Diagnosing Wireless TCP Performance Problems: A Case Study

Tianbo Kuang Fang Xiao Carey Williamson
Department of Computer Science
University of Calgary
Calgary, AB, Canada T2N 1N4
email: {kuang, xiaof, carey}@cpsc.ucalgary.ca

January 30, 2003

Abstract

Researchers have long realized that TCP performance degrades over wireless links. Many solutions, including link-layer error recovery, have been proposed to overcome this problem. This paper studies how these solutions perform in a wireless LAN network environment. The experiments are conducted using off-the-shelf products, in a controlled way. In particular, we focus on Linux 2.4 TCP performance using a USB-based Compaq 802.11b multiport wireless card, as a case study.

A multi-layer view is adopted to analyze the complex interactions between layers of the network protocol stack. Our results show that the USB bus, the TCP implementation, and the wireless link-layer protocols can all affect the overall TCP performance. While the TCP and USB implementation problems can be corrected through bug-fixing and proper setting of the USB mode, the wireless-related problems, such as the data/ACK collision problem and the network thrashing problem, may require fundamental changes to link-layer protocols in wireless LANs.

Keywords: IEEE 802.11b Wireless LANs, TCP/IP, Network Traffic Measurement

1 Introduction

As wireless networks, especially wireless LANs, gain in popularity, more and more people are connecting their laptops, handhelds, and pocket PCs to the Internet via this tetherless technology. Even though wireless networks offer opportunities for new applications, legacy applications such as WWW and email are likely to remain dominant in the near future. A common feature of these applications is that they use the Transmission Control Protocol (TCP) as the transport-layer protocol for reliable end-to-end data delivery.

Researchers have long realized that TCP performance can degrade over wireless networks. One reason is that TCP interprets a packet loss as an indication of congestion in the network and throttles its transmission rate. However, most packet losses in wireless networks are from packet corruptions due to fading and interference in the wireless channel [2, 3]. Many solutions, including link-layer and transport-layer modifications, are proposed to overcome this problem [1]. However, their effectiveness has to be verified in a real wireless network environment.

In this paper, we study TCP performance in a wireless LAN environment. As a case study, we focus on diagnosing the TCP performance problems of the Compaq 802.11b multiport wireless card used on Compaq's new *Evo 719c* laptop. The network card has a standard IEEE 802.11b [4] air interface and a modified[1] Universal Serial Bus (USB) host interface [5, 6].

The USB issue provides a second motivation for our paper. USB is an industry standard[2] for connecting a computer to its peripherals. USB is widely used today, and several manufacturers offer wireless network cards with a USB interface. However, to the best of our knowledge, the impact of the USB bus (e.g., the framing structure, and the data transfer API) on the overall network performance has not been studied before. We explore these issues in this paper.

The objectives of our case study are:

- to identify the performance bottleneck(s) when several technologies (i.e., wireless and USB) are used in the network packet processing path, and
- to determine the impacts of low-layer mechanisms (e.g., link-layer retransmission [1], and MAC-layer rate adaptation [7]) on TCP performance

Since wireless TCP performance involves complicated interactions among layers of the network protocol stack, we use a multi-layer system view, similar to that used by Ludwig *et al.* [8]. Protocol behaviour at the TCP layer is captured by a utility called `tcpdump` [9]. Link-layer behaviour is captured by a wireless network "sniffer", a commercial `SnifferPro 4.6` Wireless Network Analyzer. *TCP sequence number plots* [10, 8] are used to visualize the results and understand the protocol behaviours.

The rest of the paper is organized as follows. Section 2 provides background information on IEEE 802.11b wireless LANs and on TCP, followed by a brief discussion of related research work. Section 3 describes the experimental methodology. Section 4 presents the experimental results, and identifies the causes of the performance problems observed. Section 5 concludes the paper.

2 Background and Related Work

2.1 IEEE 802.11b Wireless LANs

The IEEE 802.11b Wireless LAN (WLAN) standard [4] defines a high-speed extension (currently up to 11 Mbps) of the original 2 Mbps standard [11] in the 2.4 GHz band. The standard

[1]The host interface is a modified USB because the physical plugin is different from a standard one. However, the higher-layer protocols comply with the USB specification.

[2]More information about USB and the Linux USB implementation is provided in Appendix A.

specifies the physical layer and Medium Access Control (MAC) layer protocols used. Among other functions, such as association, authentication, and encryption, these protocols work together to mitigate the effects of wireless channel errors caused by loss, fading, and interference.

Frame transmissions on a WLAN are subject to *collisions*, if multiple senders try to transmit frames at the same time. A collision produces garbage on the channel, destroying all frames that were in transmission. To reduce this occurrence, the MAC layer regulates access to the shared (i.e., broadcast) channel in a wireless LAN. The typical MAC protocol used in 802.11b is Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA), also called Distributed Coordination Function (DCF). When a station wants to send a frame, it first senses the channel. If the channel is idle for a certain period of time (called the Distributed Inter-Frame Space (DIFS)), it transmits the frame. Otherwise, it waits until the channel becomes idle for another DIFS plus some random time. If the channel is still busy, the station doubles the random waiting period and repeats the process.

There are several error control mechanisms in IEEE 802.11b. For example, the 802.11b standard defines a MAC-layer retransmission mechanism. The standard requires that a receiver acknowledges each correctly received MAC frame. If no acknowledgment is received shortly after transmission, the sender resends the packet (at the MAC layer), repeating this process as necessary until an acknowledgment is received or until it reaches the maximum retransmission threshold (e.g., 4), at which point the sender gives up, leaving the problem to higher layer protocols (e.g., TCP). This MAC-layer retransmission can typically recover from wireless channel errors in a few milliseconds, long before TCP could detect and recover on an end-to-end basis. In addition, many 802.11b implementations employ a rate adaptation algorithm that can dynamically adjust the data rate used for transmissions, on a frame-by-frame basis. According to the current standard, four possible rates can be used: 1 Mbps, 2 Mbps, 5.5 Mbps, and 11 Mbps. Since the higher data rates are achieved through more sophisticated modulation schemes, the high data rate transmissions are the most vulnerable to channel errors. Dynamically choosing a lower transmission rate for the retransmitted frame may increase the chances that the frame transmission is successful on a noisy wireless channel.

The 802.11b WLAN can be operated in two different modes. In *infrastructure mode*, all stations communicate via an Access Point (AP) connected to a wired network. In *ad hoc* mode, there is no AP; stations communicate with each other directly in a peer-to-peer fashion. In our study, we use infrastructure mode, with a mobile laptop communicating with a wired host.

2.2 TCP Overview

TCP is a connection-oriented, end-to-end reliable-byte-stream transport-layer protocol [10, 12]. It is widely used on the Internet and in the Web.

In TCP, the fundamental mechanism to ensure reliable transfer is acknowledgment and retransmission. Every data byte that a TCP sender transmits has a logically associated (increasing) sequence number. When a TCP endpoint receives a packet containing data up to and including sequence number N, it sends back[3] an acknowledgment (ACK) indicating the next expected

[3]The receiver is not required to generate the ACK immediately. Rather, the receiver can delay the ACK (up

sequence number $N + 1$. This is called *cumulative acknowledgment*. Since either the packet or its ACK could be lost in the network, a sender starts a timer (which is a function of the estimated round-trip time (RTT)) when it transmits a packet. If the timer expires before an ACK is received, then the sender retransmits the outstanding packets. This is called a retransmission timeout (RTO). To prevent spurious timeouts, an implementation also sets some minimum value for the timer. For example, Linux uses the value 200 ms.

A retransmission timeout is costly, for several reasons. The primary reason is the TCP congestion control algorithm [13]. It uses adaptive window-based flow control to achieve congestion control. A sender can send at most a window's worth of packets without receiving ACKs. The flow control window size is adjusted dynamically based on two TCP state variables: the congestion window (*cwnd*), and the slow-start threshold (*ssthresh*). The initial value of *cwnd* is one segment, and *cwnd* is increased as successful ACKs are received. The increase is exponential in the slow-start phase (i.e., doubling *cwnd* every RTT, until *ssthresh* is reached), and linear in the congestion avoidance phase (i.e., increasing *cwnd* by one segment for every complete window's worth of data exchanged).

TCP uses packet loss (due to buffer overflow at a router) as an implicit signal of network congestion. When a retransmission timeout occurs, TCP updates its estimate of the slow-start threshold (e.g., $ssthresh = cwnd/2$), reduces its congestion window size to one segment, and re-enters the slow-start phase. This is why a timeout is costly. In order to avoid this, another commonly used strategy is *fast retransmit* [10], which uses *duplicate ACKs* (typically 3) to trigger the retransmission of a missing packet, typically well before the retransmission timer expires. This approach works well in recovering from single packet losses [14]. To deal with multiple packet losses in a window of data, several improved mechanisms are proposed in TCP NewReno [15], Selective Acknowledgment (SACK) TCP [16], and Duplicate SACK (DSACK) [17].

2.3 Related Work

There is a growing number of wireless measurement studies in the literature, ranging from wireless channel behaviour at the link layer to mobile user behaviour at the application layer. For example, Eckhardt *et al.* [2] and Nguyen *et al.* [18] characterized and modeled wireless channel errors. While interesting, this is not the focus of our work. Rathke *et al.* [19] and Xylomenos *et al.* [20] studied the TCP and UDP performance over WLANs. Their main findings are that host location as well as host and interface heterogeneity have a great impact on TCP performance. Our work is complementary to these studies. More recently, Balachandran *et al.* [21] report on network performance and user behaviour for Internet access by several hundred wireless LAN users during the 2001 ACM SIGCOMM conference in San Diego. However, little information is provided about the low-layer aspects of wireless LAN performance (a main focus in our paper).

Our work also differs from the prior work in that we focus on the overall system performance, and identify situations where the USB bus is the bottleneck. To the best of our knowledge, this is a new contribution.

to 200 milliseconds), in the hope that the ACK can be piggybacked on an outgoing data packet. However it is also recommended that an ACK be sent out for every two incoming data packets.

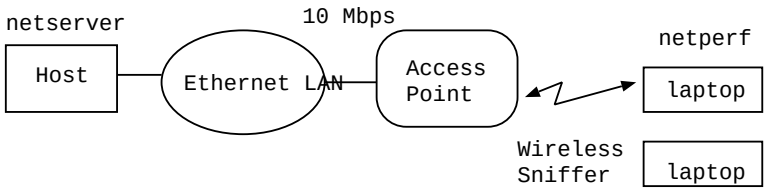


Figure 1: Network Topology

3 Experimental Methodology

3.1 Experimental Setup

Our experiments were conducted on an in-building IEEE 802.11b wireless LAN in the Department of Computer Science at the University of Calgary. A schematic illustration of this experimental setup is shown in Figure 1. The access point (AP) connecting the WLAN to the departmental 100 Mbps Ethernet LAN was a Lucent RG-1000 residential gateway, configured to operate in infrastructure mode. The stationary host, running SunOS 5.8, was connected to the LAN. The mobile laptop was a Compaq *Evo 7i19c* with a multiport wireless network adapter. The laptop was running Redhat Linux 7.2 (version 2.4.18). The USB controller has a *Universal Host Controller Interface* (see Appendix A). The controller was driven by the `usb-uhci.o` host controller driver (HCD) in the Linux kernel. The network card driver was `Linux-wlan 0.1.13` downloaded from `http://www.linux-wlan.com`.

The `netperf` and `netserver` utilities were used to generate TCP workload and to measure the overall streaming performance. `Netperf` and `netserver` were running on the laptop and the host, respectively, as shown in Figure 1.

3.2 Experimental Design

Our experiments focused on three factors that affect the overall system performance: the USB bus, the TCP implementation, and the wireless channel. Notice that since Linux 2.4 by default has TCP SACK option enabled, all our tests were conducted with TCP SACK unless explicitly stated otherwise.

The USB-related issues were studied by modifying the kernel and network card driver source code. Two categories of USB working modes were studied: *Queued bulk transfer mode* and *Queueless mode*. Within each category, there are four four different sub-modes to choose from: Breadth-First with Full Speed Bandwidth Reclamation (FSBR), Breadth-First without FSBR, Depth-First with FSBR, and Depth-First without FSBR (see Appendix A for a description of these modes). In total, there are 8 different modes. For each mode, we measured the TCP streaming throughput from 10 independent runs, and calculated the mean and standard deviation. To rule out the impact of wireless channel errors, we did these experiments when the wireless channel condition was very good.

Table 1: TCP Throughput under Linux and Windows 2000

	Throughput (Mbps)							
OS	1	2	3	4	5	6	7	8
Linux	1.52	1.52	1.53	1.54	1.51	1.54	1.53	1.51
Windows 2000	5.11	5.12	5.10	5.11	5.09	5.12	5.12	5.14

3.3 Measurement Methodology

`Tcpdump` and `SnifferPro 4.6` wireless network analyzer software were used to identify the TCP-related and wireless-related problems. The `tcpdump` trace provides a TCP-layer view of the system and `SnifferPro 4.6` provides a wireless channel view of the system. They are ideal to study the interactions between layers. When needed, the client kernel was also modified to provide further information. To generate wireless channel errors, we simply increased the distance between the laptop and the AP. In general, increasing the distance reduces the received signal strength at the receiver, making incoming frames more susceptible to channel errors.

Since the manufacturer provides a Windows 2000 driver for the network card, we measured the TCP streaming throughput under Windows 2000 before we started the Linux experiments. The results were intended to be used as a reference. A utility `SnifferUSB` [22] was used to find out the manufacturer's driver configurations and determine important settings.

4 Experimental Results

This section presents the main results from our measurement study. We begin with a discussion of USB-related performance problems in Section 4.1, followed by TCP issues in Section 4.2, and wireless issues in Section 4.3.

4.1 USB-Related Performance Issues

4.1.1 Low Throughput Problem

Before we started our experiments, we noticed that Linux TCP throughput on the WLAN is very low even when the wireless channel condition is very good. Table 1 shows the throughput results from 8 experiments when the sender is running Linux, and 8 comparable experiments when the sender is running Windows 2000. In both sets of experiments, the receiver is running SunOS. The experiments were done when the AP and the client were only 1-2 meters apart. There are few wireless errors in this condition. The server was on a local LAN so that the impacts of WAN congestion can be ruled out. We believe that these results represent the maximum achievable throughput for this hardware/software configuration, under ideal network conditions.

The results show that the TCP throughput for Linux is much lower than that for Windows 2000. Since the hardware is exactly the same, the difference must be from the implementation of the drivers and USB subsystem. Detailed analysis of the `SnifferPro` trace indicated that all

Table 2: TCP Throughput (Mbps) with Different USB Modes

	FSBR Disabled				FSBR Enabled			
	Breadth First		Depth First		Breadth First		Depth First	
	Qless	Queued	Qless	Queued	Qless	Queued	Qless	Queued
Mean	0.44	0.46	2.89	3.84	**5.16**	5.16	5.14	5.15
StdDev	0.00	0.00	0.00	0.02	**0.04**	0.03	0.05	0.06

802.11b frames sent by Linux were sent with a 2 Mbps transmission rate, while they were 11 Mbps for Windows 2000. This suggests a bug in the Linux driver for selecting the data rate.

By consulting the Prism chip set programming manual [23] and the driver source code, we found the cause of this problem. After modification, the problem was successfully[4] solved. The average TCP throughput after modification is 5.16 Mbps (with default USB setting), similar to that for Windows 2000. All remaining experiments in this paper use the modified Linux driver.

4.1.2 Performance Impact of USB Parameters

Our next experiment studies the impact of different USB modes on TCP transfer performance. We modified the kernel and network card driver source code to test all 8 combinations described in Section 3.2. These experiments used a 16 KB send socket buffer size, a 24 KB receive socket buffer size, and a 16 KB message size.

Table 2 shows the measured results. The mean and standard deviation are derived from repeating each experiment 10 times. The values in bold font show the results with the default Linux USB setting. The results in Table 2 show that the bottleneck shifts from the USB bus to the wireless network when FSBR is enabled. We elaborate on these results in the following.

A. Queueless Mode

The columns labeled *Qless* in Table 2 show the results when the network card driver sends one packet at a time to the USB. That is, only after the USB transmits the previous TCP packet successfully does the driver submit the next TCP packet.

When the USB mode is Breadth-First without FSBR, the throughput is only 0.44 Mbps. This is expected because in our case there is only one device on the USB, thus there are only two vertical linked lists: one for upstream bulk data and one for downstream bulk data (ignoring control packets, if any). The HC will process exactly one TD in each vertical linked list. It will then stop and wait for the start of the next frame. In other words, in each 1 ms frame the HC only sends or receives 64 bytes of data (the maximum size of a bulk transfer). A 1500-byte TCP packet requires $1500/64 = 24$ such transfers, and thus takes 24 ms to transmit. The maximum achievable throughput is 512 Kbps (64 data bytes per millisecond). The measured throughput of 440 Kbps is actually lower, indicating USB bus overhead of approximately 15%. The standard deviation for these results is zero, suggesting that the behaviour is deterministic. This is probably because the USB HC is a dedicated resource.

[4]By the time we figured out this bug, we noticed that a newer version (0.1.14) driver from Linux-wlan has also corrected the problem.

When the USB mode is Depth-First without FSBR, the HC processes all TDs in the first vertical linked list before processing the second one. It then stops and waits for the start of the next frame to repeat the process. Since a frame slot can hold at most 19 64-byte bulk data transfers (1216 bytes), a 1500-byte TCP packet requires at least 2 frames (2 ms). Furthermore, since an interrupt can only be generated at the end of a frame, the submission of the next USB request may miss the start of the next frame. This means the USB bus sits idle for a frame duration. In steady state, when two TCP packets are sent and one ACK is received (SunOS uses TCP Delayed-ACK), a typical pattern in the USB bus could be:

DATA DATA IDLE DATA DATA ACK IDLE

The second idle occurs because the next TCP packet triggered by the ACK misses the start of the frame. On average, two TCP packets are sent every 7 ms. The maximum throughput is thus 3.43 Mbps. Assuming 15% USB overhead, the average throughput is about 2.9 Mbps, as indicated in Table 2.

Clearly, when FSBR is disabled, the bottleneck is the USB bus. When FSBR is enabled, the idle slots on the USB bus can be avoided. This is because even if a packet misses the start of the frame, it can be transferred in the second pass when the HC loops back to the first vertical linked list. Assuming a 1500-byte packet can be transferred in two frames (2 ms), the maximum throughput is 6 Mbps. The bottleneck has thus shifted to the wireless network. This can also be confirmed by examining the standard deviations of the measured throughput. They are not zero any more because of the randomness of wireless network errors and CPU effects.

B. Queued Bulk Transfer Mode

Another set of experiments tested the performance of queued bulk transfer mode. This is interesting because queued USB requests can reduce the USB bus idle time, thus improving the performance. Since we do not know the on-board memory size of the wireless network card, we relied on the `SnifferUSB` [22] utility to find out the buffer[5] size. After figuring out the buffer size, we modified the network adapter driver so that it sends USB requests continuously until the buffer is full.

Table 2 shows the results for Queued mode. Two observations are evident:

- When FSBR is disabled, the bottleneck is still the USB bus. In Breadth-First mode, the throughput is the same as that for Queueless mode. This is because the queue does not change the fact that only 64 bytes of data can be transferred within a 1 ms frame. However, in Depth-First mode, the throughput increased, which means it reduces the idle time on the USB bus.

- When FSBR is enabled, the throughput is the same for Queued mode and Queueless mode. This means that the bottleneck now shifts to the wireless network. Using Queued mode only improves the USB throughput; it has no effect on the overall system performance. However, the Queued mode may be advantageous in a higher speed wireless network environment (e.g., 802.11a).

[5]The Windows 2000 driver was developed by the manufacturer so they should know the buffer size. A poorly chosen buffer size may cause the firmware to drop packets because of insufficient memory space.

4.2 TCP-Related Performance Problems

The second broad category of performance problems are related to TCP implementation issues. Figure 2 shows evidence of these TCP-related problems. Figure 2(a) shows a TCP sequence number plot derived from the sender-side `tcpdump` traces for a large data transfer. Figure 2(b) shows the receiver-side `tcpdump` trace, while Figure 2(c) shows the network-level view. From these graphs, three problems are evident:

1. There is a retransmission timeout in the middle of the transfer.
2. The TCP sender behaves strangely after the timeout since it appears to retransmit previously-acknowledged TCP data segments.
3. When three duplicate ACKs are received, the TCP sender does *not* do a fast retransmission.

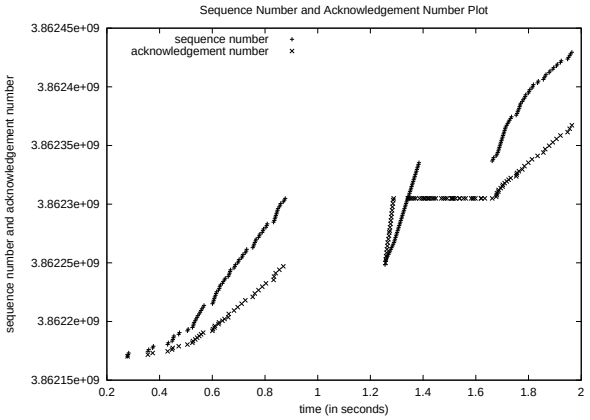
We have investigated these problems in detail and found that the wireless network adapter sometimes holds onto received TCP ACKs for a long time (e.g., hundreds of milliseconds) before passing them to the IP layer. This in turn triggers some Linux TCP implementation bugs. Problems 1 and 2 above are caused by these artifacts. Problem 3 is actually not a bug, but a deliberate feature of TCP. These findings are detailed in the following subsections.

4.2.1 The ACK Holding Problem

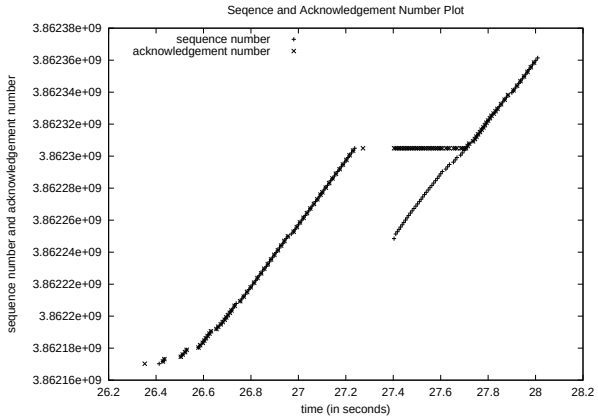
The retransmission timeout could be caused by loss of packets or ACKs in the network. However, from the receiver's sequence number plot (Figure 2(b)), it is clear that the packets were not lost. In order to see if the ACKs were lost by the wireless channel, we used `SnifferPro 4.6` to capture all frames in the wireless channel. Figure 2(c) confirms that they were not lost. Figure 2(c) also tells us important information: the TCP ACKs were sent in a timely fashion across the WLAN to the receiving network card. Comparing Figure 2(a) and Figure 2(c) indicates that the receiving network adapter has for some reason delayed the ACKs before handing them to the IP layer. This delay is large enough to cause a TCP retransmission timeout.

The question that arises is why the ACKs are held for so long. We added some print instructions to the Linux UHCI driver and redid the experiment. Figure 2(d) shows the results. The line marked "interrupt ack" shows the interrupts generated from the HC after it completes an incoming ACK reception. The time gap in the interrupts is the same as that in the TCP sequence numbers. During the gap, there are no interrupt activities. This confirms ACKs were held in the network adapter. The cause of this problem could be either a bug in the network adapter firmware, or an unexpected disabling of interrupts by the Linux operating system.

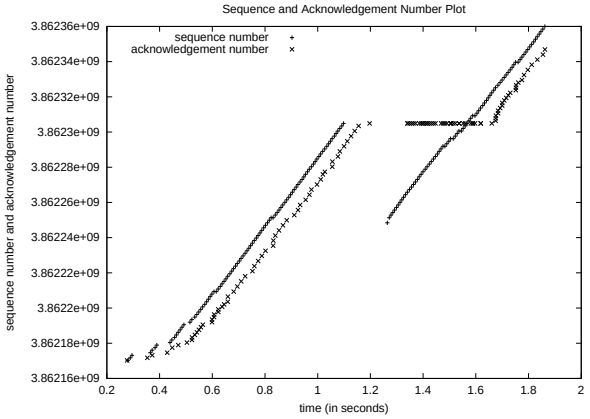
We have not yet been able to pinpoint this problem. One reason is that after recompiling the kernel (with the new instrumentation) the problem rarely occurs. In the old kernel, the problem occurred about 25% of the time. With the new kernel, the problem occurs less than 5% of the time. We hope the bug is rare enough so that it does not affect future experiments.



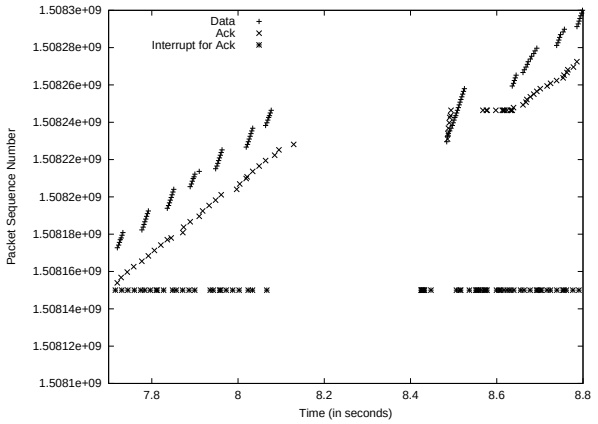
(a) Sender View



(b) Receiver View

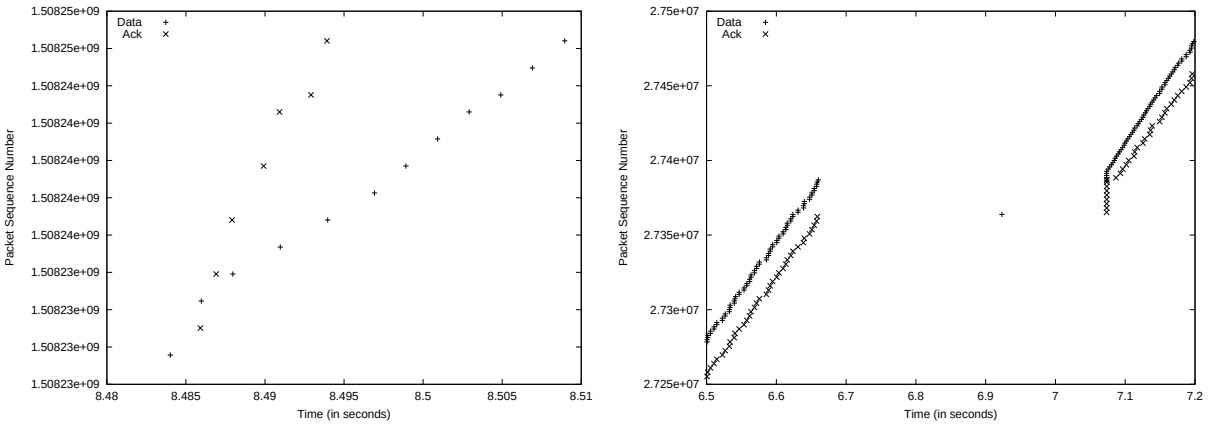


(c) Wireless Channel View



(d) USB HCD View

Figure 2: TCP Sequence Number Plots versus Time



(a) Anomalous Behaviour for the First Timeout

(b) No Slow Start for the Second Timeout

Figure 3: Illustration of the Retransmission of ACKed Data

4.2.2 Retransmission of ACKed Packets

The second TCP problem observed indicates a bug in the Linux TCP implementation. We diagnosed this problem by examining the kernel source code for the TCP module. The problem is in the Linux code that detects spurious timeouts and retransmissions. After a timeout and retransmission, Linux tries to determine if a received ACK is for the original transmission of the packet (i.e., the ACK was simply delayed by the network) or the retransmission of the packet (i.e., the ACK was lost). This is done by exploiting the TCP timestamp option [10] (Linux uses the timestamp optiion by default). If the *echoed timestamp* in the TCP ACK predates the sending of the retransmitted packet (recorded in the variable `retrans_stamp`), Linux recognizes a spurious timeout, and tries to restore the TCP state (e.g., the previous value of *cwnd*), rather than entering slow start. This implements the algorithm proposed by Ludwig and Katz [24].

In our particular experiment, the first ACK following the timeout should be considered delayed (i.e., a spurious timeout), because the ACK was delayed (but not lost) by the network card. However, Linux TCP behaves incorrectly, since it continues slow start after the timeout (see the zoomed-in TCP sequence number plot in Figure 3(a)). Worse yet, TCP seems to retransmit several data segments that were previously ACKed as received. This problem does not occur for the second (and subsequent) occurrences of ACK packets that are delayed (see Figure 3(b)).

Diagnosing this problem was difficult. We had to print out the TCP state information to debug the TCP module. In addition, since the ACK holding problem rarely occurs, we added some code to the link-layer kernel source to artificially delay incoming ACKs at random times. With these experiments we were able to solve the puzzle, as follows.

First, we noticed that there is a bug in the updating of `retrans_stamp`. Linux initializes the variable to the timestamp of the first SYN packet. However, in the TCP retransmission procedure, `retrans_stamp` is updated only if it is uninitialized (zero). This means that when Linux TCP enters the loss state for the first time, it will not be able to detect a spurious timeout

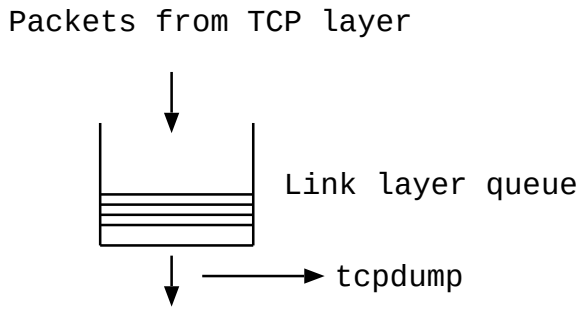


Figure 4: TCP-Layer View and tcpdump View

(i.e., the ACK timestamp will not predate the connection handshake). However, when Linux TCP exits the retransmission state, it resets `retrans_stamp` to zero, enabling proper updating of the timestamps for subsequent retransmission events. For example, Figure 3(b) shows the TCP behaviour after a second ACK holding event. It behaves exactly as required.

Second, the apparent retransmission of the ACKed packets is an artifact of `tcpdump`'s timestamp mechanism. This is illustrated in Figure 4, where the TCP-layer view and the `tcpdump` view are shown. These views differ because of the link-layer buffer: `tcpdump` sees a packet when it is actually sent to the network card, while the TCP layer may have sent the packet to the buffer a while ago. In Linux, the default link-layer buffer size is 100 packets. Since the USB bus can only send one packet at a time (when it is not in queued bulk transfer mode), whenever the TCP layer sends two or more packets in succession, `tcpdump` is likely to see a delayed version of some of the packets.

In our experiments, when the ACKs are delayed by the network card and the Linux TCP fails to detect a spurious timeout, the sender generates a burst of packets in slow start. This burst is enough to cause the link-layer queue to build up, causing inconsistency between the TCP-layer view and the `tcpdump` view. So the retransmission of ACKed packets is just an illusion: the TCP sender actually retransmitted the packets *before* their ACKs were received.

Note that although the problem studied here is triggerred by the "ack holding" problem, which is hardware-specific, spurious TCP timeouts could be common in other wireless environments because of handoffs, link-layer error recovery algorithms, or "blackouts" (due to fading). Proper handling of spurious timeouts is important to improve TCP performance [24].

4.2.3 No Fast Retransmission after Three Duplicate ACKs

The third TCP anomaly observed is the absence of a fast retransmission after three duplicate ACKs. At first glance, this seems to be a bug. However, more careful analysis shows that it is the correct behaviour in this case because no congestion is indicated (no packet drops).

For TCP NewReno, this behaviour is a deliberate design to prevent a false fast retransmission after a retransmission timeout [15]. When three duplicate ACKs are received after a retransmission timeout, it could mean that the sender has retransmitted some packets that have been received by the receiver, or it could mean a new packet drop. Since NewReno has no way of distinguishing these two scenarios, it does not do a fast retransmit. The careful design waits for

an ACK acknowledging data *beyond* the sequence number in use when slow start was entered. This is exactly what happens in our trace.

In TCP SACK [16], the sender should be able to distinguish between the two scenarios if duplicate-SACK (D-SACK) [17] is used. A Linux receiver by default uses D-SACK. However, a SunOS 5.8 receiver does not use D-SACK (only SACK). Further source code checking found that Linux TCP maintains a variable $fackets\_out$ to remember the number of out of order packets. If this number exceeds a threshold, fast retransmission is triggered. This variable is only updated when an ACK containing a SACK block is received (See `tcp_input.c::tcp_ack()`, which calls `tcp_sacktag_write_queue()` to update $fackets\_out$). However in our case there was no ACK containing SACK blocks, therefore the variable was never updated. Hence, fast retransmission is not triggered even if more than three duplicate ACKs are received.

4.3 Wireless-Related Performance Problems

The final category of performance problems relates to wireless network characteristics. In this section, we study these wireless-related problems.

4.3.1 The Data and ACK Packet Collision Problem

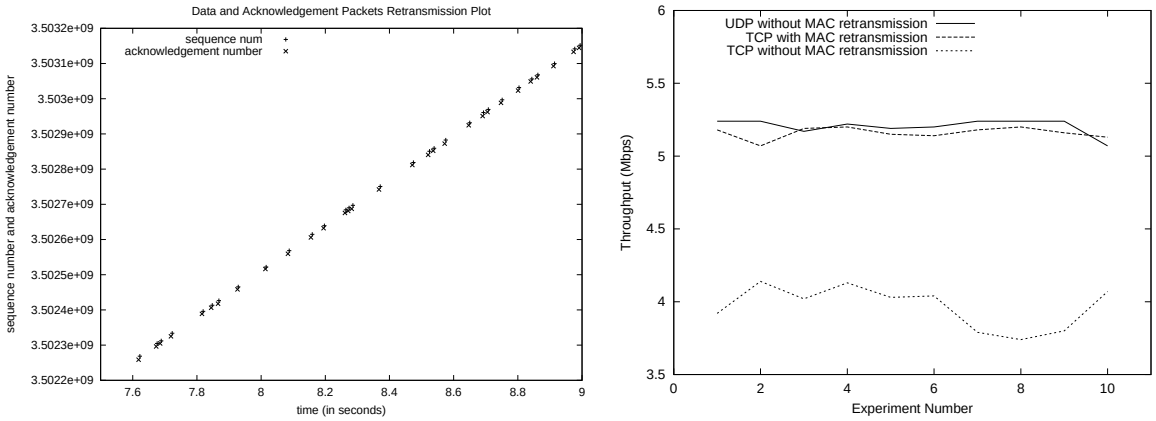
The collision avoidance mechanism of IEEE 802.11b does not prevent all collisions. Since TCP traffic is bidirectional (data packets in one direction, and ACK packets in the opposite direction), there can be TCP data packet and ACK packet collisions. These collisions cause MAC-layer retransmissions, or TCP-layer retransmissions when link-layer error recovery is not used.

Figure 5(a) shows the sequence number plot of the MAC-layer retransmissions of TCP data packets and TCP ACK packets. The retransmissions show a pattern where a retransmission of a data packet is always paired with a retransmission of an ACK packet. In this experiment, the client and the AP are in the same office and are only 1-2 meters apart. These retransmissions are unlikely to be caused by the wireless channel errors. Instead, they suggest a collision problem.

To confirm this hypothesis, we tested the retransmission rate for UDP, which does not have transport-layer acknowledgments. MAC-layer retransmissions for UDP thus reflect wireless channel errors only. The retransmission rate for UDP is about 0.47% to 0.98%. However, the retransmission rate[6] for TCP is about 4.58% to 4.73% We attribute the increase in the retransmission rate to TCP data/ACK packet collisions.

While the impact on performance may not be significant, the effect is more pronounced when MAC-layer recovery is not used. To understand this impact, we modified the network adapter driver to disable MAC-layer retransmission, and tested the streaming throughput separately for TCP and for UDP. Figure 5(b) shows the results. The TCP throughput without MAC-layer retransmission is 23% lower than that with MAC-layer retransmission. The UDP streaming performance, even without MAC-layer retransmission, is slighter higher than that of TCP with MAC-layer retransmission.

[6]We tested network cards and APs from other manufacturers as well. The retransmission rates range from 1.75% to 7.2% depending on different card and AP combinations.



(a) Illustration of Collision

(b) Impact on Performance

Figure 5: MAC Layer Collision and Its Impact on Performance

4.3.2 The MAC Layer Rate Adaptation Problem

The final problem relates to the MAC-layer rate adaptation algorithm. MAC-layer rate adaptation is supposed to increase the throughput when the channel error rate is high. However, we show in this section that a poor rate adaptation algorithm can decrease throughput.

In this experiment, we try three different MAC-layer rate adaptation algorithms:

- *Automatic*: the firmware automatically chooses the transmission rate dynamically (from 1 Mbs, 2 Mbps, 5.5 Mbps, and 11 Mbps) using its internal algorithm;
- *Static 1 Mbps*: the data rate is always set to be 1 Mbps; and
- *Static 2 Mbps*: the data rate is always set to be 2 Mbps.

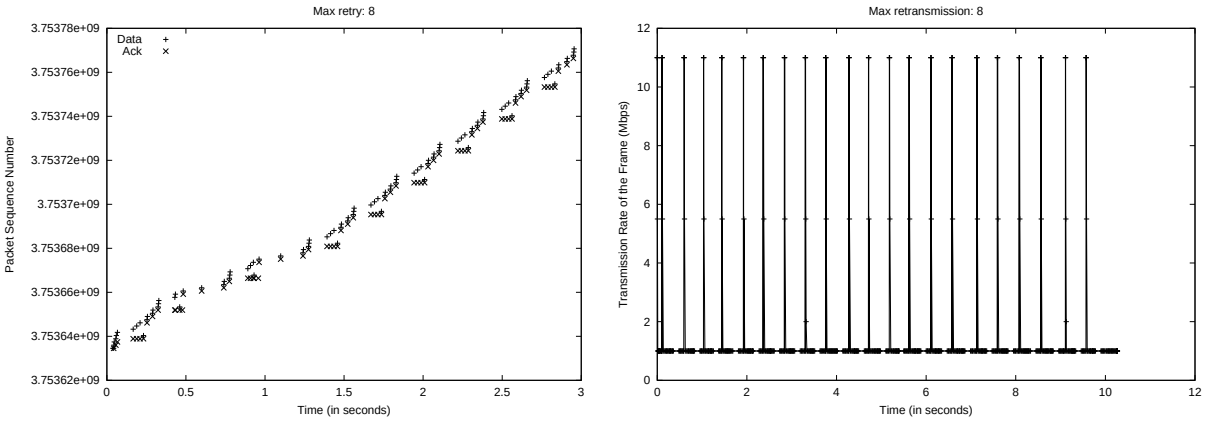
The average TCP throughput[7] over 20 runs for each of these algorithms is 0.29 Mbps, 0.77 Mbps, and 1.46 Mbps, respectively. The results clearly show that the lowest throughput is achieved with the dynamic MAC-layer rate adaptation mechanism. The explanation follows.

A. Poor Rate Adaptation

We plot in Figure 6(a) the TCP sequence number plot when rate adaptation is enabled, and the maximum retransmission count is 8. In this graph, there is a periodic loss of packets causing TCP to do repeated fast retransmits. Detailed analysis of the `SnifferPro` trace shows that the TCP packet retransmissions are for packets that reached the maximum MAC-layer retransmission limit. The curious observation is the periodicity in the pattern: about every tenth data packet requires retransmission at the TCP layer.

The explanation is found in the MAC-layer rate adaptation behaviour. Figure 6(b) shows the data transmission rate used in the transmitted frame headers; it is clearly periodic. The network

[7]These results used MAC-layer retransmission limit of 10. Other settings produce similar results.



(a) Sequence-time plot

(b) Wireless Channel View

Figure 6: TCP Performance Anomalies from Poor MAC-layer Rate Adaptation Algorithm

adapter firmware apparently sets the transmission rate to 11 Mbps after several successfully transmitted frames. However, the poor wireless channel condition corrupts many of these frames. As a result, the first transmission at 11 Mbps is unsuccessful. The sender then retransmits the frame twice more at 11 Mbps, failing each time. The rate adaptation algorithm then uses 5.5 Mbps to transmit the frame. After several more failed attempts, it finally uses 1 Mbps. At 1 Mbps, it occasionally succeeds with the transmission, but more often than not it reaches the maximum retransmission limit, and aborts the frame. The Multiplicative Increase/Multiplicative Decrease (MIMD) rate adaptation algorithm causes the periodic TCP packet retransmissions in our experiment.

This crude "bandwidth probing" mechanism causes "network thrashing" on the WLAN. The wireless channel resource is wasted on many MAC-layer retransmissions, and many (inefficient) TCP-layer retransmissions occur. Another adverse effect of this algorithm is that it unnecessarily consumes battery-power for the mobile devices.

The overall effect can be seen in Figure 6(a). Notice the (approximately) 100 ms gaps before each fast retransmission, during which the sender is retransmitting (in vain) at the MAC-layer. A static transmission rate of 1 Mbps provides higher throughput, without thrashing.

Turning off rate adaptation is not a good choice, since wireless users are mobile, and wireless network conditions are dynamic. Rather, a better rate adaptation algorithm is required. A bandwidth probing algorithm considering past transmission rate history may be an improvement. For example, the receiver-based algorithm proposed by Holland *et al.* [7] may be helpful. We will investigate this aspect in our future work.

B. A Temporary Deadlock Problem

Another subtle problem shown in Figure 6(b) is the temporary deadlock due to the interaction between TCP's Delayed-ACK algorithm at the receiver and the small congestion window size at the sender. Near 0.5 seconds, twice the sender sends only one packet, while the receiver needs two packets to trigger an immediate ACK. This creates a deadlock, which is resolved by a

Delayed-ACK timeout (about 100 ms in the graph). The channel is idle during that time. The small congestion window size in this example is caused by the periodic TCP fast retransmission.

Our future work will study the impact of Delayed-ACK on wireless network performance. When the wireless channel quality is good, Delayed-ACKs are beneficial, since they reduce data/ACK packet collisions. When the wireless channel quality is poor, TCP Delayed-ACKs reduce throughput because of the temporary deadlock problem.

5 Conclusions

In this paper, we tested the TCP performance of Compaq's USB-based 802.11b multiport wireless card in a WLAN, using Linux 2.4. We identified three factors that affect the overall system performance: the USB bus, the TCP implementation, and the wireless link-layer protocols.

Measurement results show that poor settings of the USB mode (i.e., disabling FSBR) can make the USB bus the bottleneck, limiting TCP throughput to 0.4 Mbps to 3.0 Mbps. When FSBR is enabled (the default USB mode in Linux), the wireless network is the bottleneck.

Among TCP-related problems, we observed an "ACK holding" problem at the wireless network card that occasionally results in spurious TCP timeouts. Whether it is caused by a firmware bug or interrupt-disabling is not clear. This problem in turn triggers some bugs and artifacts in the Linux TCP implementation, causing strange behaviour when viewed by `tcpdump`. Our detailed analyses explain each of these anomalies.

Finally, on the wireless front, two problems were identified: the TCP data/ACK collision problem, and the network thrashing problem. For TCP streaming, the MAC-layer retransmission rate is 1.3% to 2.3% for the multiport wireless network card, and 1.75% to 7.2% for other network cards and APs tested. The majority of these retransmissions are caused by collisions of TCP data/ACK packets. The network thrashing problem observed is caused by a poor MAC-layer rate adaptation algorithm. Careless probing of the bandwidth causes excessive MAC-layer retransmission, wasting WLAN bandwidth and battery power for mobile devices. More careful design of wireless LAN protocols is required to avoid these inefficiencies.

References

[1] G. Xylomenos, G. Polyzos, P. Mahonen, and M. Saaranen. TCP performance issues over wireless links. *IEEE Communications Magazine*, vol. 39, no. 4:pp. 52–58, 2001.

[2] D. Eckhardt and P. Steenkiste. Measurement analysis of the error characteristics of an in-building wireless network. Proceedings of ACM SIGCOMM'96, August 1996.

[3] R. Theodore. *Wireless Communications: Principles and Practice*. Prentice Hall PTR, 2002.

[4] ANSI/IEEE Standard 802.11b. Part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications: Higher-speed physical layer extension in the 2.4 ghz band. 1999.

[5] Compaq, Intel, Microsoft, and NEC. Universal serial bus specification, revision 1.1. September 23 1998.

[6] On line document. USB in a nutShell. http://www.beyondlogic.org/usbnutshell/usb1.htm.

[7] G. Holland, N. Vaidya, and P. Bahl. 'A rate-adaptive MAC protocol for multi-hop wireless networks. Proceedings of ACM/IEEE MOBICOM, July 2001.

[8] R. Ludwig, B. Rathonyi, A. Konrad, K. Oden, and A. Joseph. Multi-layer tracing of TCP over a reliable wireless link. Proceedings of ACM SIGMETRICS'99, June 1999.

[9] Tcpdump. http://www.tcpdump.org.

[10] R. Stevens. *TCP/IP Illustrated, Volume 1: the Protocols*. Addison-Wesley, 1994.

[11] ANSI/IEEE Standard 802.11. Part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. 1999.

[12] A. Tanenbaum. *Computer Networks*. Addison-Wesley, third edition, 1996.

[13] V. Jacobson. Congestion avoidance and control. Proceedings of ACM SIGCOMM'88, August 1988.

[14] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *ACM Computer Communication Review*, 26, July 1996.

[15] RFC2582. RFC2582: The NewReno Modification to TCP's fast recovery algorithm. April 1999.

[16] RFC2018. RFC2018: TCP selective acknowledgment options. October 1996.

[17] RFC2883. RFC2883: An extension to the selective acknowledgment (SACK) option for tcp. July 2000.

[18] G. Nguyen, R. Katz, B. Noble, and M. Satyanarayanan. A trace-based approach for modeling wireless channel behavior. Proceedings of the Winter Simulation Conference, December 1996.

[19] R. Rathke, M. Schlager, and A. Wolisz. Systematic measurement of TCP performance over wireless LANs. *Technical Report TKN-01BR98*, 1998.

[20] G. Xylomenos and G. Polyzos. TCP and UDP performance over a wireless LAN. Proceedings INFOCOM'99, March 1999.

[21] A. Balachandran, G. Voelker, P. Bahl, and P. Rangan. Characterizing user behavior and network performance in a public wireless LAN. Proceedings of ACM SIGMETRICS, June 2002.

[22] SnifferUSB. USB sniffer for Windows 2000 and Windows 98. http://benoit.papillault.free.fr/usbsnoop/index.en.php3.

[23] Intersil. PRISM driver programmer's manual, version 2.0. May 2001.

[24] R. Ludwig and R. Katz. The Eifel algorithm: Making TCP robust against spurious retransmissions. *ACM Computer Communications Review*, Vol. 30, No. 1, January 2000.

[25] Intel Corp. Universal host controller interface (UHCI) design guide, revision 1.1. March 1996.

[26] D. Fliegl. Programming guide for Linux USB device drivers. http://usb.cs.tum.edu, December 2000.

A Universal Serial Bus (USB) Overview

This appendix gives a short introduction to USB and its Linux implementation. More information can be found elsewhere [5, 25, 26]. See [6] for an excellent on-line tutorial.

A.1 General Information

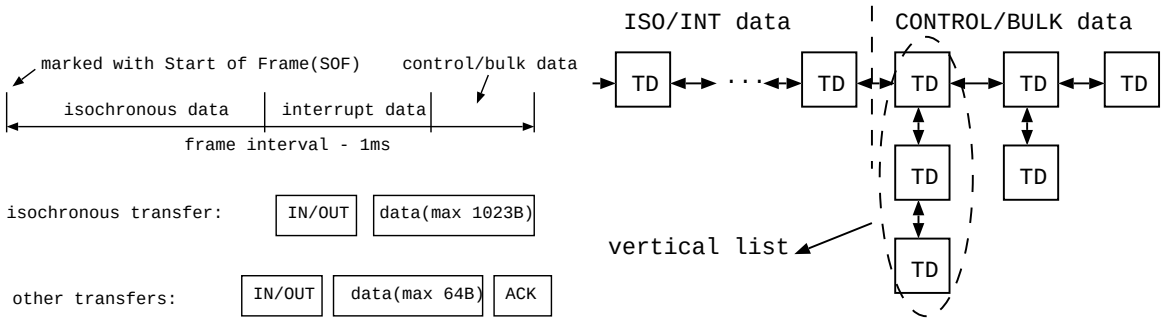
USB is an industry standard for connecting a computer to its peripheral devices, using a bus topology. Currently there are two versions, namely 1.1 and 2.0. Version 1.1 supports two data signalling rates: *low speed* at 1.5 Mbps, and *full speed* at 12 Mbps. Version 2.0 supports a third signalling rate, namely *high speed* at 480 Mbps.

All USB transfers are managed by a *Host Controller* (HC). If the transfer is from the HC to a device, it is called a *downstream* transfer; otherwise, it is called an *upstream* transfer. When an HC wants to send data to a device (usually requested by the device driver), it sends an OUT token to the device, followed by the data. After correctly receiving the data, the device returns an acknowledgment, if required. In an upstream transfer, the HC polls the device by sending it an OUT token. The device then transfers its buffered data to the HC.

The transfer time on a USB bus is divided into time slots called *frames*, each 1 ms in duration (high speed mode uses a 125 us frame interval). Within a frame, the HC classifies its data transfers into one of four types: *Isochronous*, *Interrupt*, *Control*, and *Bulk* data transfers.

The USB framing structure is illustrated in Figure 7(a). The HC first processes Isochronous data, followed by Interrupt data, followed by Control data and Bulk data if there is time left in the current frame. The standard limits Isochronous/Interrupt transfers to at most 90% of the capacity within a frame. Isochronous transfers are not reliable; therefore no acknowledgment is required from the receiver. The other three transfer types are reliable transfers, requiring an acknowledgment from the receiver.

The bus frequency and frame timing structure limit the maximum number of successful transfers for each type. For example, the maximum data length for an Isochronous transfer is 1023 bytes. The maximum achievable data rate for a full speed bus is thus 8.184 Mbps (1023



(a) Frame Structure (b) TD Linked List

Figure 7: USB Frame Structures and TD Linked List

bytes per frame). The maximum data size for each Interrupt/Bulk transfer is 64 bytes. A full speed bus can fit 19 64-byte transfers per frame (assuming no other traffic), for a maximum data rate of 9.728 Mbps. In reality, these rates are rarely achieved because of implementation issues.

Usually the HC hardware implements all of the functions described above. A Host Controller Interface (it is actually an API) allows the *Host Controller Driver* (HCD) to instruct the HC to do the desired work. More specifically, the HCD inserts Transfer Descriptors (TD) into a linked list (see Figure 7(b)), with each TD telling the HC what kind of transfer it is and how much data to transfer. The HC then automatically fetches the TDs from the linked list and conducts the required transfer. In other words, each TD corresponds to a transfer. After a transfer is completed, the hardware generates an interrupt to notify the HCD, which in turn may do a callback to the device driver to indicate the results. Normally, TDs representing transfers to the same device are linked in a vertical list. These vertical lists are then horizontally linked together to form a larger list of lists. This arrangement is shown in Figure 7(b).

The HC has two options to process the TDs: *Breadth-First* or *Depth-First*. In Breadth-First, the HC processes the first TD in the first vertical list, then the first TD in the second vertical list, and so on until it processes the first TD in the last vertical list. In Depth-First, the HC does not move to the next vertical list until it finishes all TDs in the current vertical list. Note that if the 1 ms frame time is depleted and the HC has not finished traversing the list, then it has to wait until the next frame, resuming from where it left off in the traversal. On the other hand, when the HC has processed the last list in the sequence and still has time remaining in the current frame, the HC has to wait for the start of the next frame before starting the next traversal from the first list. When this happens, part of a frame is left empty (idle). To reduce this waste, the USB specification defines *Full Speed Bandwidth Reclamation* (FSBR) mode, in which the last vertical list is circularly linked back to the first one. This mode allows the HC to continue traversing and processing TDs as long as sufficient time remains in the current frame.

The foregoing descriptions assume that the device driver submits only one packet to the USB HCD at a time. The USB specification also has a *Queued* bulk transfer mode, in which bulk data can be queued at the HC. In this way, the USB bus is filled with data to transfer, reducing the idle waiting time due to the frame structure.

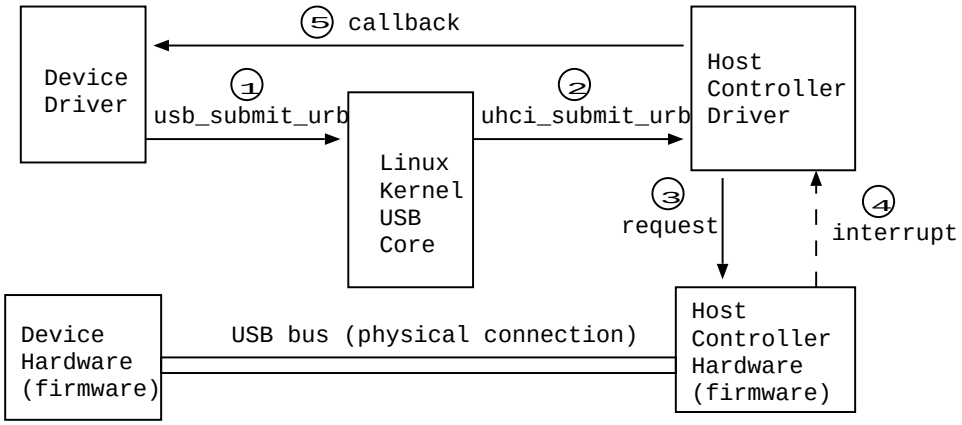


Figure 8: Linux USB Implementation

A.2 Linux 2.4 USB Implementation

The Linux 2.4 USB subsystem implements a USB core to handle USB specific issues. Two types of HCDs are implemented: Universal Host Controller Interface (UHCI), and Open Host Controller Interface (OHCI). The USB core acts as the glue between device drivers and the HCD.

Figure 8 illustrates the Linux USB operation. Data transfer requests (e.g., a 1500-byte TCP packet), are initiated by the device driver by passing to the USB core a data structure called a USB Request Block (URB), which has a pointer to the socket buffer containing the packet. The USB core then forwards this URB to the appropriate HCD, which in turn transforms this request into TDs and inserts the TDs into the linked list for HC processing. Note that since each USB transfer has a maximum length, TCP packets larger than the maximum transfer length require multiple TDs. After the requested URB transfer is completed (i.e., all TDs in a URB have been transferred), the HC generates an interrupt to inform the HCD. The latter then does a callback to inform the device driver. The device driver analyzes the URB returned. If the URB was an output request, it clears the busy bit of the device, allowing another transmission request to be sent. If the URB was an input request, it invokes the IP layer to process the received packet, and issues another URB to poll the device. In queued bulk transfer mode, successive URBs can be sent without waiting for a callback.