Late-Binding: How to Lose Fewer Packets during Handoff

Kok-Kiong Yap Te-Yuan Huang Yiannis Yiakoumis Nick McKeown Sachin Katti
Stanford University
{yapkke,huangty,yiannisy,nickm,skatti}@stanford.edu

ABSTRACT

Current networking stacks were designed for a single wired network interface. Today, it is common for a mobile device connect to many networks that come and go, and whose rates are constantly changing. Current network stacks behave poorly in this environment because they commit an outgoing packet to a particular interface too early, making it hard to back out when network conditions change. By default, Linux will drop over 1,000 packets when a mobile client associates to a new WiFi network. In this paper, we introduce the concept of *late-binding* packets to their outgoing interfaces. Prior to the binding point different flows are kept separate, to prevent unnecessarily delaying latency-sensitive traffic. After the binding point buffers are minimized—in our design, down to just two packets—to minimize loss when network conditions change. We designed and implemented a *late-binding* Linux networking stack that empirically demonstrates the value of our proposition in minimizing delay of latency-sensitive packets and packet loss when networks come and go.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Architecture and Design*

General Terms

Design, Management, Performance

Keywords

Buffer Management, Late-binding, Mobile Devices

1. INTRODUCTION

The network stacks we use on hand-held devices today were originally designed for desktop machines with only one, stable wired network connection. When transmitting data,

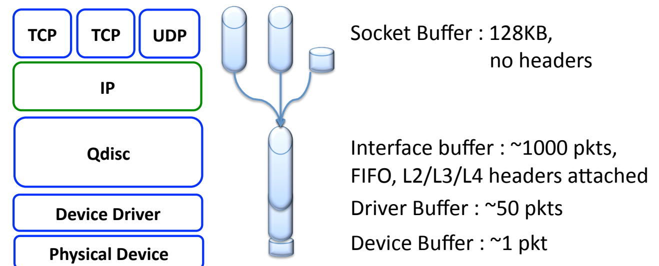


Figure 1: Layer and buffer stages in Linux.

the network stack follows a simple sequence of actions: It segments an application's data into packets, adds the network addresses, buffers each packet until its turn to depart, and then ships it out over a network interface. In wired networks with stable network connectivity, this design works well.

Things do not work so well when the device utilizes two or more networks (e.g., 3G, 4G and WiFi). In the standard network stack, typified by Linux and Android, the fate of a packet—in terms of the interface it is sent on and in what order—is determined the moment the IP addresses are added and the packet is *bound* to a particular interface. The majority of packet buffering takes place *after* the IP address has been added and after the packet has been committed to an interface. When a device starts using a new network (i.e., via a different gateway), or switches to a different network interface, we lose all the packets queued up in the buffers of the disconnected network interface. The buffers are often large (hundreds or thousands of packets), leading to a large number of lost packets.

If handoffs were rare and network conditions were constant, occasional packet loss might be acceptable. But in a world with ever smaller cells in mobile networks and many wireless networks to choose from, mobile devices frequently remap flows to new networks or interfaces (e.g., during WiFi offloading), and so we need to reduce packet loss.

The key problem is that packets are bound to an interface too early. Once an IP header has been added, and a packet is placed in the per-interface queue, it is very hard to undo the decision (e.g., if the interface switch to a new network) or if we want to send the packet to a different interface (e.g., if the interface fails, or if a preferred interface becomes available). The more packets we buffer below the binding point, the greater the commitment, and the more packets we lose if the network conditions change. We show that even in the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CellNet'13, June 25, 2013, Taipei, Taiwan
Copyright 2013 ACM 978-1-4503-2074-0/13/06 ...$15.00.

best configuration a typical mobile device loses 50 packets each time it hands off or changes interface.

A second problem caused by binding too early is that urgent packets are unnecessarily delayed. Because many transport flows are multiplexed into a single per-interface FIFO, latency sensitive traffic is held up. The problem is worst when the network is congested and data backs up in the per-interface queue. We show that urgent packets can easily be delayed by over 2.15 s on a WiFi interface with the default settings.

Our goals are to: (1) Avoid losing packets unnecessarily when we handoff to a new network, when a link goes up/down, or when we choose to use a new interface. (2) Avoid unnecessarily delaying latency-sensitive packets.

In this paper we advocate the principle of *late binding* for a mobile device network stack design, i.e., the decision on which packet to send on what interface is not made until the last possible instant. To realize this principle, we adopt a design where

1. Before the binding, flows are kept in separate interface-independent buffers.

2. After the binding, buffers are eliminated or made very shallow.

The key insight is that by minimizing buffering after the binding point we retain control of packet order and network interface until the last possible moment. Through our Linux prototype, we demonstrate that during network connectivity changes, late binding reduces the number of packet drops from 50 packets down to 0 or 1 packets. The design also reduces the delay of latency-sensitive traffic from 135 ms to less than 8 ms.

The rest of this paper is organized as follows. §2 describes the current network stack design in Linux. §3 describes the late-binding design principle and describes our prototype implementation. We present our evaluation results in §4. We discuss related work in §5 before concluding in §6.

2. THE LIFE OF A PACKET

To help us understand where a packet is bound to an interface and where the buffering takes place, we will follow a TCP packet from the application through the Linux network stack to the network interface. Figure 3 shows the four main stages of processing and buffering in the stack in transmit path.

Step 1: An application creates a socket (with a 128 kB socket buffer by default) for each TCP flow. When the application writes data to a socket, the data is transferred from user space into the kernel context (Step 1 in Figure 3). If the socket buffer is full, further writes to the buffer are blocked until there is room. The socket buffer is also where control state (e.g., congestion window, number of outstanding packets) is maintained.

Step 2: Packets leave the socket buffer according to TCP's state machine. At this point, TCP/IP adds the transport and IP headers and hands the packet to a per-interface queue called *qdisc* (short for "queueing discipline")—a 1,000 packet drop-tail FIFO that multiplexes all flows sharing a common interface. At this point, just as the packet is written into the per-interface queue, the packet is bound and committed to a particular interface.

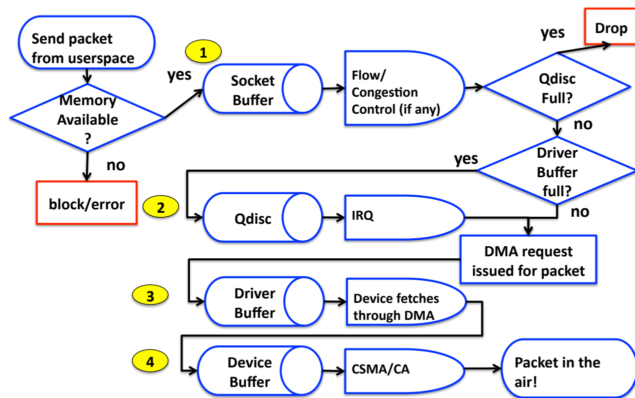


Figure 3: The flow of packets through Linux.

Step 3: Packets are handed to the driver. The device driver for each interface maintains a small *driver buffer* to hold packets waiting to be DMA'd into the network interface. For example, the Atheros `ath5k` WiFi driver creates a 50 packet driver buffer in main memory; when it fills, the buffer sets a flag to prevent the layers above from overflowing the buffer. When space becomes available, it issues an interrupt to resume transmissions.

Step 4: Packets are DMA'd from the driver buffer in main memory into the network device's local memory where they are held until they are transmitted on the air. The Atheros WiFi card that we use in our evaluations holds only one packet, although in principle the physical memory buffer might be larger.[1]

While packets pass through four queues, they are copied only twice: from user-space to the kernel's socket buffer and then via DMA from the kernel to the device buffer. The other two queue transfers are by handing a memory pointer from one layer to the next.

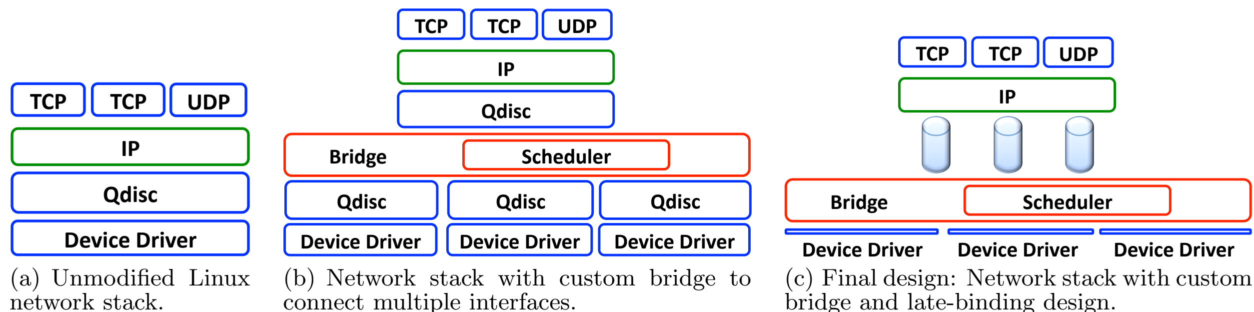
The fate of a packet is determined as soon as the packet is added to the qdisc. By this stage the IP header has been added, explicitly determining which network interface will be used. Beyond this point the packets are, by default, sent in FIFO order and an arriving packet might wait for up to 1,051 packets to be sent ahead of it (in the qdisc, driver and device buffers). If the qdisc is replaced by a priority discipline, urgent packets can still wait behind 51 packets in the device driver and network interface.

When Linux stops using a network interface, or if it hands-off to a new access point (AP): (1) Linux will drop all the packets in qdisc, the driver buffer and the device buffer; they were committed to a different set of network addresses. We will show that in practice a lot of packets are dropped unnecessarily, causing low TCP throughput and timeouts. (2) If an urgent packet arrives to qdisc, head-of-line blocking will make it wait for up to 1,051 packets (in the worst case, more than a second for a 10 Mb/s interface), or 51 packets (still over 100 ms, if qdisc is replaced).

3. DESIGN AND IMPLEMENTATION

We can overcome the two problems described above if the network stack has the following properties:

[1] It has been informally indicated that the chipset used for the OLPC project can buffer up to 4 packets [12].



(a) Unmodified Linux network stack.

(b) Network stack with custom bridge to connect multiple interfaces.

(c) Final design: Network stack with custom bridge and late-binding design.

Figure 2: Network stacks to illustrate the changes made to the Linux to implement late-binding.

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

2. Keep flows in separate queues above the binding point so that latency-sensitive packets are not unnecessarily delayed. The queues need to be interface-independent to allow us to choose which packet to send on which interface.

To make it easier for our approach to be adopted, we also require: (1) Applications should run unmodified, which means we cannot change the socket API. (2) The driver code should not be changed. (3) Our design should support existing transport protocols, including TCP, UDP, SCTP. In this section, we describe a modified Linux network stack that meets these requirements.

Starting from the default Linux network stack illustrated in Figure 2(a), we insert a *custom bridge* between the IP layer and the network interfaces, as shown in Figure 2(b) (first described in [25]). The custom bridge has two main properties. First, it remaps the IP addresses used above the bridge, to the specific interface addresses below. When an application sends data, it is initially bound to a private, virtual IP address that does not correspond to any of the physical interfaces. Once the bridge decides which interface to send the packet to, it maps the IP address to the correct address. Second, the bridge contains a packet scheduler. When an interface queue becomes available, it decides which packet to send next. This is the point at which a packet is bound to its outgoing interface. This design has the effect of leaving the socket API unchanged and making the application believe it is using a single unchanging interface. However, now there are qdisc buffers above *and* below the bridge.

To keep flows separate *above* the binding point (the bridge), we replace the default qdisc with a custom queueing discipline that keeps a separate queue for each socket. Linux makes this easy.

To minimize buffering *below* the binding point, we completely bypass the per-interface qdisc by partially reimplementing `dev_queue_xmit` to directly invoke `dev_hard_start_xmit` to deliver a packet—as soon as it has been scheduled—directly to the device driver. If the device driver buffer is full, it returns an error code that allows us to retry later. The consequence is that we bypass qdisc without having to replace it.

We reduce the driver buffer from 50 down to two packets using the Atheros `ath5k` driver configuration tool, `ethtool`.[2] Below two packets the DMA process becomes unstable and the interface disconnects.

We leave the receive path unchanged, except to forward all received packets to the virtual interface. The socket API is therefore unchanged. The final design is shown in Figure 2(c).

To evaluate our design, we implemented it in Linux 3.0.0-17 in the form of a kernel module making use of the `netdev_frame_hook` available since Linux 2.6.36. The implementation runs on a Dell laptop running Ubuntu 10.04 LTS with an Intel Core Duo CPU P8400 at 2.26 GHz processor, 2 GB RAM and two WiFi interfaces. The two WiFi interfaces are Intel PRO/Wireless 5100 AGN WiFi interface and Atheros AR5001X+ wireless network adapter connected via PCMCIA.

4. EVALUATION

In this section, we report results from experiments that show late binding almost completely eliminates packet loss during network handover, and greatly reduces the delay of urgent packets. Except where noted, our results are based on our implementation of late-binding in Linux, as shown in Figure 2(c), while varying the buffer sizes.

4.1 Reduced Packet Loss by Late-Binding

Our first goal is to avoid losing packets unnecessarily when a flow is re-routed over a different network interface. Recall that in standard Linux, the entire contents of the qdisc buffer, the driver buffer and the device buffer can be lost when a flow is mapped to a new interface, or its packets travel via a new gateway. In the default configuration, this can be over 1,000 dropped packets. Our test scenario is a Linux mobile device with two WiFi interfaces, each associated to a different access point. A TCP flow is established via interface 1; then we disconnect interface 1 and re-route the flow to interface 2. We expect packets to be dropped when interface 1 is disconnected; we measure how many are dropped as a function of the amount of buffering below the binding point (in our prototype we have already eliminated the qdisc buffering below the binding point entirely). We also measure the effect the retransmissions have on TCP throughput.

[2]The Intel and Broadcom WiFi chipset we have looked at do not let us set the driver buffer size.

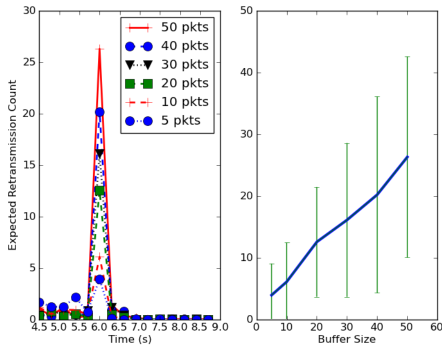


Figure 4: Left: the average number of retransmissions (in 0.3 s bins) for a TCP Cubic flow; interface 1 is disconnected at 6s. The legend shows the size of the DMA buffer. Right: the average number of retransmissions (error bars show standard deviation) immediately after disconnecting interface 1.

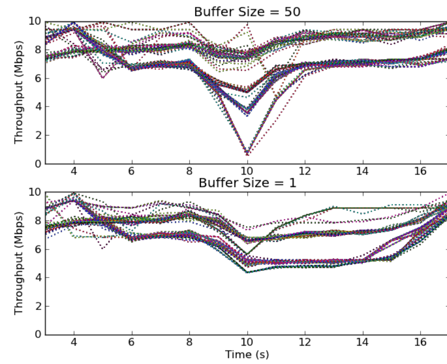


Figure 5: Throughput of a flow when 50 (above) or 1 (below) packets are dropped after 10s; for 100 independent runs.

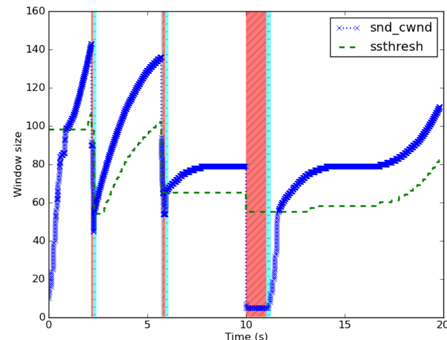


Figure 6: Sender congestion window and slow-start threshold of a single TCP Cubic flow with 50 packets dropped at 10s. The wide (red) vertical bar indicates that the socket is in recovery phase, while the narrower (cyan) vertical bars indicate Cubic's disorder phase.

Figure 4 shows the number of packets retransmitted by the TCP flow over 0.3 s intervals for an unmodified TCP Cubic flow with a throughput of ≈ 5 Mbps and RTT 100 ms. We disconnect interface 1 after approximately six seconds and repeat the experiment 100 times then average the results.

The graph clearly shows that the number of retransmissions is proportional to the size of the interface buffer. These are the packets that were bound to interface 1 and were waiting below the binding point, and were lost when the interface was turned off. With the default driver buffer of 50 packets, we lose an average of 26.3 packets. We reduce the buffer to just five packets, and the loss is reduced to an average of 3.9 packets. We did not include the experiment with a buffer of 1,000 packets because TCP will simply timeout in that case.

Next we evaluate the effect on TCP throughput when we re-route flows. Ideally TCP throughput would be unaffected, but we know TCP reacts adversely to a long burst of packet losses. In this experiment, we emulate the effect of packet loss during handover when the buffer size is down to just one packet, using a modified Dummynet [6] implementation. We establish a 10 Mb/s TCP Cubic flow (with RTT of 100 ms) through interface 1 and—to emulate disconnecting interface 1 after 10s and re-routing through interface 2—we drop either 1 or a burst of 50 packets. The experiment was run 100 times and the throughput was measured using `tcpdump` (to reconstruct the flow).

Figure 5 shows that losing a burst of 50 packets (corresponding to a driver buffer of 50 packets) the throughput can drop significantly, with some flows dropping to almost zero. If we reduce the interface buffer to only one packet, throughput is affected much less, with no flow dropping below 4Mb/s.

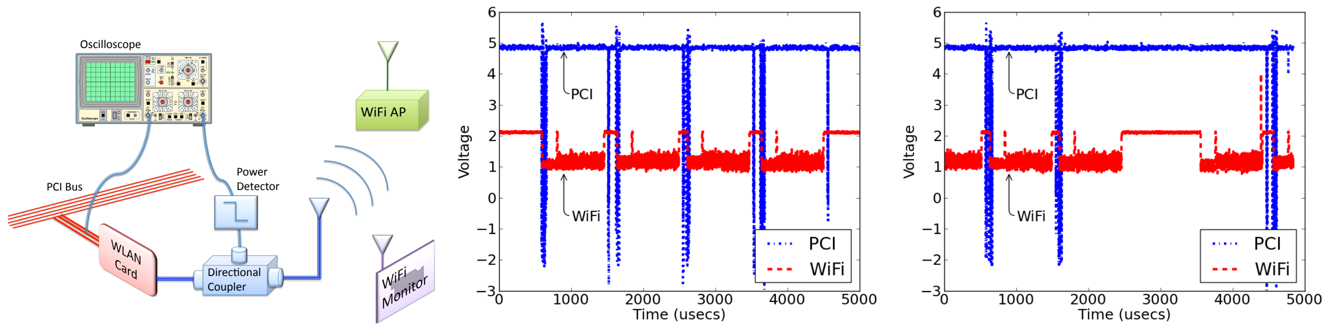
To better understand the effect of buffer size on TCP, we examine the dynamics of TCP's congestion window after the loss occurs. We modified TCP probe [13, 24] to tell us the congestion state of the socket and the sender congestion window `snd_cwnd`. The evolution of the state of the TCP flow when we drop 50 packets is plotted in Figure 6 together with the slow start threshold `ssthresh`. The burst of drops causes TCP to enter the recovery phase for over a second. The actual effect varies widely from run to run depending on the state of the TCP flow when the loss happens. This should come as no surprise as it has been observed many times that TCP throughput collapses under bursts of losses (e.g., [9]). If the packets were not unnecessarily dropped, due to early binding, the throughput would be more stable.

4.2 Latency-sensitive Traffic

Our second goal is to minimize the delay of latency-sensitive packets. As a benchmark, we start by measuring the delay of a high priority packet through the default Linux stack with a 1,000 packet qdisc. We then measure how much the delay is reduced in our prototype as we vary the size of the driver buffer.

Our experiment uses a single WiFi interface using the Atheros `ath5k` driver. For the default Linux stack, we send a marker packet, followed by a burst of 1,000 UDP packets (to fill qdisc), followed by a single urgent packet. We use `tcpdump` to measure the time from when we receive the marker packet until we receive the urgent packet. The experiment is repeated 50 times. We then repeat the experiment with our prototype late-binding stack, repeating the experiment for different device buffer sizes.

The results in Figure 7 show that an urgent packet can be delayed a long time. With the default Linux settings, the median delay of the urgent packet is 2.2 seconds. With our late-binding stack, it is reduced to 135 ms as soon as



(a) Setup for measuring buffer size of WiFi device by comparing PCI signals and antenna output.

(b) PCI and WiFi outputs for a 4-packet burst on a WiFi card. At most one packet is inside the device at any time.

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

Figure 8: Measuring buffer in WiFi device

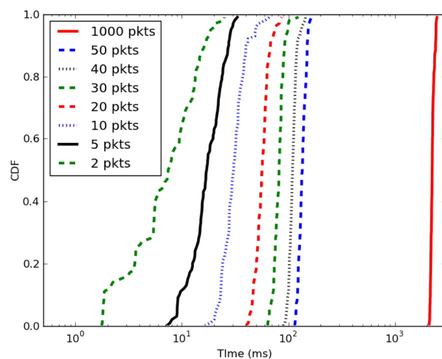


Figure 7: CDF of the time difference between the marked and prioritized packet.

we remove qdisc. And if we reduce the driver buffer from a default of 50 to only two packets, the median delay drops to just 7.4 ms, i.e., 0.3 % of the delay as the regular Linux stack. If we could reduce the driver buffer to just one packet, we expect an urgent packet to be delayed less than 5 ms.

4.3 Size of Device Buffer

To evaluate how late we can bind a packet, we need to know the size of the hardware buffer and whether we can reduce it. Manufacturers do not publish the size of the buffer inside the interface, and so we set out to measure it.

Measuring the buffer size turns out to be surprisingly hard, and there are no utilities to configure the size. Hence we designed an experiment to reverse engineer the buffer size. We used a TP-Link TL-WN350GD card equipped with an Atheros AR2417/AR5007G 802.11 b/g chipset using the `ath5k` driver. Figure 8(a) shows our experimental setup. We measure the time from when a packet is DMA'd into the WiFi chip (by monitoring the `FRAME` pin on the PCI bus) until the packet emerges from the interface and is sent to the antenna (using a directional coupler to tap the signal, and a power detector connected to an oscilloscope).

Figure 8(b) shows PCI and antenna activity when we send a burst of four 1400-bytes UDP packets at 18 Mb/s. Both signals are active low, i.e., a low voltage implies activity. On the PCI bus we see the packet being transferred to the wireless chip, and a short status descriptor being sent back to the host after the transmission. On the antenna we see a CTS-to-self packet [3], followed by a SIFS (short inter frame space) and then the actual packet transmission. Notice that as soon as one packet finishes, the DMA transfer for the next packet is triggered. This is particularly clear in Figure 8(c), which shows the retransmission of a packet—verified by WiFi monitoring sniffing the channel. There is no PCI activity during the contention and retransmission phase. This indicates a pipelined, low-latency design.

The result is encouraging—there is at most one packet in the network interface at a time. This tells us we can make the buffering very small below the binding point. We only need to change software in the operating system. The Atheros chipset is connected to the CPU via PCI. We expect (but still need to verify) that there is only one packet buffer in more integrated solutions, such as the system-on-chip designs used in modern mobile handsets.

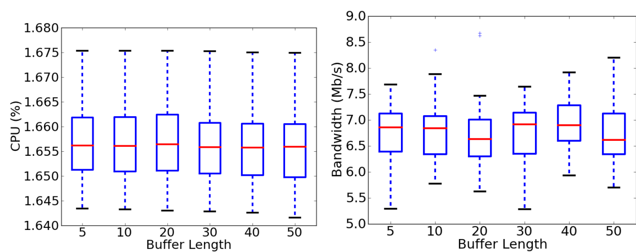
4.4 Overhead: Increased CPU interrupts

As we make the buffers smaller, we can expect more interrupts for outgoing packets. With a large device driver buffer, a DMA can shortly follow the previous, because there are still packets waiting. There is no need for an interrupt. If we make the driver buffer very small, it will go empty more often and needs to be refilled by an interrupt. Therefore, we measured the extra load placed on the CPU.

We started a maximum rate TCP flow, and measured the CPU load for difference sized driver buffers. The CPU load hovered around 1.6% and we could measure no change in load as a function of buffer size (as in Figure 9(a)). There was no change in TCP goodput either (as presented in Figure 9(b)). This is probably because wireless interfaces are quite slow for a modern CPU. For a high speed wired interface (e.g., 10 GE) the rate of interrupts would be much higher. If the same method was to be used for wireline interfaces, a deeper evaluation would be needed.

5. RELATED WORK

Many researchers have explored how to use multiple wireless interfaces at the same time [5, 7, 10, 21]. This includes work like [8, 16] that propose infrastructural changes to make better network choices. Many researchers also inves-



(a) Boxplot of CPU load. (b) Boxplot of TCP goodput.

Figure 9: Overhead of Late-binding

tigated the use of multiple interfaces (or multiple paths [11, 14, 20]) for bandwidth aggregation [15, 18, 23]. Other work like TCP Migrate [22] handoff a TCP connection to a new physical path without affecting the application. This work is orthogonal to these techniques and provides design guidelines for how these (and future) protocols can be implemented in the client network stack.

Many papers have explored buffer sizing in WAN and data-center networks [4, 17, 19]. Recent work on "bufferbloat" argues for reducing buffers (and therefore latency) in home APs and routers [1]. All these papers study the effects of large buffers in the network. Our work focuses on reducing buffers inside the client.

Our work also augments those who want to reduce web page load times, particularly when there are competing flows [2]. By keeping latency-sensitive packets separate all the way through the client stack, correct prioritization can be maintained.

6. CONCLUSION

Wireless networks are here to stay. To achieve the capacity needed, we are going to need more spatial reuse, and consequently provision the network for more handoffs. Hence, it is inevitable that over time our applications and mobile devices have to exploit these networks in a more fluid manner: choosing flexibly among the available networks, moving between them more often and more seamlessly, or even make use of multiple networks simultaneously. We believe it is time to update the client networking stack—that was originally designed with wired networks in mind—to support wireless connections that come and go, and are constantly changing. We demonstrated that current network stacks do not adequately prioritize latency-sensitive traffic, and inherently drop hundreds or even thousands of packets during a handoff or change of interface.

We introduced the principle of *late-binding* in which packets are mapped to an interface at the last possible moment, reducing the number of packets lost during a transition by three orders of magnitude. Latency-sensitive flows are also better served because they are kept separate until as close to the moment of transmission as possible. We believe late-binding is an important step towards updating the network stack for mobiles.

7. REFERENCES

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

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

[3] Part 11: Wireless lan mac and phy specifications. *IEEE Std P802.11-REVma/D8.0*, 2006.

[4] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. *SIGCOMM CCR.*, pages 281–292, 2004.

[5] B.D. Higgins, et. al. Intentional networking: opportunistic exploitation of mobile network diversity. In *Proc. ACM MobiCom '10*, Sep. 2010.

[6] M. Carbone and L. Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev.*, Apr. 2010.

[7] C. Carter, R. Kravets, and J. Tourrilhes. Contact networking: a localized mobility system. In *ACM MobiSys '03*, May 2003.

[8] S. Deb, K. Nagaraj, and V. Srinivasan. MOTA: engineering an operator agnostic mobile service. In *ACM MobiCom '11*, 2011.

[9] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. Proportional rate reduction for tcp. In *IMC*, 2011.

[10] Erik Nordstrom, et. al. Serval: An end-host stack for service-centric networking. In *NSDI*, April 2012.

[11] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational).

[12] J. Gettys. Beware, there are multiple buffers! `http://gettys.wordpress.com/2011/04/19/beware-there-are-multiple-buffers/`, April 2011.

[13] S. Hemminger. tcp_probe.c, 2004.

[14] H.-Y. Hsieh and R. Sivakumar. pTCP: an end-to-end transport layer protocol for striped connections. In *IEEE ICNP*, 2002.

[15] H.-Y. Hsieh and R. Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. *Wirel. Netw.*, 11(1-2):99–114, Jan. 2005.

[16] S. Kandula, K. C.-J. Lin, T. Badirkhanli, and D. Katabi. FatVAP: aggregating AP backhaul capacity to maximize throughput. In *NSDI*, Apr. 2008.

[17] M. Alizadeh, et. al. Data center tcp (dctcp). *SIGCOMM Comput. Commun. Rev.*, Aug. 2010.

[18] L. Magalhaes and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. In *ICNP*, 2001.

[19] N. Beheshti, et. al. Buffer sizing in all-optical packet switches. In *OThF8*. Optical Society of America, 2006.

[20] L. Ong and J. Yoakum. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286, May 2002.

[21] Ranveer Chandram et. al. MultiNet: Connecting to multiple IEEE 802.11 networks using a single wireless card. In *IEEE INFOCOM*, Mar. 2004.

[22] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MobiCom*, Sep. 2000.

[23] C.-L. Tsao and R. Sivakumar. On effectively exploiting multiple wireless interfaces in mobile hosts. In *CoNEXT '09*.

[24] K.-K. Yap. Tcp probe++. `https://bitbucket.org/yapkke/tcpprobe`.

[25] K.-K. Yap, T.-Y. Huang, and et. al. Making use of all the networks around us: a case study in android. *SIGCOMM Comput. Commun. Rev.*, Sept. 2012.