attends class on central campus, and walks roughly the same route between the two each day.

Figure 2.8 shows the percentage of scans, for each algorithm, that a usable AP was selected on each "lap" around Ann Arbor. As expected, Virgil outperforms SSS on the first lap, finding a usable AP twice as often. On the subsequent laps, however, Virgil maintains its steady success rate while the random and SSS algorithms fluctuate wildly. This is partly a consequence of unreliable AP scanning algorithms. Running a utility such



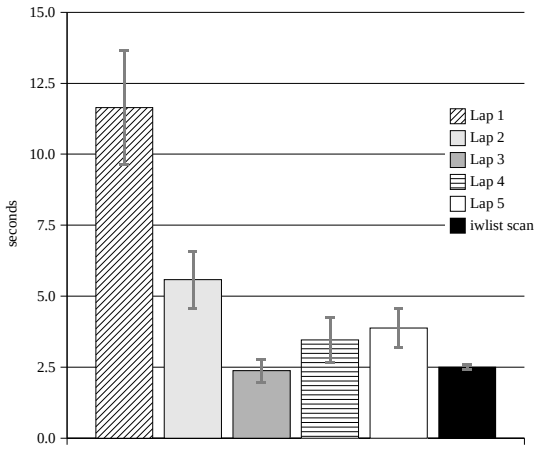
Five walks along the same 1.5 km length path from downtown, to campus, and back. The AP history database helps Virgil consistently find better access points more often.

Figure 2.8: History: Percentage of successful scans.

as Linux's `iwlist scan` several times in succession, from the same location, can return varied sets of access points. This happens because APs broadcast their beacon signals at unpredictable times, and do not always respond to beacon requests in a timely fashion [63].

On subsequent laps, all three algorithms (Virgil, SSS and random) may find new access points. In the case of Virgil, if it sees an AP that it happens to "remember" from a previous lap, Virgil will continue to use it unless the new AP proves to be of even higher quality. On the other hand, SSS does not have the benefit of such history information and has to make a spot decision based on instantaneous measurements. Thus, SSS may pick "correctly" one lap and incorrectly the next, but once Virgil finds a good AP, it will stick with it.

One of the biggest advantages of maintaining history information for Virgil is the ability to reduce the average time to complete a scan cycle. I measured the difference between laps in the average time to complete an entire scan cycle. This includes the time to discover all available access points, test each new, unencrypted AP, and finally acquire an IP address from the chosen AP. Figure 2.9 shows the average time to complete one scan cycle, for each lap. Additionally, the rightmost bar shows the mean time to only perform the scan for new APs. After the first two laps, mean time per scan cycle is nearly indistinguishable from the mean time to simply scan for APs. This is due to the effects of history. Once Virgil has "mapped-out" all the APs on a given path, it need not re-probe them. It simply scans for all available APs and chooses the best one from the list, based on its past



Time to scan for available APs, and test all new APs. The rightmost bar ("iwlist scan") is the mean time to just scan for available APs.

Figure 2.9: History: Mean time to complete one AP selection cycle.

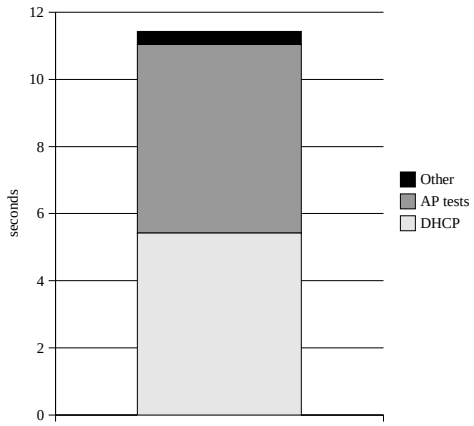
history. This confirms my belief that history will mitigate the overhead Virgil incurs in the AP probing process. Users who run Virgil every day will soon map out the routes they most commonly traverse, and per-scan overhead would be no more than current schemes such as SSS.

2.7.3 Client Overhead

I collected a diverse set of data on the time overhead inherent to our prototype implementation.

Across all five neighborhoods, for all 204 APs that granted an IP address, I calculated the time spent in each phase of the AP probing process. As Figure 2.10 shows, probing an AP took just over 11 seconds on average. This time was split fairly evenly between first acquiring an IP address via DHCP, and then running the port, RTT, and bandwidth tests. Since the timeout was 5 seconds for both of those operations, it is unsurprising that few AP tests or DHCP attempts exceeded that value.

Virgil sets this timeout to the relatively high value of five seconds, discovering more APs than it would have with a lower timeout. I argue that this scanning overhead, while not negligible, is acceptable. Virgil will quickly map the neighborhoods users spend most of their time in, erasing such overhead for subsequent visits.



Note that the time to run AP tests and to associate with the AP are comparable.

Figure 2.10: Overhead, Time to scan one AP, by phase.

Virgil would ideally be integrated with my Juggler virtual link layer, which allows one device to simultaneously associate with multiple access points (see Chapter 4). This hides most of this per-AP scan overhead, since Virgil could associate to the first candidate AP it finds, and keep scanning other APs in the background while the user is connected.

Most importantly, I argue that using Virgil to automatically select an access point, even with the overhead shown above, is still faster than the current practice of forcing users to choose manually. To reinforce this point, I measured the time required for a user to select an AP using Windows XP's integrated selection tool.

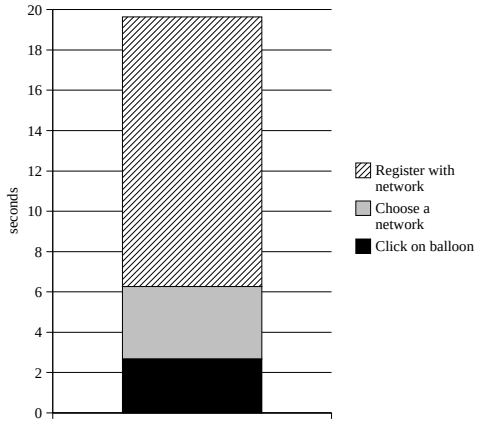
When Windows XP first boots or wakes from hibernation, it scans for all available APs. If it finds an AP which the user has previously selected as a "preferred" AP, it automatically associates to it. Otherwise, it raises the alert balloon shown in Figure 2.11(a). The user must see the alert, move the mouse to the corner of the screen and click on it. This raises a screen which lists all available APs. The only information users are given is SSID, encrypted status and signal strength (0-5 bars). Based on this information, the user chooses an AP from the list. XP then attempts to associate with the AP and receive an IP address via DHCP.

A user performed this task 10 times, and recorded the time required for three operations: (1) time between the balloon's appearance and the user clicking on it, (2) time between the AP selection window's appearance and the user clicking "Connect" to choose

(a) Windows XP's manual AP discovery notice.

	Click on balloon	Choose an AP	DHCP acquire	Total
mean	2.7	3.6	13.4	19.7
median	2.5	3.8	13.1	19.4
σ	1.1	0.8	1.8	1.4

(b) Statistics for manual AP selection in Windows XP.



(c) Breakdown of manual AP selection in Windows XP.

Many operating systems require users manually intervene to choose an AP.

Figure 2.11: Overhead, Cost of manual AP selection.

an AP, and, (3) time Windows XP required to associate with the AP, acquire an IP address, update internal state, and update the AP selection window to indicate success.

This is clearly not an exhaustive study of user behavior. However, it does provide us with some evidence of the time required to associate a Windows laptop with an AP using existing techniques. Figures 2.11(b) and 2.11(c) show it takes a user roughly 20 seconds to manually select an AP in Windows XP. I argue this is a hard bottom limit, since the user knew exactly which SSID he was looking for *a priori*, and wasted no time deciding on the selection screen, as a real user would in an unfamiliar environment.

It is curious that the time to associate with the AP dominates, since in the tests the AP was nearby and signal strength remained excellent throughout. Regardless, the time spent in active user work is significant. Furthermore, after choosing an AP, the user would need to try to load a web page, or otherwise check that the connection is usable before proceeding with her work. All of this incurs significant overhead and is burdensome to users.

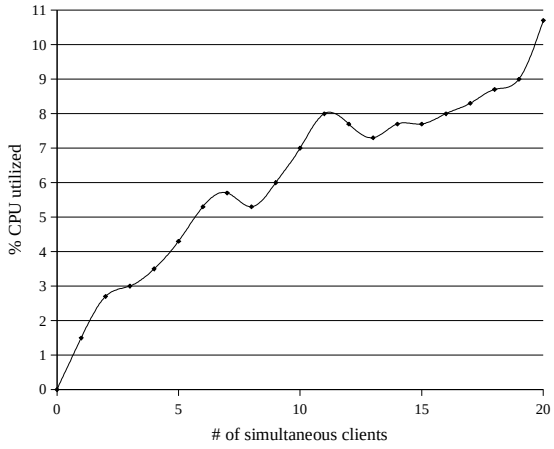


Figure 2.12: Reference server load testing results.

2.7.4 Reference Server Overhead

Finally, I sought to quantify the load the suite of AP tests would impose on the reference servers.

First, a script ran a full set of AP tests (bandwidth, round-trip-time, and 35 TCP port status tests) against the reference server. This script ran the test set 25 times in a row. This attempted to simulate a worst-case scenario, where a Virgil client found a large number of new, open APs at one location, and tested them all in rapid sequence.

Since multiple clients may be connecting to one reference server at the same time, I ran 20 trials, each time running the script described above on one additional machine.

The reference server was 2.40 GHz Intel CPU with a 512 KB L1 cache and 768 MB of system RAM. It was connected to a 10-Mbps wired Ethernet LAN. All 20 machines used to launch test clients have a 3.40 GHz Intel CPU with a 2048 KB L1 cache and 2 GB of system RAM.

To launch multiple tests as simultaneously as possible, I first pre-positioned the test script on each. To start k instances of the test script, then, from a separate machine I forked k copies of a Python script that used `ssh` to remotely launch the test script on a given machine.

Because not all copies were in fact started at the same time, I measured the peak CPU utilization during each run. This is presumably the point at which all k copies of the script are finally hammering the reference server.

Figure 2.12 shows that CPU utilization rises more or less linearly with the number of clients actively using the reference server. At the maximum load, with 20 different machines each running the AP test suite 25 times in quick succession, the CPU of the reference server was only 10.5% utilized. During none of the tests was memory a consideration.

These results suggest that a reference server with modest hardware resources can easily support dozens of client connections per second. It is unclear how many APs a typical user would need to test per day. Understanding this demand for reference server resources is crucial to ongoing work that seeks to determine how many reference servers would be needed to deploy Virgil on a large scale. Any such deployment would also need to consider the vulnerability of these servers to denial-of-service attacks, because without them Virgil clients would be unable to update their AP databases.

2.8 Chapter Summary

802.11 access point density has exploded in urban areas, to the point where users commonly have multiple APs to choose from on each scan. Since these access points are managed by a variety of individuals, businesses, and governments, a small percentage are open and usable. The quality of Internet connection APs provide often varies widely due to choice of service provider, AP load, and wireless network conditions.

A critical fact is that users' computing devices are increasingly always-on, pervasive devices with wireless radios. Such devices continually need to find the best wireless connection at new locations without any user input as their owners move through their daily routine. I argue that this vision of seamless usage is not possible without reducing the friction mobile devices currently encounter when trying to easily find the best available wireless connection.

Current selection algorithms focus on AP signal strength as an important metric. I conducted an extensive field study of three neighborhoods in Chicago, which showed that choosing an AP based on signal strength misses significant opportunities for Internet connectivity.

Motivated by the results of the field study, I presented the design and implementation of Virgil, an automatic AP discovery and selection system. Virgil quickly associates to each AP found during a scan, and runs a battery of tests designed to discover the AP's suitability for use by estimating the bandwidth and round-trip-time to a set of reference servers. Virgil also probes for blocked or redirected ports, to guide selection in favor of preserving application services currently in use.

I evaluated Virgil in five different neighborhoods across three different cities. The results show Virgil finds a usable connection from 22% to 100% more often than simply selecting based on signal strength alone. By caching AP test results, Virgil improves both performance and accuracy for neighborhoods the user commonly travels. I showed the overhead to be acceptable and less burdensome than current selection techniques which require user intervention.

CHAPTER 3

FORECASTING NETWORK CONDITIONS

The previous chapter explored wireless network management *in the moment*, reactively choosing connections only when circumstances change. This is a reasonable position to take if most users are merely nomadic, and the few truly mobile users rely on homogeneous access points.

Unfortunately, this static and simple world is fast becoming the exception, not the rule as mobile devices become primary computing platforms. Users demand continuous functionality while navigating a sea of diverse connection alternatives. In this environment applications cannot make reliable assumptions about the quality of connectivity. Instead, it fluctuates based on both the path taken through uncoordinated public deployments and the varied quality of individual access points.

This setting presents both new challenges as well as opportunities. Reactive management performs poorly, because once one has optimized for the current environment, often the device has moved and the situation has changed. Instead, one must consider the *derivative* of connectivity—how it changes over time—to properly support mobile, networked applications.

This chapter describes BreadCrumbs, a system that lets a mobile device exploit this derivative of connectivity as its owner moves around the world. BreadCrumbs maintains a personalized mobility model on the user's device, and a history of observed networking conditions. Together, these predict near-term connectivity given a user's current movement. Because people are creatures of habit, these *connectivity forecasts* can be accurate with even minimal training time. Applications, or the operating system itself, can use

these forecasts to defer less time-sensitive or low-priority work to a time that will improve performance, or reduce power consumption, or both.

To demonstrate the efficacy of this approach, I used a BreadCrumbs prototype for several weeks of day-to-day activity. During this time, both the quality and the availability of publicly-accessible APs were quite uneven. In spite of this, BreadCrumbs was able to predict the device's next-step downstream bandwidth from the Internet within 10 KB/s for over half of the time, and within 50 KB/s for over 80% of the time. These results were achieved with only one week of training.

I further explored how BreadCrumbs' connectivity forecasts could aid three example applications: (1) updating a handheld map application as the user moves, (2) streaming media content from a remote server, and (3) opportunistic writeback of created media content. Compared to prediction-ignorant baselines, BreadCrumbs' forecasts let all three applications improve the user experience in domain-specific ways.

3.1 Contributions

This chapter makes the following contributions:

- First, I introduce the concept of *connectivity forecasts* for mobile devices.
- Next, I demonstrate that such forecasts can be accurate over regular, day-to-day use, without requiring GPS hardware or extensive centralized infrastructure.
- Finally, I illustrate the potential benefits of the system through three example applications.

3.2 Background

3.2.1 Determining AP Quality

There is little point to developing a complex system for forecasting the quality and availability of public wireless connections if they are few and far between, or all access points have equivalent connection quality. I explored the current state of affairs in the pre-

vious chapter, which described Virgil—an AP selection tool that considers the application-visible quality of access points.

Much in the same way, BreadCrumbs uses a reference server to estimate the connection quality of the access points encountered by mobile devices. In addition to downstream bandwidth, BreadCrumbs also estimates upstream bandwidth via the AP. Rather than simply pinging the reference server, BreadCrumbs estimates latency by opening a TCP connection and ping-ponging a integer nonce back and forth. This was an attempt to more closely mimic how real applications would utilize a network connection. Finally, BreadCrumbs omits the port status tests in order to shorten the testing process. In summary, BreadCrumbs uses the techniques described above to estimate the following three values for each open access point the mobile device encounters: (1) downstream TCP bandwidth from an Internet host, (2) upstream TCP bandwidth to the Internet, and, (3) latency from the device to remote destinations.

If BreadCrumbs were broadly deployed and all users relied on the same reference server, the system would clearly not scale well. However, different users are free to use reference servers of their own choosing. BreadCrumbs is not attempting to quantify the quality of connection to a specific end host but rather to the more fuzzy notion of an arbitrary Internet destination.

It is true that one reference server cannot possibly represent the myriad network destinations that applications might contact. But note that the first hops—the wireless AP and its backend connection, e.g. a DSL or cable modem—are constant no matter what the remote destination of a connection ultimately is. From there, the path through the network core depends on the peering agreements between the AP's ISP and that of the destination. I argue that when choosing between two APs, it is far more likely that the overall quality of an end-to-end link depends on edge effects rather than core routing issues. This claim is validated by a recent measurement study [27] that found residential broadband links are overwhelmingly the bottleneck in end-to-end Internet paths.

3.2.2 Estimating Client Location

In order for a device to predict its future mobility, it needs some way to determine its location. This location could be descriptive ("at the Union"), relative to known locations, or absolute. In this case, BreadCrumbs uses latitude and longitude coordinates as the basic building blocks of each device's mobility model. Typically, this can be provided by GPS. Even for devices without GPS technology, it is possible to estimate one's position with reasonable accuracy, using technologies like Place Lab [47]. This project exploits the fact that a plethora of fixed-position beacons exist in the everyday environment—namely, WiFi access points and GSM mobile phone towers. A nice benefit of Place Lab is that it works well when GPS does not—indoors and in urban canyons.

Place Lab relies on public *wardriving* databases, which map beacon MAC addresses to GPS locations. For example, `wigle.net` currently tracks over 11 million distinct access points in its database. Place Lab generates a GPS fix by first scanning for all beacons in the device's vicinity, then triangulating based on the GPS location of each beacon source. Their evaluation results (in 2005) found the mean accuracy of Place Lab's location estimates to be on the order of 20-30 meters from the GPS "ground truth" when only WiFi beacon sources were utilized. As we shall see, such error is acceptable for the needs of BreadCrumbs.

3.3 Connectivity Forecasting

By leveraging Virgil and either Place Lab or GPS data, one can determine both the locations a user has previously visited and the application-level quality of network connectivity at those locations. My goal is to combine these two sets of data to yield what I will call *connectivity forecasts*. A connectivity forecast is an estimate of the quality of a given facet of network connectivity at some future time. An example would be the estimated upstream bandwidth from the client to a remote host 20 seconds in the future. This is a function both of the user's mobility—which APs will be in range at that time—and of the quality of these APs' network connections.

A wide variety of applications can exploit such forecasts. For example, consider a distributed file system client that needs to re-integrate some data to a remote file server. If energy consumption is a first-class concern—as it is for handheld devices—the best policy for the client would be to transmit data to the file server when the mobile device has the highest-bandwidth network connection that it will enjoy in the near future.

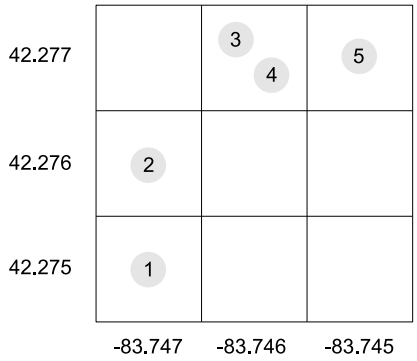
This section first discusses how BreadCrumbs maintains a personalized device mobility model, based on the past sequence of GPS locations the user visits. Next, I describe how BreadCrumbs applies the principles of Virgil to estimate the quality of different access points, and combines this data with the predictions of the mobility model. The section concludes with a concrete example of how connectivity forecasts are generated.

3.3.1 Predicting Future Mobility

Mobility prediction is a well-studied area, particularly in the domain of mobile phone networks. The majority of applications of such techniques focus on allowing a central authority to track the movement of devices to pre-provision network resources [3, 4, 13, 49, 58, 71, 77]. As did Place Lab, I note that tracking mobility history at a central point is problematic. When such databases are compromised—either accidentally, maliciously, or under subpoena—the precise movements of users are disclosed without consent. Furthermore, mobile devices may need this information the most at precisely the times when they are disconnected from the network and cannot query the centralized server.

Synthetic mobility models [75] or aggregate models derived from the movements of many users [45, 76] are useful when a network provider needs the big picture of how their network will be utilized. However, such models have little chance of accurately capturing the very unique paths one user takes through their environment.

The most compelling reason to maintain the mobility model on the device itself is that, unlike for a mobile phone network, there exists no one centralized authority who controls all public WiFi APs that the user encounters. This limits the choice of mobility models to those that can reasonably be maintained on resource-constrained, handheld devices. Song et al [72] previously evaluated the accuracy of several common mobility prediction models, using mobility data collected on the campus of Dartmouth College during the



State	Last GPS	Current GPS
1	—	(80.275,-80.747)
2	(80.275,-80.747)	(80.276,-80.747)
3	(80.276,-80.747)	(80.277,-80.746)
4	(80.277,-80.746)	(80.277,-80.746)
5	(80.277,-80.746)	(80.277,-80.745)

Each state in the second-order Markov model encodes the current GPS location and the previous location. GPS fixes are estimated at a set period τ that is the time interval between state transitions in the model.

Figure 3.1: Generating states from mobility history.

2003-2004 academic year [46]. This dataset tracks the AP association history of over 7000 users to over 550 WiFi access points of known location.

Their evaluation found a second-order Markov model, with fallback to a first-order model when the second-order model has no prediction, was the most accurate of all techniques examined. Conveniently, Markov models are ideal for use on resource constrained devices. Their CPU needs are low because model querying and maintenance involves merely reading and writing individual entries in arrays. Since these arrays are generally sparse, storage requirements are modest.

I chose geographic longitude and latitude coordinates as the fundamental building block of the model. Since I have chosen a second-order Markov model, each state consists of two sets of coordinates: the location where the device was during the last state, and its current location. Tracking this second-order state is useful for distinguishing between different mobility paths that share a common point. For example, this can disambiguate between the user walking eastbound and westbound on the same street.

Model resolution is bounded both by the accuracy of location sensing and the resource constraints of mobile devices. To avoid a state space explosion, BreadCrumbs rounds GPS values to three decimal places. While the size of one degree of latitude is constant everywhere, the distance between two degrees of longitude shrinks as one moves further away from the equator. In Ann Arbor, a $0.001^\circ \times 0.001^\circ$ grid square is 110 m$\times$80 m.

While a higher degree of location precision than 110×80 meters would seem desirable, this was impractical for two reasons. First, location estimates inherently have some amount of error from the "ground truth". As noted above, BreadCrumbs relies on PlaceLab [25] to estimate GPS location from observed WiFi beacons. The authors of PlaceLab found an average error of ± 20 or 30 meters for their technique, as compared to GPS. Second, even if a GPS antenna is available on a mobile device, one must be mindful of the state-space explosion in the Markov model that would occur if a small grid size were chosen. BreadCrumbs is intended for small devices with limited storage, CPU and battery power. When choosing a type of mobility model, I was mindful of these constraints. For example, maintaining a type of model that required a large number of complex, synchronous floating point operations would result in more CPU activity (and more power consumption) than the simpler Markov-based solution I chose.

The frequency with which BreadCrumbs estimates the device's GPS location bounds the resolution of the mobility model. This model can be thought of as a discrete-time Markov chain where a state transition fires every τ seconds. Figure 3.1 illustrates how the model generation process works. The first state is state 1. This is a special state with no "Last GPS" component, just the initial location. Then, τ seconds later BreadCrumbs fixes the device's location at $(80.276, -80.747)$, and creates the new state 2. The remaining states in the example are generated in a similar fashion.

For each state in the model, BreadCrumbs updates the Markov transition matrix whenever the model is in the state and transitions to another. These transitions occur every τ seconds. Note that if the user remains at one location for long periods, the model will have a heavy transition probability towards the self-loop (back to the same state) at that location. This is an easy way for BreadCrumbs to identify what others have termed *hubs* [32]—popular, long-term destinations.

3.3.2 Forecasting Future Conditions

Chapter 2 above described prior work on determining the application-visible quality of WiFi access points. I use similar techniques here to build an AP quality database. The purpose of maintaining this database is to estimate the "quality" of a connection to the

```
BBW (state x)
    best ← 0.00
    foreach ap ∈ {APs seen at state x}
        if ap.bandwidth > best
            best ← ap.bandwidth
    return best
```

(a) Best bandwidth algorithm

$$\text{CF (state } x_i\text{, int } steps\text{)}$$
$$\textbf{if } steps \leq 1$$
$$\quad \textbf{return } \textstyle\sum_{\forall j}\{p_{ij} \cdot \text{BBW}(x_j)\}$$
$$\textbf{else}$$
$$\quad \textbf{return } \textstyle\sum_{\forall j}\{p_{ij} \cdot \text{CF}(x_j, steps-1)\}$$

(b) Connectivity forecast algorithm

The best bandwidth algorithm has been simplified to assume BreadCrumbs tracks one type of bandwidth, when in fact it differentiates between upstream and downstream connectivity.

Figure 3.2: Pseudocode: best bandwidth at a state and connectivity forecasts.

Internet, for all the different access points a mobile device encounters. As with Virgil, when BreadCrumbs first encounters an unencrypted AP, it attempts to associate and obtain an IP address through DHCP. If successful, BreadCrumbs then estimates downstream and upstream bandwidth, and latency to remote Internet destinations.

Building an AP quality database from scratch is admittedly taxing on mobile devices, given their limited battery life. In the previous chapter, I discussed how caching results in a local database hides this expense after an initial training period. As part of future work, I hope to deploy BreadCrumbs on the COPSE mobile device testbed[1] to investigate how sharing of these databases among co-located users can reduce this scanning overhead further.

A subtle point is that one access point may be visible from multiple grid locations, since the chosen grid size ($0.001^\circ \times 0.001^\circ$) is only 110m×80m at Ann Arbor's latitude. The quality of an AP may vary at different grid locations, however, because of varying distances from the AP, physical interference, et cetera. BreadCrumbs therefore tags all AP test results with the GPS coordinates at which they were taken. Multiple test results for a single AP co-exist in the quality database if they were probed at different GPS grid locations.

[1] `http://copse.cs.duke.edu/`

The test database tracks access points both by ESSID and by MAC address. This is crucial to differentiate between APs sharing the same ESSID, either intentionally as part of a coordinated deployment or unintentionally because the default ESSID (e.g. `linksys`, `netgear`) has not been changed.

This test process incurs a reasonable but non-trivial overhead in terms of time and energy. BreadCrumbs therefore caches test results for performance. When an access point is detected, BreadCrumbs checks if a test results exists in the database for that AP at the GPS grid location containing the user's current position, and does not retest the AP if one exists. In order to age stale test results out of the database, however, BreadCrumbs retests such previously-probed APs probabilistically a small fraction of the time.

BreadCrumbs combines the custom user mobility model and the AP quality database to provide *connectivity forecasts*. Figure 3.2 describes a simplified version of this algorithm. This example takes two arguments: a state in the mobility model, and an integer number of steps in the future. In my actual implementation of BreadCrumbs, the algorithm also considers what network quality is to be forecast (downstream/upstream bandwidth, or latency). To simplify the pseudocode I assume the algorithm only considers one network quality metric, *bandwidth*.

First, consider the limiting case where $steps$ is one. This is a request for the projected network bandwidth one transition past the specified state. In other words, for the model transition period τ, one step is τ seconds in the future. BreadCrumbs calculates this forecast as the weighted sum, across all states in the model, of the best bandwidth previously seen from an AP at that potential next state. This sum is weighted by the transition probability that model will transition from state x_i to a state x_j. Thus, the best bandwidth seen at states which are likely successors of the state contributes more to the connectivity forecast than transitions which are unlikely. In practice, the number of successor states from any given state will be small as compared to the whole state space, because states are grounded in geographic reality.

If $steps$ is greater than one, connectivity forecasts are calculated recursively. At each step up the recursion tree, results from leaf nodes are weighted-summed in proportion to the transition probabilities.

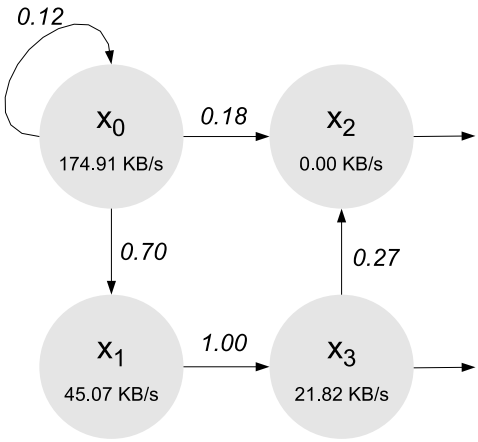


Figure 3.3: Example Markov model with best-bandwidth results.

3.3.3 Example

Consider the Markov chain in Figure 3.3. The value below each state's name is the best downstream bandwidth probed while at that state—for a state x_i, this is $\mathrm{BBW}(x_i)$. The current state is x_0. One wants to know the expected downstream bandwidth at the next time step. From Figure 3.2(b) above, this yields:

$$\mathrm{CF}(x_0, 1) = \sum_{\forall j} p_{0j} \cdot \mathrm{BBW}(x_j) \tag{3.1}$$

In other words, the expected downstream network bandwidth one step in the future is the sum (over all states in the Markov chain) of the best bandwidth observed at each state, weighted by the probability that the Markov chain will transition from the current state x_0 to each given state x_j. When calculating a connectivity forecast, one need not actually sum across all the states in the Markov chain, but only across those with a non-zero transition probability. Returning to the example, one sees from Figure 3.3 that the only possible transitions out of state x_0 are to states x_1 and x_2, and a self-loop back to x_0. Therefore, Equation 3.1 above is simplified to:

$$\begin{aligned}
\mathrm{CF}(x_0, 1) &= p_{00} \cdot \mathrm{BBW}(x_0) + p_{01} \cdot \mathrm{BBW}(x_1) + p_{02} \cdot \mathrm{BBW}(x_2) \\
&= 0.12 \cdot 174.91 + 0.70 \cdot 45.07 + 0.18 \cdot 0.00 \\
&= 52.54\ KB/s
\end{aligned}$$

For instance, if the time step of the model was ten seconds, then this would be the estimated downstream network bandwidth available to the device ten seconds from the current time. To calculate connectivity forecasts further into the future, the connectivity forecast algorithm calls itself recursively as shown in Figure 3.2(b). The downstream bandwidth 20 seconds ahead (two steps) is therefore the following:

$$\begin{aligned} \mathrm{CF}(x_0, 2) &= \sum_{\forall j} p_{0j} \cdot \mathrm{CF}(x_j, 1) \\ &= p_{00} \cdot \mathrm{CF}(x_0, 1) + p_{01} \cdot \mathrm{CF}(x_1, 1) + p_{02} \cdot \mathrm{CF}(x_2, 1) \end{aligned}$$

3.4 Implementation

I have implemented a BreadCrumbs prototype on Linux, as a user-level privileged process. This process consists of two threads, each of which is described in a subsection below.

3.4.1 Scanning Thread

One thread periodically scans for access points and fixes the device's GPS coordinates by triangulating on the locations of AP beacons in the Place Lab database. This scanning period is a configurable parameter (τ), set to 10 seconds in the current implementation. The scanning thread also handles the probing of AP connection quality, as described in Section 3.2.1, whenever an open AP is encountered that has not been probed at the current GPS grid location. Test results are then stored in a local database.

After fixing its current GPS location every τ seconds, this thread then updates the Markov model. This consists of updating the transition probability from the previous state to the new current state (because of the new location estimate).

The reference server used to estimate AP connection quality was located on the University of Michigan campus, connected directly to the Internet on the wired EECS network with no firewall. Given that the subsequent evaluation took place in the same city, one might be skeptical that connecting to this server from different wireless access points in the same city would truly approximate the average latency and bandwidth one would encounter when connecting to arbitrary remote destinations. Due to peering agreements

between the university's ISP and ISPs providing service elsewhere in town (usually Comcast or AT&T), network traffic between the reference server and off-campus endpoints passes into the network core before returning to Ann Arbor. In fact, for a subset of locations I performed a `traceroute` to the reference server, and in all cases the shortest path from the wireless AP to the departmental network detoured several hundred kilometers away before returning to Ann Arbor. I am therefore confident that this configuration reasonably approximates the latency and bandwidth one would encounter when contacting typical Internet destinations that require a trip through the network core.

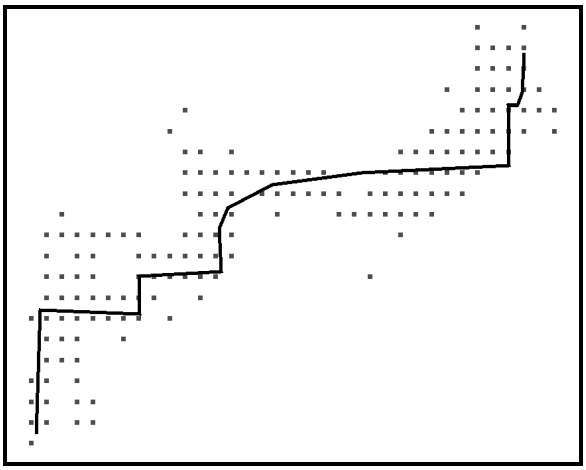
3.4.2 Application Interface

The other thread handles application requests for connectivity forecasts. Applications send requests to BreadCrumbs via a named pipe. These requests consist of two values: (1) the criterion of interest—downstream bandwidth, upstream bandwidth, or latency—and (2) an integer number of seconds in the future.

BreadCrumbs converts the value in seconds into the number of corresponding state transitions in the future of the model. This depends both on the scanning period τ and the number of seconds left until the start of the next scan, because the mobility model is a discrete time Markov chain where a state transition fires every τ seconds.

First, BreadCrumbs subtracts the time left until the start of the next scan from the value passed by the application. Then, it performs integer division of the remaining time by τ. The result is the number of steps in the future of the model at which to generate a connectivity forecast.

For example, assume that BreadCrumbs scans for APs and updates the mobility model every 10 seconds (as in the implementation), starting at $t = 0$. At $t = 9$, an application queries for the forecasted downstream bandwidth 25 seconds in the future (at $t = 36$). This is $\lfloor (36-1)/10 \rfloor = 3$ steps in the future. BreadCrumbs then generates the connectivity forecast at that point in the future, for the given criterion, and returns the value to the calling application through the named pipe.



Small squares are all GPS grid locations fixes from two weeks of user mobility traces collected. The black line is the *ground truth* path through the map taken by the user on his daily commute between home and work.

Figure 3.4: Visited grid locations and *commute* ground truth.

3.5 Sample Applications

In evaluating the usefulness of BreadCrumbs, I designed several simple applications that one might commonly find on mobile devices. I then examined how well BreadCrumbs can improve the user experience for these applications, as compared the best effort one could make without any connectivity forecast information.

The error bars in all subsequent figures in this section represent the standard error of the mean: $S_E = \sigma/\sqrt{n}$.

3.5.1 Methodology

Rather than rely on existing mobility traces or synthetic models, I installed BreadCrumbs on an iPAQ h5555 handheld, with an integrated 802.11b WiFi card, running Familiar Linux (a distribution targeted for handheld devices [35]). I carried the handheld with me continuously for two weeks during weekday, daytime hours (before seven pm).

Clearly, most users are stationary for large portions of their day (e.g. sitting at a desk). Predicting connectivity in such situations is trivial. I was more concerned with how well BreadCrumbs forecasts connectivity when users are in motion. I therefore edited the logs by hand to remove portions of time when I was stationary for more than five minutes.

	mean	σ	max	min	n
APs per scan	10.23	7.73	32	0	5227

unique APs	1621
open APs	282 (17.40%)
encrypted APs	1339 (82.60%)
grid locations visited	110
locations with usable AP	61 (55.45%)

Locations with usable AP are those grid locations where at least one access point had a probed downstream bandwidth greater than zero.

Table 3.1: Access point statistics.

BreadCrumbs ran continuously in the background, scanning for new access points every ten seconds. After each scan, BreadCrumbs estimated the device's current GPS coordinates by cross-referencing the MAC addresses of detected APs with the Place Lab database (as described in Section 3.2.2). The GPS coordinates and MAC addresses were then logged, along with a timestamp. For each AP in the scan set that had not been previously probed at those coordinates, BreadCrumbs attempted to associate and probe AP quality as described in Section 3.2.1. The probe results (upstream bandwidth, downstream bandwidth, latency) were then appended to a test results database.

Recall from Section 3.3.1 that BreadCrumbs divides the world into *grid locations*, where each grid box is 0.001° of latitude by 0.001° of longitude. At Ann Arbor's latitude, this is 110 m×80 m. All GPS fixes that fall within the same box are considered to be the same position. The small squares in Figure 3.4 are all the unique grid locations visited during the two weeks of user traces. The solid black line represents the *ground truth path* of my daily commute between home and work. This trip is a mix of walking and bus riding, and is responsible for the vast majority of motion during the two week period. The spread of visited grid locations is not strictly limited to the commute path, however. This is a result both of Place Lab GPS error and noise introduced by other, non-commuting trips. For example, the trace set includes instances of me walking from home to various downtown destinations, and driving to several different locations.

As noted above, the logs were split into discrete *trips* by excising any stationary periods lasting longer than five minutes. This resulted in 26 different trips over the two week

	mean	σ	max	min	n
down BW	68.38	114.41	385.54	0.00	110
down non-zero	123.30	129.74	385.54	0.29	61
up BW	33.98	49.85	241.66	0.00	110
up non-zero	64.44	52.44	241.66	4.10	58

Values in KB/s. According to Place Lab estimates, during the evaluation period the mobile device visited 110 unique grid locations (0.001° latitude by 0.001° longitude). *Non-zero* refers to omitting those locations where no encountered AP had a probed bandwidth greater than zero.

Table 3.2: Bandwidth at grid locations.

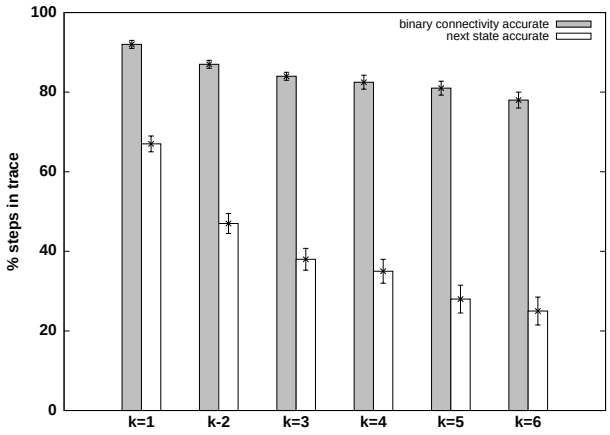
period—the longest lasted 52 minutes and 55 seconds, while the shortest was only three minutes and 50 seconds. The mean trip duration was 24:49, with a standard deviation of 12:14, indicating a large variance in the length of trips in the traces.

Tables 3.1 and 3.2 summarize the frequency and quality of network connectivity that BreadCrumbs encountered during the course of the evaluation. BreadCrumbs scanned for available APs 5553 times during the two weeks of traces. For only 368 of those scans (6.63%) were no APs detected whatsoever. As Table 3.1 shows, on average BreadCrumbs detected roughly 10 APs per scan, but this value has a high variance as well. While only 17% of all access points encountered were unencrypted, BreadCrumbs was able to discover a usable AP at over half of all visited grid locations. I define *usable* to mean there existed an AP at that location whose probed downstream bandwidth was greater than zero.

As Table 3.2 shows, I found that the quality of publicly-available access points varies significantly. For each of the 110 grid locations visited during the two weeks of trace collection, I calculated the best upstream and downstream bandwidth available. Even when those locations where no AP had a non-zero bandwidth are omitted, the variance is quite large. This bolsters my claim that the quality of WiFi connectivity fluctuates significantly as users move around the world.

3.5.2 Forecast Accuracy

I first wanted to quantify how accurate connectivity forecasts are, given the two weeks of traces I collected. As a reminder, BreadCrumbs estimates its GPS coordinates at a fixed frequency. For the evaluation I set this period to ten seconds. Thus, the traces are a



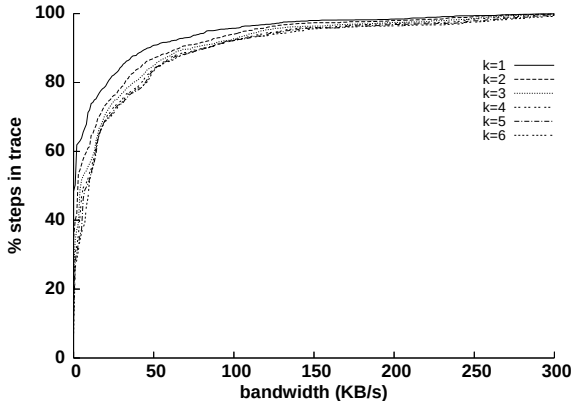
k indicates the number of steps into the future BreadCrumbs forecasts.

Figure 3.5: Mobility model prediction accuracy.

series of scan sets—listing all AP beacons detected, plus current GPS coordinates and a timestamp—separated by ten seconds of real time.

I used the first week of traces as the training set that built BreadCrumbs' mobility model. The second week of traces was then the evaluation set. For each step (scan set) in the evaluation set of traces, I compared the grid location where BreadCrumbs predicted the device would be in the next step with where it actually did move. I then repeated this, varying the number of steps BreadCrumbs looked ahead (k) from one through six. The white bars in Figure 3.5 indicate the percentage of steps across all two weeks of traces where BreadCrumbs' predicted grid location was correct, for $1 \leq k \leq 6$. The accuracy is over 70% for $k = 1$ but quickly degrades as BreadCrumbs must extrapolate further into the future. This is intuitive because when predicting many states into the future, if the mobility model chose incorrectly at any previous junction in the projected walk, then the odds of ending up at the correct grid location by the kth prediction are quite low. Thus, errors in prediction compound as BreadCrumbs looks further into the future.

The crucial insight, however, is that BreadCrumbs need not predict the user's mobility perfectly. If BreadCrumbs predicts the user will move to one location, and they in fact move to another, as long as the quality of network connectivity available at the two locations is comparable this "mistake" is unimportant. The gray bars in Figure 3.5 repre-



k indicates the number of steps into the future BreadCrumbs forecasts.

Figure 3.6: CDF, bandwidth prediction error.

sent the percentage of steps where BreadCrumbs' prediction and the actual next location matched with regard to binary connectivity. A given location is considered *connected* if at least one AP seen at that location had a probed downstream bandwidth greater than zero. BreadCrumbs was over 90% accurate in predicting binary connectivity one step ahead. This accuracy remained high when looking further into the future—nearly 80% accurate six steps ahead.

Next, I examined how the bandwidth predicted by connectivity forecasts matched the bandwidth actually encountered. Figure 3.6 charts the difference between predicted and actual bandwidth as a cumulative distribution function (CDF). Even six steps in the future, BreadCrumbs' bandwidth forecasts were within 10 KB/s of the actual value for over 50% of the trace period, and within 50 KB/s for over 80%.

It is important to note that these results were achieved with a training set of only one week duration. As users run BreadCrumbs for increasingly-long periods, the device-centric mobility model can only benefit from increased exposure to the user's patterns.

3.5.3 Sample Applications

The goal of BreadCrumbs is to improve application- and user-visible experiences for mobile devices. To evaluate the system, it was necessary to examine how different mobile applications could benefit from connectivity forecasts.

I evaluated the performance of different applications using the collected traces, rather than executing the applications "live" on a mobile device. This makes it possible to directly compare the performance of prediction-unaware algorithms and BreadCrumbs on identical sequences of user motion and APs seen, to ensure an accurate comparison.

The subsections that follow investigate three such scenarios. Clearly, connectivity forecasts are most useful for background or opportunistic tasks, where an application has some flexibility in when a network operation must occur.

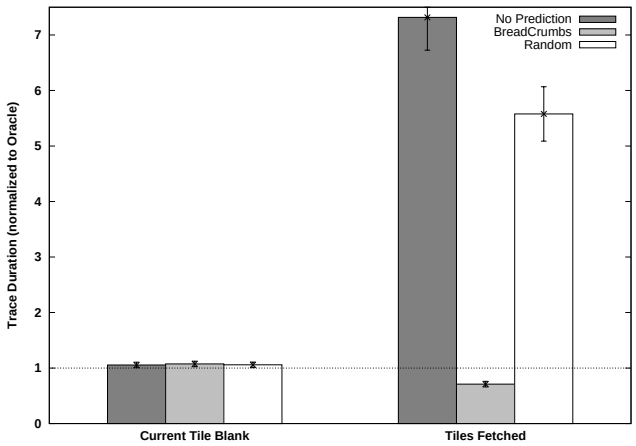
As in Section 3.5.2, the first week of traces was the training set that built the mobility model, and the second week the evaluation set. For each scenario I devised three algorithms that accomplished the same objective—one that was ignorant of any future predictions, another that utilized BreadCrumbs' connectivity forecasts, and a third that used a random walk mobility model. For each trace in the evaluation set, I ran all three algorithms, recorded the results, and subsequently averaged across all the runs. A "step" in each trace corresponds to 10 seconds of real time.

At each step in the trace, for all algorithms, the simulation declared the device associated to the AP with the best downstream bandwidth among all APs present that that location. This corresponds to the device using the aforementioned Virgil AP selection system to choose the current AP, rather than simply selecting based on signal strength. The *No Prediction* algorithm therefore represents the best one could do making no predictions of future connectivity, but using the best AP available a the current location for each step.

Map Viewer

The first sample application is a map viewer, commonly found on mobile devices like the Nokia N800. This application displays a map of the user's current location, and is typically linked to a GPS receiver so as the user moves, the currently-displayed map tile is updated to reflect this movement. Beyond simple street maps, these map tiles can contain rich contextual information—such as menus and reviews of nearby restaurants—or detailed geographic information, such as provided by Google Earth.

When the user moves out of one map tile and into another, the tile's information must either be fetched synchronously or already be present in a cache. Otherwise the user



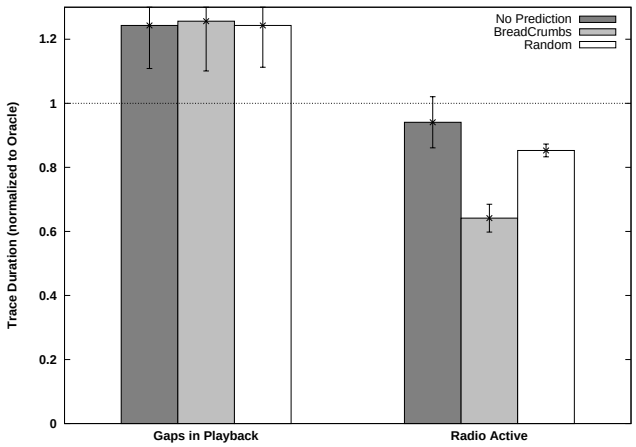
Current Tile Blank are steps in the trace where the tile corresponding to the current GPS location is not present in the device's cache, and insufficient network bandwidth exists to download it synchronously. *Tiles Fetched* is the total number of map tiles fetched over the course of each trace. All values are normalized to those of an Oracle algorithm that uses perfect knowledge of future mobility to minimize *Current Tile Blank*. The *BreadCrumbs* algorithm avoids unnecessary network traffic by not pre-fetching neighboring tiles when upcoming network conditions are predicted to be good, while incurring a slightly higher rate of missing tiles.

Figure 3.7: Evaluation, Map Viewer.

experience degrades as blank tiles appear in the map. A policy of always pre-fetching all neighboring map tiles provides good coverage, but at the cost of wasted network operations if those tiles are never visited or displayed.

I investigated if BreadCrumbs' forecasts could be used to "roll the dice" and avoid wasteful pre-fetching in cases where BreadCrumbs predicted that the device will have sufficient network bandwidth available to synchronously fetch a new map tile as soon as the user moves to that location. I therefore designed four algorithms for comparison.

First, *No Prediction* ensures that the user's current map tile, and all eight surrounding tiles, are in the cache whenever sufficient network bandwidth exists to do so. Second, at each step in each trace, the *BreadCrumbs* algorithm generates a connectivity forecast one step in the future. If the forecast indicates that the device will have enough bandwidth to synchronously download the new tile, the *BreadCrumbs* algorithm does not pre-fetch neighboring blocks. If the predicted next-step bandwidth is low, however, it pre-fetches neighboring blocks just as *No Prediction*. Third, the *Random* algorithm is identical to *BreadCrumbs*, but instead of using BreadCrumbs' connectivity forecasts, this algorithm



Gaps in Playback is the trace duration where the stream was not playing on the device because the buffer was empty and inadequate network connectivity existed at that location. *Radio Active* is the trace duration that the WiFi radio was actively downloading data. All values are normalized to those of an Oracle algorithm that minimizes *Gaps in Playback* by downloading the entire stream as fast as possible with an infinite buffer size. The *BreadCrumbs* algorithm avoids pre-filling the buffer if forecasts indicate that upcoming network bandwidth will be sufficient to service the stream. This conserves energy while not significantly increasing playback gaps.

Figure 3.8: Evaluation, Streaming Media.

chooses a random successor state to the current state, and takes the best bandwidth observed at that state as the bandwidth the device will have at the next step in the trace.

Finally, the *Oracle* algorithm uses perfect knowledge of future mobility to achieve the minimum possible number of blank tiles per trace. Note that minimizing one criterion (*Current Tile Blank*) does not necessarily optimize for the other (*Tiles Fetched*). In fact, we will see that the *BreadCrumbs* algorithm fetches fewer tiles than the *Oracle* because it risks blank tiles in order to fetch as few tiles as possible from the remote server.

Tiles correspond to the 110×80 meter tiles of the model, and each tile is assumed to be 100 KB in size. The results in Figure 3.7 are all normalized to those of the *Oracle* algorithm. One sees that by not pre-fetching neighboring tiles when upcoming connectivity is predicted to be good, the *BreadCrumbs* algorithm avoids wasting energy by fetching tiles that will never be displayed. At the same time, this gamble results in only a three percent higher rate of missing map tiles than the *No Prediction* algorithm.

Streaming Media

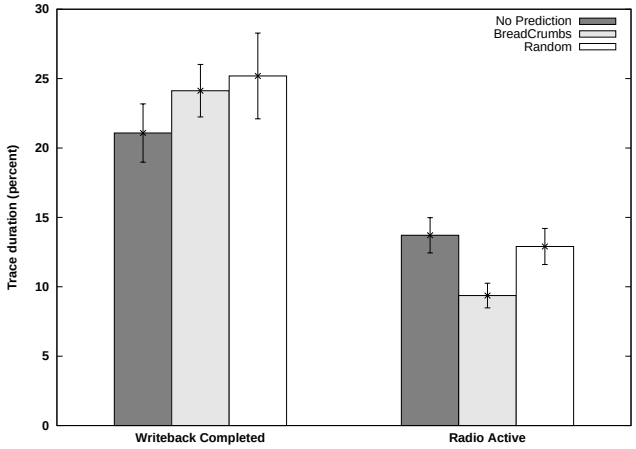
Next, I considered issues raised when streaming media content from a remote server onto a handheld device while the user is in motion. A media stream has a well-defined quality-of-service metric—specifically, the encoded bit rate of the stream. When mobile in public, however, the user's device moves from connection to connection at different locations. Some locations may have sufficient bandwidth to service the stream, some may not, and some locations may be devoid of network connectivity altogether.

One option is to define a buffer size and fetch the stream as fast as possible at every given moment, up to the point where the buffer is filled. This is the strategy commonly employed today by streaming media applications, corresponding to the *No Prediction* algorithm for this application. This algorithm downloads the stream as fast as possible at each step in the trace, given the available network connectivity, up until the point that a two-minute buffer has been filled.

Second, at each step the *BreadCrumbs* algorithm generates connectivity forecasts for each of the next six steps—up to one minute in the future. If the future connectivity is predicted to be sufficient to service the media stream, the algorithm does not pre-fetch data into the buffer. In other words, this algorithm risks incurring the (hopefully) rare consequence of guessing incorrectly—an interrupted media stream—in order to aggresively reduce the amount of network traffic it generates. The *Random* algorithm is identical, except that instead of using connectivity forecasts to predict future network conditions, it generates a random walk from the current location into the future.

Finally, the *Oracle* algorithm downloads the entire stream as fast as possible, without a buffer size cap. This is less of an "oracle" than a bound on how quickly any algorithm could fetch the data comprising the stream for the trace duration. Note that while this minimizes *Gaps in Playback* it results in the radio being active for longer than any of the other algorithms.

For this evaluation, I simulated a 64 KB/s video stream, of over two hours in length—comparable to that of a feature film. Note that, as described above, I broke the traces up into segments demarcated by idle periods of five minutes or more. None of the trace



Writeback Completed is the total elapsed trace time until all data was safe on the remote server. *Radio Active* is the total trace time that the WiFi radio was actively transmitting data. All values are percentages of total trace duration. By utilizing BreadCrumbs' connectivity forecasts, the prediction-aware algorithm delays data writeback briefly to selectively use high-bandwidth access points. As a result, the total time until data is safe on the remote server is comparable, but BreadCrumbs activates the WiFi radio 30% less often than with no prediction, translating into significant energy savings.

Figure 3.9: Evaluation, Opportunistic writeback.

segments were longer than the length of the stream. Figure 3.8 shows that all three algorithms (*No Prediction*, *BreadCrumbs*, and *Random*) result in comparable gaps in playback. The *BreadCrumbs* algorithm, however, activates the WiFi radio 30% less often than the prediction-unaware algorithm. By employing BreadCrumbs' connectivity forecasts, that algorithm is able to provide the same playback experience to the user while using significantly less of the mobile device's battery as compared to a prediction-ignorant algorithm.

Opportunistic Writeback

The final scenario considers a user who has generated some content on his handheld device while away from home. These files are digital photos taken by the camera on his smartphone. The user previously configured a distributed file system client to ensure all content he generates will be safely reintegrated to his remote file server. This file server could be a dedicated machine at his home or work, or a web service such as Flickr. I assume the only network connectivity available to the smartphone is whatever open WiFi is available.

For evaluation purposes, I set the number of photos that our hypothetical user took at eight, each with a filesize randomly uniform between 1 MB and 5 MB. The filesizes were generated once and then the same set used across the entire evaluation for consistency.

The *No Prediction* algorithm simply tried to transmit the eight image files as quickly as possible, at each step using the AP with the best upstream bandwidth available at that location. The algorithm that utilized BreadCrumbs sought to reduce the amount of time the WiFi radio was active, while not delaying data writeback unreasonably. The simple prediction-aware algorithm worked as follows. At each step of trace playback:

1. Determine which AP has the best upstream bandwidth at the current location.

2. Query BreadCrumbs for its connectivity forecast of upstream bandwidth 10, 20, and 30 seconds in the future. If any of those three future points are predicted to have better upstream bandwidth, do nothing at this time. Else, transmit data to the remote server as fast as possible during this step.

This algorithm is admittedly somewhat naïve. This was intentional as I sought to evaluate how useful BreadCrumbs' connectivity forecasts could be for applications that have made very minimal modifications. A third algorithm, *Random*, operated the same as the BreadCrumbs algorithm but instead of using connectivity forecasts to predict future network conditions, *Random* simply generated a random walk through the geographic neighbors of a given state in order to "predict" future mobility and connectivity.

I ran all three algorithms once for each of the traces in the evaluation set. The evaluation metrics were (1) total elapsed time until the all data was safely on the remote server, and (2) total time the WiFi radio was actively transmitting. Figure 3.9 illustrates the results. On average, the BreadCrumbs algorithm completes writeback only slightly slower than the aggressive, prediction-ignorant algorithm. In fact the difference is nearly within the error bounds of the mean for both algorithms.

On the other hand, utilizing BreadCrumbs' connectivity forecasts lets the prediction-aware algorithm activate the WiFi radio 30% less often. By attempting to only transmit data at high-bandwidth locations, the prediction-aware algorithm makes more efficient use of the wireless radio. While small for desktops or even laptops, this is significant for

# states in model	652
model size	27984 bytes (42.92 B/state)
# test results	1335
test DB size	92132 bytes (69.01 B/entry)

The test database is currently stored in unoptimized, ASCII format.

Table 3.3: Overhead, space requirements.

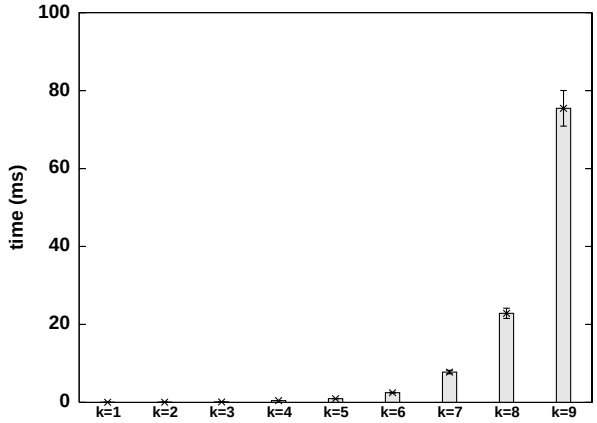
mobile devices where wireless NIC usage is a large fraction of total energy expenditure. For example, Anand et al [7] found that, for an iPAQ handheld, the power required to actively transmit data over the WiFi interface (even in power-save mode) was nearly equal to the measured quiescent power consumption of the entire device when the radio was inactive.

3.5.4 Overhead

Table 3.3 shows the storage required on the iPAQ to store the mobility model and test database generated in the course of the evaluation. With 652 different states in the model, the total model size is approximately 27.3 KB, or 43 bytes per state on average. Recall that, because this is a second-order Markov model, each state represents the current GPS grid location of the user and their previous location. As Table 3.1 shows, BreadCrumbs visited 110 different grid locations during the evaluation period. If every combination of current location and previous location were generated as a state, the model would have $110 \times 110 = 12100$ states. Even a model of such complexity would only require 508 KB of space on the mobile device. Given the sparseness of these models in practice, a model of that size would be most likely be sufficient to cover an entire metropolitan area.

Likewise, the overhead imposed to store the test database is reasonable—69 bytes per test entry on average. For convenience, the database was implemented as an ASCII flat file, unoptimized. Even so, the records for the 1335 test results generated by the evaluation require 90 KB of storage space, but only 7.04 KB when in compressed form.

Figure 3.10 examines the CPU overhead imposed when generating connectivity forecasts. The parameter k is the number of steps in the future of the model, given a current state, that an application requested a connectivity forecast of downstream bandwidth from



Results on a Compaq iPAQ handheld (400 MHz CPU), 128 MB RAM.

Figure 3.10: Connectivity forecast overhead.

BreadCrumbs. This graph represents only the instrumented CPU time required for the calculation, not any communications overhead between BreadCrumbs and the application requesting the forecast. All results were measured on a Compaq iPAQ h5555, with a 400 MHz ARM processor and 128 MB of system RAM.

An application requested a connectivity forecast for each of the 652 states in the model the evaluation generated, varying the size of k from 1 to 10. Because this is a recursive algorithm (see Figure 3.2) one would expect the overhead to grow exponentially. Up to six steps ahead, the overhead is less than 2.5 ms. Even the mean overhead of 75 ms at $k = 9$ is not prohibitive for applications that perform such intensive operations rarely. Note that I did not implement caching of calculated forecasts or other possible optimizations in the implementation.

3.6 Chapter Summary

Operating systems currently focus on immediate conditions when managing wireless networking. But today, users are highly mobile, utilizing a patchwork of public access points of varying capabilities and uneven distribution. Applications would like to opportunistically perform background or low-priority work, but cannot make reliable assumptions about connection quality at any given moment in the future.

I argue that the increased mobility of users demands a focus on how connectivity changes over time—its *derivative*. This chapter described BreadCrumbs, my system that let a mobile device track this trend of connectivity quality as its owner moves around the world. BreadCrumbs maintains a personalized mobility history on the device, and tracks the APs encountered at different locations. BreadCrumbs also probes the application-level quality—bandwidth and latency to the Internet—of the open connections the device encounters.

Together, the predictions of the mobility model and the AP quality database yield *connectivity forecasts*. These forecasts let applications take domain-specific action in response to upcoming network conditions. I evaluated the efficacy of these forecasts with several weeks of real-world usage. BreadCrumbs was able to predict downstream bandwidth at the next step of the model within 10 KB/s for over 50% of the evaluation period, and within 50 KB/s for over 80% of the time, with only one week of training data to build the model and AP quality database. I also evaluated how three example applications, with minimal modification, can utilize connectivity forecasts. The results showed that with as little as one week training time, BreadCrumbs can provide improved performance while reducing power consumption, a critical concern for resource-constrained mobile devices.

CHAPTER 4

EXPLOITING AMBIENT CONNECTIVITY

Up to this point, this dissertation has focused first on discovering wireless network connectivity, and subsequently on predicting future connectivity based on device mobility patterns and the discovered AP deployment in the user's environment. The underlying assumption throughout was that devices connect to only one access network (e.g. WiFi access point) at a time. The log data from the Virgil and BreadCrumbs field evaluation, however, show that devices are often within range of more than one usable AP. Restricting devices to just one results in underutilization of the potential wireless connectivity available.

I showed in previous chapters that the quality of wireless data connections can vary widely. This is a result of many causes, including distance from the access point, interference from buildings, collisions from other clients transmiting on the same frequency, and poor quality of the AP's backhaul link to the Internet. No matter what the cause, however, given the unreliable quality of such connections one cannot afford to ignore potential bandwidth opportunities.

There is increasing recognition that, in this situation, wireless clients can often benefit from additional radio interfaces. For example, multiple interfaces can increase effective bandwidth through provider diversity [60], alleviate spot losses with spectrum diversity [53], and improve mobility management through fast handoff [8]. Despite such compelling advantages, however, software developers cannot assume all target devices will feature multiple radios.

VirtualWiFi seeks to provide these benefits with one radio [21]. It virtualizes a single wireless interface, multiplexing it across a number of different end points. While promising, this work remains incomplete. Switching times, even with chipsets supporting software MAC layers, are at least 25 ms. This may still be too high for many potential multi-interface applications. Furthermore, VirtualWiFi's API can be cumbersome, exposing the multiplexed interfaces at the application layer. This forces the application to explicitly manage networks that come and go, complicating applications whether they can benefit from this functionality or not. Finally, VirtualWiFi has primarily been applied to point-to-point, ad hoc communication [1, 9, 23]. The benefit of such techniques when clients are communicating with Internet destinations over infrastructure APs is still unclear.

FatVAP [41] somewhat hides the complexity of multiple virtual networks from users, unlike VirtualWifi. This system connects simultaneously to many Wifi access points, and bundles the bandwidth provided into one logical connection. Applications simply send and receive packets as normal, and the operating system handles the details of multiplexing connections across different links. As we will see in this chapter, however, data striping is not the only useful application enabled by the capability to connect to many networks at once.

This chapter presents Juggler, a refinement of VirtualWiFi's virtual network scheme. It provides switching times of approximately 3 ms, and less than 400 μs when switching between endpoints on the same channel. Juggler provides a single network interface to applications that desire such simplicity, but provides a mechanism for applications to manage connectivity explicitly if they can benefit from doing so.

I present the design and implementation of Juggler, with a prototype built in the Linux 2.6 kernel. Juggler is able to multiplex across infrastructure base stations, ad hoc peers or mesh networks, and a passive beacon-listening mode with minimal delay. Juggler is implemented as a stand-alone kernel module, together with a user-level daemon, `jugglerd`. The latter manages the configuration of multiple endpoints and the transmission schedule across them, making experimentation easy.

The bulk of this chapter evaluates this prototype across a variety of benchmarks, exploring the benefits and drawbacks of virtual interfaces in wireless networks for three

different scenarios. The first, AP handoff, demonstrates that by devoting only 10% of the wireless duty cycle to AP scanning, a client can switch APs within tens of milliseconds of detecting lost connectivity. Importantly, this 10% duty cycle loss reduces foreground transfer throughput by only a few percent.

The second scenario explores the degree to which various applications can exploit data striping and bandwidth aggregation. I evaluate three applications—a multi-threaded file transfer, a streaming video application, and a peer-to-peer file sharing client. Typically, these applications benefit most when the bandwidth on the wireless side of the AP is significantly higher than the back-end, wired side. For example, the file sharing client obtains benefit through data striping up to back-end bandwidths of 2.4 Mbps—a typical rate for private broadband access.

The final scenario demonstrates Juggler's ability to support a small side channel for ad hoc connections to nearby peers without interrupting primary flows to the infrastructure APs. The TCP throughput offered by this scheme is relatively low, due primarily to time-outs induced by the short ad hoc duty cycle. Nevertheless, the achieved rate of 320 Kbps for a 10% share of a 4 Mbps connection is reasonable for many opportunistic applications.

4.1 Contributions

This chapter makes the following contributions:

- First, I show how Juggler significantly improves on the switching overhead of VirtualWiFi by tightly integrating with the lowest layers of the OS software stack.
- Second, I introduce Juggler's novel interface to the upper layers of the software stack. Unlike other extant virtual link layers, Juggler hides the complexity of multiple wireless networks by presenting an unchanging, static network interface to the network layer and up. This removes the burden of dealing with this complexity from users and application designers.
- Third, I present evaluation results of the Juggler prototype implementation in use in three application scenarios of importance and interest to mobile devices.

4.2 Background

Juggler's design is based on VirtualWiFi [21]. This system maintains a set of virtual networks that are each active on the WiFi radio in turn. When a virtual network is not active, any outbound packets are buffered for delivery the next time the network is activated. Switching from one AP or ad hoc network to the next involves updating such wireless parameters as the SSID, BSSID (station MAC address), and radio frequency on the wireless card.

Most current WiFi cards perform the IEEE 802.11 protocol in firmware rather than a software device driver. The problem is that hardware designers and firmware authors did not envision a scenario where it would be advantageous to change the radio frequency or SSID every 100 ms. The firmware of such legacy cards often performs a card reset when changing certain wireless parameters.

VirtualWiFi reduced switching time from three or four seconds to 170 ms by suppressing the media connect/disconnect messages that wireless cards generate when these parameters are changed. Otherwise, these notifications cause upper layers of the networking stack to believe that the network interface is briefly disabled, and no data can flow for several seconds.

They further reduced switching time to 25 or 30 ms when *Native WiFi* cards were used. These are cards that perform the MAC layer in software, not on the card itself. The software device driver can therefore be modified to perform only those operations that are necessary, and omit any wasteful firmware resets. The Native WiFi cards used in the evaluation of VirtualWiFi still performed the 802.11 association procedure automatically—in firmware—whenever the network was rotated.

Juggler uses wireless cards that implement the MAC layer entirely in the device driver. This lets it suppress the association process to further reduce switching time. When Juggler first communicates with an AP, it must perform the slow 802.11 association sequence in order to make itself known to the AP. Subsequently, Juggler only associates to an AP again if it receives an explicit 802.11 deauthentication message. This may occur if Juggler fails to respond to too many ACKs because it was tuned to a different radio frequency.

Another problem when connecting to multiple networks simultaneously is that packets destined for the device may arrive at an AP while the WiFi radio is communicating with a different AP or ad hoc peer. Because the first AP does not know this, it will transmit data but the client's radio will not detect the packets because it is tuned to a different channel.

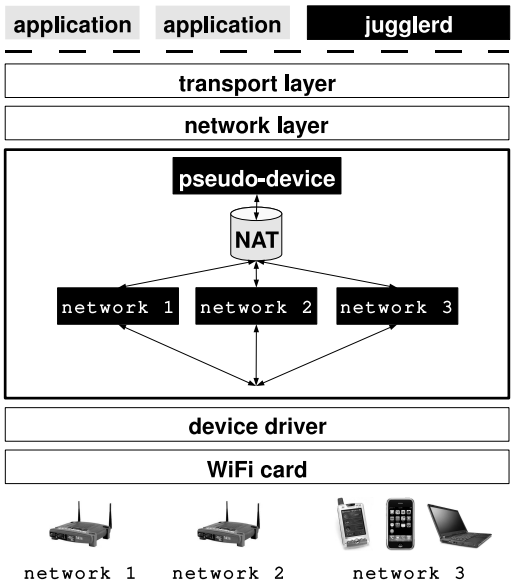
VirtualWiFi uses the 802.11 power saving mode (PSM) to coerce APs into buffering downstream packets intended for the client while the client is communicating with another AP or peer. In standard PSM operation, a client is connected to one base station but periodically deactivates its WiFi interface to conserve power. Before turning off the WiFi radio, the client sends a null IEEE 802.11 frame to the base station, with a PSM mode bit set. At a fixed frequency, the client reactivates its radio and listens passively for the AP's beacon frame. One field of the beacon—a Traffic Indicator Map (TIM)—indicates which of the many clients connected to the AP have buffered packets waiting for them. Clients are uniquely identified by an association ID (AID) previously received as part of the 802.11 association process.

If the client finds it has no buffered packets waiting, it deactivates its radio until the next timeout. But if data is pending, the client transmits a special PSPOLL frame to the access point. The AP then transmits the first buffered packet to the client. Each packet received by the client has a bit in the 802.11 header indicating if there are yet more packets buffered on the AP. The client continues to transmit PSPOLL packets until all buffered data has been retrieved.

Downstream packet buffering was described in the original VirtualWiFi paper, and subsequently implemented in follow-up work [1]. I also have implemented this technique in the Juggler prototype.

4.3 Juggler

Figure 4.1 illustrates a standard network stack, modified to include Juggler. Rather than force all applications to explicitly bind their data flows to specific access points [21, 78], Juggler presents a single, unchanging network interface to upper layers of the stack. This pseudo-device impersonates a wired Ethernet interface with a static, private IP address. All data flows are bound to this network interface and IP address.



One unchanging network interface is visible to upper layers of the network stack and to applications. Juggler maintains connection parameters (SSID, frequency, DHCP configuration) for each virtual network with which it is associated, assigns sockets to virtual networks, buffers packets destined for inactive networks, and performs network address translation (NAT) between internal and external IP addresses.

Figure 4.1: Juggler network stack

The system consists of two main parts. The first is an in-kernel component that sits between the network and link layers of the OS networking stack. The second is an application-level, privileged process that handles access point discovery and configuration.

Juggler can connect to 802.11 ad hoc or mesh networks as well as infrastructure access points. I use the general term *virtual network* in the remainder of this paper to refer the configuration for either an infrastructure AP or an ad hoc group. For every configured virtual network, Juggler tracks the following state:

- Network type (infrastructure or ad hoc)
- SSID
- MAC address (BSSID)
- Frequency (channel number)
- IP address, default gateway, netmask, DNS server(s)
- An outbound packet queue

- An ARP cache
- Radio duty cycle fraction that the network is active

Data flows are distributed among virtual networks at the granularity of the socket abstraction. A process can therefore "stripe" data across many virtual networks by creating multiple sockets with appropriate options, but all packets belonging to a given socket are always transmitted over the same AP. I made this design decision to preserve the semantic definition of a socket endpoint as an (IP address, port) pair. This preserves communication with unmodified end hosts that need not even be aware that a client is using Juggler.

4.3.1 Assigning Flows to Networks

Juggler was designed with flexibility and ease of use as primary concerns. Applications need not specify which virtual network should handle a given data flow, but they are provided with a simple interface to do so if desired. After creating a socket, applications may set a new socket option with the MAC address of a preferred network. This is analogous to using the `SO_BINDTODEVICE` socket option to bind a socket to an interface when multiple NICs are available.

When Juggler receives data for a previously unseen socket from the network layer, it assigns the socket to a virtual network. If a preferred network's MAC address was previously set via the new socket option, the socket is assigned to that virtual network. Otherwise, Juggler simply assigns it to whichever network is currently active on the WiFi radio.

Thus, a data flow created without specifying an AP preference is pseudo-randomly assigned to one of the active virtual networks. My ongoing work examines how Juggler can handle this matchmaking in a more intuitive fashion. I intend to add a socket option so applications can specify the general properties of a data flow (e.g. background bulk transfer, interactive session). Juggler will then match these needs with the connection quality of different virtual networks, probed in a fashion similar as my Virgil AP selection daemon 2.

4.3.2 Sending and Receiving Packets

As illustrated in Figure 4.1, upper layers of the network stack see only one network device. This pseudo-device emulates a wired Ethernet interface, with an IP address in the private address range. All sockets are bound to this interface and IP address when they are created.

It is critical that Juggler maintain a unique ARP cache for each virtual network, bypassing the system-wide ARP cache completely. IP address namespaces of different virtual networks may collide because access points commonly use network address translation (NAT) to share a wired link and assign IP addresses from private blocks. If Juggler relied on the system-wide ARP cache instead, this cache would need to be flushed constantly because, for example, different hosts connected to different APs might be assigned the address 192.168.1.1 but have different MAC addresses.

This pseudo-device is implemented by the kernel component of Juggler. All outbound data flows therefore pass through Juggler before reaching the WiFi device driver. Handling an outbound data packet is a four-stage process:

1) Determine the owning virtual network If this is the first time data has been seen on this socket, assign the flow to a virtual network.

2) Construct the Ethernet header If the destination IP address falls inside the subnet, as determined by the virtual network's assigned IP address and netmask, then get the destination MAC address from the network's ARP cache. Otherwise, use the MAC address of the default gateway.

3) Network address translation Because all sockets are bound to the internal pseudo-device, packets received from the network layer will have their IP source address set to the internal IP address. The different virtual networks have different external IP addresses, however, that were either assigned to them by a DHCP server running on an access point, or statically configured. Juggler therefore rewrites the IP and transport-layer headers as needed to reflect the real source IP address.

4) Forward for transmission If the virtual network that owns this socket is currently active on the WiFi radio, Juggler immediately passes the packet to the WiFi device driver for transmission. This is done through the same interface that the network layer would use to contact the device driver if Juggler were not installed. The device driver therefore thinks the packet has arrived from the network layer, and proceeds as expected. If the virtual network that owns the socket is not active, however, the packet is enqueued in the virtual network's outbound packet queue.

When constructing an Ethernet header, Juggler may not find the MAC address it needs in the virtual network's ARP cache. In that case, Juggler first enqueues the outbound data packet in the virtual network's outbound packet queue. Next, it constructs an ARP request for that IP address and broadcasts the request when the virtual network is next active on the WiFi radio. Once the device owning that IP address responds with its MAC address, Juggler adds the mapping to the virtual network's ARP cache, and continues with the outbound transmission of the original data packet.

Receiving data packets is easier than sending. Juggler simply performs NAT to translate the destination IP address in the packet to that of the internal pseudo-device and forwards the packet up to the network layer.

4.3.3 Switching Between Virtual Networks

Each active virtual network is allotted an adjustable fraction of the radio's duty cycle. Virtual networks are active in a round-robin fashion, each for their configured time. After activating a given virtual network, Juggler sets a kernel timer to be invoked again once the new network's timeslice has expired. Thus, Juggler need not run at a constant frequency but only when needed to switch to the next virtual network.

Switching the WiFi radio from one AP or ad hoc network to the next is a multi-stage process. First, Juggler coerces the current access point into buffering packets destined for the client while the radio is communicating with another virtual network. This is done by transmitting a null IEEE 802.11 frame with the power-save mode (PSM) bit set, indicating that the client is entering PSM.

Next, Juggler updates the radio's wireless parameters via the device driver. If the next virtual network is not on the same channel as the previous one, the radio frequency must be modified. Juggler updates the SSID and MAC address to that of the new AP or ad hoc group, and updates the mode (infrastructure or ad hoc) and/or encryption parameters if these have changed.

If this is the first time the virtual network has been activated—because it was just added—or if Juggler has recently received a deauthentication message from the access point, Juggler must force the WiFi device driver to perform the entire association process in order to obtain an association ID.

Juggler then transmits a power-save poll (PSPOLL) frame to the new AP, indicating the client returned from its (fake) power-save mode. If the AP has any enqueued packets destined for the client, it transmits the first one. Juggler sends PSPOLL packets until all enqueued packets have been received. Finally, Juggler transmits any outbound packets that were enqueued for this virtual network when it was previously inactive.

In addition to infrastructure APs and ad hoc networks, Juggler recognizes a third, special type of network: a scanning slot. When this virtual network comes up in the rotation, Juggler simply sets the link status of the WiFi card to unlinked (to passively listen for beacons) and changes the frequency of the WiFi radio. Each time the scanning slot is scheduled Juggler listens on a different frequency, so that the entire channel space is eventually searched. In our current implementation, Juggler rotates among the three non-overlapping channels 1, 6, and 11.

4.3.4 User-level Daemon

A user-level process, `jugglerd`, is responsible for general configuration of the Juggler kernel module. The two communicate via the `/proc` filesystem in Linux. To add a virtual network to the rotation, `jugglerd` sends the kernel module the following information:

- Mode (infrastructure, ad hoc, or scanning slot)
- SSID and MAC address of the AP or ad hoc group
- Channel number

When the kernel module receives the request, it creates a virtual network structure (containing the outbound packet queue, ARP queue, et cetera) and adds the new network to the end of the round-robin rotation. The new network is assigned the default timeslice duration—100 milliseconds. If the new network is an infrastructure AP, Juggler will perform the slow 802.11 association the first time the network is activated.

Optionally, `jugglerd` can include an IP configuration (address, netmask, default gateway, and DNS server) all at once with the network add request, or update those values at a later time. No data flows will be assigned to a virtual network until its network layer parameters have been configured.

To delete a virtual network, `jugglerd` simply writes the network's MAC address to another `/proc` file. If the network is currently active, Juggler pre-emptively switches to the next network before deleting the network's state.

To adjust the relative timeslices of active virtual networks, `jugglerd` writes network MAC addresses, and a relative weight for each, to the kernel. These weights are interpreted as multiples of the current default switching timeout. For example, consider the case where two APs are active and the default switching timeout is 100 ms. To give AP1 90% of the radio duty cycle and AP2 10%, `jugglerd` would give AP1 a weight of 9 and AP2 a weight of 1. Because the default timeout was 100 ms, AP1 would then be active for 900 ms, followed by 100 ms of AP2, then 900 ms of AP1. The default switching timeout is also configurable at runtime, allowing `jugglerd` to assign a radio schedule of desired granularity.

4.3.5 Implementation Details

The vast majority of the Juggler kernel code is a standalone kernel module. A small kernel patch was required for two reasons. First, I modified the `socket()` system call to automatically bind all sockets to the pseudo-device created by Juggler in order to capture all outbound flows. Second, network device drivers forward inbound data packets up to the network layer by calling a well-known function (`netif_rx()`). To allow Juggler to perform inbound NAT processing, I added one line to `netif_rx()` that calls Juggler's inbound NAT function before performing network-layer processing.

I used WiFi cards with the Realtek 8185 chipset for development and testing. This chipset performs all MAC-layer functions in software, letting Juggler optimize the repeated switching process. I used the open-source `rtl-wifi` driver[1], which leverages the common Linux `ieee80211` software MAC layer.

To encourage future portability, I made as few changes to the `rtl-wifi` driver and `ieee80211` MAC layer as possible. The `ieee80211` layer maintains one large global structure containing information on the currently-associated AP—channel number, SSID, association ID, sequence numbers, et cetera. Juggler stores a copy of this global structure for each active network in a linked list, and each time Juggler switches between networks it updates `ieee80211` to point to the correct version of the structure.

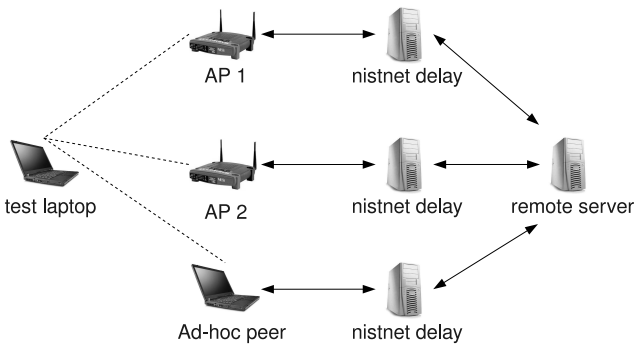
In the `rtl-wifi` driver, I modified the overly-cautious delay imposed whenever the device driver writes a value to the card over the PCI bus. For example, changing the radio frequency requires 6 sequential writes to the card. By default, the driver waits 5 ms between each write to allow the PCI bus to stabilize. I was able to reduce this delay to 500 μs without problem. Thus, Juggler can switch the radio frequency in $6 \times 500\mu s = 3$ ms rather than 30 ms.

4.4 Experimental Setup

Before evaluating Juggler, one must consider what sort of usage environment is intended to be modelled. Previous evaluations of virtual link layers focused primarily on communicating with peers over ad hoc, point-to-point links [9, 21, 23]. Throughput in such situations is limited by the 802.11 link speed (e.g. 10 or 54 Mbps) and interference on the wireless channel.

I am focused on mobile devices that primarily communicate with remote Internet destinations, by means of wireless access points where wireless bandwidth outstrips that of the AP's back-end connection. This is certainly the case when the back-haul link is a DSL line or cable modem, typical for residential settings, coffee houses, and other opportunistic public connectivity. Note that this assumption may be invalid on corporate or academic campuses where APs may connect directly to Gigabit Ethernet networks.

[1] `http://rtl-wifi.sourceforge.net/`



A test laptop running Juggler can connect wirelessly to one of two 802.11g access points, or to another laptop in ad hoc mode. Two gateway routers use NIST Net to selectively throttle the bandwidth between each AP and the remote server. This allows simulation of varied link quality between the test laptop and a remote server across the Internet. In real usage, network bandwidth is dependent both on wireless link characteristics and the quality of an AP's backplane link to the Internet.

Figure 4.2: Laboratory setup

Figure 4.2 illustrates the test setup in the laboratory. The test laptop at left represents a mobile client with one WiFi network card. I configured two Linksys WRT54G 802.11g access points, on disjoint channels (1 and 11) and different subnets (192.168.0.x and 192.168.1.x). A second laptop was also present to act as an ad hoc peer for certain experiments. The remote server at the far right represents an arbitrary Internet end host with which the mobile client wishes to communicate. This machine was configured with a static IP address of 192.168.2.5, outside either AP's subnet.

As illustrated in Figure 4.2, APs and the remote server were connected by gateways. The gateways used IP forwarding and NAT to forward packets from each access point's subnet to the subnet (192.168.2.x) of the remote server. To evaluate the effect of different back-haul bandwidths from APs to remote Internet hosts, I installed NIST Net [19] on all gateway machines. NIST Net configures a Linux host to act as a router, delaying or dropping packets to shape flows to a desired bandwidth or emulate a given loss rate.

4.5 Microbenchmarks

Juggler works by interposing between the network layer of the Linux kernel and the link layer functionality provided by the WiFi interface's device driver and the `ieee80211`

	mean	std. err
switch (different channel)	3.328 ms	0.021 ms
switch (same channel)	0.381 ms	0.011 ms
process ingress packet	284.0 cycles	1.6 cycles
process egress packet	6975.3 cycles	39.2 cycles

Table 4.1: Juggler, CPU overhead benchmarks

software MAC layer. To quantify the overhead this introduces, I instrumented Juggler to measure the overhead imposed for (1) switching from one virtual network to the next, and (2) performing network address translation (NAT) on ingress and egress data packets.

The minimum resolution of a standard kernel timer in Linux depends on the frequency with which the scheduler timer fires. For the Linux 2.6 kernel, this time interval—known as a *jiffy*—is 4 ms. This is clearly too coarse-grained when we want to time operations that occur in microsecond timeframes. Instead, I use an x86 assembly language instruction, `rdtsc`, which reads the current value of the processor timestamp counter. This counter holds the number of processor cycles executed since the processor was last reset. By wrapping a set of instructions with calls to `rdtsc`, one can estimate the number of CPU cycles that elapsed in the interim.

To ensure reliable results, prior to benchmarking I disabled the second processor in the multi-core CPU of the test laptop, and disabled CPU frequency scaling as to ensure a constant conversion rate between CPU cycles and time. The test machine contained an Intel Core 2 Duo CPU, 1.79 GHz per core. The wireless network interface was a CardBus adapter based on the Realtek 8185 chipset.

I loaded Juggler, connected to two different APs on different channels, and recorded the time required to switch networks over 10000 times. I next repeated the experiment while connected to two APs that share the same channel. As Table 4.1 shows, the time to switch the radio's frequency clearly dominates switching time.

This switching time of just over 3 ms allows very fine-grained multiplexing of virtual networks. Even when switching as often as every 100 ms, only 3.3% of each usage period would be lost to overhead. VirtualWiFi's best cited switching time was was 25 ms, resulting in 25% overhead for the same switching frequency.

As discussed above, changing the radio frequency on the Realtek chipset requires six sequential writes to the interface over the PCI bus. The driver must pause in between each write to allow the PCI bus to stabilize. I set this timeout as 500 μs, for a total of 3 ms delay to make six writes. It was not possible to lower this timeout much below 500 μs without degraded performance and lost data.

I also examined inbound and outbound packet processing overhead. The most heavy-weight operation is rewriting network-layer headers to perform NAT to and from the internal IP address of the pseudo network device. The results in Table 4.1 have been left in units of cycles due to their extremely small size. The overhead required is clearly minimal. Note that this does not account for packet queuing delay when an outbound packet is destined for a virtual network that is currently inactive. I was interested here in the CPU overhead imposed by the presence of Juggler in the critical path of the network stack.

4.6 Application Scenarios

The primary contribution of this chapter is the exploration of several realistic usage scenarios where the ability to multitask one wireless interface is beneficial. In this section, I apply Juggler to three application domains: (1) soft handoff between WiFi APs, (2) data striping and bandwidth aggregation, and (3) mesh and ad hoc connectivity.

I use NIST Net as described above to simulate different network conditions on the link between a wireless AP and the Internet core. During real usage, the bandwidth and latency a mobile device experiences when communicating with a given remote destination depends on several factors:

- Properties of the wireless link (interference, link speed)
- Congestion from many clients sharing one AP
- Quality of the AP's wired back-haul link to its ISP
- Network core congestion
- Edge delays in the destination network

Residential broadband providers promise fairly high data rates. In the United States, for example, SBC advertises DSL links of 384 to 768 Kbps upstream and 768 to 6144 Kbps down, while Comcast claims the same upstream bandwidth and 4096 to 8192 Kbps downstream over a cable modem. Verizon's FiOS fiber optic service is even faster—on the order of 10 or 20 Mbps.

These are theoretical maximum rates, however, from the client to the service provider's edge network, not through the network core. As shown in Chapter 2 above, in real public deployments the bandwidth achievable by an application-level TCP flow is far lower—typically several hundred Kbps. Independent measurements of broadband connectivity quality support these results [42].

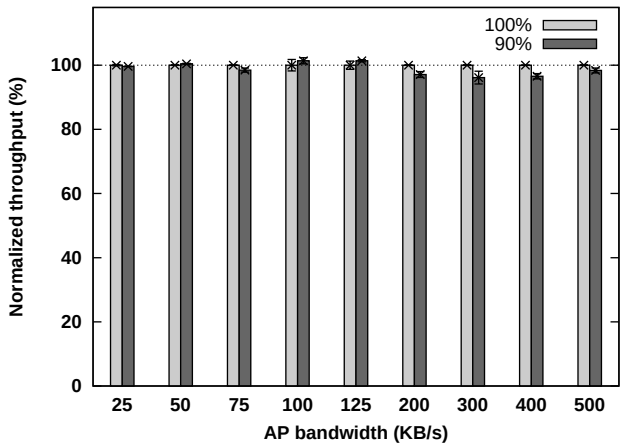
For all figures in the remainder of this section, error bars represent $\pm$ the standard error of the mean ($\sigma/\sqrt{n}$).

4.6.1 Soft Handoff

Handoff between WiFi APs is far from seamless. The IEEE 802.11 protocol requires a time-consuming association and authentication process be completed before a client can communicate with an access point. If a device can only be connected to one AP at a time, migrating to a new AP requires a significant gap between the time data was last sent over the previous AP and association to the new AP is complete. This overhead can be reduced by either requiring two physical radios or modifying AP firmware [22, 63]. Once associated to a new AP, however, the client must often configure IP-layer settings through DHCP before any useful data can flow.

Ideally, WiFi handoff would be as seamless as the handoff a mobile phone makes from one GSM tower to the next. Such fluid transfers would be possible if, *before* the current AP becomes unusable, the device (1) knew which AP it will use next, (2) had already completed association, and, (3) had already received a DHCP configuration (if applicable).

In this section, I use Juggler to do just this. Juggler uses variable timeslicing to assign 90% of the radio's duty cycle to the current "primary" AP. This is the highest-quality AP at the device's current location. The remaining 10% of radio cycles are devoted to scanning for new APs and maintaining association with one or more secondary APs.



90% of the radio's activity period is devoted to a "primary" AP that handles all data flows, while 10% is used to discover new APs and maintain association with the backup AP(s). The reduction in primary bandwidth is small despite the loss of 10% of the radio duty cycle.

Figure 4.3: Soft handoff, throughput of primary AP

While the device is using the primary AP to transfer data, Juggler scans for new APs in the background, and pre-emptively associates with them and obtains DHCP leases. The user-level Juggler daemon probes the application-visible quality of newly-discovered APs using techniques adapted from Virgil. Low-quality APs are dropped, and high-quality ones are assigned a small portion of the 10% background slice in order to maintain association. When the primary AP later becomes unusable or its signal fades, Juggler promotes the best secondary AP to be the new primary.

First, I wanted to ensure that reducing the primary AP's radio slice from 100% (without Juggler) to 90% would not adversely impact foreground data traffic. I used a simple TCP client and server to transfer data from the test laptop, through one AP, to a remote server representing an Internet host. As illustrated in Figure 4.2, the gateway machines between each AP and the remote server made it possible to simulate a range of bandwidths. Figure 4.3 plots TCP throughput as a function of AP bandwidth between the client and a remote Internet server. For each bandwidth value, there are two data series: 100% (entire radio devoted to one AP) and 90% (radio split between the AP at 90% and background scanning at 10%). The results show that reserving 10% of the WiFi radio's duty cycle for background tasks has a negligible impact on foreground data throughput.

	mean	standard error
Association	1.071	0.167
DHCP	1.817	0.191
Failover time	1.008	0.055
Socket timeout	1	—

Juggler listened for AP beacons for 10% of the duty cycle. *Association* is the total elapsed time from when the new AP began broadcasting beacons until Juggler finished associating with it. *DHCP* is the total elapsed time to obtain a network configuration via DHCP. *Failover time* is the total elapsed time from when the primary link was deactivated to when the remote server received the next packet in the data flow (over the new AP). *Socket timeout* is the minimum time required to detect failure of the primary AP. All times in seconds, 20 trials.

Table 4.2: Soft handoff, discovery and fail-over

I next quantified how quickly Juggler can discover and configure new access points. The client connected to a primary AP with 90% duty cycle, and the remaining 10% was allotted for background scanning. I then activated a new access point on a different channel from the primary AP, and measured the time between when the new AP began broadcasting beacons and the client completed the 802.11 association process. Table 4.2 shows that on average, Juggler discovered and associated with the new AP within one second of its introduction to the environment. The second row of Table 4.2 is the time required for the client to obtain a DHCP configuration from the new AP, after the association process is completed. This takes just under two seconds due to the connectionless nature of DHCP (atop UDP) and the fact that the background discovery operations are limited to only 10% of the radio cycles.

Finally, I examined how quickly Juggler could perform soft handoff from one AP to the next. A simple user-level process transferred data bi-directionally over TCP with the remote server as fast as possible over the current primary AP at 90% timeslice. The secondary AP was already configured and associated at a 5% timeslice, with scanning and discovery allocated the last 5%. I then deactivated the link of the primary AP. The user-level process detected this failure through the standard TCP socket timeouts (`SO_SNDTIMEO`,`SO_RCVTIMEO`). I set these timeouts to one second for this evaluation. After detecting a socket timeout, the user-level process requested that Juggler fail over to the secondary AP, and then resumed the data transfer.

I measured the total time elapsed between when the primary AP deactivated its link and the remote server received a new TCP connection, signalling the resumption of data transfer. As Table 4.2 shows, the total time the data transfer lapsed is just slightly longer than the socket timeout value—on average, 8 ms longer. This is roughly the time required for one round-trip between the client and server in the laboratory, to establish a new TCP connection. It is clear that if the link failure of the primary AP could be detected more quickly then the response would be even faster. There is a tension, however, between the sensitivity of this detection and the false positive rate. Even this gap of one second is usable, however, for such real-time applications as Internet telephony and video streaming.

4.6.2 Data Striping and Bandwidth Aggregation

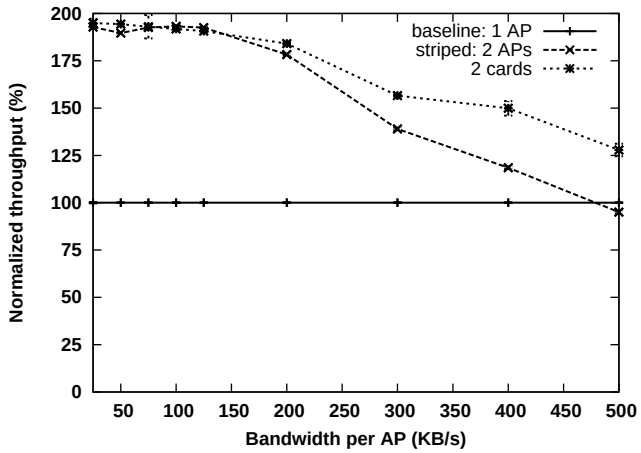
Outside of corporate and campus settings, bandwidth to Internet hosts via a wireless AP is rarely constrained by the 802.11 link rate. Rather, it depends on the quality of the AP's back-end link (e.g. DSL, cable modem), congestion on the AP, or interference. A wireless radio that transmits at 10 or 54 Mbps can often push bits into the network faster than the AP can forward them.

Striping is a well-known technique for improving throughput by breaking one logical flow into multiple chunks, which are then transmitted in parallel over different paths. Prior work has shown the efficacy of this technique when multiple network interfaces are present on a device [60, 61, 64]. This section explores how well Juggler lets applications and users enjoy the benefits of striping while avoiding the costs of multiple network interfaces.

I first quantified how the throughput improvement gained by striping is affected by the bandwidth available through each access point. Next, I simulated the behavior of a video streaming client that had been modified to fetch video frames over multiple APs. Finally, I modified *KTorrent*, a popular BitTorrent client, to stripe data torrent downloads across multiple access points.

Throughput Improvement

Recall the laboratory setup shown in Figure 4.2. I used a simple TCP client on the test laptop to repeatedly download a 10 MB file from the remote server. For the baseline



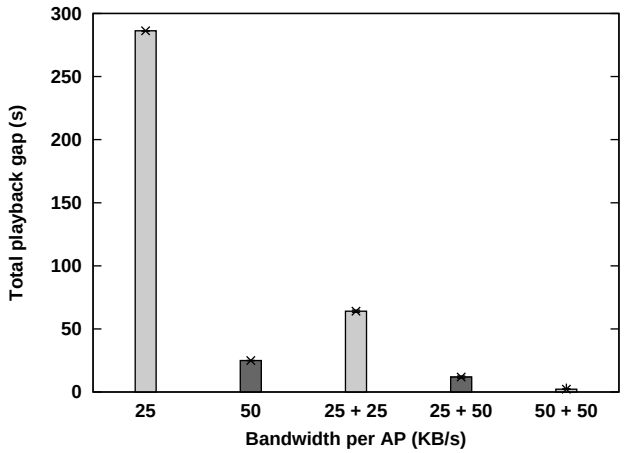
For various AP bandwidths, the client downloaded a 10 MB file from a remote server. The "baseline" case was the device associated with one AP, using one TCP socket for the download. The "striped" case was Juggler associated with two APs simultaneously, 50% duty cycle for each AP, and using two TCP sockets to each download 1/2 of the file over a different AP. "2 cards" used two physical WiFi cards, associated with different APs, each downloading 1/2 of the file over a TCP socket in parallel.

Figure 4.4: Data striping, throughput improvement

case, the client used one AP exclusively to transfer the entire file over one TCP socket connection. For the second case, Juggler associated simultaneously with both APs, each with 50% duty cycle, switching between APs every 100 ms. The client made one request for each half of the file using two threads, assigning each thread to a different AP by setting the new Juggler socket option. The multithreaded server then sent each half of the file in parallel. The third case—two cards—was the same, except that the client used two physical WiFi cards instead of having Juggler share one radio. Each card was associated with a different access point, and each of the two sockets bound to a different interface (using the `SO_BINDTODEVICE` socket option).

The remote server in the lab configuration represents an arbitrary Internet destination. By using the gateways lying between each AP and the remote server to throttle bi-directional bandwidth, I explored a range of application-level bandwidths between the TCP client and server. I repeated each case for the range of AP bandwidths. The throttled bandwidths for each AP were always equal and changed together.

Figure 4.4 shows the mean throughput achieved during the download as a function of available bandwidth on each AP. All values are normalized to those of the one card, single



Times in seconds. Video length was 204.8 s (10 MB encoded at 400 Kbps). Series labels refer to bandwidth available through the AP(s) over which the video was streamed. For instance, *25 + 50* means the client was connected to two APs at once, one of which had 25 KB/s of bandwidth, the other 50 KB/s.

Figure 4.5: Streaming video, total playback gap per run

AP case. For modest AP bandwidths, striping using Juggler results in the same throughput as using two physical radios—close to the theoretical speed-up limit. As AP bandwidth increases, gains from striping decrease more rapidly for Juggler than for the two radio case. However, striping over Juggler is still beneficial until AP bandwidth reaches approximately 500 KB/s (4 Mbps). This is far higher than upstream data rates for residential broadband, and roughly equal to the downstream quality over cable or DSL links under ideal conditions.

Streaming Video

Unlike simple bulk downloading, streaming video is concerned with *when* specific parts of the video are downloaded. Blocks toward the beginning of the file will be needed earlier, because the purpose of the application is to allow the user to watch the beginning of the video while further content is still being transferred. A simulator modeled a simple video player that uses an earliest deadline first policy to chose which block to download next. This experiment assumes all blocks are of uniform size, a condition that may not hold true for some real-world video encoding schemes. The TCP streaming client creates one thread per available AP and each thread downloads the earliest unfetched block. For

example, if there were two threads downloading at the same rate, downloading the earliest unfetched block should have the effect of assigning one thread all the even-numbered blocks and the other all the odd-numbered blocks. However, if the APs have any asymmetry in available bandwidth, this scheme may not minimize the finish time of each block. To compensate for any asymmetry in the available bandwidth at each AP, each thread tracks which block it previously downloaded and subtracts the next block number to download from the number of the previously-downloaded block to obtain a "delta". In the symmetric case, each delta should be two—the current thread just downloaded one block and in that time, the other thread downloaded one block. If delta is greater than two, the thread's AP must be slower than the other thread's AP, so we download the block that is delta blocks after the earliest unfetched block to compensate.

Streaming video clients typically buffer data to compensate for transient fluctuations in available bandwidth. If the buffer is emptied during playback, clients stop playing video until the buffer is again filled. However, buffering and displaying video to the user do not affect the optimal assignment of blocks to APs, so the simulator simply emulated the network behavior of the client, recorded the finish times of each block, and *post facto* calculated the time spent buffering. This calculation derives a deadline for each block from the video bitrate and block size, taking into account the fact that the buffer is filled before the video begins playing. If a block misses its deadline, video playback stops, and the time to refill the buffer is added to the total buffering time.

For this experiment, the simulated video client repeatedly streamed a 10 MB video—encoded at a bitrate of 400 Kbps—from the remote server. This filesize and encoding rate corresponds to 204.8 seconds of simulated video. The client block size was 16 KB. For the first baseline case, the client used one AP exclusively with only 25 KB/s bandwidth to the server available to transfer blocks. As a second baseline, I repeated the baseline, but increased the available bandwidth to 50 KB/s. For the striping cases, the client used Juggler to associate simultaneously with both APs, for various combinations of AP bandwidth. The server from Section 5.2.1 was reused, as it simply responds to requests for a number of bytes at a given offset in a file.

Figure 4.5 shows the results. Note that the video encoding rate of 400 Kbps is equivalent to 50 KB/s. For the first case, where the available bandwidth is only half the video bitrate, the total playback gap is nearly 300 s. This is not merely a case of a long up-front buffering time. I calculated the average size of playback gaps and the period in between gaps—during which time the video is playing. For the case of one AP at 25 KB/s, the average gap size (6.091 seconds) is larger than the average inter-gap period (4.931 s). This results in an incredibly poor user experience, because the video is constantly starting and stopping.

When the single AP bandwidth is increased to 50 KB/s, one sees a small glitch here and there but overall the video player is able to stream the video with one-tenth the wait time. The third case attempts to aggregate two 25 KB/s links into a logical 50 KB/s stream. This lowered wait time by a factor of four over the single AP, 25 KB/s case, though the buffering time was still three times that of using one AP at 50 KB/s. Using one 25 KB/s AP and one 50 KB/s AP nearly eliminates all wait time. Finally, streaming over two APs, each offering 50 KB/s bandwidth, avoids wait time completely for 95% of the test runs.

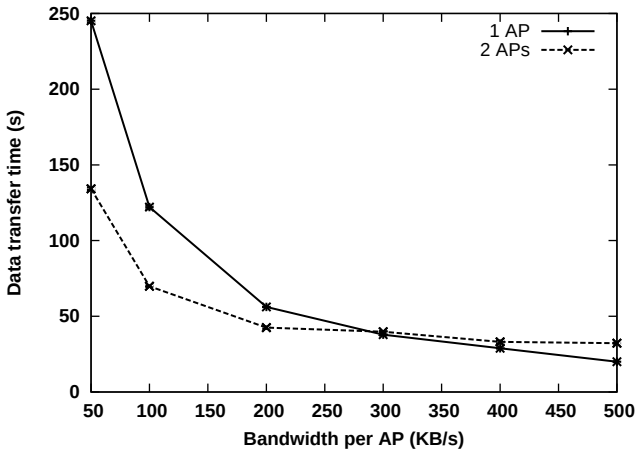
BitTorrent

BitTorrent is a popular peer-to-peer file transfer protocol. A given file is broken into equal-sized chunks, and clients fetch a file by downloading a unique subset of chunks from different peers that are *seeding* the same file. This portion of the evaluation uses a modified KTorrent 2.4[2], a popular open-source BitTorrent client, to evaluate the usefulness of striping a torrent download across multiple APs.

This case closely resembles the striping results in Section 4.6.2. Because KTorrent opens one socket per peer and uses wrapper libraries to hide the socket interface, data is striped by assigning peers to each AP evenly. As stated in Section 4.3.1, obviating the need for developers to bind flows to APs explicitly is future work. The torrent was a 10 MB file seeded on a 2.8 GHz Pentium 4 and a 550 MHz Pentium III Xeon, both running Debian "lenny"[3] with Linux kernel version 2.6.22. The client used on both seed machines

[2] `http://www.ktorrent.org/`

[3] `http://www.debian.org/releases/lenny/`



For both cases, blocks were downloaded from both seed servers. 10 MB data file.

Figure 4.6: BitTorrent, torrent download time

was the official BitTorrent client version 3.4.2, packaged with Debian[4]. The Pentium 4 seed also ran the tracker for the torrent.

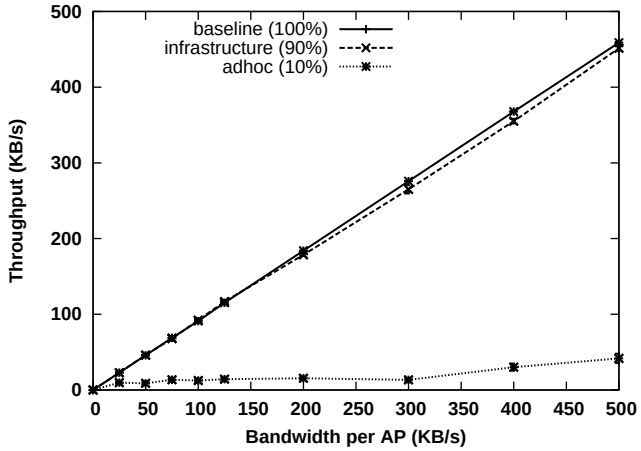
The baseline case used KTorrent to download the 10 MB torrent over a single AP. The second case used the modified KTorrent client to stripe the data at peer granularity, as described above, and used Juggler to associate with two APs simultaneously at 100 ms switching granularity with 50% duty cycle each.

Figure 4.6 shows the results. As before, when bandwidth between the client and remote peer is poor, Juggler downloads the file over 1.75 times faster than when using a single access point. However, BitTorrent performance degrades faster than the simple striping client's performance as the available bandwidth increases. While performing the evaluation, one notices that the application-level BitTorrent protocol takes longer than standard TCP to accelerate to using the full available bandwidth. I attribute the performance gap between these results and the results in Section 4.6.2 to this protocol overhead.

4.6.3 Mesh and Ad Hoc Connectivity

The primary motivation of the original VirtualWiFi work was to let clients be simultaneously connected to an infrastructure AP and to peers in ad hoc mode [21]. Such a

[4] `http://www.bittorrent.com/`



"Baseline" is the maximum TCP throughput from the test server for varied values of effective link bandwidth, when Juggler was not active. I then used Juggler to connect simultaneously to an infrastructure AP (with 90% of the radio) and a nearby device in ad hoc mode (with 10% of the radio). The results show that Juggler can maintain a usable background mesh connection without significantly degrading the quality of the "primary" infrastructure link to the Internet.

Figure 4.7: Mesh connectivity, TCP throughput

side channel is clearly useful for communicating with devices in the user's personal area network (PAN) [6], participating in mesh networks [28], or exploiting physical proximity for reasons of security [10].

VirtualWiFi has been used to create an ad hoc side channel while preserving foreground infrastructure connectivity. *WiFiProfiler* [23] allocates 800 ms to foreground traffic and 500 ms to peer traffic (61.5% to 38.5%). Their results show the penalty on the primary link is modest but non-trivial. Also, only one value of network bandwidth was evaluated—approximately 70 KB/s from Figure 3 of the paper.

Juggler's switching time optimizations allow for a much finer-grained trade off between foreground and background traffic. As for the evaluation of soft handoff, I allocate 90% of the radio's duty cycle to the "primary" virtual network—an infrastructure AP representing the device's connection to the Internet. With the remaining 10% duty cycle, Juggler connected to another test laptop in ad hoc mode on a non-overlapping channel to that of the infrastructure AP. For the experiment, the WiFi radio rotated between the infrastructure AP for 450 ms and the ad hoc peer for 50 ms.

Both laptops had 802.11g cards and communicated on a well-known SSID, with static IP address assignment. Due to interference and link conditions, however, in real situations two ad hoc peers may not be able to communicate at the full 54 Mbps bitrate. I therefore configured the peer laptop as an IP forwarding gateway, connected via its wired Ethernet link to the second NIST Net gateway, which was connected in turn to the remote server. This lets one throttle bandwidth between the ad hoc peers in the same fashion as for infrastructure APs throughout our evaluation, in order to give a more realistic picture of data throughput.

I ran two instances of a simple TCP server on the remote server. The first instance handled connections from the test laptop via the infrastructure AP, passing through the first NIST Net delay router. The second instance handled connections from the test laptop to the peer laptop in ad hoc mode, passing through the second NIST Net router. A TCP client on the test laptop used two threads to download data as fast as possible over both links. I then ran a baseline case, where the test laptop was only connected via the infrastructure AP with 100% of the radio duty cycle.

Figure 4.7 shows negligible throughput difference between using the entire radio capacity and reserving 10% for a side channel, even for high values of AP bandwidth. As expected, the throughput of the 10% ad hoc channel is modest—roughly 40 KB/s for a TCP flow when total AP bandwidth is 500 KB/s. This is due to problems with TCP timeouts because the radio is tuned away from the ad hoc channel for such long periods.

Note that I have throttled the ad hoc bandwidth in order to present a pessimistic estimate of the bandwidth available via that channel. Nonetheless, this side-channel is usable for low-priority background communication between local peers, while foreground throughput is reduced by at most a few percent.

4.7 Chapter Summary

Mobile devices with multiple network interfaces enable many capabilities of interest and value to users. Such benefits, however, are negated by added cost in terms of physical form factor, money, and energy consumption. Multiplexing one wireless radio across multiple *virtual networks* has been proposed as a solution, but there are several drawbacks

to existing work in this area. Switching times may still be too high for certain potential applications, and application-level interfaces too cumbersome for software developers to realize full benefit.

This paper presented Juggler, a virtual WiFi link layer I have developed for the Linux operating system. By leveraging network cards that perform the MAC layer in software, rather than in device firmware, Juggler switches between wireless networks in just over three milliseconds, or less than 400 microseconds if networks share the same wireless channel. Rather than force applications to choose between a fluctuating set of wireless networks, Juggler presents one unchanging network interface to upper layers and either automatically assigns data flows to one of the many active virtual networks, or lets applications exert explicit control.

The primary contribution of this work was an evaluation of the prototype implementation's performance in several realistic usage scenarios. I show that mobile clients can enjoy nearly instantaneous 802.11 handoff by reserving 10% of the radio duty cycle for background AP discovery, while minimally impacting foreground data throughput. Juggler also enhances data throughput in situations where wireless bandwidth is superior to that of the wired, back-end connection of an access point. I show how striping data across virtual networks is useful in such situations. Finally, I show that Juggler can maintain a low-bandwidth side-channel, suitable for intra-PAN or point-to-point communication, without adversely impacting foreground connectivity to the Internet.

CHAPTER 5

RELATED WORK

In this chapter, I discuss prior art from the literature that is relevant to my work described thus far in this dissertation. The three sections below address the related work pertaining to Virgil, BreadCrumbs, and Juggler, respectively.

5.1 Discovering Network Connectivity

Several previous "wardriving" studies collected 802.11 AP beacon information [2, 25], which contributed to the many Internet databases of wardriving maps [38, 39, 74]. Users must manually scour these maps to find access points, however, while Virgil is fully automatic. None of these systems associate with APs to run performance tests as Virgil does. As my results showed, the information gleaned from these tests allows Virgil to outperform selection schemes driven solely by the signal-strength data in such datasets. Furthermore, these static maps become unreliable and outdated over time [17], while Virgil continuously rediscovers and probes the user's environment.

SyncScan [63] modifies access points as well as clients, forcing APs to synchronize their beacon frame broadcast schedules. Since clients know in advance what channels will be broadcasting at which times, they can quickly collect all beacons and return to their original channel before any user-perceivable service disruption is noticeable. Shin et al. [69] similarly optimize the scan process, since AP discovery has been shown to dominate the 802.11 handoff process. Neither technique associates with the scanned APs. Therefore, such techniques only speed up the selection process, but neither makes the choice any more accurate than existing strongest-signal-strength algorithms.

Lee and Miller [48] propose adding information to the access point beacon signal, to help guide clients' AP selection. While their focus was on facilitating roaming between commercial wireless access networks, the concept could be generalized. One could envision access points broadcasting their current load, current estimated latency to reference servers in the Internet, etc. I argue it is preferable for clients to discover this information for themselves. As I showed, testing one AP takes at most a matter of seconds, a reasonable overhead. Furthermore, when clients are roaming in public, they have no reason to trust the stranger who administers an access point. In fact, it is likely in the AP administrator's self-interest to falsely advertise his AP as low-quality, to prevent anonymous traffic from overloading it.

Judd and Steenkiste [40] recognized that basing AP selection policy solely on signal strength results in uneven loading of multiple access points. They suggested AP load as a beneficial metric, since their work was more focuses on balancing load between access points than directly focusing on client performance as the primary goal.

It has become accepted that the push toward ubiquitous computing makes automatic service discovery in new environments more important than ever [67]. Existing work, however, has focused on application-level services [26, 31], but is silent on how the client chooses an appropriate network connection in the first place. Virgil seeks to fill this gap.

Several systems seek to allow clients of one wireless service provider to access foreign wireless hotspots when roaming [14, 29, 52, 66]. Virgil is complementary, since users must find and associate to an access point before negotiating such roaming agreements. The service discovery Virgil provides is similarly critical for grassroots wireless collective initiatives [11, 57, 68].

5.2 Mobility Modelling and Path Prediction

Rahmati and Zhong [62] investigated the problem of choosing between WiFi and cellular data networks, given that a large number of mobile devices now feature both radios (e.g. the Apple iPhone). Rather than build and maintain a mobility model as BreadCrumbs does, they use the set of cellular tower IDs currently seen and some time-of-day heuristics to estimate the expected quality of WiFi connectivity at the current location. Their system

does not predict future conditions, as BreadCrumbs does. Rather, it decides whether at a given time and place, it is more advantageous to power on the WiFi interface or to use the lower-bandwidth, but ubiquitious, cellular connection. Also, identifying location solely by cell tower signals is necessarily more coarse-grained than our approach that leverages WiFi beacons or GPS. Cellular signals reach at least several kilometers, and tens of kilometers under good conditions. WiFi signals have a far shorter range, typically several hundred meters at best.

MobiSteer [54] focuses on improving wireless network connectivity in one specific usage setting—while in motion in a motor vehicle. Their system uses a directional antenna to maximize the duration and quality of connectivity between a moving vehicle and stationary access points in the community. This goal is complementary to that of BreadCrumbs, because MobiSteer performs well in situations where BreadCrumbs does not, and vice versa. While portions of the evaluation traces, collected in in the course of my evaluation of BreadCrumbs, track the user riding on a city bus, during this period the user only has reliable connectivity while stopped at intersections. As explored in detail by Bychkovsky et al [16], this reduced performance was due to the brief time the client has to associate with the AP, obtain a DHCP address, and do useful work. On the other hand, BreadCrumbs does not require any specialized hardware and works with whatever users already carry in their pocket. MobiSteer's cached mode operation is also reminiscent of the way BreadCrumbs and Virgil optimize future resource discovery by caching historical access point quality information.

Song et al. [71] studied the efficacy of applying different mobility prediction methods to the problem of improving bandwidth provisioning and handoff for VoIP telephony. Much like my work, they use real client traces to evaluate the success of a concrete application that is prediction-aware. They assume the existence of a centralized authority, however, that collects all mobility information, makes predictions, and disseminates instructions to the various wireless access points. I am focused on applications that are still useful when the device itself keeps its mobility history, and this information need not be disclosed to any other party.

Ghosh et al. [32] predict the probability that users visit popular locations, known as *hubs*. Their focus is on extrapolating *sociological orbits* from the client mobility data by identifying the frequency with which users encounter one another at these hubs. The authors do not evaluate how accurately their Bayesian techniques predicted explicit client paths (rather than just the hubs they visit). I therefore was unable to compare the accuracy of their technique with that of BreadCrumbs' second-order Markov model.

Yoon et al. [76] concentrated, as did Kim et al [45], on deriving realistic mobility models from actual user mobility traces. The idea is to take many different client traces and build a probabilistic model that can be used to generate arbitrary client tracks. These traces, while still artificial, more closely model the real movements of users than do synthetic models like Random Waypoint [75]. In this dissertation, I focus on the situation where devices maintain their actual mobility history themselves, and predict their future behavior "on-the-fly" rather than base predictions on mobility models derived from multiple users' behavior.

Marmasse and Schmandt [51] argue, as I do, in favor of a user-centric mobility model. Their *comMotion* system is concerned chiefly with tracking users' movement through various semantically meaningful locations, such as "home" or "work". BreadCrumbs, on the other hand, focuses on lower-level waypoints—namely, GPS grid locations. The semantic concept of user-defined locations could easily be layered atop such low-level information, however.

Haggle [37] is a framework for disseminating data between mobile users based on the fleeting occasions when they come into physical contact with each other. In these situations infrastructure such as WiFi networks need not be used, because users are within range of low-power, point-to-point link technologies like Bluetooth or ZigBee. Their system is clearly dependent on user-centric mobility information, but seeks to predict when pairs of users will come into contact with each other. My work, on the other hand, is focused more on leveraging information about wireless access points the user will soon encounter.

Most applications of location prediction have been in mobile phone networks. Typically, a central network operator seeks to know the sequence of network towers with which a handset will associate. Given this information, the network operator can reserve

resources, such as bandwidth, at the upcoming nodes, so handoff proceeds as smoothly as possible. Bhattacharya and Das [13] use a variant of the LZ predictor described above to predict the next cell users will associate with. Yu and Leung [77] extend this idea to predict not only where a mobile device will hand off but also when this will occur. Liang and Haas [49] use a Gauss-Markov model in a similar way. Others use Robust Extended Kalman Filtering (REKF) [58], integrate individual path information with system-wide aggregate data [3], or estimate future locations through trajectory analysis [4]. Liu et al [50] use a similar hybrid approach for mobility prediction in wireless ATM networks, rather than for mobile telephony. They combine system-wide information with local mobility history and path trajectories to reduce system resource consumption while maintaining user QoS.

All of these location predictors are enabled by accurate estimates of a mobile device's location. In some cases, all that is needed is information on which access point or mobile phone tower the device is associated with. For predictors and applications requiring more fine grained location information, there are a wide variety of solutions. Place Lab leverages public *war-driving* databases of WiFi AP GPS coordinates to triangulate one's location based on the APs seen at a given location and their signal strengths [47]. The same idea has recently been extended to use GSM phone towers rather than WiFi APs [24]. Fox et al [30] showed the benefit of Bayesian filtering to coalesce results from multiple location sensors and smooth transient uncertainty in location estimates. Other work focuses on indoor localization at very small scales, either by deploying custom hardware [70] or mapping existing WiFi beacon sources [34].

5.3 Utilizing Multiple Networks

5.3.1 Virtual link layers, multiple interfaces

Virtual WiFi— [21] virtualizes a device's wireless interface, fooling applications into believing the device is connected simultaneously to different APs on different channels. This is a step in the right direction, because devices can now exploit all available connectivity in their vicinity. Unfortunately, the complexity of assigning data flows to interfaces

is still present. All available APs are presented to higher layers of the network stack as if the device had a wireless radio for each AP that is present. Applications are still responsible for evaluating the quality of each connection on their own. With Juggler, unmodified applications and uninterested users are only aware of a single unchanging network interface. Unlike Virtual WiFi, for the default case of unmodified applications the kernel is responsible for assigning data flows to a certain virtual interface. However, as described in Chapter 4 above, applications and users can override this mechanism to bind data flows explicitly to a certain access point or ad hoc group when desirable.

FatVAP [41], developed concurrently with Juggler, achieves similar performance but is focused primarily on the bundling of multiple 802.11 connections into one logical pipe. While Juggler also supports this sort of data striping, I have shown above how the adjustable radio duty cycle feature enables a richer set of potential applications than this.

Bahl et al. [8] examined scenarios where multiple physical network interfaces are useful to mobile devices, such as handoff and link aggregation. This discussion inspired several of our usage scenarios that address similar issues while using only one radio.

Contact Networking [20] hides the differences between local and remote communication from users. All communication appears to be local—like a direct Bluetooth connection between two devices—even if infrastructure such as the Internet is actually involved. As does my work, the authors recognize that mobile devices typically have several, heterogeneous wireless radios at their disposal. Contact Networking is also conscious of the properties of different link layers. Their primary focus, however, is on neighbor discovery, name resolution, and (ultimately) the preservation of application-level sessions in the face of user mobility. My work does find common ground with the idea that all network connectivity options are not equivalent, and ideally the operating system should dynamically assign data flows to the most appropriate link.

Zhao et al. [78] attack similar problems as Contact Networking. Their work lies firmly within the framework of Mobile IP [59], because the user's Home Agent is required to arbitrate the routing of various data flows. Applications must explicitly bind a data flow to a specific interface through their `SO_BINDTODEVICE` socket option. I envision a decentralized solution, where the operating system automatically assigns flows to interfaces.

5.3.2 Network discovery and handoff

SyncScan [63] coordinates AP beacon transmission in a global fashion, based on AP channel number. Because clients know precisely when the APs on a certain channel will broadcast their beacon, AP discovery becomes a quick process of hopping briefly through the channel space rather than listening passively on a channel for hundreds of milliseconds. SyncScan requires changes to both wireless clients and AP firmware, however, hindering rapid adoption. Juggler's strategy for soft handoff, described in Section 4.6.1 of Chapter 4 above, requires no such changes to access points.

Shin et al. reduce 802.11 handoff latency by maintaining *neighbor graphs*—sequences of AP handoffs [69]. Clients build graphs by direct observation and through sharing with cooperative peers. When a client's current AP becomes unusable, instead of scanning the entire channel space the client only searches those channels on which a successor AP to the current AP has been seen in the neighbor graph. Rather than incur the overhead to track such history, Juggler scans for APs, associates, and obtains a DHCP configuration before the current AP has even become unusable.

In SMesh [5], all wireless clients and stationary access points are members of one mesh network. Handoff is efficient because access points collectively decide when to transfer responsibility for a given device. Clients are unmodified, but SMesh requires custom access point software and a homogeneous deployment, managed by a single entity. This is at odds with Juggler's target environment—heterogeneous, unmanaged public connectivity.

5.3.3 Data striping and aggregation

MAR is a standalone hardware device that aggregates heterogeneous wireless links into one logical, high-bandwidth pipe [64]. Its focus is on combining the capacity of many physical radios, while Juggler connects to multiple networks through only one radio.

Horde [61] is similar to MAR, but is a middleware layer on the mobile client itself rather than a separate device. Horde also lets applications dictate quality of service (QoS) requirements for their flows. The authors subsequently deployed a real-time video streaming application that aggregates many low-bandwidth links to provide high QoS while in motion, using a dynamic set of mobile phone data networks [60].

PRISM [44] is a proxy-based inverse multiplexer that allows cooperative mobile hosts to aggregate and share their wireless infrastructure bandwidth. The authors focus on supporting TCP traffic. PRISM stripes packets of one TCP flow across disjoint links. Because this may result in out-of-order delivery, their system reorders ACKs to preserve the expected TCP semantics at the client end. PRISM requires an additional congestion control mechanism to handle TCP windows sizes properly. Their results are intriguing for the future development of Juggler, because some of our throughput inefficiency is a result of the sorts of TCP side-effects noted in their work.

Hacker et al. studied the effect of parallel TCP flows on total throughput and flow fairness [33]. Experimental results showed that during periods of congestion, the distribution of total bandwidth among all competing parallel flows can be severely unbalanced.

5.3.4 Mesh networks and side channels

VirtualWiFi has been applied to help diagnose faults in wireless LANs. This often is difficult because clients need help or advice the most when they find themselves disconnected from the infrastructure network. Both Client Conduit [1] and WiFiProfiler [23] share the common strategy of using VirtualWiFi to let clients connect simultaneously to nearby nodes and to an infrastructure AP. Nodes that have infrastructure connectivity then help diagnose the problems suffered by their peers who are disconnected from the network but can still contact their neighbors in ad hoc mode. The mesh connectivity scenario in Section 4.6.3 provides a similar channel via Juggler, but at a more responsive switching resolution while imposing a minimal penalty on the infrastructure connection.

Prior work has leveraged the properties of point-to-point links, such as Bluetooth or WiFi in ad hoc mode, to aid in the establishment of security relationships between users [10, 18]. For example, exchanging public keys over the Internet puts users at risk for a man-in-the-middle attack, while communicating directly forces attackers to be physically present. Juggler allows users to establish these sorts of temporary, low-bandwidth side channels without adversely impacting their primary infrastructure connection.

5.3.5 Robustness through diversity

Multi-Radio Diversity (MRD) uses redundant wireless channels to reduce packet losses and improve throughput [53]. Devices receive on different channels simultaneously over multiple network interfaces, and transmit upstream in parallel to multiple, coordinated access points to ensure faithful reception. MRD requires tight coordination among access points, an assumption that Juggler does not make. It is also unclear how closely Juggler could approximately the redundant downstream channel of MRD, because they leverage the fact that many radios are receiving the same packets simultaneously—on different frequencies—in order to detect and correct bit errors.

Vergetis et al. performed an extensive study of how packet-level diversity could be beneficial in 802.11 data transmission [73]. They evaluated the effectiveness of encoding data with an erasure code and transmitting over multiple paths as a form of forward error correction. Their results found that multiple physical interfaces are not mandatory for the scheme to be beneficial, provided that switching delays could be reduced below one millisecond. An interesting extension of Juggler would be to evaluate how well such an error-correcting code scheme could be deployed atop the current implementation of Juggler, with its somewhat higher 3 ms switching overhead.

CHAPTER 6

CONCLUSIONS

Wireless network availability and quality must approach that of wired network connections if mobile computing is to be completely seamless to users. This dissertation argues that this vision is not yet reality. Prior work has attacked the problem piecemeal, but often places heavy burdens on users (scouring wardriving databases) or application developers (manually matching data flows with one of many network connections).

This dissertation advances the argument that mobile systems must move beyond optimzing for sets of local conditions, at discrete geographic locations, to considering how the connectivity presented to a device changes over time. Comprehending this *derivative of connectivity* must therefore be a first-order concern of systems software for mobile devices.

The results show that through low-level software and modifications to the operating itself, the wireless networking experience of mobile devices can be greatly improved, while imposing little or no burden on users and application developers. Until the promise of a ubiquitious, high-quality wireless data network is realized, the techniques outlined in this dissertation will help stitch the current norm of islands of connectivity into a more cohesive whole.

This remainder of this chapter states the specific contributions this dissertation makes in the field of computer science, and reflects on future work suggested by the current state of the implementation and investigation.

6.1 Contributions

The main contribution of this dissertation is a deeper understanding of how much the quality and availability of network connectivity for mobile devices can be improved over this use of current techniques, without overburdening users or application developers, and without requiring centralized network infrastructure. This solution has several complementary aspects.

The dissertation began by studying actual WiFi AP deployments in three cities in the United States, to determine the quality, quantity, and distribution of publicly-accessible wireless connectivity. The results show that connectivity is sufficiently pervasive that mobile users typically can be connected to the Internet a high percentage of the time. However, the most common AP selection algorithm—choosing by strongest signal strength—misses a large fraction of usable APs. This led to the design of an AP selection daemon, known as Virgil, that evaluates the application level qualities (e.g. bandwidth, latency) of each AP before settling on a decision. This method results in a 22-100% increase in AP discovery success, depending on the neighborhood in question. Virgil is the first instance in the literature of an access network selection system that actively connects and probes application-level quality before settling on a decision.

This technique of probing application-level quality of access points was then augmented with a user-centric mobility model to generate *connectivity forecasts*. These forecasts are a prediction of the quality of the upcoming wireless connectivity a mobile device will have. A new system, BreadCrumbs, generates connectivity forecasts first by building and maintaining a mobility model tracking the motion of the user's device. This model is implemented as a second-order Markov chain, with GPS coordinates as the building blocks of the state space. Atop this mobility model, BreadCrumbs layers the AP test results of all access points encountered while the model was in a given state. The efficacy and usefulness of connectivity forecasts was then evaluated by devising three applications of benefit to users of mobile devices. The evaluation compared the success of prediction-ignorant and prediction aware algorithms. This showed that even with modest training time and minimal application modification the connectivity forecasts that BreadCrumbs provide can

yield significant benefit to applications in terms of energy savings or application-specific efficiencies.

The final contribution of this dissertation is a virtual link layer, Juggler, which allows a mobile device to connect simultaneously to many wireless access networks through just one physical WiFi radio. The creation of Juggler was motivated by the observation that mobile devices often encounter multiple usable WiFi APs at once, and choosing only one results in forgoing the full potential for network access that that location. Given the varied quality of such wireless links, one can ill afford such under-exploitation if a consistent, high-quality user experience is to be maintained. We present the design and implementation of a Juggler prototype that is integrated with the Linux kernel, but the primary contribution of that portion of the dissertation is an extensive evaluation of how the capabilities provided by Juggler can be used by different applications. Juggler enables nearly instantaneous WiFi handoff, striping of data flows across multiple low-quality links in parallel, and maintaining simultaneous foreground Internet connectivity and a low-bandwidth side channel to mesh networks, the user's PAN, or other ad-hoc groups.

6.2 Impact of Future Device and Connectivity Technologies

The work presented in this dissertation focuses primarily on the usage scenario where devices are connected to the Internet via WiFi access points of varying quality, and are disconnected from the network when no such AP is present. The reader may wonder how relevant this work will remain in the near future, when third- or fourth- generation cellular networks provide nearly ubiquitous coverage.

The techniques described above will remain useful in the future because of the inherent tension in wireless networking between range and bandwidth. If all other conditions (e.g. interference) are equal, then a short-range technology such as WiFi will often be able to offer superior bandwidth to long-range, cellular technologies. Another important consideration is energy usage, because it takes more power to communicate with a distant cellular tower at the same bitrate as with a nearby WiFi AP.

In this ubiquitous 4G future, then, mobile devices will want to use WiFi APs when available to maximize throughput and minimize power consumption, and resort to cellular

data connections when no such AP is available. Clearly the techniques proposed in this dissertation are equally as useful for negotiating these sorts of tradeoffs as they are for the navigating current state of wireless networking.

Along with link technologies, devices themselves will continue to evolve and improve. While this work focused primarily on mobile devices with a WiFi radio, it can be extended to any link protocol or combination of possible link technologies. Unlike prior work that relies on specific attributes of the 802.11 link protocol to predict quality, the techniques described here are fully generalizable because they are link-layer agnostic and consider only the application- and user-level experience. Any current (e.g. ZigBee, WiMax, GPRS/EDGE) or future wireless protocol could be supported with minimal effort.

6.3 Future Work

One consequence of Juggler's design is that multiple routes to the Internet, of heterogeneous quality, exist when a device is associated with several APs. This begs the following question: how can applications easily exploit this situation, without imposing a burden on programmers or users? It would be interesting to explore how well the operating system can infer application intent and assign data flows to the most appropriate access point. For example, consider a remote terminal client such as ssh. In general, ssh does not need a high-bandwidth connection to the remote server but latency is critical or the user experience will suffer. Applications could indicate their requirements through hints to Juggler, but it would be even better if the system could learn the needs of different applications over time, by observing the characteristics of the data flows they generate.

There is also potential for applying collaboration and social networking to connectivity forecasting systems such as BreadCrumbs. An army of users running BreadCrumbs on their phone or PDA would quickly map the quality and location of access points in even a large city. Simply exchanging AP test results with strangers, however, raises important privacy concerns that require careful investigation. It would be useful to deploy BreadCrumbs on a larger scale on the COPSE (Concurrent OPportunistic Sensor Environment) mobile device testbed[1]. COPSE is designed to be a common laboratory environment for

[1] `http://copse.cs.duke.edu/`

deploying applications on real handheld devices—what PlanetLab is for wide-area, distributed network applications. Deploying BreadCrumbs on hundreds of mobile devices, carried daily by actual users, and gathering actual mobility traces and models generated by many different users will resolve certain unanswered questions, such as whether the mobility models of strangers are sufficiently similar that sharing them might be useful.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Atul Adya, Paramvir Bahl, Ranveer Chandra, and Lili Qiu. Architecture and techniques for diagnosing faults in IEEE 802.11 infrastructure networks. In *Proceedings of the Tenth International Conference on Mobile Computing and Networking (MobiCom)*, Philadelphia, Pennsylvania, USA, September 2004. 63, 66, 96

[2] Aditya Akella, Glenn Judd, Srinivasan Seshan, and Peter Steenkiste. Self-management in chaotic wireless deployments. In *Proceedings of the 11th International Conference on Mobile Computing and Networking (MobiCom)*, pages 185–199, Köln, Germany, August 2005. 89

[3] Ian F. Akyildiz and Wenye Wang. The predictive user mobility profile framework for wireless multimedia networks. *IEEE/ACM Transactions on Networking*, 12(6):1021–1035, 2004. 3, 40, 93

[4] A. Aljadhai and T.F. Znati. Predictive mobility support for QoS provisioning in mobile wireless environments. *IEEE Journal on Selected Areas in Communications*, 19(10):1915–1930, October 2001. 3, 40, 93

[5] Yair Amir, Claudiu Danilov, Michael Hilsdale, Raluca Musaloiu-Elefteri, and Nilo Rivera. Fast handoff for seamless wireless mesh networks. In *Proceedings of the Fourth International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 83–95, Uppsala, Sweden, June 2006. 95

[6] Manish Anand and Jason Flinn. PAN-on-demand: Leveraging multiple radios to build self-organizing energy-efficient PANs. In *Proceedings of the Fifth Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, Dublin, Ireland, July 2008. 86

[7] Manish Anand, Edmund B. Nightingale, and Jason Flinn. Self-tuning wireless network power management. In *Proceedings of the Ninth International Conference on Mobile Computing and Networking (MobiCom)*, pages 176–189, San Diego, California, USA, September 2003. 59

[8] Paramvir Bahl, Atul Adya, Jitendra Padhye, and Alec Walman. Reconsidering wireless systems with multiple radios. *ACM SIGCOMM Computer Communication Review*, 34(5):39–46, October 2004. 62, 94

[9] Paramvir Bahl, Ranveer Chandra, and John Dunagan. SSCH: Slotted Seeded Channel Hopping for Capacity Improvement in IEEE 802.11 Ad-Hoc Wireless Networks. In *Proceedings of the Tenth International Conference on Mobile Computing and Networking (MobiCom)*, Philadelphia, Pennsylvania, USA, September 2004. 63, 73

[10] D. Balfanz, D. Smetters, P. Stewart, and H. Wong. Talking to strangers: Authentication in ad-hoc wireless networks. In *Proceedings of the Ninth Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, February 2002. 86, 96

[11] Bay area wireless users group. `http://bawug.org/`. 4, 11, 90

[12] BBC News. 'Evil Twin' Fear for Wireless Net, 20 January 2005. 11

[13] Amiya Bhattacharya and Sajal K. Das. Lezi-update: an information-theoretic approach to track mobile users in pcs networks. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 1–12, New York, NY, USA, 1999. ACM Press. 3, 40, 93

[14] Mauro Brunato and Danilo Severina. WilmaGate: A new open access gateway for hotspot management. In *Proceedings of the Third ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH)*, pages 56–64, Köln, Germany, September 2005. 90

[15] United States Census Bureau. 2000 census of population and housing, summary population and housing characteristics, Washington, DC, USA, 2002. 12, 13, 26

[16] V. Bychkovsky, B. Hull, A.K. Miu, H. Balakrishnan, and S. Madden. A measurement study of vehicular internet access using in situ Wi-Fi networks. In *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2006. 91

[17] Simon Byers and Dave Kormann. 802.11b access point mapping. *Communications of the ACM*, 46(5):41–46, May 2003. 89

[18] Srdjan Capkun, Jean-Pierre Hubaux, and Levente Buttyan. Mobility helps security in ad-hoc networks. In *Proceedings of the Fourth ACM International Symposium on Mobile Ad-hoc Networking and Computing (MobiHoc)*, pages 46–56, Annapolis, Maryland, USA, June 2003. 11, 96

[19] Mark Carson and Darrin Santay. NIST Net—A Linux–based Network Emulation Tool. *ACM SIGCOMM Computer and Communication Review*, June 2003. 74

[20] Casey Carter, Robin Kravets, and Jean Tourrilhes. Contact networking: A localized mobility system. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 145–158, San Francisco, California, USA, May 2003. 94

[21] R. Chandra, P. Bahl, and P. Bahl. MultiNet: Connecting to multiple IEEE 802.11 networks using a single wireless card. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 882–893, Hong Kong, China, March 2004. 63, 65, 66, 73, 85, 93

[22] Ranveer Chandra, Jitendra Padhye, Lenin Ravindranath, and Alec Wolman. Beaconstuffing: Wi-Fi without associations. In *Proceedings of the Eighth IEEE Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2007. 77

[23] Ranveer Chandra, Venkata N. Padmanabhan, and Ming Zhang. WifiProfiler: Cooperative Diagnosis in Wireless LANs. In *Proceedings of the Fourth International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Uppsala, Sweden, June 2006. 63, 73, 86, 96

[24] Mike Y. Chen, Tim Sohn, Dmitri Chmelev, Dirk Haehnel, Jeffrey Hightower, Jeff Hughes, Anthony LaMarca, Fred Potter, Ian Smith, and Alex Varshavsky. Practical metropolitan-scale positioning for GSM phones. In *Proceedings of the Eighth International Conference on Ubiquitous Computing (UbiComp)*, pages 225–242, Irvine, California, USA, September 2006. 93

[25] Y. Cheng, Y. Chawathe, A. LaMarca, and J. Krumm. Accuracy characterization for metropolitan-scale Wi-Fi localization. In *Proceedings of the Third International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 233–245, Seattle, Washington, USA, June 2005. 42, 89

[26] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. In *Proceedings of the Fifth International Conference on Mobile Computing and Networking (MobiCom)*, pages 24–35, Seattle, Washington, USA, August 1999. 90

[27] M. Dischinger, A. Haeberlen, K.P. Gummadi, and S. Saroui. Characterizing residential broadband networks. In *Proceedings of IMC*, October 2007. 38

[28] Richard Draves, Jitendra Padhye, and Brian Zill. Routing in multi-radio, multi-hop wireless mesh networks. In *Proceedings of the Tenth International Conference on Mobile Computing and Networking (MobiCom)*, pages 114–128, Philadelphia, Pennsylvania, USA, September 2004. 86

[29] Elias C. Efstathiou and George C. Polyzos. A peer-to-peer approach to wireless LAN roaming. In *Proceedings of the First ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH)*, pages 10–18, San Diego, California, USA, September 2003. 90

[30] Dieter Fox, Jeffrey Hightower, Lin Liaoand Dirk Schulz, and Gaetano Borriello. Bayesian filtering for location estimation. *IEEE Pervasive Computing*, 2(3):24–33, July–September 2003. 93

[31] Adrian Friday, Nigel Davies, Nat Wallbank, Elaine Catterall, and Stephen Pink. Supporting service discovery, querying and interaction in ubiquitous computing environments. *Wireless Networks*, 10(6):631–641, November 2004. 90

[32] Joy Ghosh, Matthew J. Beal, Hung Q. Ngo, and Chunming Qiao. On profiling mobility and predicting locations of campus-wide wireless network users. In *REALMAN '06: Proceedings of the Second International ACM/SIGMOBILE Workshop on Multi-hop Ad Hoc Networks (MobiHoc)*, pages 55–62, Florence, Italy, May 2006. 42, 92

[33] Thomas J. Hacker, Brian D. Noble, and Brian Athey. Improving throughput and maintaining fairness using parallel TCP. In *Proceedings of INFOCOM*, Hong Kong, China, March 2004. 96

[34] Andreas Haeberlen, Eliot Flannery, Andrew M. Ladd, Algis Rudys, Dan S. Wallach, and Lydia E. Kavraki. Practical robust localization over large-scale 802.11 wireless networks. In *Proceedings of the 10th annual international conference on Mobile computing and networking (MobiCom)*, pages 70–84, Philadelphia, Pennsylvania, USA, 2004. 93

[35] Familiar Linux. `http://www.handhelds.org/`. 13, 48

[36] Tristan Henderson, David Kotz, and Ilya Abyzov. The changing usage of a mature campus-wide wireless network. In *Proceedings of the Tenth International Conference on Mobile Computing and Networking (MobiCom)*, pages 187–201, Philadelphia, Pennsylvania, USA, September 2004. 21

[37] Pan Hui, Augustin Chaintreau, James Scott, Richard Gass, Jon Crowcroft, and Christophe Diot. Pocket switched networks and human mobility in conference environments. In *Proceedings of the ACM SIGCOMM Workshop on Delay-tolerant Networking*, pages 244–251, Philadelphia, Pennsylvania, August 2005. 92

[38] Intel Research Seattle. Place Lab: A privacy-observant location system. `http://placelab.org/`. 89

[39] Wi-fi hotspot locator. `http://jiwire.com/`. 7, 89

[40] Glenn Judd and Peter Steenkiste. Fixing 802.11 access point selection. *ACM SIGCOMM Computer Communications Review*, 32(3):31, July 2002. 90

[41] Srikanth Kandula, Kate Ching-Ju Lin, Tural Badirkhanli, and Dina Katabi. FatVAP: Aggregating AP backhaul capacity to maximize throughput. In *Proceedings of the Fifth USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, April 2008. 63, 94

[42] Jeremy A. Kaplan. Real world testing: The best ISPs in America. PC Magazine, May 2007. 77

[43] Orin S. Kerr. Cybercrime's scope: Interpreting "access" and "authorization" in computer misuse statutes. *New York University Law Review*, 78(5):1596–1668, November 2003. 11

[44] Kyu-Han Kim and Kang G. Shin. Improving TCP performance over wireless networks with collaborative multi-homed mobile hosts. In *Proceedings of the Third International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 107–120, Seattle, Washington, USA, June 2005. 96

[45] Minkyong Kim, David Kotz, and Songkuk Kim. Extracting a mobility model from real user traces. In *Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, Barcelona, Spain, April 2006. 40, 92

[46] David Kotz, Tristan Henderson, and Ilya Abyzov. CRAWDAD trace set dartmouth/campus/movement (v. 2005-03-08), March 2005. 41

[47] Anthony LaMarca, Yatin Chawathe, Sunny Consolvo, Jeffrey Hightower, Ian Smith, James Scott, Tim Sohn, James Howard, Jeff Hughes, Fred Potter, Jason Tabert, Pauline Powledge, Gaetano Borriello, and Bill Schilit. Place Lab: Device positioning using radio beacons in the wild. In *Procedings of the Third International Conference on Pervasive Computing*, pages 116–133, Munich, Germany, May 2005. 39, 93

[48] Yui-Wah Lee and Scott Miller. Network selection and discovery of service information in public WLAN hotspots. In *Proceedings of the Second ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH)*, pages 81–92, Philadelphia, Pennsylvania, USA, October 2004. 90

[49] Ben Liang and Zygmunt J. Haas. Predictive distance-based mobility management for multidimensional PCS networks. *IEEE/ACM Transactions on Networking (TON)*, 11(5):718–732, October 2003. 3, 40, 93

[50] T. Liu, P. Bahl, and I. Chlamtac. Mobility modelling, location tracking, and trajectory prediction in wireless atm networks. *IEEE Journal on Selected Areas in Communications*, 16(6):922–936, August 1998. 93

[51] Natalia Marmasse and Chris Schmandt. A user-centered location model. *Personal and Ubiquitous Computing*, 6(5–6):318–321, December 2002. 92

[52] Yasuhiko Matsunaga, Ana Sanz Merino, Takashi Suzuki, and Randy Katz. Secure authentication system for public WLAN roaming. In *Proceedings of the First ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH)*, pages 113–121, San Diego, California, USA, 2003. 11, 90

[53] Allen Miu, Hari Balakrishnan, and Can Emre Koksal. Improving loss resilience with multi-radio diversity in wireless networks. In *Proceedings of the 11th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 16–30, Cologne, Germany, 2005. 62, 97

[54] Vishnu Navda, Anand Prabhu Subramanian, Kannan Dhanasekaran, Andreas Timm-Giel, and Samir R. Das. MobiSteer: Using steerable beam directional antenna for vehicular network access. In *Proceedings of the Fifth International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Juan, Puerto Rico, June 2007. 91

[55] Anthony J. Nicholson, Junghee Han, David Watson, and Brian D. Noble. Exploiting Mobility for Key Establishment. In *Proceedings of the 7th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, Blaine, Washington, USA, April 2006. 11

[56] Anthony J. Nicholson, Ian E. Smith, Jeff Hughes, and Brian D. Noble. LoKey: Leveraging the SMS Network in End-to-end, Decentralized Trust Establishment. In *Proceedings of the Fourth International Conference on Pervasive Computing*, Dublin, Ireland, May 2006. 11

[57] NYCWireless. `http://nycwireless.net/`. 4, 11, 90

[58] Pubudu N. Pathirana, Andrey V. Savkin, and Sanjay Jha. Mobility modelling and trajectory prediction for cellular networks with mobile base stations. In *MobiHoc '03: Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*, pages 213–221, New York, NY, USA, 2003. ACM Press. 3, 40, 93

[59] C.E. Perkins. Mobile networking through Mobile IP. *IEEE Internet Computing*, 2(1):58–69, January–February 1998. 94

[60] Asfandyar Qureshi, Jennifer Carlisle, and John Guttag. Tavarua: Video streaming with WWAN striping. In *Proceedings of ACM Multimedia (MM)*, pages 327–336, Santa Barbara, California, USA, October 2006. 62, 80, 95

[61] Asfandyar Qureshi and John Guttag. Horde: Separating network striping policy from mechanism. In *Proceedings of the Third International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 121–134, Seattle, Washington, USA, June 2005. 80, 95

[62] Ahmad Rahmati and Lin Zhong. Context-for-wireless: Context-sensitive energy-efficient wireless data transfer. In *Proceedings of the Fifth International Conference on Mobile Systems, Applications and Systems (MobiSys '07)*, pages 165–178, San Juan, Puerto Rico, June 2007. 90

[63] I. Ramani and S. Savage. SyncScan: Practical fast handoff for 802.11 infrastructure networks. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 675–684, Miami, Florida, USA, March 2005. 29, 77, 89, 95

[64] Pablo Rodriguez, Rajiv Chakravorty, Julian Chesterfield, Ian Pratt, and Suman Banerjee. MAR: A commuter router infrastructure for the mobile Internet. In *Proceedings of the Second International Conference on Mobile Systems, Applications*

and Services (MobiSys), pages 217–230, Boston, Massachusetts, USA, June 2004. 80, 95

[65] Naouel B. Salem, Jean-Pierre Hubaux, and Markus Jakobsson. Reputation-based Wi-Fi Deployment Protocols and Security Analysis. In *Proceedings of the Second ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots*, October 2004. 11

[66] Naouel B. Salem, Jean-Pierre Hubaux, and Markus Jakobsson. Reputation-based Wi-Fi Deployment Protocols and Security Analysis. In *Proceedings of the Second ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH)*, pages 29–40, Philadelphia, Pennsylvania, USA, October 2004. 90

[67] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001. 90

[68] SeattleWireless. `http://seattlewireless.net/`. 4, 11, 90

[69] Minho Shin, Arunesh Mishra, and William A. Arbaugh. Improving the latency of 802.11 hand-offs using neighbor graphs. In *Proceedings of the Second International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 70–83, Boston, Massachusetts, USA, June 2004. 89, 95

[70] Adam Smith, Hari Balakrishnan, Michel Goraczko, and Nissanka Priyantha. Tracking moving devices with the cricket location system. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 190–202, New York, NY, USA, 2004. ACM Press. 93

[71] Libo Song, Udayan Deshpande, Ulas C. Kozat, David Kotz, and Ravi Jain. Predictability of WLAN mobility and its effects on bandwidth provisioning. In *Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, Barcelona, Spain, April 2006. 40, 91

[72] Libo Song, David Kotz, Ravi Jain, and Xiaoning He. Evaluating location predictors with extensive Wi-Fi mobility data. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1414–1424, March 2004. 3, 40

[73] Evangelos Vegetis, Eric Pierce, Marc Blanco, and Roch Guerin. Packet-level diversity–from theory to practice: An 802.11-based experimental investigation. In *Proceedings of the 12th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 62–73, Los Angeles, California, USA, September 2006. 97

[74] WIGLE: Wireless geographic logging engine. `http://wigle.net/`. 7, 89

[75] Jungkeun Yoon, Mingyan Liu, and Brian Noble. Random waypoint considered harmful. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1312–1321, March 2003. 40, 92

[76] Jungkeun Yoon, Brian D. Noble, Mingyan Liu, and Minkyong Kim. Building realistic mobility models from coarse-grained traces. In *Proceedings of the Fourth International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 177–190, Uppsala, Sweden, June 2006. 40, 92

[77] Fei Yu and Victor C.M. Leung. Mobility-based predictive call admission control and bandwidth reservation in wireless cellular networks. In *Proceedings of the 20th Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 518–526, Anchorage, Alaska, USA, April 2001. 3, 40, 93

[78] Xinhua Zhao, Claude Castelluccia, and Mary Baker. Flexible network support for mobility. In *Proceedings of the Fourth International Conference on Mobile Computing and Networking (MobiCom)*, pages 145–156, Dallas, Texas, USA, 1998. 66, 94