Juggler: Virtual Networks for Fun and Profit

Anthony J. Nicholson, Scott Wolchok, and Brian D. Noble

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 a link-layer 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 percent 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.

Index Terms—Wireless communication, mobile computing, virtual networks, Juggler.

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 [22], alleviate spot losses with spectrum diversity [18], and improve mobility management through fast handoff [3]. Despite such compelling advantages, 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 endpoints. 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 multiinterface applications. VirtualWiFi also made no modifications to wireless device drivers, and consequently, incurred unavoidable overhead as a result of delays and device resets inherent in the third-party driver code. 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.

In this paper, we present Juggler, a refinement of VirtualWiFi's virtual network scheme. Juggler is a virtual networking stack implemented at the link layer, with support from the device driver. 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 percent of the wireless duty cycle to AP scanning, a client can switch APs within tens of milliseconds of detecting lost connectivity. Importantly, this 10 percent 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 multithreaded 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 backend, wired side. For example, the file sharing client obtains benefit through data striping up to backend 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

- A.J. Nicholson is with Google, Inc., 20 W. Kinzie St., Chicago, IL 60610. E-mail: ajnicholson@gmail.com.
- S. Wolchok and B.D. Noble are with the Software Systems Laboratory, University of Michigan, 2260 Hayward Street, Ann Arbor, MI 48109-2121. E-mail: swolchok@umich.edu, bnoble@umich.edu.

Manuscript received 12 Apr. 2008; revised 7 Feb. 2009; accepted 27 Apr. 2009; published online 14 May 2009.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-2008-04-0138. Digital Object Identifier no. 10.1109/TMC.2009.97.

1536-1233/10/$26.00 © 2010 IEEE Published by the IEEE CS, CASS, ComSoc, IES, & SPS

relatively low, due primarily to packet losses induced by the short ad hoc duty cycle. Because Juggler is communicating with an ad hoc peer, it cannot leverage the power save mode buffering feature as if the other party was a standard access point. As a result, packets from the ad hoc peer to the test laptop will often be dropped during the time that Juggler is tuned to the other frequency. Nevertheless, the achieved rate of 320 Kbps for a 10 percent 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 WiFi cards perform part of the IEEE 802.11 protocol in firmware rather than in a software device driver. The problem is that this does not support 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 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 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 software MAC layer. This lets us suppress the association process to further optimize switching. When Juggler first communicates with an AP, it must perform the slow 802.11 association sequence. Subsequently, Juggler only associates to an AP again if it receives an explicit 802.11 deauthentication message. This may occur if Juggler is associated with a given AP but rarely sends any data through that virtual network, since access points periodically deauthenticate "inactive" clients.

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.

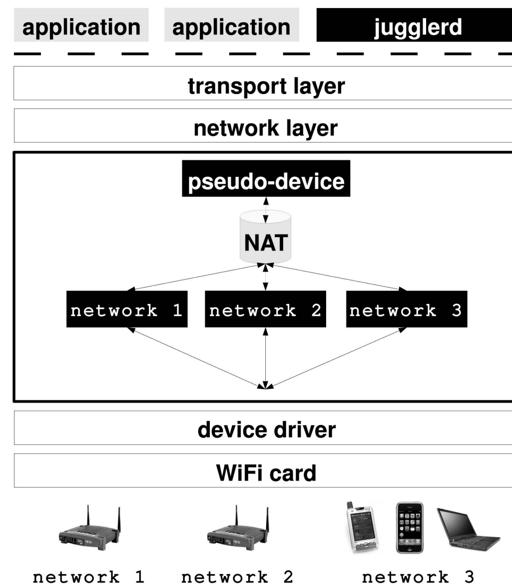

Fig. 1. **Juggler network stack.** One unchanging network interface is visible to upper layers of the network stack and to applications. Juggler maintains connection parameters (SSID, channel, DHCP configuration) for each virtual network with which it is associated, assigns sockets to virtual networks,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

If the client finds it has no buffered packets waiting, it deactivates its radio until the next timeout. But if data are 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 have been retrieved.

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

3 Juggler

Fig. 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], [30], we present a single, unchanging network interface to upper layers of the stack. This pseudodevice impersonates a wired Ethernet interface with a static, private IP

address and a synthetic Ethernet MAC address to distinguish it from the device's "real" network interfaces. 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.

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 to 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; and
- Radio duty cycle fraction that the network is active.

Tracking AP MAC address is critical in order to distinguish between different access points that share the same SSID (as is common in enterprise or campus deployments). 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 are 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.

Juggler is not designed to handle flow failover when moving from one wireless network to another. Existing solutions such as Mobile IP [21] and/or TCP Migrate [28] should be employed atop Juggler for this purpose.

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 pseudorandomly 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 and interactive session). Juggler will then match these needs with the connection quality of different virtual networks. We envision leveraging our prior work [20], which tracked user mobility and network availability to predict the quality and properties of access points users will encounter.

In the future, mobile devices may increasingly make use of services provided by other endpoints within an AP's private network—for example, an access point that doubles as a streaming media server. This is not a problem if all the wireless access points to which Juggler is connected have been configured to assign IP addresses from different private subnets. If subnets overlap, however, unmodified applications may have their flows go to an unintended destination. Consider the case where Juggler is connected to two APs that both assign client IP addresses in the 192.168.0.x subnet. The destination address 192.168.0.5 would be ambiguous in this situation. Juggler does not currently handle this usage case unless applications utilize the socket override option.

3.2 Sending and Receiving Packets

As illustrated in Fig. 1, upper layers of the network stack see only one network device. This pseudodevice 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 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.

A consequence of this NAT/reverse NAT architecture is that a security protocol such as IPsec would break unless explicit support were built into the Juggler layer of the networking stack. For example, Juggler could perform a man-in-the-middle "attack" during the initial key negotiation, in order to be able to rewrite the IP layer headers on the fly for subsequent data packets.

This pseudodevice 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:

First, 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.

Second, 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. Juggler may not find the MAC address it needs in the virtual network's ARP cache. In that case, Juggler enqueues the outbound data packet, 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 transmission of the original packet.

Third, perform network address translation. Because all sockets are bound to the internal pseudodevice, 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.

Fourth, forward for transmission. If the virtual network that owns this socket is currently active on the WiFi radio, Juggler immediately transmits the packet. This is done through the same function call interface that the network layer would use to contact the device driver if Juggler were not installed. If the virtual network that owns the socket is not active, the packet is enqueued.

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 pseudodevice 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 multistage 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 PSM bit set, indicating that the client is entering the 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 the 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 nonoverlapping channels 1, 6, and 11. In the current design, this is strictly passive scanning (no active probes are sent). This was a design decision to conserve battery life, because listening for packets or beacons consumes far less energy than actively transmitting a packet.

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 SSID and MAC address and channel number of the network, along with the mode (infrastructure, ad hoc, or scanning slot).

When the kernel module receives the request, it creates a virtual network structure (containing the outbound packet queue, ARP queue, etc.) and adds the new network to the end of the round-robin rotation. The new network is assigned the default timeslice duration—100 ms. 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 preemptively 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 percent of the radio duty cycle and AP2 10 percent, `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 2.6.22.14 kernel was required to automatically bind all sockets to the pseudodevice created by Juggler in order to capture all outbound flows, and to allow Juggler to perform

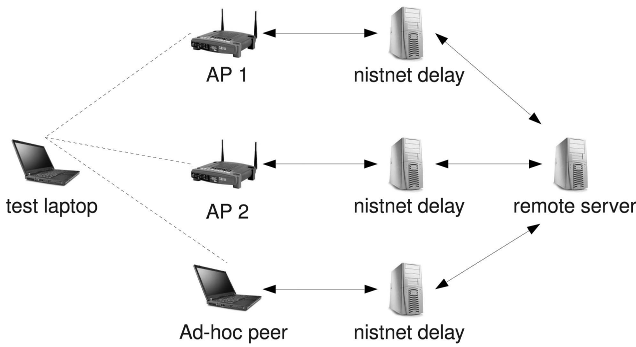

Fig. 2. **Laboratory setup.** A test laptop running Juggler can connect wirelessly to one of two 802.11g access points, or to another laptop in ad hoc mode. Three gateway routers use NIST Net to selectively throttle the bandwidth between each AP and the remote server. This simulates varied link quality between the test laptop and an Internet destination.

inbound NAT processing before packets are delivered to the network layer.

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, which leverages the common Linux `ieee80211` software MAC layer. The vast majority of our implementation is chipset-agnostic, however. Indeed, Juggler has since been ported to the Broadcom chipset with minimal effort.

We reduced the `rtl-wifi` driver's overly-cautious delay imposed whenever writing a value to the card over the PCI bus. For example, changing the radio frequency requires six 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 $500\ \mu\text{s}$, letting Juggler switch the radio frequency in $6 \times 500\ \mu s = 3\ \text{ms}$ 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 [4], [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 access points where wireless bandwidth outstrips that of the AP's back-end connection. This is certainly the case for DSL lines or cable modems, typical for residential settings, coffee houses, and other opportunistic public connectivity. This assumption may be invalid on corporate or academic campuses where APs connect directly to Gigabit Ethernet networks.

Fig. 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.

	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

Fig. 3. **Juggler: CPU overhead benchmarks.**

As illustrated in Fig. 2, the APs and the ad hoc peer were all connected with the remote server 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 backhaul 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.

We sought to minimize interference between the APs and wireless clients by distributing them throughout our laboratory. The laboratory environment was not electromagnetically shielded, however, introducing the possibility of interference from elsewhere in the building. For the case where two cards were used in one laptop, one card was the built-in WiFi card in the laptop and the other was a PCMCIA card. This minimized interference between the two cards as much as possible, because the internal WiFi's antenna was located high on the lid of the laptop, not next to the PCMCIA slot.

5 Microbenchmarks

Juggler works by interposing between the network and link layers of the operating system's protocol stack. 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 depends on the frequency with which the scheduler timer fires (4 ms for the Linux 2.6 kernel). 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` that allowed us to 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 multicore CPU of the test laptop, and disabled CPU frequency scaling so 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 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 10,000 times. We next repeated the experiment while connected to two APs that share the same channel. As Fig. 3 shows, the time to switch the radio's frequency clearly dominates switching time.

This switching overhead of just over 3 ms allows very fine-grained multiplexing of virtual networks. For example, if the AP duty cycle were set at 100 ms, only 3.3 percent of each usage period would be lost to overhead. VirtualWiFi's best-stated switching time was 25 ms, resulting in 25 percent overhead for the same 100 ms duty cycle. Users and applications must be mindful of this 3 ms overhead when tuning the switching frequency. While it is theoretically possible to configure Juggler to switch between networks every 4 ms, doing so would waste over 75 percent of the radio duty cycle to the unavoidable switching overhead.

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 pseudonetwork device. The results in Fig. 3 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 multitasking one wireless interface is beneficial. 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 depends on many factors, including interference or wireless link speed, load on the AP, quality of the AP's wired link to its ISP, and core and/or edge network congestion.

Residential broadband providers promise fairly high data rates. In the United States, for example, SBC advertises DSL links of 384-768 Kbps upstream and 768-6,144 Kbps down, while Comcast claims the same upstream bandwidth and 4,096-8,192 Kbps downstream over a cable modem. Verizon's FiOS fiber optic service is even faster—of 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 kilobits per second [19]. Independent measurements of broadband connectivity quality support these results [15].

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 that a time-consuming association and authentication process be completed before a client can communicate with an access point. One must also consider the time required to discover the next access point (by scanning for its beacon). Mhatre and Papagiannaki [17] showed that this fail-over time results in handoff delays of the order of 1-2 seconds.

Furthermore, our prior work [19] showed that the final part of the process—obtaining a DHCP configuration—takes several seconds under ideal conditions, and tens of seconds at worst. Emerging security protocols, such as WPA-PSK and IEEE 802.1x, add even further overhead to connection establishment. Migrating to a new AP therefore requires a significant data flow interruption. This overhead can be reduced by either requiring two physical radios or modifying AP firmware [10], [24]. Once associated to a new AP, however, the client must still configure IP-layer settings through DHCP before any useful data can flow.

Ideally, WiFi handoff would be as seamless as mobile phone handoff. 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.

In this section, we explore using Juggler to do just this. We assign 90 percent 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 percent of radio cycles are devoted to scanning for new access points and maintaining association with one or more secondary APs.

While the device is using the primary AP to transfer data, Juggler scans for new APs in the background, 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 [19]. Low-quality APs are dropped, and high-quality ones are assigned a small portion of the 10 percent 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 percent (without Juggler) to 90 percent 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 Fig. 2, the gateway machines between each AP and the remote server allowed us to simulate a range of bandwidths. Fig. 4 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 percent (entire radio devoted to one AP) and 90 percent (radio split between the AP at 90 percent and background scanning at 10 percent). The results show that reserving 10 percent 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 percent duty cycle, and assigned 10 percent to background scanning. We then powered up a new access point on a

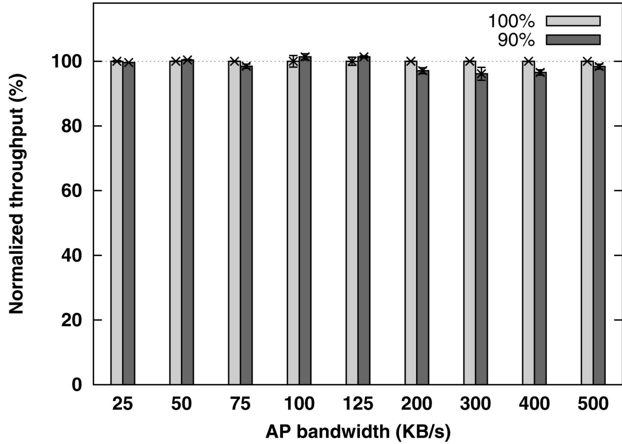

Fig. 4. **Soft handoff: throughput of primary AP.** Ninety percent of the duty cycle is devoted to a "primary" AP that handles all data flows, while 10 percent is used to discover new APs and maintain association with the backup AP(s). The reduction in primary bandwidth is small despite the loss of 10 percent of the radio duty cycle.

different channel from the primary AP, and measured the time between the new AP beginning to broadcast its beacon and the client completing the 802.11 association process. Table 1 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 1 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 percent 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 bidirectionally over TCP with the remote server as fast as possible over the current primary AP at 90 percent timeslice. The secondary AP was already configured and associated at a 5 percent timeslice, with scanning and discovery allocated the last 5 percent. 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. Clearly, faster and more sophisticated techniques for detecting a network disruption exist than simply relying on a socket timeout. We wanted to test a worst-case scenario, however, of how much benefit Juggler could bring to this problem without requiring any further modification to the system.

As Table 1 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

TABLE 1
Soft Handoff: Discovery and Fail-Over

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

Association *is the time from when the new AP began broadcasting beacons until Juggler finished associating with it.* DHCP *is the time to obtain a network configuration via DHCP.* Failover time *is the 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 time required to detect failure of the primary AP. Twenty trials.*

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 [19]. Rather, it is limited by the quality of the AP's back-end link (e.g., DSL and 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 its effectiveness when multiple network interfaces are present [22], [23], [25], [26]. 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 a BitTorrent client to stripe data torrent downloads across multiple access points.

6.2.1 Throughput Improvement

Recall the laboratory setup shown in Fig. 2. We used a 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 to transfer the entire file over one TCP connection. For the second case, we used two WiFi cards, each associated with a different access point, and created two sockets that were each bound to a different interface. The multithreaded server then sent each half of the file in parallel over the two sockets. The third case was the same, but using Juggler to associate simultaneously with both APs, each with 50 percent duty cycle, switching between APs every 100 ms.

The remote server represents an arbitrary Internet destination. By using the gateways lying between each AP and the remote server to throttle bidirectional bandwidth, we explored a range of application-level bandwidths between the client and server, from 25 up to 2,000 KB/s. We repeated

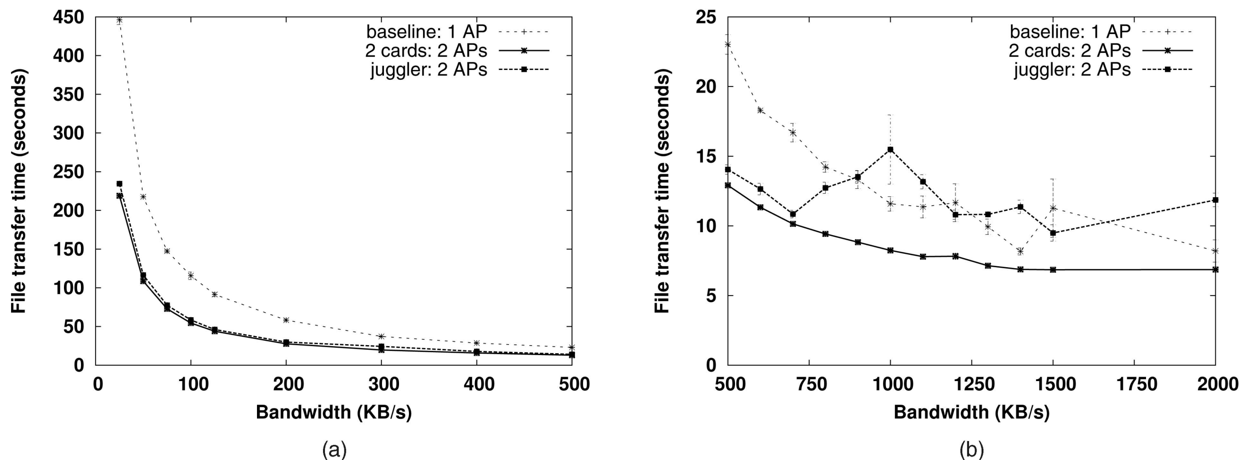

Fig. 5. **Data striping: throughput improvement.** A 10 MB file downloaded from a remote server. *Baseline*: associated with one AP, one TCP socket. *Juggler: 2 APs*: Juggler associated with two APs, 50 percent duty cycle for each AP, two TCP sockets each downloading half of the file over a different AP. *2 cards: 2 APs*: two WiFi cards, associated with different APs, to stripe the download. Mean over five runs. (a) 25 through 500 KB/s. (b) 500 through 2,000 KB/s.

each case for the range of AP bandwidths. The bandwidths for each AP were always equal and changed together.

Fig. 5 shows the time required to download the 10 MB file from the remote server, as a function of AP wired bandwidth. The results have been split into two graphs to provide more detail on performance in high-bandwidth situations. When the wired bandwidth between the APs and the remote server is relatively low, striping the file over two APs using Juggler results in nearly identical performance to using two physical cards—roughly twice as fast as when using only one access point. These performance gains hold until wired bandwidth exceeds 700 KB/s. This is far higher than typical residential broadband upstream data rates and comparable to ideal downstream quality.

To determine the cause of the degradation past this point, we used `iptables` to record TCP sequence numbers for all flows generated during the test runs. The sequence numbers were recorded at the server, for outbound flows (because the client was downloading from the server). Fig. 6 presents the results for selected values of wired AP bandwidth. Regressions in sequence number—downward "jags" in a graph—represent packet retransmissions. Across the board, using two physical cards, results in relatively few retransmissions. But as AP wired bandwidth increases, one can see an increase in the number of retransmissions for flows striped by Juggler.

Just as crucial to overall throughput are the relative slopes of the lines in each graph. A steeper slope corresponds to more data being stuffed into a given packet, because the 10 MB file transfer is completed using a smaller number of packets. The difference between these slopes for the "2 cards" and "juggler" cases is an artifact of TCP window size. It is clear that when striping over two cards, TCP ramps up the window size more quickly than for the Juggler case—a direct result of retransmissions. This window size effect is more pronounced for higher bandwidth cases, and explains why Juggler's performance degrades at those levels.

One possible cause of this behavior is Juggler's use of access points to buffer inbound packets while clients are tuned to a different AP. As bandwidth increases, more bytes are be transmitted from the server to the client while the client's radio is servicing a different AP. The IEEE 802.11 standard does not impose any required buffer size—indeed, APs are free to drop data as they see fit. We speculate this may be the cause of the increase in retransmissions for high-bandwidth situations.

Furthermore, one can see from Fig. 5 that even when two physical cards are used, one cannot double the throughput of the one AP case at high levels of wired AP bandwidth. We attribute this to the inherent overhead in TCP connection establishment, and artifacts of our laboratory setup (e.g., network interference).

6.2.2 Streaming Video

Unlike simple bulk downloading, streaming video is concerned with *when* specific parts of the video are downloaded. Earlier blocks have high priority, because users can watch the beginning of the video while later content is still being transferred. We modeled a simple video player that uses an earliest deadline first policy. 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, this has 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

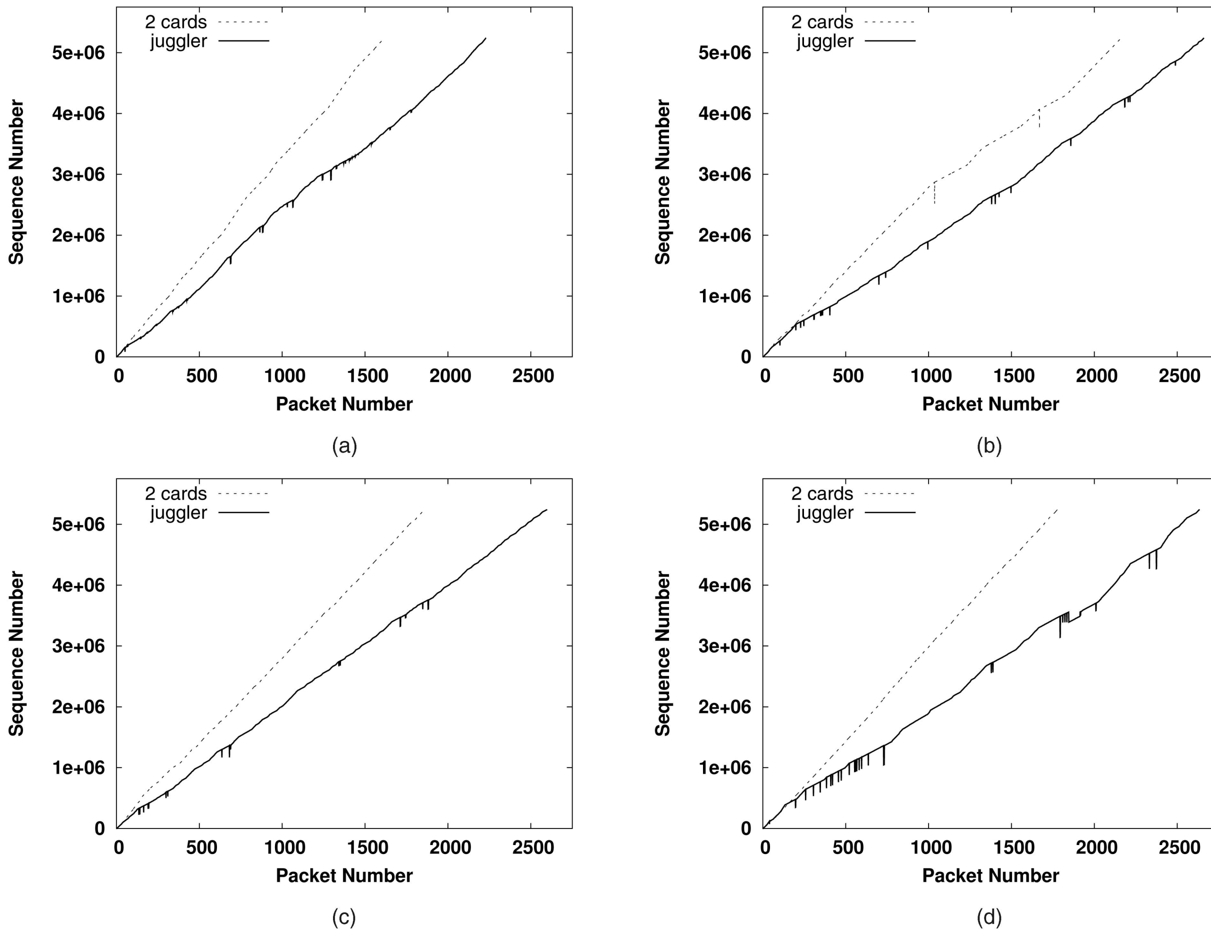

Fig. 6. **TCP sequence numbers.** "Packet Number" starts at zero for each TCP flow (a discrete test run from Fig. 5). "Sequence Number" is the sequence number in the TCP header for a given packet in the flow. (a) 100 KB/s. (b) 500 KB/s. (c) 1,000 KB/s. (d) 1,500 KB/s.

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 *postfacto* 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.

Fig. 7 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 upfront buffering time. We calculated the average size of playback gaps and the period in between gaps—during which time the video is playing. For one AP at 25 KB/s, the average gap size (6.091 seconds) is larger than the average intergap period (4.931 s). This results in a poor user experience, with the video constantly starting and stopping.

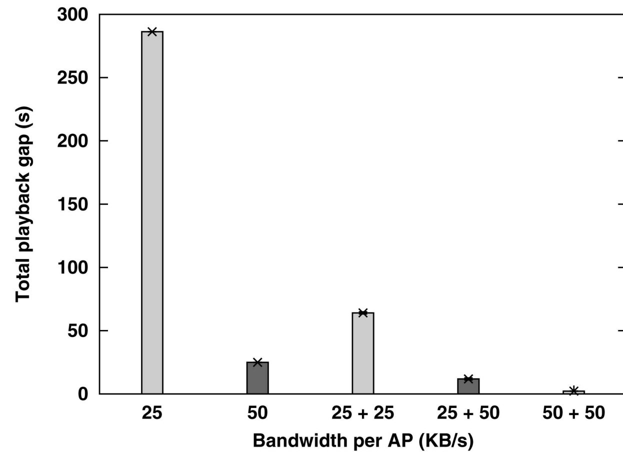

Fig. 7. **Data striping: streaming video.** Total playback gap per run, times in seconds. Video length was 204.8 s (10 MB encoded at 400 Kbps). Series labels refer to bandwidth available through the AP(s) over which the video was streamed. For instance, *25 + 50* means the client was connected to two APs at once, one of which had 25 KB/s of bandwidth, the other 50 KB/s.

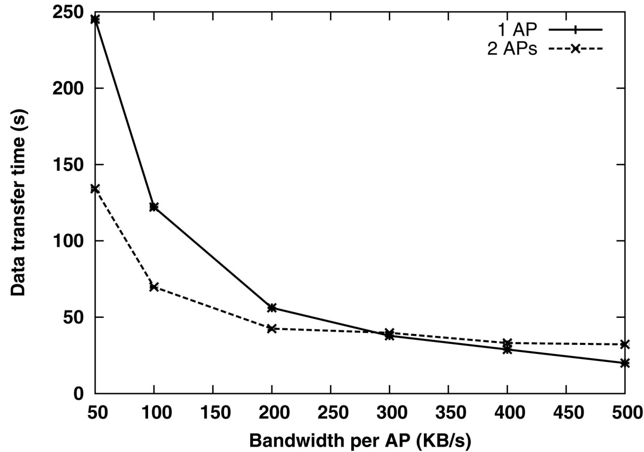

Fig. 8. **Data striping: BitTorrent.** Torrent download time. For both cases, blocks were downloaded from both seed servers. 10 MB data file.

When the single AP bandwidth is increased to 50 KB/s, we see 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. This is a result of TCP slow start, because each of the separate 25 KB/s links cannot ramp up as quickly as the one 50 KB/s link that never has to buffer packets. Using one 25 KB/s AP and one 50 KB/s AP, however, nearly eliminates all wait time. Finally, streaming over two APs, each offering 50 KB/s bandwidth, avoids wait time completely for 95 percent 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, 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. The client used on both seed machines was the official BitTorrent client version 3.4.2. The Pentium 4 seed also ran the tracker for the torrent.

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 percent duty cycle each.

Fig. 8 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 believe this overhead is due to artifacts in how the BitTorrent application-level protocol is affected by Juggler's buffering. 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 original motivation of VirtualWiFi was to let clients connect simultaneously to an infrastructure AP and to peers in ad hoc mode [9]. Such a side channel is clearly useful for communicating with the user's personal area network (PAN) [2], participating in mesh networks [12], or exploiting physical proximity for security [5].

Juggler's switching optimizations allow for a much finer-grained trade-off between foreground and background traffic than in prior work that has leveraged VirtualWiFi (such as WiFiProfiler [11]). We allocated 90 percent of the radio's duty cycle to an infrastructure AP representing the device's connection to the Internet. With the remaining 10 percent duty cycle, Juggler connected to another test laptop in ad hoc mode on a nonoverlapping 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 percent of the radio duty cycle.

Fig. 9 shows negligible throughput difference between using the entire radio capacity and reserving 10 percent for a side channel, even for high values of AP bandwidth. As expected, the throughput of the 10 percent 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

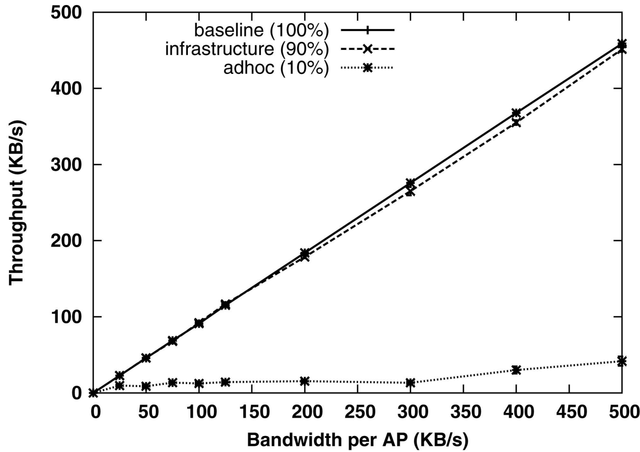

Fig. 9. **Mesh connectivity: TCP throughput.** "Baseline" is the maximum TCP throughput when Juggler was not active. Juggler connects simultaneously to an infrastructure AP (with 90 percent of the radio) and a nearby device in ad hoc mode (with 10 percent of the radio).

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

7.1 Interface Multiplexing

VirtualWiFi [9] was the first system to multiplex a single wireless interface across multiple connection points, exposing each of those points as a separate interface to the application layer. While the switching time between interfaces was significant, the authors suggest several mechanisms to minimize that cost.

Two contemporaneous projects have implemented these mechanisms, but with very different interfaces—our own work, and FatVAP [14]. Both modify wireless drivers to allow fast channel switching, use an NAT/reverse NAT architecture to multiplex different AP network endpoints, and exploit the power save mode feature on APs for packet buffering. The performance of AP switching and stream multiplexing are similar, with FatVAP achieving throughput gains closer to the theoretical ideal of multiple physical radios, transmitting in parallel, than does Juggler.

Where Juggler and FatVAP differ is in the interface and services they present to applications and users. FatVAP is designed primarily to aggregate network bandwidth into one logical pipe. Applications simply send and receive packets as usual, and FatVAP handles the details of multiplexing data flows across multiple APs.

In contrast, Juggler acts as a low-level service that higher layers can make use of in various ways, in an attempt to provide a separation of concerns. This allows several uses of interface multiplexing beyond bandwidth aggregation, as this paper has shown. Many of these were suggested by Bahl et al. [3]. This work 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.

7.2 Network Discovery and Handoff

Brik et al. [6] aim to hide 802.11 handoff latency by using a second physical radio, which scans for new access points and preassociates with them before the current connection becomes unusable. This is quite similar to our evaluation scenario in Section 6.1. Their implementation achieves zero handoff latency, while our results show a data interruption of 8 ms. Juggler achieves this, however, by using only one radio instead of requiring extra hardware on mobile devices where power consumption and form-factor are primary concerns.

Mhatre and Papagiannaki [17] reduce 802.11 handoff latency by continuously monitoring beacon strengths, and maintaining histories of these trends. Their system then triggers a handoff based both on current conditions and these past histories, reducing handoff overhead by 50 percent. In this paper, we similarly used Juggler preemptively with the next access point before the current AP became unusable. The only handoff latency we incur is the time required to trigger the transition. We evaluated the simplistic situation where the current AP suddenly becomes completely unavailable. Combining their sophisticated fail-over triggering scheme with Juggler's ability to associate with multiple APs would reduce handoff latency to near zero.

SyncScan [24] 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. maintain *neighbor graphs*—sequences of AP handoffs [27]. Clients build graphs by direct observation and through cooperative peer sharing. When an 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.

7.3 Data Striping and Aggregation

MAR is a stand-alone physical device that aggregates many heterogeneous wireless links into one logical, high-bandwidth pipe [26]. Its focus is on combining the capacity of many physical radios, while Juggler connects to multiple networks through only one radio.

Horde [23] 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 [22].

PRISM [16] aggregates and shares wireless infrastructure bandwidth among mobile nodes, by striping packets of one TCP flow across disjoint links. Because this may result in out-of-order delivery, their system reorders ACKs to preserve

expected TCP semantics. Their results are intriguing for the future development of Juggler, because some of our throughput inefficiency is a result of these TCP side effects.

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.

7.4 Mesh Networks and Side Channels

Both Client Conduit [1] and WiFiProfiler [11] use 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 [5], [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.

7.5 Robustness through Diversity

Multiradio Diversity (MRD) uses redundant wireless channels to reduce packet losses and improve throughput [18]. Devices receive different channels simultaneously over multiple network interfaces, and transmit upstream in parallel to multiple, coordinated access points to ensure faithful reception. A system like Juggler could be employed as an interesting enhancement to MRD, leveraging the seamless handoff application described in our evaluation section.

Vergetis et al. [29] 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, if switching delays could be reduced below 1 ms. 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 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 link-layer implementation of an 802.11 virtual networking service. By leveraging network cards that perform the MAC layer in software, Juggler switches between wireless networks in just over 3 ms, or less than 400 μs 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's performance in several realistic usage scenarios. We show how mobile clients can enjoy nearly instantaneous 802.11 handoff by reserving 10 percent of the radio duty cycle for background AP discovery, while minimally impacting foreground transfers. Juggler also enhances data throughput when wireless bandwidth is superior to that of an AP's wired, backend connection, by striping data across many networks. 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.

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) under award numbers CNS-0509089 and CNS-0615086, and the Ford Motor Company. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the NSF or Ford Motor Company.

References

[1] A. Adya, P. Bahl, R. Chandra, and L. Qiu, "Architecture and Techniques for Diagnosing Faults in IEEE 802.11 Infrastructure Networks," *Proc. ACM MobiCom*, Sept. 2004.

[2] M. Anand and J. Flinn, "PAN-on-Demand: Building Self-Organizing WPANs for Better Power Management," Technical Report CSE-TR-524-06, Univ. of Michigan, 2006.

[3] P. Bahl, A. Adya, J. Padhye, and A. Walman, "Reconsidering Wireless Systems with Multiple Radios," *ACM SIGCOMM Computer Comm. Rev.*, vol. 34, no. 5, pp. 39-46, Oct. 2004.

[4] P. Bahl, R. Chandra, and J. Dunagan, "SSCH: Slotted Seeded Channel Hopping for Capacity Improvement in IEEE 802.11 Ad-Hoc Wireless Networks," *Proc. ACM MobiCom*, Sept. 2004.

[5] D. Balfanz, D. Smetters, P. Stewart, and H. Wong, "Talking to Strangers: Authentication in Ad-Hoc Wireless Networks," *Proc. Ninth Ann. Network and Distributed System Security Symp. (NDSS '02)*, Feb. 2002.

[6] V. Brik, A. Mishra, and S. Banerjee, "Eliminating HandOff Latencies in 802.11 WLANs Using Multiple Radios," *Proc. Fifth ACM SIGCOMM Conf. Internet Measurement (IMC '05)*, 2005.

[7] S. Capkun, J.-P. Hubaux, and L. Buttyan, "Mobility Helps Security in Ad-Hoc Networks," *Proc. ACM MobiHoc*, pp. 46-56, June 2003.

[8] M. Carson and D. Santay, "NIST Net—A Linux-Based Network Emulation Tool," *ACM SIGCOMM Computer Comm. Rev.*, June 2003.

[9] R. Chandra, P. Bahl, and P. Bahl, "MultiNet: Connecting to Multiple IEEE 802.11 Networks Using a Single Wireless Card," *Proc. IEEE INFOCOM*, pp. 882-893, Mar. 2004.

[10] R. Chandra, J. Padhye, L. Ravindranath, and A. Wolman, "Beacon-Stuffing: Wi-Fi without Associations," *Proc. Eighth IEEE Workshop Mobile Computing Systems and Applications (HotMobile)*, 2007.

[11] R. Chandra, V.N. Padmanabhan, and M. Zhang, "WifiProfiler: Cooperative Diagnosis in Wireless LANs," *Proc. ACM MobiSys*, June 2006.

[12] R. Draves, J. Padhye, and B. Zill, “Routing in Multi-Radio, Multi-Hop Wireless Mesh Networks,” *Proc. ACM MobiCom,* pp. 114-128, Sept. 2004.
[13] T.J. Hacker, B.D. Noble, and B. Athey, “Improving Throughput and Maintaining Fairness Using Parallel TCP,” *Proc. IEEE INFOCOM,* Mar. 2004.
[14] S. Kandula, K.C.-J. Lin, T. Badirkhanli, and D. Katabi, “FatVAP: Aggregating AP Backhaul Capacity to Maximize Throughput,” *Proc. Fifth USENIX Symp. Networked Systems Design and Implementation (NSDI ’08),* Apr. 2008.
[15] J.A. Kaplan, “Real World Testing: The Best ISPs in America,” *PC Magazine,* May 2007.
[16] K.-H. Kim and K.G. Shin, “Improving TCP Performance Over Wireless Networks with Collaborative Multi-Homed Mobile Hosts,” *Proc. ACM MobiSys,* pp. 107-120, June 2005.
[17] V. Mhatre and K. Papagiannaki, “Using Smart Triggers for Improved User Performance in 802.11 Wireless Networks,” *Proc. ACM MobiSys,* June 2006.
[18] A. Miu, H. Balakrishnan, and C.E. Koksal, “Improving Loss Resilience with Multi-Radio Diversity in Wireless Networks,” *Proc. ACM MobiCom,* pp. 16-30, 2005.
[19] A.J. Nicholson, Y. Chawathe, M.Y. Chen, B.D. Noble, and D. Wetherall, “Improved Access Point Selection,” *Proc. ACM MobiSys,* pp. 233-245, June 2006.
[20] A.J. Nicholson and B.D. Noble, “Breadcrumbs: Forecasting Mobile Connectivity,” *Proc. ACM MobiCom,* Sept. 2008.
[21] C.E. Perkins, “Mobile IP,” *IEEE Comm. Magazine,* vol. 35, no. 5, May 1997.
[22] A. Qureshi, J. Carlisle, and J. Guttag, “Tavarua: Video Streaming with WWAN Striping,” *Proc. ACM Multimedia (MM ’06),* pp. 327-336, Oct. 2006.
[23] A. Qureshi and J. Guttag, “Horde: Separating Network Striping Policy from Mechanism,” *Proc. ACM MobiSys,* pp. 121-134, June 2005.
[24] I. Ramani and S. Savage, “SyncScan: Practical Fast Handoff for 802.11 Infrastructure Networks,” *Proc. IEEE INFOCOM,* pp. 675-684, Mar. 2005.
[25] P. Rodriguez and E.W. Biersack, “Dynamic Parallel Access to Replicated Content in the Internet,” *IEEE/ACM Trans. Networking,* vol. 10, no. 4, pp. 455-465, Aug. 2002.
[26] P. Rodriguez, R. Chakravorty, J. Chesterfield, I. Pratt, and S. Banerjee, “MAR: A Commuter Router Infrastructure for the Mobile Internet,” *Proc. ACM MobiSys,* pp. 217-230, June 2004.
[27] M. Shin, A. Mishra, and W.A. Arbaugh, “Improving the Latency of 802.11 Hand-Offs Using Neighbor Graphs,” *Proc. ACM MobiSys,* pp. 70-83, June 2004.
[28] A.C. Snoeren and H. Balakrishnan, “An End-to-End Approach to Host Mobility,” *Proc. ACM MobiCom,* pp. 155-166, 2000.
[29] E. Vegetis, E. Pierce, M. Blanco, and R. Guerin, “Packet-Level Diversity—from Theory to Practice: An 802.11-Based Experimental Investigation,” *Proc. ACM MobiCom,* pp. 62-73, Sept. 2006.
[30] X. Zhao, C. Castelluccia, and M. Baker, “Flexible Network Support for Mobility,” *Proc. ACM MobiCom,* pp. 145-156, 1998.

Anthony J. Nicholson received the PhD degree in computer science and engineering from the University of Michigan in 2008. He is currently a software engineer at Google in Chicago, Illinois. His research interests include mobile and pervasive computing and networking.

Scott Wolchok received the BSE degree in computer science from the University of Michigan in 2008. He is currently working toward the PhD degree in the Electrical Engineering and Computer Science Department at the University of Michigan. His research focuses on security in mobile and embedded systems.

Brian D. Noble received the PhD degree in computer science from Carnegie Mellon University in 1998 and is a recipient of the US National Science Foundation CAREER Award. He is an associate professor in the Electrical Engineering and Computer Science Department at the University of Michigan. His research centers on software supporting mobile devices and distributed systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.