Juggler: Virtual Networks for Fun and Profit

Anthony J. Nicholson, Scott Wolchok and Brian D. Noble
Department of Electrical Engineering and Computer Science
University of Michigan
{tonynich,swolchok,bnoble}@umich.edu

Abstract

There are many situations in which an additional network interface—or two—can provide benefits to a mobile user. Additional interfaces can support parallelism in network flows, improve handoff times, and provide sideband communication with nearby peers. Unfortunately, such benefits are outweighed by the added costs of an additional physical interface. Instead, *virtual interfaces* have been proposed as the solution, multiplexing a single physical interface across more than one communication endpoint. However, the switching time of existing implementations is too high for some potential applications, and the benefits of this approach to real applications are not yet clear. This paper directly addresses these two shortcomings. It describes an implementation of a virtual 802.11 networking layer, called Juggler, that achieves switching times of approximately 3 ms, and less than 400 μs in certain conditions. We demonstrate the performance of this implementation on three application scenarios. By devoting 10% of the duty cycle to background tasks, Juggler can provide nearly instantaneous handoff between base stations or support a modest sideband channel with peer nodes, without adversely affecting foreground throughput. Furthermore, when the client issues concurrent network flows, Juggler is able to assign these flows across more than one AP, providing significant speedup when wired-side bandwidth from the AP constrains end-to-end performance.

1 Introduction

There is increasing recognition that wireless clients can often benefit from additional radio interfaces. For example, multiple interfaces can increase effective bandwidth through provider diversity [18], alleviate spot losses with spectrum diversity [16], and improve mobility management through fast handoff [4]. Despite such compelling advantages, however, devices with multiple interfaces remain the exception rather than the rule.

VirtualWiFi seeks to provide these benefits with a single radio [9]. 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, 5, 11]. The benefit of such techniques when clients are communicating with Internet destinations over infrastructure APs is still unclear.

In this paper, we present 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.

We 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, 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 the paper 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. We 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 timeouts 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.

2 Background

Juggler's design is based on VirtualWiFi [9]. 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 rely on a full software MAC layer. This lets us suppress the association process to further optimize the switching time. The first time Juggler 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 our 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]. We also have implemented this technique in our Juggler prototype.

3 Juggler

Figure 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 [9, 24], we present 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.

Our 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.

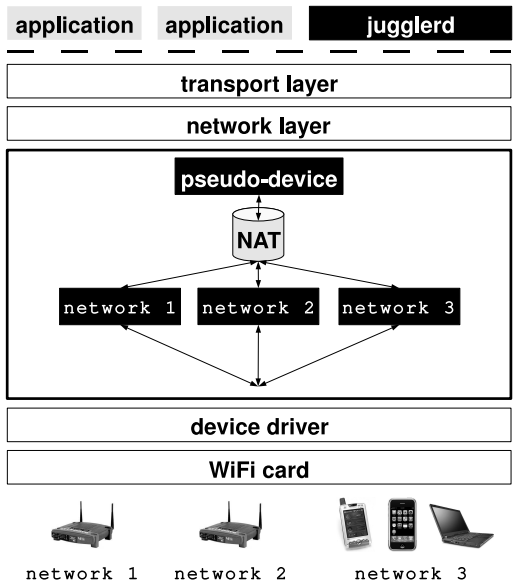

Figure 1: **Juggler network stack.** One unchanging network interface is visible to upper layers of the network stack and to applications. Juggler maintains connection parameters (SSID, frequency, DHCP configuration) for each virtual network with which it is associated, assigns sockets to virtual networks, buffers packets destined for inactive networks, and performs network address translation (NAT) between internal and external IP addresses.

Juggler can connect to 802.11 ad hoc networks as well as infrastructure access points. We 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 data belonging to one socket is transmitted via the same virtual network. We made this design decision to preserve the semantic definition of a socket endpoint as an (IP address, port) pair.

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. Our ongoing work examines how Juggler can handle this matchmaking in a more intuitive fashion. We 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 our Virgil AP selection daemon [17].

3.2 Sending and Receiving Packets

As illustrated in Figure 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.

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, we coerce 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 mode.

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 new AP. This indicates that the client has returned from its (fake) power-save mode. If the AP has any enqueued packets destined for the client, it transmits the first one. Juggler continues sending PSPOLL 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.

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.

3.5 Implementation Details

The vast majority of the Juggler kernel code is a standalone kernel module. A small patch to the Linux 2.6.19 kernel was required for two reasons. First, we 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, we added one line to `netif_rx()` that calls Juggler's inbound NAT function before performing network-layer processing.

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

To encourage future portability, we 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. We store a copy of this global structure for each active network in a linked list, and each time Juggler switches between networks we update `ieee80211` to point to the correct version of the structure.

In the `rtl-wifi` driver, we 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. We were 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 Experimental Setup

Before evaluating Juggler, we must consider what sort of usage environment we intend to model. Previous evaluations of virtual link layers focused primarily on communicating with peers over ad hoc, point-to-point links [5, 9, 11]. Throughput in such situations is limited by the 802.11 link speed (e.g. 10 or 54 Mbps) and interference on the wireless channel.

We are 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.

Figure 2 illustrates the test setup in our laboratory. The test laptop at left represents a mobile client with one WiFi network card. We 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 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, we installed NIST Net [8] 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.

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

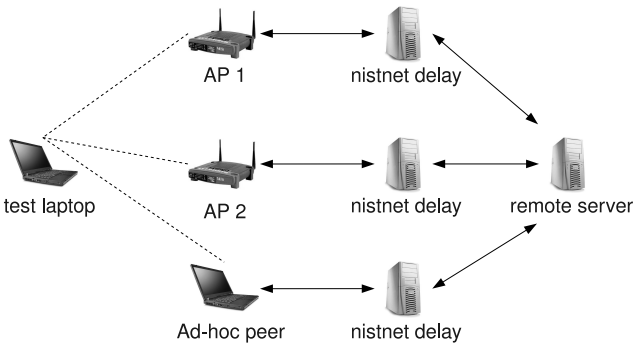

Figure 2: **Laboratory setup.** A test laptop running Juggler can connect wirelessly to one of two 802.11g access points, or to another laptop in ad hoc mode. Two gateway routers use NIST Net to selectively throttle the bandwidth between each AP and the remote server. This allows simulation of varied 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.

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` software MAC layer. To quantify the overhead this introduces, we 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, we 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 we 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.

We loaded Juggler, connected to two different APs on different channels, and recorded the time required to switch networks over 10000 times. We next repeated the experiment while connected to two APs that share the same channel.

	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 1: **Juggler: CPU overhead benchmarks.**

As Table 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. We set this timeout as 500 μs, for a total of 3 ms delay to make six writes. We were unable to lower this timeout much below 500 μs without degraded performance and lost data.

We also examined the overhead incurred when processing inbound or outbound data packets. The most heavyweight operation required is rewriting network-layer headers to perform NAT to and from the internal IP address of the pseudo network device. The results in Table 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 in situations where an outbound packet is destined for a virtual network that is currently inactive. We were merely interested here in the CPU overhead imposed by the presence of Juggler in the critical path of the network stack.

6 Application Scenarios

The primary contribution of this paper is the exploration of several realistic usage scenarios where the ability to multitask one wireless interface is beneficial. In this section, we 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.

We 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 our prior work showed, in real public deployments the bandwidth achievable by an application-level TCP flow is far lower—typically several hundred Kbps [17]. Independent measurements of broadband connectivity quality support these results [14].

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

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 [10, 20]. 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, we explore the efficacy of using Juggler to do just this. We use variable timeslicing to assign 90% of the radio's duty cycle to the current "primary" AP. This is the highest-quality access point detected at the mobile device's current location. The remaining 10% of radio cycles are devoted to scanning for new access points and maintaining association with one or more secondary APs.

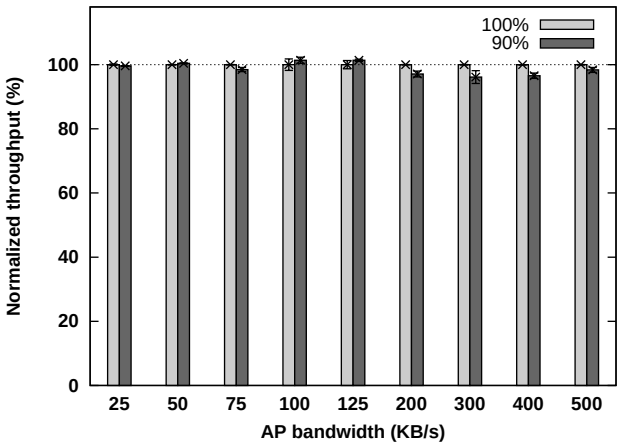

Figure 3: **Soft handoff: throughput of primary AP.** 90% of the radio's activity period is devoted to a "primary" AP that handles all data flows, while 10% is used to discover new APs and maintain association with the backup AP(s). The reduction in primary bandwidth is small despite the loss of 10% of the radio duty cycle.

While the device is using the primary AP to transfer data, Juggler scans for new APs in the background, and preemptively 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 our prior work [17]. 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, we wanted to ensure that reducing the primary AP's radio slice from 100% (without Juggler) to 90% would not adversely impact foreground data traffic. We 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 2, the gateway machines between each AP and the remote server allowed us to simulate a range of bandwidths. Figure 3 plots TCP throughput as a function of AP bandwidth between the client and a remote Internet server. For each bandwidth value, we show 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.

We next sought to quantify how quickly Juggler can discover and configure new access points. We configured the client to be connected to a primary AP with 90% duty cycle, and assigned 10% to background scanning. We 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 its beacon and the client completed the 802.11

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

Table 2: **Soft handoff: discovery and fail-over.** 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.

association process. Table 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 2 is the time required for the client to obtain a DHCP configuration from the new access point, after the association process is completed. This takes on average 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. For static roaming situations, even this modest overhead would not be required.

Finally, we 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%. We 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`). We 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.

We 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 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 our 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.

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 [17]. 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 [18, 19, 21]. In this section, we explore how well Juggler lets applications and users enjoy the benefits of striping while avoiding the costs of multiple network interfaces.

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

6.2.1 Throughput Improvement

Recall the laboratory setup shown in Figure 2. We used a simple TCP client on the test laptop to repeatedly download a 10 MB file from the remote server. For the baseline case, the client used one AP exclusively to transfer the entire file over one TCP socket connection. For the second case, we used Juggler to associate 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 we used two physical WiFi cards instead of having Juggler share one radio. Each card was associated with a different access point, and we bound each of the two sockets to a different interface (using the `SO_BINDTODEVICE` socket option).

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

Figure 4 shows the mean throughput achieved during the download as a function of available bandwidth on each AP.

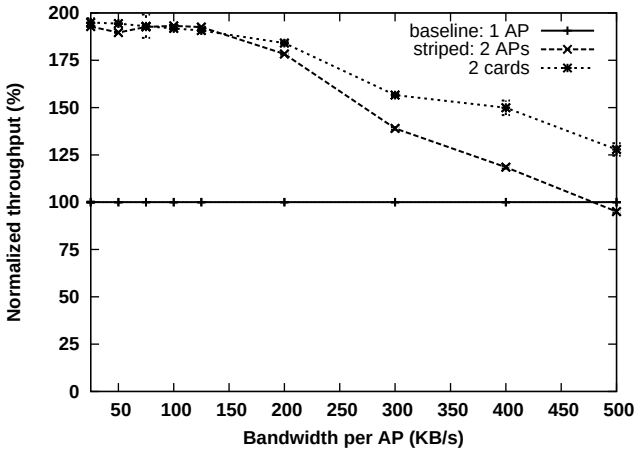

Figure 4: **Data striping: throughput improvement.** For various AP bandwidths, we downloaded a 10 MB file from a remote server. The "baseline" case was the device associated with one AP, using one TCP socket for the download. The "striped" case was Juggler associated with two APs simultaneously, 50% 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.

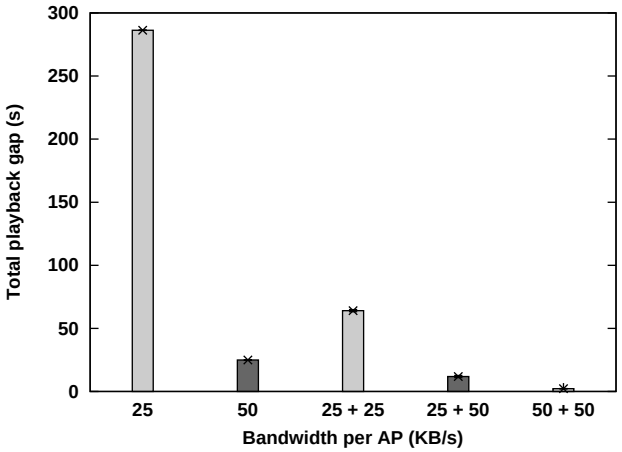

Figure 5: **Streaming video: total playback gap per run.** Times in seconds. Video length was 204.8 s (10 MB encoded at 400 Kbps). Series labels refer to bandwidth available through the AP(s) over which the video was streamed. For instance, *25 + 50* means the client was connected to two APs at once, one of which had 25 KB/s of bandwidth, the other 50 KB/s.

All values are normalized to those of the one card, single 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.

6.2.2 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. We modeled a simple video player that uses an earliest deadline first policy to chose which block to download next. 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 we simply simulated 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, we

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 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. We 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, we see a small glitches 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.

6.2.3 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. We 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 6.2.1. 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 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 was the official BitTorrent client version 3.4.2, packaged with Debian[4]. The Pentium 4 seed also ran the tracker for the torrent.

[2] http://www.ktorrent.org/
[3] http://www.debian.org/releases/lenny/
[4] http://www.bittorrent.com/

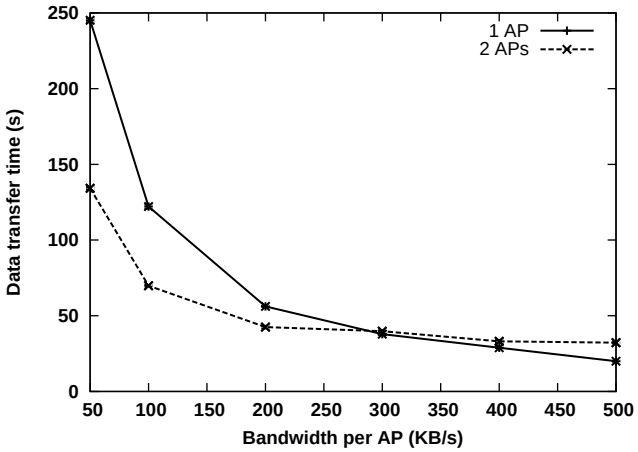

Figure 6: **BitTorrent: torrent download time.** For both cases, blocks were downloaded from both seed servers. 10 MB data file.

For the baseline case, we used KTorrent to download the 10 MB torrent over a single AP. For the second case, we modified KTorrent 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 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, we noticed that the application-level BitTorrent protocol takes longer than standard TCP to accelerate to using the full available bandwidth. We attribute the performance gap between these results and the results in Section 6.2.1 to this protocol overhead.

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 [9]. Such a side channel is clearly useful for communicating with devices in the user's personal area network (PAN) [3], participating in mesh networks [12], or exploiting physical proximity for reasons of security [6].

VirtualWiFi has been used to create an ad hoc side channel while preserving foreground infrastructure connectivity. *WiFiProfiler* [11] 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.

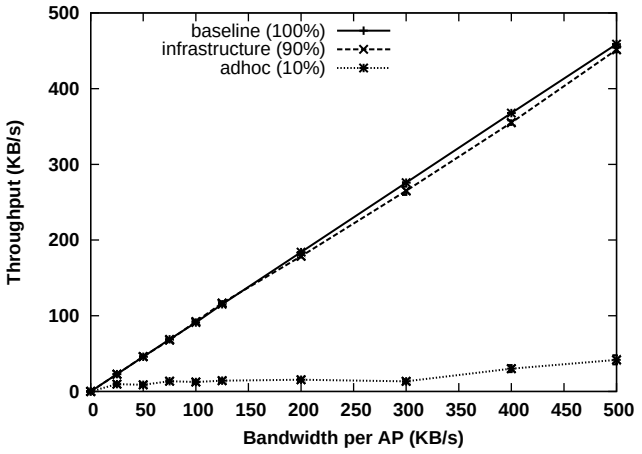

Figure 7: **Mesh connectivity: TCP throughput.** "Baseline" is the maximum TCP throughput from the test server for varied values of effective link bandwidth, when Juggler was not active. We then used Juggler to connect simultaneously to an infrastructure AP (with 90% of the radio) and a nearby device 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.

Juggler's switching time optimizations allow for a much finer-grained trade off between foreground and background traffic. As for our evaluation of soft handoff, we 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. We 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 let us throttle bandwidth between the ad hoc peers in the same fashion as we have for infrastructure APs throughout our evaluation, in order to give a more realistic picture of data throughput.

We 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. We then ran a baseline case, where the test laptop was only connected via the infrastructure AP with 100% of the radio duty cycle.

Figure 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 we 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.

7 Related Work

Virtual link layers, multiple interfaces VirtualWiFi [9] is described in detail throughout the paper. Apart from VirtualWiFi and Juggler, we are aware of no other systems for virtualizing a wireless network connection. Bahl et al [4] 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.

Network discovery and handoff SyncScan [20] 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 6.1 above, requires no such changes to access points.

Shin et al reduce 802.11 handoff latency by maintaining *neighbor graphs*—sequences of AP handoffs [22]. 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 [2], all mobile 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 mobile device. Clients are unmodified, but their system 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.

Data striping and aggregation MAR is a standalone physical device that aggregates many heterogeneous wireless links into one logical, high-bandwidth pipe [21]. Its focus is on combining the capacity of many physical radios, while Juggler connects to multiple networks through only one radio.

Horde [19] 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 [18].

PRISM [15] 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.

Our prior work studied the effect of parallel TCP flows on total throughput and flow fairness [13]. Experimental results showed that during periods of congestion, the distribution of total bandwidth among all competing parallel flows can be severely unbalanced.

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 [11] 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. Our mesh connectivity scenario in Section 6.3 provides a similar channel, 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 [6, 7]. 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.

Robustness through diversity *Multi-Radio Diversity (MRD)* uses redundant wireless channels to reduce packet losses and improve throughput [16]. 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 [23]. 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.

8 Conclusion

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 we have developed for the Linux operating system. By leveraging network cards that perform the MAC layer in soft-

ware, 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 our prototype implementation's performance in several realistic usage scenarios. We show that mobile client 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. We show how striping data across virtual networks is useful in such situations. Finally, we 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.

References

[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.

[2] 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.

[3] Manish Anand and Jason Flinn. PAN-on-Demand: Building self-organizing WPANs for better power management. Technical Report CSE-TR-524-06, University of Michigan, 2006.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] Anthony J. Nicholson, Yatin Chawathe, Mike Y. Chen, Brian D. Noble, and David Wetherall. Improved access point selection. In *Proceedings of the Fourth International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 233–245, Uppsala, Sweden, June 2006.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.