The Effects of Asymmetry on TCP Performance[1]

Hari Balakrishnan[2], Venkata N. Padmanabhan[3], and Randy H. Katz

{hari,padmanab,randy}@cs.berkeley.edu

Computer Science Division, Department of EECS

University of California at Berkeley, Berkeley, CA 94720-1776.

Abstract

In this paper, we study the effects of network asymmetry on end-to-end TCP performance and suggest techniques to improve it. The networks investigated in this study include a wireless cable modem network and a packet radio network, both of which can form an important part of a *mobile ad hoc network*. In recent literature (e.g., [18]), asymmetry has been considered in terms of a mismatch in bandwidths in the two directions of a data transfer. We generalize this notion of *bandwidth asymmetry* to other aspects of asymmetry, such as *latency* and *media-access,* and *packet error rate*, which are common in wide-area wireless networks.

Using a combination of experiments on real networks and simulation, we analyze TCP performance in such networks where the throughput achieved is not solely a function of the link and traffic characteristics in the direction of data transfer (the *forward* direction), but depends significantly on the *reverse* direction as well. We focus on bandwidth and latency asymmetries, and propose and evaluate several techniques to improve end-to-end performance. These include techniques to decrease the rate of acknowledgments on the constrained reverse channel (*ack congestion control* and *ack filtering*), techniques to reduce source burstiness when acknowledgments are infrequent (*TCP sender adaptation),* and algorithms at the reverse bottleneck router to schedule data and acks differently from FIFO (*acks-first scheduling*).

Keywords: TCP, asymmetry, cable modem, wireless networks

1. Introduction

The Transmission Control Protocol (TCP) is widely used in the Internet for reliable, unicast communications. The robustness of TCP in a wide variety of networking environments is the primary reason for its widespread usage. However, emerging networking technologies pose new challenges to TCP in terms of performance, which motivate the development of new TCP algorithms. In this paper, we focus on the challenges to end-to-end TCP performance that arise due to network asymmetry, including in the context of wide-area wireless networks. The increased interest in asymmetric networks is motivated by technological and economic considerations as well as by the popularity of applications such as Web access, which involve a substantially larger data flow towards the client (the *forward* direction) than from it (the *reverse* direction).

Examples of networks that exhibit asymmetry include *cable modem* networks (both wireline and wireless), *direct broadcast satellite* networks, and *Asymmetric Digital Subscriber Loop* (ADSL) networks, where bandwidth in the forward direction is often orders of magnitude larger than that in the reverse. Such asymmetry is accentuated when the channel is unidirectional, necessitating the use of a different, often low-bandwidth channel (e.g., a dialup line or a bandwidth-constrained and lossy wireless channel) for communication in the reverse direction.

Our study is not limited to networks where the asymmetry is explicit because of mismatched bandwidths — we also study TCP dynamics in *packet radio networks*, where traffic flowing simultaneously in different directions can adversely affect performance. This is because most packet radio networks use half-duplex radio units, which cannot transmit and receive data frames at the same time. In addition, we combine two wireless technologies — wireless cable and packet radio — in our study to understand the problems that arise in these situations when different types of asymmetry are tied together.

Asymmetry is inherent in several wireless networks, where it is often the case that a central transmitter (or base station) can transmit at high power to receiving portable/mobile units. However, to reduce power consumption, these units transmit to the base station at relatively low power. In addition, they often have to contend with other mobile units to gain access to the channel.

Such asymmetric networks are often used in *mobile ad hoc environments*. As an example, consider a detachment of soldiers in the battlefield who communicate with each other via a packet radio network (which exhibits latency/media-access asymmetry) and whose link to the outside world is via an asymmetric-bandwidth satellite or wireless cable modem network. A TCP connection between an one of the mobile nodes (e.g., a soldier's computer) and a host in the wired Internet (e.g., a database server at the command center) will have to contend with both forms of asymmetry. Thus, it is important to consider the aspect of asymmetry while analyzing TCP performance in mobile ad hoc networks.

We generalize the various phenomena and examples described above to the following definition of asymmetry: *a network is said to exhibit network asymmetry with respect to TCP performance, if the throughput achieved is not solely a function of the link and traffic characteristics of the forward direction, but depends significantly on those of the reverse direction as well*. In addition to the *bandwidth asymmetry* described above, this definition extends to other types of asymmetry, such as *latency* and *media-access,* and *packet error rate*. In this paper, we study bandwidth, and latency and media-access asymmetries, both individually and in combination. We use measurements on a real testbed as well as simulations experiments with different choices of topology and workload, to identify the performance problems. Based on these results, we propose and evaluate several techniques to improve performance. The wireless networks that serve as the basis for our work include a wireless cable modem network and a packet radio network.

Fundamentally, network asymmetry affects the performance of reliable transport protocols such as TCP because these protocols rely on feedback in the form of cumulative acknowledgments from the receiver to ensure reliability. In addition, TCP is *ack-clocked*,

1. A more detailed treatment of the material in this paper appears in the theses [2] and [21].

2. Presently at MIT's Lab for Computer Science and EECS Dept.

3. Presently at Microsoft Research.

relying on the *timely* arrival of acknowledgments, to make steady progress and fully utilize the available bandwidth of the path [12]. Thus, any disruption in the feedback process could potentially impair the performance of the forward data transfer. For example, a low bandwidth acknowledgment path could significantly slow down the growth of the TCP sender window during slow start, *independent* of the link bandwidth in the direction of data transfer. A second example is from packet radio networks, where variable latencies in the presence of bidirectional traffic (caused, for instance, by acknowledgements flowing in a direction opposite to data packets) causes the sender's round-trip time estimate to be highly variable. This inflates TCP's retransmission timeout value, thereby impairing loss recovery.

The following are our major results and conclusions:

- SLIP header compression [13] alleviates some of the performance problems due to bandwidth asymmetry, but does not completely eliminate all problems, especially those that arise in the presence of bidirectional traffic.
- Connections traversing packet radio networks suffer from large variations in round-trip time caused by the half-duplex nature of the radios and asymmetries in the media-access protocol. This adversely affects TCP's loss recovery mechanism and results in degraded performance.
- The various end-to-end and router-based techniques that we propose help improve performance significantly in many asymmetric situations, both in simulations and in experiments on the real testbed. These include decreasing the frequency of acknowledgments (acks) on the constrained reverse channel (*ack congestion control* and *ack filtering*), reducing source burstiness when acknowledgments are infrequent (*TCP sender adaptation),* and scheduling data and acks intelligently at the reverse bottleneck router.
- In addition to improving throughput for individual connections, our proposed modifications also help improve the fairness and scaling properties when several connections contend for scarce resources in the network. We demonstrate this via simulations of bulk and Web-like transfers.

The rest of the paper is organized as follows. Section 2 describes some related work and Section 3 discusses the details of our experimental testbed and methodology. We then analyze the problems that arise due to bandwidth asymmetry (Section 4) and latency/media-access asymmetry (Section 5) using measurements and packet traces from our network testbed. In Section 6, we present several end-to-end and router-based techniques to alleviate the problems due to asymmetry. An evaluation of these techniques via simulation appears in Section 7. We implemented the techniques that seemed promising based on the simulation experiments, in our testbed. The implementation is described in Section 8, and a performance evaluation of the implementation, that confirms the trends observed in simulation, is presented in Section 9. We present our conclusions in Section 10 and plans for future work in Section 11.

2. Related Work

Several researchers have identified and proposed solutions to transport protocol problems that arise in single-hop wireless networks [1, 3, 4, 6]. The main issue considered in these papers is the impact of packet losses due to reasons other than congestion (wireless error, handoff, etc.) on TCP performance. We view our work as being in the natural progression of such research, with the overall goal of understanding and improving the performance of reliable transport protocols like TCP in the face of ever-increasing heterogeneity in network technologies and characteristics. The specific measurements reported in this paper were taken over a wireless cable modem network and a packet radio network.

There has been some previous work on understanding the effects of two-way traffic on TCP performance. In [24], the authors demonstrate how two-way traffic can lead to *ack compression*, where closely-bunched acknowledgments disrupt the smooth ack-clocked transmission at the sender. More recently, there has been interest in how asymmetric-bandwidth networks exacerbate this problem. In [18], the authors model a network with bandwidth asymmetry and derive analytical expressions for throughput in terms of packet loss probability and the *normalized asymmetry ratio* under certain ideal assumptions. They also propose the use of a *drop-from-front* strategy for dropping acknowledgments at the bandwidth-constrained reverse link. In [15], the authors demonstrate how bidirectional traffic over asymmetric links leads to ack compression, and consequently, degraded performance. They investigate a backpressure mechanism to limit data flow in the reverse direction, but conclude that this alone is not enough for good performance.

There have also been studies of bandwidth asymmetry in the context of satellite networks [9, 22]. The main distinction between such a network and the wireless cable modem network we consider in this study is that the former has a much larger bandwidth-delay product, which could be the dominating factor in performance. Finally, some basic performance measurements of Metricom's Ricochet packet radio network are presented in [7].

While the results and analysis in [15] and [18] are very useful, the set of problems is far from being understood or solved. In addition to proposing and evaluating other schemes to alleviate the adverse effects of bandwidth asymmetry, we also characterize other types of asymmetry that occur in wide-area wireless networks. We evaluate our solutions for these networks in terms of connection throughput, fairness, and scaling behavior.

3. Experimental and Simulation Methodology

In this section, we describe our experimental testbed, simulation setup, and traffic workloads used in the study.

3.1 Experimental Testbed

We use a combination of simulation and actual experimentation on a heterogeneous network testbed to evaluate the performance of TCP, understand the reasons for observed performance, and design end-host and router-based techniques to improve performance. Our simulation topologies and parameters are derived from the following networks in our testbed:

- *Wireless cable modem network:* This is a wireless cable modem network using technology developed by Hybrid Networks, Inc. (*www.hybrid.com*). The aggregate bandwidth of the (unidirectional) forward channel is 10 Mbps[4] and the one-way link latency is about 5ms. The topology for this testbed is shown in Figure 1. The downstream channel for data operates in the 2.4 GHz range and is down-converted at the receiving end to standard television channel frequencies. The reverse channel could be a dialup line, an ISDN line, a wireless channel using a wide-area packet radio network, etc.
- *Packet radio network:* Our packet radio network is based on Metricom Inc.'s Ricochet network (*www.metricom.com*). The

4. A 30 Mbps 2-way wireless cable system is currently under development; this system also exhibits bandwidth asymmetry.

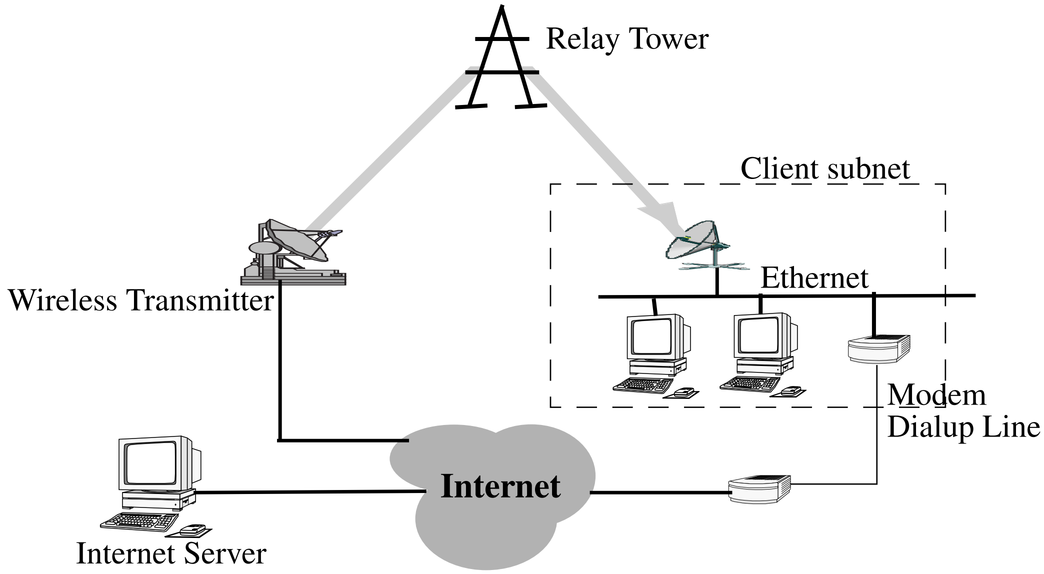

Figure 1. The network topology of the wireless cable modem network which illustrates bandwidth asymmetry. Hosts on the client subnet receive data from the Internet via the 10 Mbps wireless cable link (the *forward* channel) and send out data via a low bandwidth (e.g., dialup) link (the reverse channel).

topology for this network is shown in Figure 5. The packet radios operate in the 915 MHz ISM band and have a raw link speed of 100 Kbps. The poletop units typically have a range of several hundred meters.

The wireless cable modem testbed is an example of a network with bandwidth asymmetry depending on the return path used, which could be a dialup phone line (e.g., 14.4 Kbps or 28.8 Kbps), or a wireless channel. In a good installation of the wireless cable modem network, the bit-error rate of the forward channel is negligible. We therefore do not focus on the effects of bit-errors on the forward channel in this paper.

The packet radio network we study is an example of a network that does not have explicit (bandwidth) asymmetry, but has the characteristic that the flow of traffic (e.g., TCP data) in one direction is affected by the flow of traffic (e.g., TCP acks) in the opposite direction. This arises because of the half-duplex nature of the radio units that cannot simultaneously send and receive packets, having to incur a significant overhead in turning around from one mode to the other.

After studying the impact of each type of asymmetry independently, we combine networks with these different asymmetric characteristics and study the performance characteristics of TCP connections through them. We now describe the details of our simulation setup, workloads, and performance metrics.

3.2 Simulation Setup and Performance Metrics

We used `ns` [20], an event-driven packet-level network simulator from Berkeley and LBNL for our work. We developed several extensions to this simulator to model the networks of interest to us. We added the notion of a shared link (LAN) to the simulator with the ability to incorporate arbitrary link-layer and media-access protocols. Our simulations of the packet radio network use a MAC protocol loosely based on Ricochet's protocol. The details of these additions are described in Section 5.1.

Our simulation parameters (such as link bandwidth, latency, and packet-radio turn-around time) and topologies are derived from the characteristics of the networks in our experimental testbed. We validated the simulated performance of standard TCP and constant-rate UDP traffic with actual measurements in the real networks. We experiment with two kinds of workloads — large bulk transfers and short Web-like transfers. We also consider simultaneous transfers in opposite directions.

Our main performance metrics are *throughput* measured at the receiver and a metric for fairness, called the *fairness index* [8]. The fairness index, f, is defined as follows: if there are n concurrent connections in the network and the throughput achieved by connection i is equal to x_i, $1 <= i <= n$, then

$$f = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \sum_{i=1}^{n} x_i^2}$$

The fairness index always lies between 0 and 1 (since throughput is non-negative), and as explained in [14], is equal to (k/n) if k of the n connections receive equal throughput and the remaining none. Thus, f cannot be less than $1/n$ in a network with n connections. We use the fairness index to understand and analyze the scaling properties of the network when multiple connections are simultaneously active.

4. Bandwidth Asymmetry

In this section, we discuss the network topology and performance problems that arise due to bandwidth asymmetry. These include the slowdown and increased burstiness of a TCP sender due to the disruption of ack clocking, and highly variable performance when there are simultaneous TCP transfers in both the forward and the reverse directions.

4.1 Network Topology

The network topology of the wireless cable system is shown in Figure 1. The bandwidth of the forward channel is 10 Mbps. The

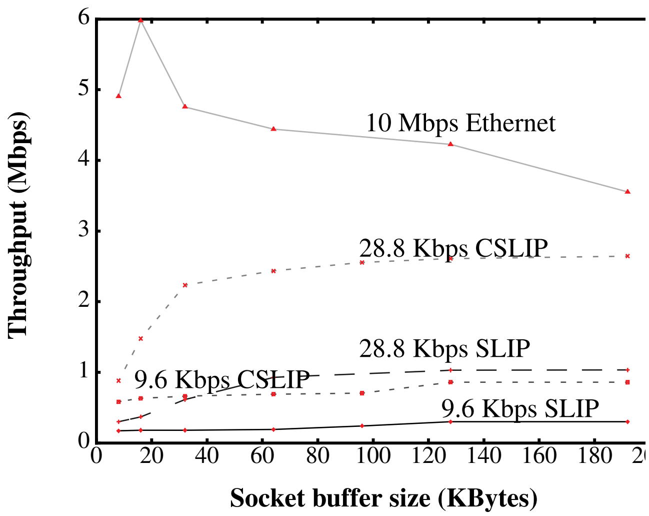

Figure 2. Measured performance of the Hybrid wireless cable network using different return channels, across a range of socket buffer sizes. Each run of the experiment involved the transfer of 1 MB of data in the forward direction between two BSD/OS hosts.

reverse channel is much slower, for instance a 28.8 Kbps dialup phone line. In our experiments, we also considered an Ethernet reverse channel. While such a configuration is possibly unrealistic, it serves as a useful data point for comparison. In the following sub-sections, we discuss several performance problems that we observed based on experiments conducted in the real testbed as well as in the simulator.

4.2 Analysis of performance problems

We now discuss the problems that arise due the limited bandwidth of the reverse channel in an asymmetric-bandwidth network.

4.2.1 One-way Transfers

We first discuss the case where TCP transfers happen only in the forward direction. A common example of this is a user downloading data from a server. For simplicity, we initially restrict ourselves to the case of a single TCP transfer in the forward direction.

We define the *normalized bandwidth ratio*, k, (as defined in [12]) between the forward and reverse paths as the ratio of the raw bandwidths divided by the ratio of the packet sizes used in the two directions. For example, for a 10 Mbps forward channel and a 100 Kbps reverse channel, the raw bandwidth ratio is 100. With 1000-byte data packets and 40-byte acks, the ratio of the packet sizes is 25. So, k is 100/25 = 4. This implies that if there is more than one ack for every $k = 4$ data packets, the reverse bottleneck link will get saturated before the forward bottleneck link does, possibly limiting the throughput that can be achieved in the forward direction.

The main effect of bandwidth asymmetry in this case is that TCP ack clocking can break down. Consider two data packets transmitted by the sender in quick succession. While in transit to the receiver, these packets get spaced apart according to the bottleneck link bandwidth in the forward direction. The principle of ack clocking is that the acks generated in response to these packets preserve this spacing (in time) all the way back to the sender, enabling the sender to clock out new data packets with the same spacing.

However, the limited reverse bandwidth and consequent queuing effects could alter the inter-ack spacing. When acks arrive at the bottleneck link in the reverse direction at a faster rate than the link can support (which happens when $k > 1$ assuming every data packet is acknowledged), they get queued behind one another. The spacing between them when they emerge from the link is dilated with respect to their original spacing. (This is in contrast to ack compression which happens when acks get queued at a fast link, i.e. $k < 1$.) Thus the sender clocks out new data at a slower rate than if there had been no queuing of acks. Furthermore, the sender's window growth is slowed down.

This is part of the reason why the measured throughput for a dialup reverse channel without SLIP header compression (Figure 2) is so low. SLIP header compression (CSLIP) reduces the sizes of acks and decreases k, improving performance. For example, consider the case of a 10 Mbps forward and 28.8 Kbps reverse channel, with a data packet size of 1 KB. With the TCP timestamp option enabled, the ack size is 52 bytes with SLIP and 18 bytes with CSLIP. So k is 18.05 with SLIP and is 6.25 with CSLIP. With TCP delayed acks (one ack for every two data packets), throughput is limited to 10*2/18.05 = 1.1 Mbps and 10*2/6.25 = 3.2 Mbps, with SLIP and CSLIP respectively. These numbers closely match the measured throughputs shown in Figure 2.

In comparison, the performance with an Ethernet return channel is much better because of the absence of bandwidth asymmetry (k is 0.052) and a much smaller link delay than the dialup lines. As an aside, the throughput shows a dip beyond a socket buffer size of 16 KB because larger socket buffer sizes lead to overflow of some router queue along the forward path.

In practice, the reverse bottleneck link will also have a finite amount of buffer space. If the TCP transfer lasts long enough, this buffer can fill up and cause acks to get dropped. If the receiver acknowledges every packet, on average $(k-1)$ out of every k acks get dropped at the reverse channel buffer. Since in effect only one ack traverses the reverse bottleneck link for every k data packets, acks may not directly limit forward throughput. However, this situation leads to several other problems because the sender now receives fewer acks than it would have otherwise.

First, the sender could become bursty. If the sender receives only one ack in k, it ends up sending out data in bursts of k packets. This increases the chance of data packet loss, especially when k is large. Second, since conventional TCP senders base their window increase on *counting* the number of acks and not on how much actual data is acknowledged, fewer acks imply a slower rate of growth of the congestion window. Third, the receipt of fewer acks could disrupt the sender's fast retransmission algorithm when there is a data packet loss. The sender may not receive the threshold number of duplicate acks although the receiver may have sent out more than the required number. And finally, the loss of the (now infrequent) acks further down the path to the sender could cause long idle periods while the sender waits for subsequent acks to arrive.

4.2.2 Two-way Transfers

We now consider the case when TCP transfers simultaneously occur in the forward and reverse directions. An example of this is a user sending out data (for example, an e-mail message) while simultaneously receiving other data (for example, Web pages). We restrict our discussion to the case of one connection in each direction.

In this scenario, the effects discussed in Section 4.2.1 are more pronounced, because some of the reverse direction bandwidth is used by the reverse transfer. This increases the degree of bandwidth asymmetry for the forward transfer.

In addition, there are other effects that arise due to the interaction between data packets of the reverse transfer and acks of the forward transfer. Suppose the reverse connection is initiated first and that it has saturated the reverse channel and buffer with its data packets at the time the forward connection is initiated. There is then a high probability that many acks of the newly initiated forward connection will encounter a full reverse channel buffer and hence get dropped. Even after these initial problems, acks of the forward connection could often get queued up behind large data packets of the reverse connection, which could have long transmission times (e.g., it takes about 280 ms to transmit a 1 KB data packet over a 28.8 Kbps line). This causes the forward transfer to stall for long periods of time.

Figure 3 and Figure 4 show concurrent reverse and forward connections, measured in the wireless cable modem network. The reverse connection is initiated first. As discussed above, the forward connection starts off very slowly. Figure 4 clearly shows large idle times until about 160 seconds into the transfer. It is only at times when the reverse connection loses packets (due to a buffer overflow at an intermediate router) and slows down that the forward connection gets the opportunity to make rapid progress and quickly build up its window. This is evident from the sharp upswings in the forward connection's data rate just before the times at which the reverse connection retransmits packets, marked by circles in Figure 3.

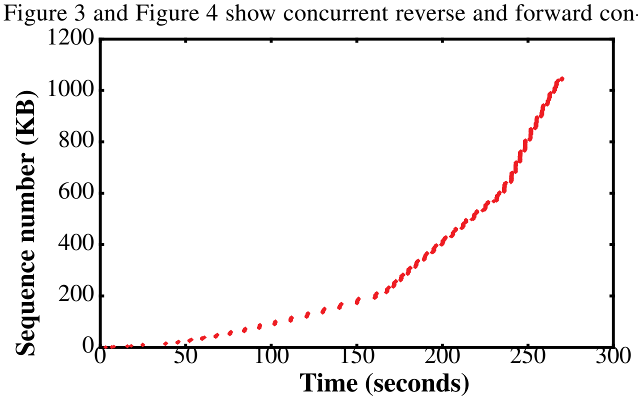

Figure 4. Measurements of the forward connection operating over the 10 Mbps Hybrid wireless cable network. The sharp upswings in its data rate occur whenever the reverse connection suffers a loss and slows down.

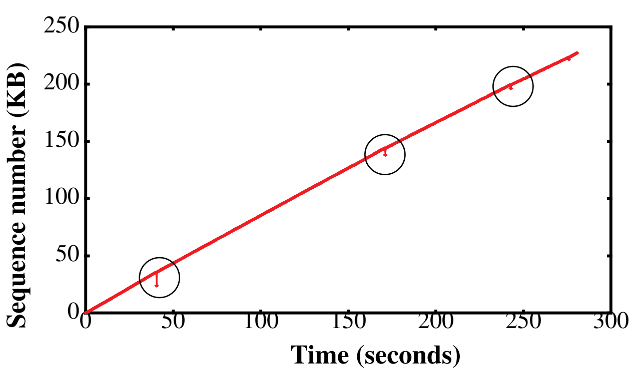

Figure 3. Measurements of the reverse connection operating over a 9.6 Kbps dialup line. The circles indicate times when the reverse connection retransmits packets.

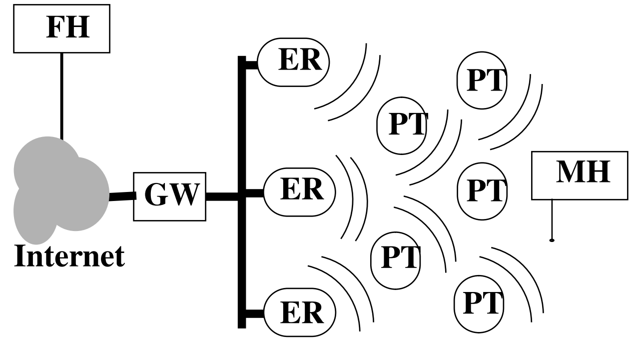

Figure 5. Topology of the Ricochet packet radio network. The Mobile Host (MH) has a modem attached to it, which communicates with a Fixed Host (FH) on the Internet through the Poletop Radios (PT) and Ethernet Radios (ER). The Gateway (GW) routes packets between the packet radio network and the Internet.

5. Latency Asymmetry

In this section, we discuss the effects of latency and media-access asymmetry on TCP performance. As before, we use a combination of measurements and simulations to obtain our results. We focus on TCP connections through a packet radio network as an example of a situation with this type of asymmetry. We start by describing the topology of the network and the media-access and link-layer protocols. We then discuss the results of our experiments and solutions to observed problems. Finally, we discuss some scaling and fairness issues in this network.

5.1 Network Topology and Underlying Protocols

Topology: The topology of the packet radio network is shown in Figure 5. The maximum link speed between two nodes in the wireless cloud is 100 Kbps. Packets from a fixed host (FH) on the Internet are routed via the Metricom Gateway (GW) and through the poletop radios (PT), to the end mobile host (MH). The number of wireless hops is typically between 1 and 3.

Radio Turnarounds: The radio units in the network are *half-duplex*, which means that they cannot simultaneously transmit and receive data. Moving from transmitting to receiving mode takes a non-trivial amount of time, called the transmit-to-receive turnaround time, T_{TR}. Similarly, going from receiving to transmitting mode takes a time equal to the receive-to-transmit turnaround time, T_{RT}.

MAC Protocol: The radios are frequency-hopping, spread-spectrum units operating in the 915 MHz ISM band. The details of the frequency-hopping protocol are not relevant to this paper, since the predominant reason for variability is the MAC protocol. The MAC protocol is based on a polling scheme, similar to (but not identical to) the RTS/CTS ("Request-To-Send/Clear-To-Send") protocol used in the IEEE 802.11 standard. A station wishing to communicate with another (called the peer) first sends it an RTS message. If the peer is not currently communicating with any other station, it sends a CTS message acknowledging the RTS. When this is received by the initiator, the data communication link is established. A data frame can then be sent to the peer. If the peer cannot currently communicate with the sender because it is communicating with another peer, it does not send a CTS, which causes the sender to backoff for a random amount of time and schedule the

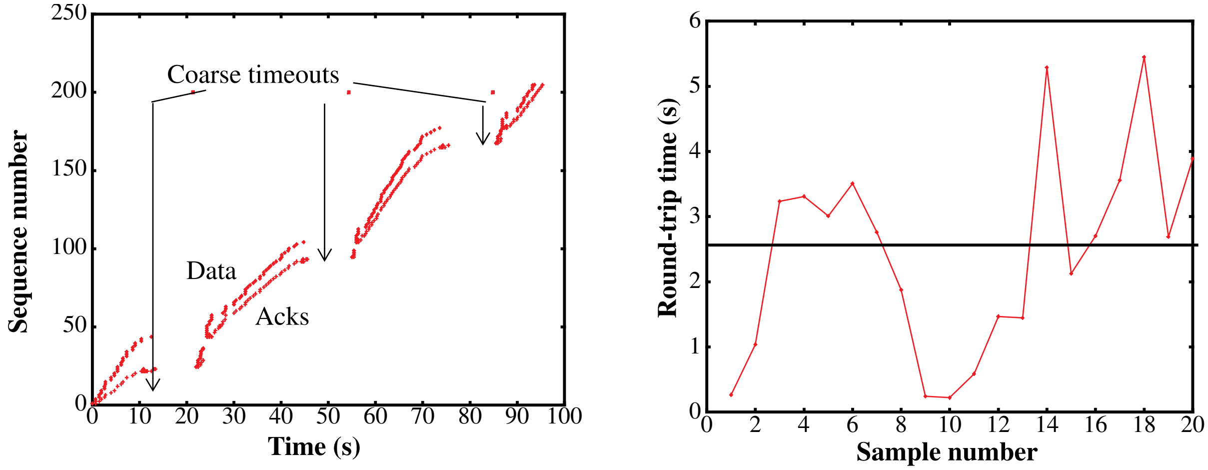

Figure 6. (a) Packet and ack sequence trace of a 200 KB TCP bulk transfer measured over one wireless hop in the Ricochet network. The three pauses are sender timeouts, lasting between 9 and 12 seconds each because large round-trip time variations cause the retransmission timeout estimate to be very long. (b) Twenty round-trip time samples collected during this connection are shown. The samples have a mean of about 2.5 s and a standard deviation of about 1.5 s.

transmission for later. It could also send a NACK-CTS to the sender, which achieves the same effect. In all this, care is taken by both stations to ensure that messages and data frames are not lost because the peer was in the wrong mode, by waiting enough time for the peer to change modes. To do this, each station maintains the value of the turnaround times of its neighbors in the network.

Link-Layer Protocol: The reliable link-layer protocol used in this network is a simple frame-by-frame protocol with a window size of 1. When a frame is successfully received, the receiver sends a link-level ACK to the sender. If the frame is not received successfully, the sender retransmits after a timeout. Such simple link-layer protocols are the norm in several packet radio networks (see, e.g., [16]).

5.2 Analysis of problems

We now discuss the results of several measurements and simulations under various network topologies and traffic workloads. We start with the simplest case of a bulk TCP transfer across one wireless hop running the MAC and link-layer protocols described above. Although these particular measurements were made using the Phase 1 Ricochet modems, we have observed similar effects in measurements made with the newer Phase 2 modems as well.

Variable Delays: The need for the communicating peers to first synchronize via the RTS/CTS protocol and the significant turnaround time for the radios result in a high per-packet overhead. Further, since the RTS/CTS exchange needs to back off when the polled radio is otherwise busy (for example, engaged in a conversation with a different peer), the overhead is variable. This is the main reason for large and variable latency in packet-radio networks. It is also clear why an increase in “interfering” traffic (like TCP acks) can significantly impact the flow of TCP data packets.

Figure 6(a) shows the packet sequence trace of a measured 200 KB TCP transfer across one wired and one wireless hop in the Ricochet network. This clearly shows the effect of the radio turnarounds and increased variability affecting performance. The connection is idle for 35% its duration, as a result of only three coarse timeouts (six other losses are recovered by TCP’s fast retransmission mechanism). Ideally, the round-trip time of a data transfer will be relatively constant (i.e., have a low deviation). Unfortunately, this is not true for connections in this network, as shown in Figure 6(b). This figure plots the individual round-trip time estimate samples during a TCP connection over the actual Ricochet network. The mean value of these samples is about 2.5 seconds and the standard deviation is about 1.5 seconds. Because of the high variation in the individual samples, the retransmission timer, set to *srtt + 4*mdev,* is on the order of 10 seconds, causing long idle periods. In general, it is correct for the retransmission timer to trigger a segment retransmission only after an amount of time dependent on both the round-trip time and the linear (or standard) deviation, since this avoids spurious retransmissions. Thus, techniques are needed to alleviate the problems caused by large deviations in TCP round-trip times to the loss recovery process. These problems are exacerbated in the presence of two-way traffic as well as other competing traffic.

6. Solutions

Most of the problems discussed in the preceding sections arise because of contention for the bottleneck resources in the reverse direction — link bandwidth and buffer space, or because of the interaction between traffic (data and acks) flowing in opposite directions in the network. This observation serves as the starting point for the solutions discussed below.

We first present two techniques — ack congestion control and ack filtering — for reducing the frequency of ack packets on the reverse channel. We then discuss changes at the TCP sender to enable it to adapt well to the situation where acks are received infrequently. Finally, we present a simple scheduling algorithm for data and ack packets at the reverse channel router to improve performance when there are two-way transfers.

6.1 Ack Congestion Control (ACC)

The idea here is to extend congestion control to TCP acks, since they do make non-negligible demands on resources at the low-bandwidth bottleneck link in the reverse direction. Acks occupy slots in the reverse channel buffer, whose capacity is often limited to a certain number of *packets* (rather than bytes), as is the case in our BSD/OS systems.

Our approach is to use the RED (*Random Early Detection*) algorithm [11] at the gateway of the reverse link to aid congestion control. The gateway detects incipient congestion by tracking the average queue size over a time window in the recent past. If the average exceeds a threshold, the gateway selects a packet at random and marks it, i.e. sets an *Explicit Congestion Notification* (ECN) bit using the RED algorithm[5]. This notification is reflected to the sender of the packet by the receiver. Upon receiving a packet with ECN set, the sender reduces its sending rate.

The important point to note is that with ACC, *both* data packets and TCP acks are candidates for being marked. The TCP receiver maintains a dynamically varying delayed-ack factor, *d*, and sends one ack for every *d* data packets. When it receives a packet with the ECN bit set, it increases *d* multiplicatively, thereby decreasing the frequency of acks also multiplicatively. Then for each subsequent round-trip time (determined using the TCP timestamp option) during which it does not receive an ECN, it linearly decreases the factor *d*, thereby increasing the frequency of acks. Thus, the receiver mimics the standard congestion control behavior of TCP senders in the manner in which it sends acks.

There are bounds on the delayed-ack factor *d*. Obviously, the minimum value of *d* is 1, since at most one ack is sent per data packet. The maximum value of *d* is determined by the sender's window size, which is conveyed to the receiver in a new TCP option. The receiver should send at least one ack (preferably more) for each window of data from the sender. Otherwise, it could cause the sender to stall until the receiver's delayed-ack timer (usually set at 200 ms) kicks in and forces an ack to be sent.

When the reverse RED gateway gets full, it needs to drop a packet. The gateway can choose from a variety of schemes to pick a packet to drop — in particular, it can drop from the tail (ACC-D), or it can drop a packet that is already enqueued at random (ACC-R). We experimented with both policies in our experiments and found that the choice of drop policy makes a difference to performance in some cases.

6.2 Ack Filtering (AF)

The ACC mechanism described above modifies the TCP stack at the receiver in order to decrease the frequency of acks on the constrained reverse link. Ack filtering, based on an idea suggested by Karn [17], is a gateway-based technique that decreases the number of TCP acks sent over the constrained channel by taking advantage of the fact that TCP acks are cumulative.

When an ack from the receiver is about to be enqueued, the router (or the end-host's routing layer, if the host is directly connected to the constrained link) traverses its queue to check if any previous acks belonging to the same connection are already in the queue. It then removes some fraction (possibly all) of them, depending on how full the queue is. The removal of these "redundant" acks frees up space for other data and ack packets. The policy that the filter uses to drop packets is configurable and can either be deterministic or random (similar to a random-drop gateway, but taking the semantics of the items in the queue into consideration). There is no need for any per-connection state to be maintained at the router — all the information necessary to implement the drop policy is already implicitly present in the packets in the queue.

In the experiments reported in this paper, AF deterministically clears out all preceding acks belonging to a connection whenever a new ack for the same connection with a larger cumulative ack value enters the queue.

5. The gateway can also be configured to drop the selected packet (Random Early Drop), but we chose to mark it instead.

6.3 TCP Sender Adaptation (SA)

ACC and AF alleviate the problem of congestion on the reverse bottleneck link by decreasing the frequency of acks, with each ack potentially acknowledging several data packets. As discussed in Section 4.2.1, this can cause problems such as sender burstiness, a slowdown in window growth, and a decrease in the effectiveness of the fast retransmission algorithm.

We combat sender burstiness by placing an upper bound on the number of packets the sender can transmit back-to-back, even if the window allows the transmission of more data. If necessary, more bursts of data are scheduled for later points in time computed based on the connection's data rate. The data rate is estimated as the ratio *cwnd/srtt*, where *cwnd* is the TCP congestion window size and *srtt* is the smoothed RTT estimate. Thus, large bursts of data get broken up into smaller bursts spread out over time.

The sender can avoid a slowdown in window growth by simply taking into account the amount of data acknowledged by each ack, rather than the number of acks. So, if an ack acknowledges *s* segments, the window is grown as if *s* separate acks had been received. This policy works because the window growth is only tied to the available bandwidth in the *forward* direction, so the number of acks is irrelevant.

Finally, we solve the fast retransmission problem by not requiring the sender to count the number of duplicate acks. Instead, with ACC when the receiver observers a threshold number of out-of-order packets, it marks all of the subsequent duplicate acks with a bit to indicate that a fast retransmission is requested. With AF, the reverse channel router takes similar action when it has filtered out a threshold number of duplicate acks. The receipt of even one such marked packet causes the sender to do a fast retransmission.

6.4 Ack Reconstruction (AR)

Ack reconstruction is a technique to reconstruct the ack stream after it has traversed the reverse direction bottleneck link. AR is a local technique designed to prevent the reduced ack frequency from adversely affecting the performance of standard TCP sender implementations (i.e., those that do not implement sender adaptation). This enables us to use schemes such as ACC and AF without having to modify sources and perform sender adaptation, and is something asymmetric network providers may be able to locally deploy.

The main idea here is for the reconstructor to fill in the gaps in the ack sequence to "smooth out" the ack stream seen by the sender. Suppose two acks, $a1$ and $a2$ arrive at the reconstructor after traversing the constrained reverse link at times $t1$ and $t2$ respectively. Let $a2 - a1 = \Delta a > 1$. When $a2$ reaches the sender, at least Δa packets are sent by the sender (if the flow control window is large enough), and the congestion window increases by at most 1, independent of Δa. AR remedies this situation by interspersing acks to provide the sender with a larger number of acks at a consistent rate, which reduces the degree of burstiness and causes the congestion window to increase faster.

One of the configurable parameters of the reconstructor is δa, the ack threshold, which determines the spacing between interspersed acks. Typically, we set δa to 2, which follows TCP's standard delayed-ack policy. Thus, if successive acks arrive at the reconstructor separated by Δa, we interpose $ceil(\Delta a/\delta a) - 2$ acks, where *ceil()* is the ceiling operator. The other parameter needed by the reconstructor is δt, which determines the *temporal* spacing between the reconstructed acks. In order to do this, we measure the rate at which acks arrive at the *input* to the reconstructor — this rate depends on the output rate from the constrained reverse channel and on the presence of other traffic on that link. We use an exponentially weighted moving average estimator to monitor this rate; the output of the estimator is Δt, the average rate at which acks are arriving at the reconstructor (and the average rate at which acks would reach the sender if there were no further losses or delays). If we set δt equal to Δt, then we would essentially operate at a rate governed by the reverse bottleneck link, and the resulting performance would be determined by the rate at which *unfiltered* acks arrive out of the reverse bottleneck link. If sender adaptation were being done, then the sender behaves as if the rate at which acks arrive us $\Delta a/\Delta t$. Therefore, a good method of deciding the temporal spacing of reconstructed acks, δt, is to equate the rates at which *increments* in the ack sequence happen in the two cases. That is, the reconstructor sets δt such that $\Delta a/\Delta t = \delta a/\delta t$, which implies that $\delta t = (\delta a/\Delta a)*\Delta t$. The later ack, *a2*, is held back for a time roughly equal to Δt.

Thus, by carefully controlling the number and spacing between acks, unmodified senders can be made to increase their congestion window at the right rate and also avoid bursty behavior. AR can be implemented by maintaining only soft state at the reconstructor that can easily be regenerated. Note that no spurious acks are generated by the reconstructor and the end-to-end semantics of the connection are completely preserved. The trade-off in AR is between obtaining less bursty performance and a better rate of congestion window increase, versus a modest increase in the round-trip time estimate at the sender. We believe that it is a good trade-off in the asymmetric environments we are concerned with.

6.5 Scheduling Data and Acks

In the case of two-way transfers, data as well as ack packets compete for resources in the reverse direction (Section 4.2.2). In this case, a single FIFO queue for both data and acks could cause problems. For example, if the reverse channel is a 28.8 Kbps dialup line, the transmission of a 1 KB sized data packet would take about 280 ms. So if two such data packets get queued ahead of ack packets (not an uncommon occurrence since data packets are sent out in pairs during slow start), they would shut out acks for well over half a second. And if more than two data packets are queued up ahead of an ack, the acks would be delayed by even more.

To alleviate this problem, we configure the router to schedule data and ack packets differently from FIFO. A particular scheduling algorithm we consider is one that always gives higher priority to acks over data packets (*acks-first* scheduling). The motivation for this is that with techniques such as header compression [8], the transmission time of acks becomes small enough that it affects subsequent data packets very little (unless the per-packet overhead of the reverse channel is large, as is the case in packet radio networks). At the same time, it minimizes the idle time for the forward connection by minimizing the amount of time acks remain queued behind data packets.

Note that as with ACC, this scheduling scheme does not require the gateway to explicitly identify or maintain state for individual TCP connections.

In summary, ACC and AF are schemes to reduce the frequency of acks on the reverse channel, while SA and AR are schemes to overcome the adverse effects of reduced ack feedback. In our experiments, we use ACC in conjunction with SA, and AF in conjunction with SA or AR. Finally, the acks-first scheduling scheme is designed to prevent the forward transfer from being starved by data packets of the reverse transfer.

7. Simulation Results

7.1 Bandwidth Asymmetry

In this section, we present the results of several simulations of one-way and two-way TCP transfers on a network that exhibits bandwidth asymmetry.

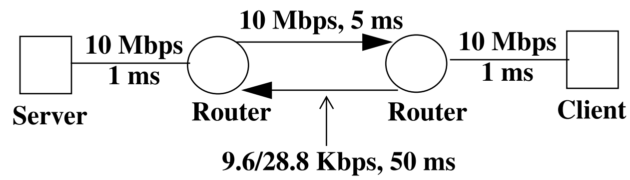

Figure 7. The simulation topology used to model a network with bandwidth asymmetry. The bandwidth and delay parameters have been chosen to closely model the Hybrid wireless cable modem network.

The simulation topology we used to investigate the effects of bandwidth asymmetry is shown in Figure 7. The parameters of the forward channel are based on our measurements of the Hybrid wireless cable network. We experimented with reverse channels of different bandwidths, but fixed delay. Although reverse channels ranging from slow to high speed dialup lines to ISDN have different latencies in reality, keeping the delay constant (at 50 ms) in the simulation experiments helps us focus on the bandwidth asymmetry aspect.

7.1.1 Single One-way Transfers

We conducted two sets of experiments, each involving a 50-second transfer in the forward direction. There was no traffic in the reverse direction other than the acks for the forward transfer in both cases. In the first set of experiments (A), there was sufficient buffering to ensure that there were no data losses in the forward direction. The second set of experiments (B) was designed to investigate how the

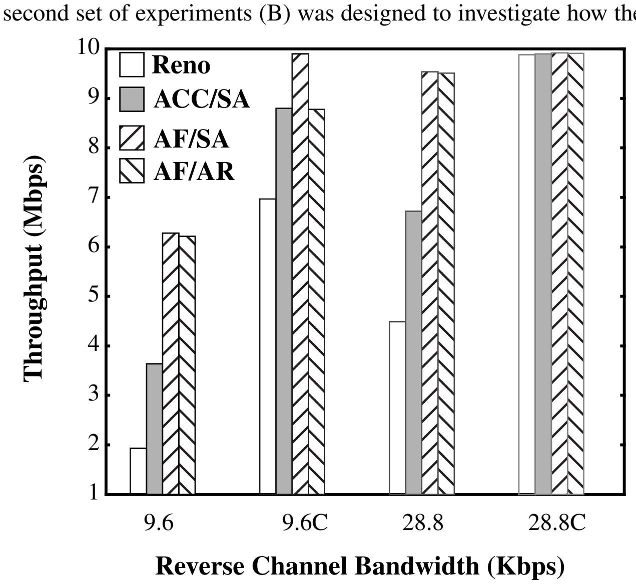

Figure 8. Throughputs from the simulation of a single one-way transfer in the forward direction with no data losses (case (A)). "C" indicates the use of SLIP header compression.

different protocols performed when losses occurred in the forward direction due to insufficient forward buffer capacity, and in particular to highlight the benefits of using SA or AR when the ack stream is lossy.

Lossless transfers: Figure 8 shows the throughputs obtained for four protocol configurations — TCP Reno, Reno with ACC/SA[6], Reno with AF/SA, and Reno with AF/AR — with different types of return channels, in case (A). The socket buffer size at the sender and receiver was set to 100 KB and each data packet was 1 KB in size. The buffer size at the reverse bottleneck router was set to 10 packets, and at all other routers to 20 packets. The ack size was set to 6 bytes and 40 bytes, respectively, with and without header compression. The motivation is to understand how the bandwidth and queue-length characteristics of the return channel affect forward throughput when there is no data loss.

The main observation here is that since the transfers are long, the reverse buffer fills up early on for Reno. Beyond that point, only one ack in *k* gets through on average, causing the sender to send out bursts of *k* packets. As long as *k* does not exceed the bottleneck buffer size in the forward direction, the increased burstiness of the sender does not lead to losses.

The factor *k* (the *normalized bandwidth ratio* as defined in Section 4.2.1) exceeds 20 for the cases of SLIP without header compression, resulting in bursts larger than the size of the buffers in the forward direction. This explains the poor throughput of TCP Reno in those cases (1.93 and 4.48 Mbps). The sender adaptation or ack reconstruction employed in conjunction with ACC and AF breaks up potential bursts, thereby avoiding performance degradation suffered by vanilla TCP Reno.

For the 9.6 Kbps reverse channel with header compression, *k* is 6.25, which is less than 20. Still, the throughput obtained with TCP Reno (6.67 Mbps) is worse than that for the other schemes. This happens because the reverse channel buffer gets filled with acks (totalling 10*6 = 60 bytes), which adds a significant delay (60*8/9.6 = 50 ms) to the connections round-trip time (RTT). The same effect also explains why the performance with ACC is somewhat worse than that with AF for both the 9.6 Kbps and 28.8 Kbps cases. The former only tries to ensure that the reverse channel queue does not get completely filled up. The latter ensures that there is not more than one ack per connection in the queue, which minimizes the effect of queuing on the round-trip time.

The following are the key results from the set of experiments (A):

1. C-SLIP is a big win, and in some cases (when there are no losses in the forward direction) it eliminates the problem entirely.
2. AF and ACC lead to significant improvements when the reverse buffer is small, especially when *k* is large.
3. Large reverse buffers cause Reno performance to severely degrade, and cause ACC to perform poorer This is because Reno now progresses at the rate at which acks leave the reverse queue, and it takes ACC does not kick in until the number of reverse acks is a large fraction of the reverse queue.

Lossy transfers: We now discuss the results of the second set of experiments (B), designed to study the effects of forward losses on performance, and to investigate how Reno, ACC and AF alter the inter-ack spacing and how SA and AR prevent burst transmissions. Table 1 shows the results of these experiments for the different protocols. The maximum window size was set to 120 packets and all queue sizes were set to 10 packets. We used a 28.8 Kbps header-compressed reverse channel.

6. In the rest of this paper, SA is implicitly bundled with each of ACC and AF, unless it is explicitly stated that AR is used instead or that SA is excluded.

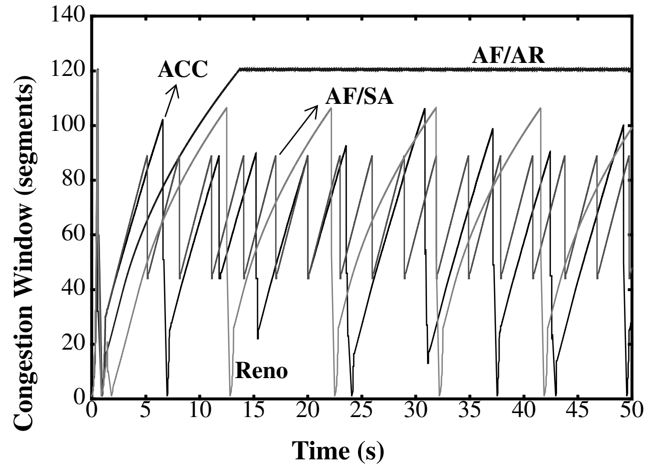

Figure 9. Congestion window evolution for the different schemes. ACC, AF/SA, AF/AR don't fluctuate as much as Reno, achieving better performance.

Metric	Reno	ACC/ SA	AF/ SA	AF/ AR	AF alone
Throughput (Mbps)	6.71	6.95	**7.82**	**8.57**	**5.16**
Average cwnd (pkts)	66.7	62	**65.3**	**104.6**	**43.8**
Average rtt (ms)	79	70	65	97	65

Table 1. Performance of different protocols in the presence of losses in the forward direction. The highlighted fields show the benefits of SA and AR and demonstrate that AF alone is not enough.

AF/AR and AF/SA perform the best, achieving throughputs between 15% and 21% better than Reno. ACC/SA performs about 5% better than Reno for this configuration. The important point to note is that the degree of burstiness is reduced significantly, while the reverse router queue is no longer perpetually full because of AF or ACC. This can be seen from Figure 9, which shows the time-evolution of congestion windows for the different protocols. Table 1 shows the time-averaged TCP congestion window and round-trip times for the different protocols. It is clear from the table that reducing the frequency of acks alone is not sufficient, and that techniques like SA or AR need to be used as well.

We note that AR results in a larger round-trip time than the other protocols, but this leads to the best performance for this setting of parameters because it reduces the number of losses (since packet transmissions from the source are now spread over a longer duration). For ACC/SA, we use a RED gateway to mark packets (acks) and drop packets when the queue is full. We found that using a random drop policy is superior to dropping from the tail when an ack has to be dropped. This is because tail drops sometimes lead to long, consecutive sequences of acks being dropped, leading to increased sender burstiness.

In summary, our results show that SA or AR is important to overcome the burstiness that results from a lossy ack stream, and that a random drop policy at the RED gateway was better for performance.

7.1.2 Two Simultaneous One-way Transfers

We now consider two simultaneous one-way transfers with the same topology as in Section 7.1.1 and the reverse channel fixed to be a 28.8 Kbps dialup line with header compression. The first transfer is initiated at time 0 and continues for 50 seconds. The second transfer starts at a randomly picked time between 5 and 10 seconds and ends at time equal to 50 seconds. Ten runs were conducted for each configuration. The goal here is to see how the two connections share the reverse channel bandwidth and buffer, which impacts the throughput of each.

Table 2 summarizes the results obtained in terms of the aggregate throughput for the two connections and the fairness index (as defined in Section 3.2) computed over the period during which both connections are active. We see that unmodified TCP Reno yields the best aggregate throughput but has a much worse fairness index value than the others.

Metric	Reno	ACC	AF
Total throughput	9.80	8.59	8.98
Fairness index	0.5	0.95	0.99

Table 2. The aggregate throughput (in Mbps) and the fairness index based on the simulation of two one-way transfers in the forward direction. The reverse channel is a 28.8 Kbps dialup line with header compression.

The high degree of unfairness with TCP Reno arises because the acks of the first connection quickly fill up the reverse channel buffer. So, when the second connection starts up, it suffers ack losses early on, leading to timeouts and hence a lack of progress. Even if all acks of the second connection were not lost, the growth of its window during the slow start phase would be slowed down because of the large queuing delay that its acks would encounter.

By decreasing the frequency of acks, ACC and AF keep the reverse channel queue small, so that the new connection does not face problems such as the ones that happen with unmodified TCP Reno. Consequently, the fairness indices in these cases are close to the maximum value of 1.

7.1.3 Two-way Transfers

We now consider the case when two simultaneous transfers are simultaneously active, one each in the forward and reverse directions. Again we fix the reverse channel to be a header-compressed 28.8 Kbps dialup line with a 50ms latency, with a buffer size of 10 packets. The forward transfer is initiated at time 0. The reverse transfer is initiated at a randomly picked time between 5 and 10 seconds. Both transfers continue until time equal to 50 seconds. The forward direction buffer size was set to 30 packets, thereby eliminating packet losses in the forward direction.

Table 3 shows the results of these two sets of experiments. The forward and reverse throughputs were computed over the period when transfers in both directions were active, and were averaged over 50 runs.

We make several interesting observations. With unmodified TCP Reno, acks of the forward connection could completely fill up the reverse channel buffer. In these experiments with no forward losses, this happened about 20% of the time (11 times out of 50). Consequently, the packets of the reverse connection get dropped with high probability. When this happens a few times in succession, the retransmission timer of the reverse connection backs off to such an extent that the connection makes no progress, resulting in zero throughput.

Protocol Combination	Forward Throughput (Mbps)	Reverse Throughput (Kbps)
TCP Reno (80%)	1.80	22.7
TCP Reno (20%)	9.93	0.00
ACC/SA	1.50	19.4
ACC/SA + acks-first	2.38	25.9
AF/SA	0.54	24.0
AF/SA + acks-first	9.93	0.00
AF/AR	2.03	19.0
AF/AR + acks-first	9.92	0.00

Table 3. The throughput from the simulation of simultaneous forward and reverse transfers. The reverse channel is 28.8 Kbps dialup line with TCP header compression. The forward-direction buffer size is 30 and reverse-direction buffer size is 10 packets. The large forward-direction buffer ensures that there is no packet loss due to buffer overflow.

However, the scenario was quite different in the remaining 80% of the runs. While the reverse channel buffer does tend to fill up with acks of the reverse connection, it is not 100% full at all times because packets are being dequeued regularly for transmission. In fact, the average queue size was 8.5 packets as opposed to the buffer capacity of 10. It is, therefore, possible for 1 or 2 data packets of the reverse connection to get into the queue. When this happens, later acks of the forward connection suddenly experience a much longer delay because they are queued behind 1000-byte packets, causing the forward connection to *time out* and cut down its congestion window to 1, even though no data packet has been lost. At this point, the reverse connection can progress, until it experiences a loss, whereupon the forward connection takes over again, and so on. Thus, the 2 connections take turns in making progress in bursts, rather than in a simultaneous manner.

AF achieves relatively poor throughput for the forward transfer but close to optimal throughput for the reverse transfer. The reason this happens is that when the reverse transfer starts up, 1 KB sized data packets start entering the reverse channel queue. The transmission time of each data packet over the 28.8 Kbps line is 280 ms. Because of FIFO scheduling, acks of the forward transfer get queued behind these data packets for this entire duration, causing the sender of the forward transfer to stall. Many acks are also lost during this period. These may cause the sender to time out while waiting for acks. But the reverse connection continues building up its window, so as time progresses, ack packets get queued behind not one but several data packets. The end result is that the forward connection makes progress in short bursts interspersed by multi-second idle times. Figure 10 illustrates this for a simulation experiment with AF.

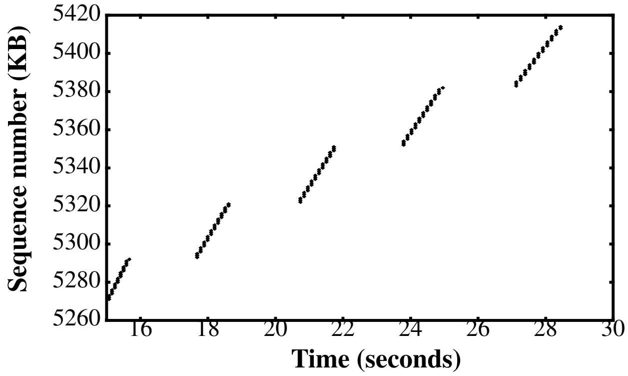

Figure 10. Simulation results showing a portion of the sequence number trace for the forward transfer after the reverse transfer has started up. The reverse channel router uses ack filtering. The multi-second idle times are caused by acks getting queued behind multiple 1 KB data packets belonging to the reverse transfer.

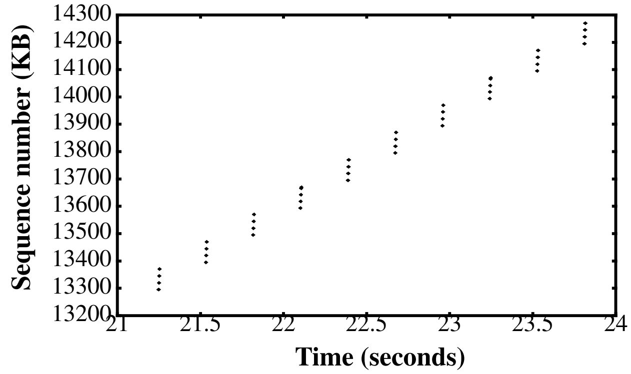

Figure 11. Simulation results showing a portion of the ack trace for the forward transfer after the reverse transfer has started up. ACC is used in conjunction with acks-first scheduling. There is an idle time of about 280 ms between bursts of acks because of the 1 KB data packets belonging to the reverse transfer.

With ACC (and the reverse channel router employing the RED algorithm), the throughput of the reverse transfer is reasonably good (19.4 Kbps as against the maximum possible of 28.8 Kbps). At the same time, the throughput of the forward transfer (1.5 Mbps) is much better than for the AF/SA configuration. The reason for this is that feedback from the RED gateway prevents the reverse transfer from filling up the reverse gateway with its data packets. The reverse connection can sustain optimal throughput without having to grow its window to more than 1-2 packets. (Even assuming a rather large RTT of 500 ms for the reverse connection, the bandwidth-delay product is 28.8 Kbps * 500 ms = 1.8 KB which is less than two 1 KB packets.) Thus, the reverse connection can decrease the impact that its data packets have on ack packets of the forward transfer, while sustaining optimal throughput.

Even with the RED algorithm in operation, ack packets could get queued behind more than one data packet, which decreases forward throughput. The acks-first scheduling scheme (Section 6.5) avoids this by prioritizing acks over data. The assumption is that such scheduling will not add significantly to the queuing delay of data packets. With ACC (which decreases the frequency of acks) and header compression (which makes them small in size), data packets are indeed not affected significantly. As shown in Table 3, ACC with acks-first scheduling achieves a forward throughput of 2.38 Mbps while maintaining a close-to-optimal reverse throughput (25.9 Kbps).

A simple calculation shows that with the parameters we have chosen, we cannot do better than 2.85 Mbps while maintaining optimal reverse throughput. While a data packet of the reverse connection is undergoing transmission on the 28.8 Kbps link (lasting 280 ms), the forward connection sender does not receive any new acks. Figure 11 illustrates this effect through simulation. So it can send at most one window's worth of data in 280 ms. With the socket buffer size of 100 KB that we have chosen, the maximum sender throughput works out to 100*8/280 = 2.85 Mbps.

In contrast to ACC, combining acks-first scheduling with AF leads to starvation of data packets of the reverse transfer. This is because ack packets arrive at the queue at a rate faster than they can be drained out, so there is always an ack waiting to be sent in the queue. Note that an ack undergoing transmission is no longer in the queue, and so is not considered by the ack filtering algorithm.

Finally, to point out the benefits of using RED feedback to do ACC, we consider the case where feedback from the RED gateway is only applied to data (of the reverse connection) and not to acks, which in effect disables ACC. The forward throughput is higher than before (3.03 Mbps), but the reverse throughput is only 17.84 Kbps (these numbers not shown in Table 3). Since acks are not subject to congestion control like data, they cause the reverse connection to lose packets and time out periodically. During these idle periods of the reverse connection, the forward transfer makes rapid progress, resulting in a higher forward throughput than before.

7.2 Latency Asymmetry

In this section, we perform a detailed analysis of the problems caused by latency asymmetry and present some solutions that alleviate the adverse effects of increased round-trip time variability. Based on several experimental measurements of the Ricochet network, we modeled the system in the ns simulator. We extended the point-to-point link abstraction of ns to a more general shared LAN and added support for arbitrary MAC and link-layer protocols. The simulation parameters used to obtain the results described in Section 7.2 are shown in Table 4. We do not consider the impact of wireless bit errors in these simulations, to isolate the impact of variability due to the MAC protocol on performance. In practice, link-layer retransmissions of corrupted packets will only add to the variability of the network.

Parameter	Value
Link bandwidth	100 Kbps
Fixed link latency	10 ms
T_{TR}	11.125 ms
T_{RT}	13.25 ms
Radio queue size	10 packets
Receiver buffer size	32 KBytes

Table 4. Simulation parameters of the multi-hop packet radio network. The number of wireless hops varies between 1 and 3.

Our first set of experiments are for single one-way transfers through a network including the wireless cloud, with the number of wireless hops varying between 1 and 3, similar to the topology of the commercial Ricochet network. Then, we experiment with multiple simultaneous TCP transfers through the wireless cloud, to highlight the issues of fairness and scale in this wireless network.

We start by discussing an optimization to the underlying link-layer protocol.

7.2.1 Piggybacking Link-Layer Acks with Data

This scheme is motivated by the observation that the radios turnaround both for data frames as well as for link-layer acks. The presence of traffic in both directions, even when caused by TCP acknowledgments, already causes turnarounds to happen. Thus, link-layer acks can be piggybacked with data frames, thereby avoiding some extra radio turnarounds.

The basic reliable link-layer protocols in several systems do not piggyback acks with data. However, recent releases of the radio software in the Ricochet network attempt to do this whenever possible. Our simulations of the multi-hop wireless network assume that the radio units piggyback link-layer acks with data.

Despite this optimization, the fundamental problem of additional traffic and underlying protocols affecting round-trip time estimates and causing variabilities in performance still persists. Connections traversing multiple hops of the wireless network are more vulnerable to this effect, because it is now more likely that the radio units may already be engaged in conversation with other peers.

7.2.2 Single One-Way Transfers

We vary the number of wireless hops in the simulated network from 1 to 3 and measure the performance of bulk TCP transfers. The workload consists of a 100-second TCP transfer, with no other competing traffic and a maximum receiver window size of 32 KBytes. Congestion losses occur as a result of buffer overflow, and lead to sender timeouts if multiple packets are lost in a transmission window. The protocols we investigated include unmodified TCP Reno, Reno with ACC/SA, and Reno with AF/SA.

Figure 12 shows the results of these experiments, as a function of the number of wireless hops. The performance of AF and ACC with SA are better than Reno, and AF/SA is better than ACC/SA. The performance improvement for AF/SA over Reno is shown in Table 5 — the degree of improvement in throughput varies from 20% (1 hop) to 30% (3 hops).

# hops	Reno (Kbps)	Reno+ACC/SA (Kbps (%age))	Reno+AF/SA (Kbps (%age))
1	40.7	42.8 (5%)	51.0 (20%)
2	19.0	22.0 (14%)	24.0 (25%)
3	12.1	14.5 (17%)	17.1 (30%)

Table 5. The performance of Reno, ACC/SA, and AF/SA as a function of the number of wireless hops in the simulated multihop wireless network. Throughputs are in Kbps, and the numbers in parentheses are percentage improvements compared to Reno for the same configuration.

The main reasons for this improvement are the reduced number of packets and reduced round-trip variability of the two enhanced

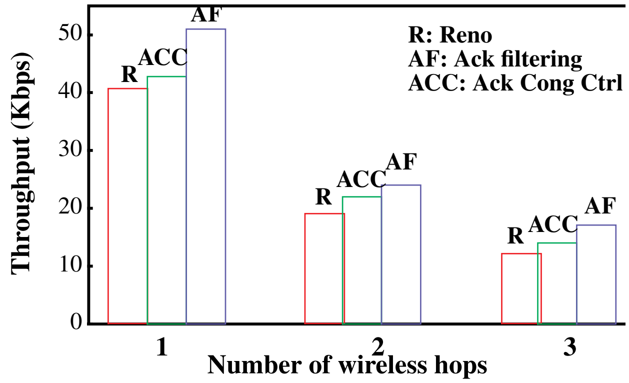

Figure 12. TCP throughputs from simulations of Reno, ACC and AF, as a function of the number of wireless hops.

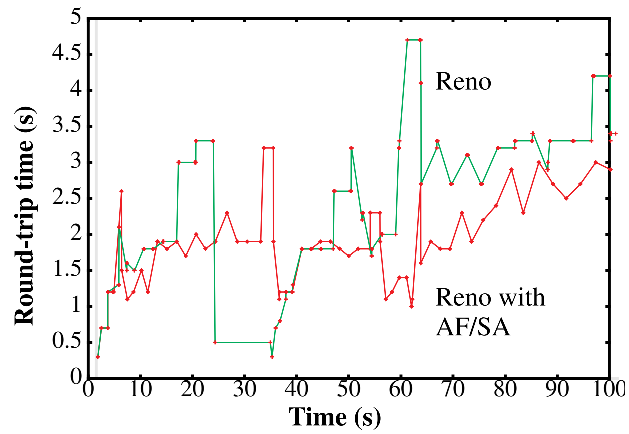

Figure 13. Round-trip times obtained from simulations of TCP Reno and TCP with AF/SA for a connection over two wireless hops. The round-trip times for the Reno connection are much more variable, with a mean of 2.67 secs and a standard deviation of 1 sec, whereas AF reduces the mean to 1.85 secs and the standard deviation to 0.6 secs.

protocols, compared to Reno. Figure 13 shows the round-trip times of simulations of a TCP Reno connection and a TCP connection with AF, over two wireless hops (in a chain-like topology between sender and receiver). For Reno, the mean round-trip time is about 2.67 seconds and the standard deviation is about 1 second. AF reduces the mean round-trip time to 1.85 seconds and the corresponding standard deviation to only 0.6 seconds. The number of packets traversing each node also drops, reducing the amount of contention. These factors result in a 25% improvement in end-to-end throughput, from 19 Kbps (Reno) to 24 Kbps over 2 wireless hops. Similar improvement in performance is seen for connections traversing three wireless hops — end-to-end throughput improves on average from 12.1 Kbps (Reno) to 17.0 Kbps (AF), an improvement of 30%. While the exact values of the round-trip time and the deviation are a strong function of the window size and the amount of competing traffic, these trends toward improvement are observed in other configurations as well.

Finally, we note that AF outperforms ACC because the former completely eliminates all redundant acks and reduces the amount of "interfering" traffic caused by TCP acknowledgments to a greater extent.

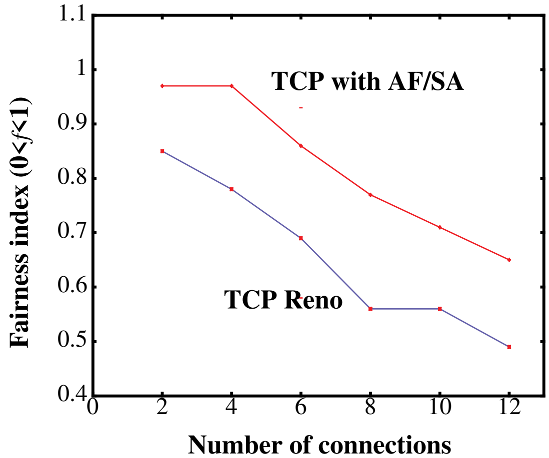

Figure 14. Simulation results for the fairness index of TCP Reno and TCP with AF, as a function of the number of connections traversing one hop of the packet radio network.

7.2.3 Multiple Simultaneous Transfers

We now experiment with multiple simultaneous transfers in the multihop wireless network, in order to understand performance as a function of the number of connections traversing the network. These concurrent connections compete for resources in the network. For simplicity, we focus on the results of connections over one wireless hop in this section for the best performing protocol, Reno+AF/SA, and compare it to Reno.

There are two important metrics to consider while studying the impact of increasing the number of connections in the network: *utilization* and *fairness*. Network utilization is defined as the ratio of the aggregate throughput of all connections to the maximum achievable throughput of the network. Fairness is quantified using the fairness index defined in Section 3.2. We are interested in obtained large values of both the network utilization as well as the fairness index for any configuration.

Table 6 shows the simulated aggregate throughput achieved by all the connections in the network, as a function of the number of competing connections for Reno and AF/SA. The table also shows the standard deviation of the resulting performance, which shows the degree of variation in the throughputs seen by the different concurrent connections. The aggregate performance varies between 44 and 47 Kbps for TCP Reno and between 48 and 52 Kbps with AF/SA, as the number of connections varies, implying that there is little change in utilization as a function of the number of connections. However, the standard deviation of the performance, calculated over concurrent connections, is much higher for Reno than for Reno+AF/SA. This suggests that the distribution of throughputs is more spread out and less equal across any set of connections, especially as their number increases.

Simultaneous Connections	Reno throughput (Kbps) [std-dev]	Reno+AF/SA (Kbps) [std-dev]
1	40.7 [-]	51.0 [-]
2	42.8 [13.1]	51.8 [10.0]
4	45.2 [24.9]	49.3 [8.9]
6	47.1 [32.5]	49.3 [20.2]
8	45.0 [37.6]	49.6 [28.0]
10	45.6 [42.2]	48.4 [32.4]
12	45.8 [48.0]	48.8 [36.4]

Table 6. Throughputs of TCP Reno and Reno with AF/SA as a function of the number of connections over one wireless hop. The numbers in brackets are the standard deviations of the throughputs calculated over the simultaneous connections.

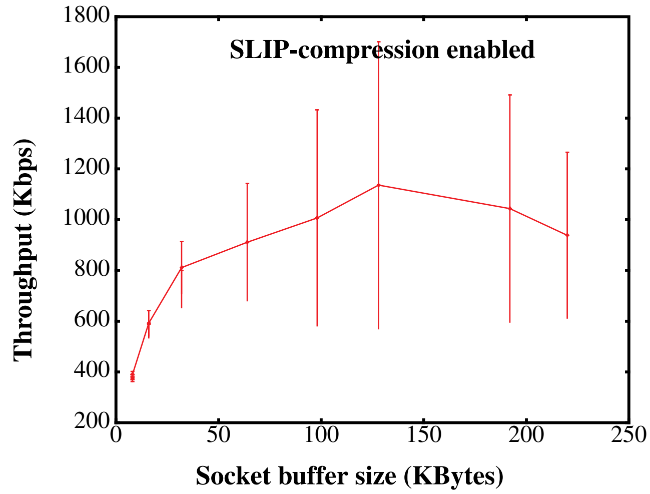

Figure 15. Measured performance of a 1 MByte TCP transfer across a Hybrid wireless cable downlink and Ricochet return channel. The large error bars are a consequence of the inherent variability of the Ricochet return channel.

We quantify this effect further in Figure 14, where the fairness index of throughput is plotted as a function of the number of simultaneous connections. TCP Reno is often grossly unfair in the distribution of throughput, reaching a value as low as 0.49 (n=12) and not exceeding 0.85 (n=2). In contrast, AF/SA substantially improves the fairness index of the throughput distribution, while also improving the overall utilization of the network. The reasons for the improvements in fairness are the reduced number of interfering and competing packets in the network.

Thus, we see that our enhancements to TCP and the router algorithms help improve overall utilization and fairness as the number of connections increases. In Section 10, we revisit further investigate some scaling issues by simulating a Web-like workload in a network with both a high-bandwidth forward channel and a packet radio return channel.

7.3 Combining Wireless Technologies: Wireless Cable and Packet Radio

In this section, we investigate the effects of combining different types of asymmetry on TCP performance. We focus on a network topology with a high-bandwidth forward path modeled after Hybrid's wireless cable channel, and a low-bandwidth, packet radio reverse path modeled after the Ricochet network. Such composite network topologies are relevant in several application scenarios. For example, a disaster relief vehicle or ambulance with a unidirectional high-bandwidth link would use a wide-area wireless network as its reverse channel.

Figure 15 shows the measured performance of 1MB TCP transfers as a function of the receiver socket buffer size using a Hybrid Networks' wireless cable forward path and a one-hop wireless Ricochet return path. The error-bars show the standard deviation of

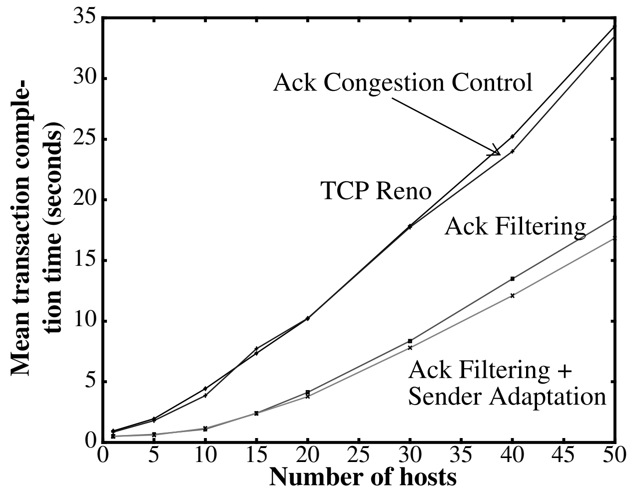

Figure 16. Simulated scaling behavior of *TCP Reno*, *AF*, and *ACC* as a function of the number of connections for a Web-like micro-benchmark over a packet radio return path. The graph shows average completion time vs. *n*, the number of hosts (4 concurrent connections per host). The ack filtering protocol performs the best as *n* increases.

measured performance (20 runs each). These controlled measurements were performed in the absence of any cross-traffic and the inherent variability of the return path manifests itself in the significant error-bars on the graph.

We focus on a Web-like benchmark in the following simulations and study the performance of this network as the number of hosts and connections increases. We investigate the various modifications to the transport and router protocols to help reduce the average completion time of a Web request in such networks.

We model the Web micro-benchmark as a 500 byte Web request followed by set of 4 concurrent TCP transfers of 10 KB each to the client. This is not intended to be an accurate model of reality, but rather to understand the effects of small and concurrent connections, as well as competing and interacting users, on performance. Since latency is a critical factor that impacts performance in the packet radio network, we implicitly assume that the Web requests use pipelining [23] and that one 500 byte request results in 4 downloads.

We vary the number of hosts from 1 to 50 in the simulations. Hosts make requests independent of each other at a time uniformly distributed in [0,5] seconds. We measure the mean and standard deviation of the completion time for the entire Web transaction (i.e., all 4 connections).

Figure 16 shows the mean completion time for a transaction as the number of hosts varies, for TCP Reno, Reno with ACC (Section 6.1) and Reno with AF (Section 6.2). Two curves are shown for the AF case — with sender adaptation enabled based on the round-trip time and the congestion window, and without (Section 6.3). These results indicate that AF is very beneficial in reducing the response time and improving the throughput of the network as the system scales. The reason ACC is not as beneficial as in the cases investigated in Section 4 is the shorter transfer lengths in this benchmark. No connection exceeds 10 packets, so the sender's window is never very large[7]. This limits the extent to which ACC can be performed. The other reason is the larger number of acks traversing the network with ACC, compared to AF. One critical factor in this network is the latency and variability associated with each packet on the wireless network. AF results in significant gains because it purges all redundant acks from the

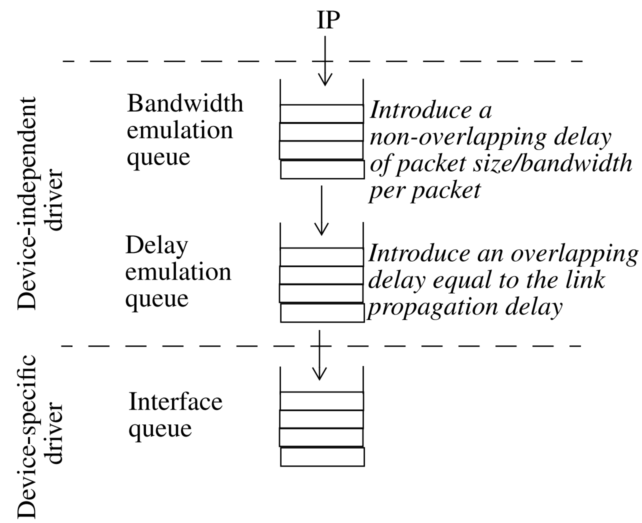

Figure 17. Structure of link emulation module. Note that the delay emulation queue is organized as a callout queue to enable emulation of (overlapping) propagation delays.

queue independent of the state of congestion, thereby reducing the number of packets in the wireless cloud. Finally, we note that the lack of sender adaptation does not significantly hurt performance, since each connection is rather small. The maximum possible burst in the network is automatically limited by this short length and the slow start process starting from 1 segment.

8. Implementation

Having analyzed the various solution techniques via simulation, the next step is the implementation and evaluation of the promising techniques in our network testbed. We used a Pentium-based PC platform running the BSD/OS 3.0 operating system from BSDI, Inc. [5] for both the end-host and the router implementations. The algorithms that we have implemented are: ack congestion control (ACC), ack filtering (AF), sender adaptation (SA), and acks-first scheduling.

One practical problem we encountered while experimenting with a modem reverse channel was the packets were getting queued in the on-board memory on the modem rather than in the operating system's interface queue. This poses a problem for ack filtering because it requires access to the queue. Ideally, the AF algorithm should be implemented in the modem firmware where it has access to the queue. However, this was difficult for us to do. Therefore, to evaluate ack filtering, we chose to emulate a modem link in software so that the queuing occurs in software. We begin by describing the link emulation module.

8.1 Link Emulation

Our link emulation module emulates a link of a certain bandwidth and delay in software. This code is part of the device-independent driver for the Ethernet class of devices, so it can be used with a wide variety of physical channels including 10 Base-T and 100 Base-T Ethernet, and WaveLAN.

7. With persistent-connection HTTP [23], recommended by HTTP/1.1 [10], connections will tend to be longer. This should help ACC.

The link emulation code is structured as shown in Figure 17. It operates in two stages — bandwidth emulation first and then delay emulation. The reason for this separation is that the transmission of the individual packets happens in succession (i.e., two packets cannot overlap) but the propagation can happen concurrently.

To emulate bandwidth, the transmission time of the packet at the head of the input queue is computed. A timer is set to expire at the time when the transmission of this packet is supposed to end. Any packets arriving in the mean time are queued up in the bandwidth emulation queue, and algorithms such as ack filtering or RED packet marking can operate on the them. When the timer expires, the designated packet is moved over to the delay emulation queue.

This second queue used a callout queue data structure and operates strictly in FIFO order. The purpose of the queue is to insert a delay, corresponding to the propagation delay of the emulated link, into the outgoing data path. This is done by setting timers corresponding to the propagation delay. Each time the timer expires, the packet at the head of the queue is dequeued and handed down to the device-dependent driver for actual transmission. If the callout queue still has other packets, a fresh timer is set corresponding to the *remaining* propagation delay for the new packet at the head of the queue.

There are two issues that need to be considered in the context of our link emulation algorithm. First, the transmission and propagation due to the underlying physical channel are not considered. We believe this simplification is appropriate for emulating a slow link over a fast physical channel (e.g., modem link over 10 Base-T Ethernet), which is precisely our goal. Second, the granularity of the software clock in the BSDI operating system is 10 ms, which limits the accuracy with which we can schedule packets. Fortunately, for slow links with relatively large transmission and propagation delays, the 10 ms scheduling granularity becomes less significant. Furthermore, when packets tend to get queued up at the input queue of the emulated link, there is an increased possibility for the scheduling inaccuracies to cancel out over successive packets (e.g., one packet may undershoot a bit and a later one may overshoot a bit).

8.2 Ack Congestion Control

The basic objective of ACC is for the receiver to decrease the frequency of acks in order to alleviate congestion on the bandwidth-constrained reverse channel. In general, there are two modes which the receiver could operate in.

1. It could decrease the frequency of acks to the minimum possible based on preset limits and information from the sender (as explained below), or
2. It could do so in response to explicit congestion notifications from the reverse channel router.

In addition to changing the TCP algorithm at the receiving and sending sides, supporting both these modes of operation require two changes to the on-the-wire protocol:

ECN bit: The TCP and IPv4 headers have no designated field for explicit congestion notification from the routers to the end hosts. However, there are a few unused bits in the TCP header, and one of these could be used as the ECN bit. Note that this bit is needed only if the receiver operates in the second mode mentioned above.

Sender Window Option: The ACC algorithm requires the TCP receiver to have knowledge of the sender's window size to make sure that the receiver does not decrease the frequency of acks to such an extent that the sender stalls. To convey this information from the sender to the receiver, we added a peerwin TCP option.

Sender algorithm:

```
w <- min(cwnd, sb_dat)  /* min. of congestion window and amount
                           of data in send socket buffer */
w <- w/mss              /* scale down by max. segment size tc
                           make it a 2-byte quantity */
Send w in a peerwin TCP option to the receiver
```

Receiver algorithm:

```
Update the variable peerwin based on the TCP option received
w <- min(peerwin, sb_sz/mss)  /* min. of sender's window and
                                 receiver's socket buffer size */
delack <- w/min_acks_per_win  /* send at least min_acks_per_win
                                 per window of data */
delack <- min(maxdelack, delack) /* upper bound on delack */
Send an ack every delack segments (instead of the every 2 seg-
ments as is done with standard TCP delayed acks)
```

Figure 18. Sender and receiver algorithms for dynamically computing delack for ack congestion control

The sender computes a 2-byte quantity which the minimum of its congestion window size and the amount of data it has in its socket buffer as a multiple of the maximum segment size (MSS), and inserts it into the peerwin option. Another 2 bytes of overhead make the total length of the option 4 bytes. Since this option is added only to packets traversing the high-bandwidth forward path, we do not believe that this 4-byte overhead is significant.

In our implementation, the receiver operates in the first mode mentioned above, so the ECN bit is not required. The TCP receiver maintains the following additional variables in its protocol control block to do ACC:

- delack: the delayed ack factor. The receiver sends back one ack for every delack segments received.
- max_delack: upper bound on delack. This can be set using a user-level configuration program.
- peerwin: the size of the peer's window (expressed as a multiple of the MSS). This is updated each time a segment with the peerwin option enabled is received from the sender. The sender fills in this option by computing the minimum of its congestion window size and the amount of data there is in its socket buffer
- min_acks_per_win: the minimum number of acks to send per window of data that the sender. Clearly, this should be set at least to 1. However, it may have to be set to a somewhat larger value to alleviate ill effects (such as sender stalls) due to delays and/or losses that acks could experience as they travel from the receiver to the sender.

In the BSDI implementation, the TCP receiver sends an ack every time the receiving application reads in enough data (thereby clearing up enough space in the receive socket buffer) that at least 2 segments' worth of additional flow control window can be advertised to the sender. With ACC, the receiver checks for delack segments of space instead of 2 segments.

The receiver dynamically computes delack as shown in Figure 18. It first computes the sender's effective window size, w, as the minimum of peerwin and the receive socket buffer size (expressed as a multiple of the MSS). Since the receiver must send at least min_acks_per_win acks per window, delack is set to w divided by

Tying window growth to amount of data acknowledged:

```
acked <- new_ack - snd_una  /* value of new ack minus sequence
                               # of first unacknowledged byte */
incr_count <- acked/(2*mss)  /* # of window increase operations */
if (slow start phase) cwnd += incr_count*mss
if (cong. avoidance phase) cwnd += incr_count*mss/cwnd
```

Scheduling data bursts:

```
if (burst timer is pending) return
if (segs_to_send > maxburst) {
   timeout(maxburst/(cwnd/t_srtt_exact))   /* schedule next burst */
   segs_to_send = maxburst
}
Transmit segs_to_send  segments
```

Figure 19. Sender adaptation algorithm

min_acks_per_win. Finally, the receiver makes sure that delack does not exceed the preset limit of max_delack.

The standard delayed ack timer is still used just as before. The receiver forces out an ack if and when this timer expires. Since this only happens infrequently, these forced acks are not a significant addition to the reverse direction traffic.

To summarize, our implementation of ack congestion control operates purely in an end-to-end mode with no special role played by the routers in between. While this simplifies the implementation, it requires the user/system administrator to configure the delack and min_acks_per_win parameters on hosts that expect to see asymmetric connectivity.

8.3 Sender Adaptation

The goal of sender adaptation is to enable the TCP sender to operate successfully even when the frequency of acks is decreased as a result of ack congestion control or ack filtering. There are two aspects to sender adaptation:

1. Tying congestion window growth to the amount of data acknowledged rather than the number of ack packets received.
2. Avoiding large bursts of data when each ack acknowledge several new data segments.

The sender adaptation algorithm is shown in Figure 19. In order to satisfy the first requirement mentioned above, the sender computes acked, the number of bytes newly acknowledged by the most recent ack. The quantity acked is scaled down by 2*mss to determine the number of window increase operations to be performed (either in the slow start phase or in the linear increase phase). The reason for scaling down by 2*mss is to make it equivalent to TCP Reno with standard delayed acks, which involves one window increase operation for every 2 data segments acked.

To avoid large bursts of data, the sender limits the maximum number of segment that can be sent out back-to-back to maxburst. If still more segments remain to be sent, a software timer is scheduled for a later time. The wait time for the timer is computed by dividing maxburst by the ratio of the congestion window size (cwnd) to the round trip time (cwnd/rtt equals the rate of the connection). The underlying principle here is that the scheduling of the bursts (each of size maxburst) should match the overall rate of the connection. Upon expiry of the timer, the next burst of data is transmitted.

In order to compute the rate, we need to measure the round trip time far more accurately than is done using the timestamp option in standard TCP (500 ms granularity). We solve this problem by having the sender insert the current time with microsecond granularity in the timestamp option, which is then echoed by the receiver. With 4 bytes to represent the time value in, we don't run into wrap-around problems so long as the RTT is under 4000 seconds, which is certainly true in practice. The quantity t_exact_srtt thus computed is used in the rate calculation mentioned above. It is also translated into a 500 ms granularity quantity, t_srtt, which is used to compute the retransmission timeout value as in standard TCP.

8.4 Ack Filtering

The goal of ack filtering is to remove all redundant acks from the queue of the reverse router, when a newer cumulative ack arrives there. We describe below our experiences with implementing this scheme and how we did our experiments, and also mention some practical problems especially in the context of the packet radio network that we are working to overcome.

Since both the reverse connection for the bandwidth asymmetry case and the network connectivity for the latency asymmetry case use Hayes-compatible modems running PPP, we implemented this scheme in the PPP layer of the BSD/OS 3.0 kernel. The basic idea is quite straightforward — when a new ack arrives, we go through the queue of packets and remove all redundant acks for the connection, taking care not to remove any acks that have data associated with them. A connection is uniquely identified by the 4-tuple <source address, destination address, source port, destination port>. In practice, packets on the PPP queue are often header-compressed using the algorithm described in [13], which makes it hard to offset into the header and obtain the necessary fields.

There are several alternative approaches to solving this problem:

1. Decompress the compressed packets in the queue each time a new ack arrives and compare fields. This requires maintaining extensive decompression state at the compressor (ack filter) and is expensive and hard to implement.
2. Maintain two queues explicitly at the PPP layer, one for header-compressed packets and the other for the TCP/IP headers alone. Then, upon every enqueue/dequeue to the packet queue, the corresponding header must be enqueued/dequeued on the header queue as well. We chose not to adopt this method because dequeueing of packets can be done from a variety of underlying disciplines, and more importantly, we need to maintain the consistency between the two queue data structures.
3. Modify the PPP queue data structure to maintain not just pointers to kernel mbufs [19] (corresponding to the header-compressed packets), but a pointer to a structure containing the relevant fields of the packet header as well as the kernel mbuf. This is the simplest alternative that isn't inefficient, and is the one we implemented.

Finally, acks are purged if their sequence numbers are smaller than the newly arrived ack, or if they are the same and the new ack advertises a different window from the enqueued one (since later window updates subsume earlier ones). Currently, we do not filter out duplicate acks that could trigger a fast retransmission at the sender. We could do so by adding a TCP option that specified the number of purged duplicate acks and propagating that to the sender from the router, and by modifying the sender to recognize and react to this option.

When we implemented this mechanism and deployed it, we found that the queueing of packets occurs inside the modem and not in the kernel's PPP queue, with both the Ricochet and standard telephone modems. Therefore, in our experiments bandwidth asymmetry with AF, we emulated the modem link in software. Since in this case packets are not compressed, we can offset to the TCP and IP headers and purge redundant acks. For the experiments with latency asymmetry, we are currently working with Metricom to obtain research modems with the ability to control queues on the modem and cause queueing to happen in the kernel's PPP queue. We will be able to measure the benefits of our proposed enhancements once that happens.

9. Implementation Results

In this section, we present experimental results based on the implementation described in Section 8. We present results of bandwidth asymmetry experiments here. We are currently in the process of evaluating our improvements with latency asymmetry in our experimental Ricochet network. We expect to have performance results for this network in the next few weeks.

9.1 Bandwidth Asymmetry Experiments

At the time we conducted these experiments, the Hybrid wireless cable modem network (Section 3.1) was unfortunately unavailable. Given the Ethernet-like characteristics of this network (10 Mbps raw bandwidth, 2 ms latency, negligible error rate), we decided to substitute it with an Ethernet segment for the forward channel. We used a 28.8 Kbps dialup reverse channel for all experiments except those involving ack filtering. For the reasons mentioned in Section 8, we emulated a dialup-like link (with 28 Kbps bandwidth and 80 ms one-way latency) in software for the ack filtering experiments. The size of the reverse channel buffer was set to 32 packets, the default for a PPP dialup interface under BSDI 3.0.

Standard header compression [13] was enabled on the dialup reverse channel. However, upon tracing through the code, we discovered that the compression code was never being invoked because as it stands it cannot deal with TCP headers that include options (such as the timestamp option) that change in value from one segment to the next. It is possible in principle to change the compression algorithm to accommodate TCP options. But this was not feasible in our situation because we had no control over terminal server we were dialing in to. However, we observe that in a situation were it is feasible to deploy a link-local scheme such as AF/AR, it should also be possible to deploy a new header compression algorithm.

Each experiment described below involves one or both of a TCP data transfer in the forward direction and one in the reverse direction. The length of the forward transfer was 3 MB and the sender and receiver socket buffer sizes were set to 150 KB. The length of the reverse transfer was 100 KB and the sender and receiver socket buffer sizes were set to 100 KB also. The TCP maximum segment size (MSS) was set to 1424 bytes for transfers in both directions.

9.1.1 Sender Adaptation: Effect of maxburst

The parameter maxburst determines the size of the largest burst (i.e., sequence of back-to-back packets) that a sender could transmit. This parameter is of significance when the sender receives acks infrequently, as is the case with ack congestion control or ack filtering.

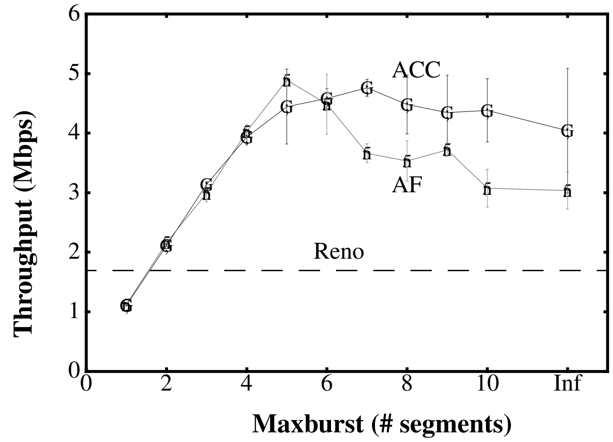

Figure 20. Throughput of a 3 MB forward transfer versus maxburst. For ACC, min_acks_per_win = 5 and max_delack = 8. maxburst = Inf (infinity) means that there is no limit on the burst size. The data points show the average of 10 runs and the vertical bars correspond to +/- one standard deviation.

Each run of our experiment involved a 3 MB data transfer in the forward direction. We did 10 runs for each value of maxburst ranging from 1 through 10. We also conducted a set of runs with maxburst set to infinity which corresponds to there being no limit on the burst size.

Figure 20 plots the throughput of the forward direction data transfer versus maxburst with ACC and AF. The throughput increases steadily at first and then either levels off or drops off.

The initial increase corresponds to the situation where the throughput is limited by the 10 ms granularity of the software timer used to schedule bursts. For example, when maxburst is set to 3, the maximum possible throughput is obtained by sending 3 segments every 10 ms. With 1424-byte segments, this works out to a throughput of 3.4 Mbps. This explains why as maxburst is increased from 1 through 4, the average throughput increases almost linearly.

The levelling off or the drop off of average throughput, and the significant increase in variability (as evidenced by the longer vertical bars), happen because as maxburst gets large, the large bursts result in buffer overflow (most likely at the Ethernet network interface card) at least some of the time. The poor throughput on runs that involve packet loss pulls down the average and increases variability.

Based on the data in Figure 20, it seems best to set maxburst to 4 or 5. Standard TCP with delayed acks could send bursts of size 3 during slow start. Therefore, setting maxburst to 4 seems conservative enough because it would result in bursts at most one larger than already happen in the Internet today.

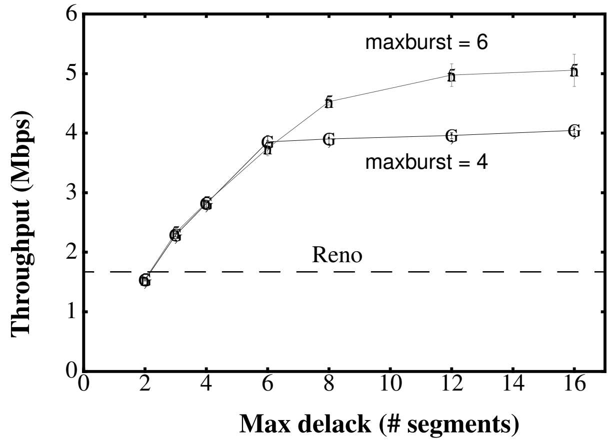

Figure 21. Throughput of a 3 MB forward transfer versus max_delack. The parameter min_acks_per_win = 5 and max-burst = 4 or 6. The data points show the average of 10 runs and the vertical bars correspond to +/- one standard deviation.

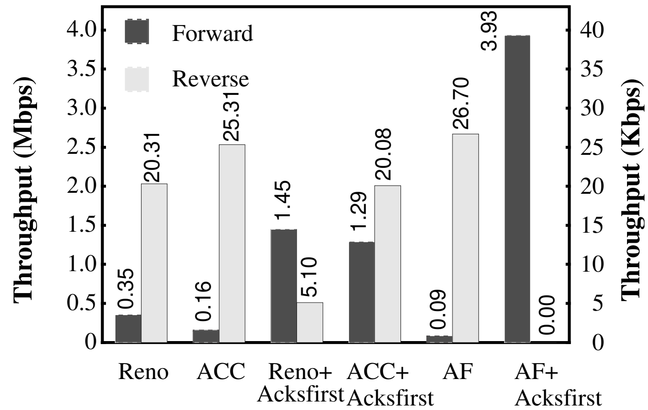

Figure 22. Throughput of a 3 MB forward transfer and a 100 KB reverse. The reverse transfer is initiated 2-3 seconds after the forward transfer. Only the time during which both transfers are active is considered for computing throughput. Note that the reverse throughput is 0 for the “AF+Acksfirst” case.

9.1.2 ACC: Effect of max_delack

With ACC, the parameter max_delack determines the minimum frequency of acks from the receiver. Since max_delack is an upper bound on delack, the receiver has to send at least 1 ack every max_delack segments of data that it receives.

We conducted a set of 3 MB data transfers similar to the ones discussed above, but varied max_delack between 2 and 16 over the different sets of runs. The throughput obtained is plotted against max_delack in Figure 21.

As max_delack is increased, throughput also increases initially. The reason for this is that the decreasing frequency of acks alleviates congestion of acks on the reverse channel. However, beyond max_delack equal to 8, there is little increase in throughput, presumably because congestion on the reverse channel has already been eliminated.

9.1.3 Effect of Two-way Traffic

In this experiment, we consider the situation of simultaneous transfers in the forward and reverse directions. As we observed in simulation experiments (Section 7.1.3), there are situations where data transfer in one direction shuts out the one in the opposite direction.

In each run of our experiment, we first initiated a 3 MB data transfer in the forward direction, and 2-3 seconds later initiated a 100 KB reverse transfer. For the forward-direction connection, maxburst was set to 4. Also, when ACC was used, max_delack was set to 8. In Figure 22, we report the throughputs of the forward and reverse transfers computed over the time during which both transfers were active.

The trends we observe are in conformance with the simulation results reported in Section 7.1.3. With standard FIFO scheduling on the reverse channel, the large data packets of the reverse transfer tend to starve out the acks of the forward transfer, resulting in poor forward throughput (0.35 Mbps with Reno, 0.16 Mbps with ACC and 0.09 Mbps with AF). The decreased frequency of acks with ACC and AF increases the chances that the sender of the forward transfer stalls waiting for acks to arrive, which explains the worse forward throughput in these cases as compared to Reno. For the same reason, the reverse throughputs show the opposite trend, with a throughput with 10% of the 28.8 Kbps link speed in the cases of ACC and AF.

Using acks-first scheduling on the reverse channel helps improve forward throughput substantially. However, acks-first together with standard TCP Reno hurts reverse throughput significantly (bringing it down to 5.1 Kbps). When acks-first is used in conjunction with a forward transfer that does ack filtering, the reverse transfer is completely shut out, achieving zero throughput whereas the forward transfer achieves a high throughput of nearly 4 Mbps (equivalent to the one-way transfer case). The reason for the poor performance of the reverse transfer in these two cases is that acks of the forward transfer are queued up in the reverse channel buffer most of the time. While such acks are present, data packets of the reverse transfer are not served at all because acks are given a higher priority. Even ack filtering, which allows at most one such ack to reside in the reverse buffer, does not help because so long as a new ack arrives before the current one has completed transmission, data packets of the reverse transfer do not get served.

When acks-first is used together with ACC, both the forward and reverse transfers achieve relatively good throughput (nearly 1.3 Mbps and 20.1 Kbps respectively). We can repeat the calculation done in Section 7.1.3 for the best possible forward throughput when the reverse throughput is close to the reverse link speed of 28.8 Kbps. With 150 KB socket buffers and 1424-byte segments, this works out to 2.75 Mbps. The throughput actually achieved (1.3 Mbps) is less than half this. We suspect that the shortfall is due to a combination of several factors, including the time it takes the forward connection’s congestion window to grow up to a large enough value and the non-negligible transmission time for acks. We still investigating this matter to determine the reason(s) for sure.

10. Conclusions

In this paper, we investigated the effects on network asymmetry on TCP performance in the context of wide-area wireless networks. We studied the impact of the reverse path, used primarily for acknowledgments and data requests, on end-to-end performance in the forward direction. We distinguished between bandwidth asymmetry, latency and media-access asymmetry, and loss asymmetry, and focused on the first two types.

Our methodology has involved three major steps. First, we measured the various asymmetric networks, including the Hybrid wireless cable network and the Ricochet packet radio network, to identify performance bottlenecks. Then, we modeled these networks in the ns simulator and experimented with several techniques to improve performance. Finally, we implemented the promising techniques in our experimental testbed and confirmed the performance trends observed in the simulations.

The following are our main results:

- SLIP header compression [13] alleviates some of the problems due to bandwidth asymmetry, but does not completely eliminate all problems, especially those that arise in the presence of bidirectional traffic.
- Connections traversing packet radio networks suffer from large variations in round-trip times caused by the half-duplex nature of the radios and asymmetries in the media-access protocol, which lead to variable latencies. This adversely affects TCP's loss recovery mechanism and results in degraded performance.
- The various end-to-end and router-based techniques that we propose help improve performance significantly in many asymmetric situations. We have verified this both via simulations and via experiments on the real testbed. These include decreasing the frequency of acks on the constrained reverse channel (*ack congestion control* and *ack filtering*), reducing source burstiness when acks are infrequent (*TCP sender adaptation* and *ack reconstruction),* and scheduling data and acks intelligently at the reverse bottleneck router (*acks-first scheduling*).
- In addition to improving throughput for individual connections, our proposed modifications also help improve the fairness and scaling properties when several connections contend for scarce resources in the network. We have demonstrated this via simulations of bulk and Web-like transfers.

Finally, we note that asymmetric networks are becoming increasingly important both in the mass market (e.g., Internet access to the home via cable modems) and in specialized situations (e.g., battlefield communications via mobile ad hoc networks). Therefore, we believe that the findings reported in this paper are relevant in a wide context.

11. Future Work

There are several areas of future work that we plan to investigate.

- The asymmetry in loss and error rates, especially in the context of wireless return channels, poses new challenges. Current packet radio networks use link-layer protocols for local error recovery, but this results in increased latency and variability in latency. We plan to extend Explicit Loss Notification (ELN) schemes proposed in the context of single-hop cellular networks [3] to multi-hop wireless networks, with the goal of reducing variability without sacrificing local error recovery.
- Cellular Digital Packet Data (CDPD) networks exhibit media-access asymmetry. Communication from the base station to the end stations is unimpeded, but end stations contend with each other for channel access in the reverse direction. It will be informative and useful to study the impact of this asymmetry on reliable transport performance.

12. Acknowledgments

We are grateful to several people for their help in setting up and debugging various networks in our experimental testbed: Mike Ritter, Mike Cunningham, Bob Luxemburg, Will SanFilippo, David Paulsen, and Sheela Rayala (Metricom, Inc.); Ed Moura (Hybrid Networks, Inc.); Andrew Nestor (MetroNet, Inc.); and Ken Lutz, Steve Hawes, and Fred Archibald (UC Berkeley). We appreciate their time and assistance.

We thank Mary Baker, Sally Floyd, Tom Henderson, Mike Ritter, Srinivasan Seshan, Andrew Swan, and the anonymous Mobicom reviewers for several comments and suggestions that helped improve the quality of this paper. Our special thanks to Giao Nguyen for his contributions to ns, which greatly facilitated our work.

This work was supported by DARPA contract DAAB07-95-C-D154, by the State of California under the MICRO program, and by the Hughes Aircraft Corporation, Metricom, Fuji Xerox, Daimler-Benz, Hybrid Networks, and IBM. Hari is partially supported by a research grant from the Okawa Foundation.

13. References

[1] A. Bakre and B. R. Badrinath. Handoff and System Support for Indirect TCP/IP. In *Proc. Second Usenix Symp. on Mobile and Location-Independent Computing*, April 1995.

[2] H. Balakrishnan. *Challenges to Reliable Data Transport over Heterogeneous Wireless Networks*. PhD thesis, University of California at Berkeley, August 1998. Also available as Technical Report UCB/CSD-98-1010.

[3] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R.H. Katz. Comparing the Performance of Transport Protocols in Wireless Networks. In *Proc. ACM SIGCOMM '96*, August 1996.

[4] H. Balakrishnan, S. Seshan, and R.H. Katz. Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks. *ACM Wireless Networks*, 1(4), December 1995.

[5] Berkeley Software Design, Inc. http://www.bsdi.com.

[6] R. Caceres and L. Iftode. Improving the Performance of Reliable Transport Protocols in Mobile Computing Environments. *IEEE Journal on Selected Areas in Communications*, 13(5), June 1995.

[7] S. Cheshire and M. Baker. A Wireless Network in MosquitoNet. *IEEE Micro*, Feb 1996.

[8] D-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.

[9] R. Durst, G. Miller, and E. Travis. TCP Extensions for Space Communications. In *Proc. ACM Mobicom Conference*, November 1996.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC, Jan 1997. RFC-2068.

[11] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[12] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM 88*, August 1988.

[13] V. Jacobson. *Compressing TCP/IP Headers for Low-Speed Serial Links*, February 1990. RFC 1144.

[14] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, 1991.

[15] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan. Performance of Two-Way TCP Traffic over Asymmetric Access Links. In *Proc. Interop '97 Engineers' Conference*, May 1997.

[16] P. Karn. MACA – A New Channel Access Method for Packet Radio. In *Proc. 9th ARRL Computer Networking Conference*, 1990.

[17] P. Karn. Dropping TCP acks. Mail to the end-to-end mailing list, February 1996.

[18] T. V. Lakshman, U. Madhow, and B. Suter. Window-based Error Recovery and Flow Control with a Slow Acknowledgement Channel: A study of TCP/IP Performance. In *Proc. Infocom 97*, April 1997.

[19] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, Reading, MA, 1996.

[20] UCB/LBNL/VINT Network Simulator - ns (version 2). http://www-mash.cs.berkeley.edu/ns/.

[21] V. N. Padmanabhan. *Addressing the Challenges of Web Data Transport*. PhD thesis, University of California at Berkeley, September 1998. Also available as Technical Report UCB/CSD-98-1016.

[22] V. N. Padmanabhan, H. Balakrishnan, K. Sklower, E. Amir, and R. H. Katz. Networking Using Direct Broadcast Satellite. In *Proc. Workshop on Satellite-Based Information Systems*, November 1996.

[23] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proc. Second International World Wide Web Conference*, October 1994.

[24] L. Zhang, S. Shenker, and D. D. Clark. Observations and Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proc. ACM SIGCOMM '91*, pages 133–147, 1991.

Author Biographies

Hari Balakrishnan is an Assistant Professor of EECS and a member of the Laboratory for Computer Science at the Massachusetts Institute of Technology. He received his M.S. and Ph.D. degrees in Computer Science from the University of California at Berkeley in 1995 and 1998 respectively, and a B. Tech. in Computer Science and Engineering from the Indian Institute of Technology (Madras) in 1993.

Hari's research interests are in the areas of computer networks and protocol architectures, wireless networks and mobile systems, adaptive network applications, and large-scale communication systems. He has received awards for papers at the ACM/IEEE Mobicom Conference and the Usenix Technical Conference. He is also a recipient of the C. V. Ramamoorthy award for distinguished graduate research at Berkeley and the winner of a Segasoft research grant from the Okawa Foundation. He is a member of the ACM and the IEEE. His email address is hari@lcs.mit.edu and his WWW URL is http://www.sds.lcs.mit.edu/~hari

Venkata N. Padmanabhan (ACM '94, IEEE '94) is a Researcher in the Systems and Networking group at Microsoft Research. He received his B.Tech. degree in Computer Science and Engineering at the Indian Institute of Technology, Delhi in 1993, and his M.S. and Ph.D. degrees in Computer Science at the University of California at Berkeley in 1995 and 1998, respectively.

Venkat's research interests are in the areas of Computer Networks, Mobile Computing, and Operating Systems. The focus of his Ph.D. dissertation was on developing network protocols for efficient Web access and effective data transport over asymmetric networks. His work on persistent-connection HTTP has been adopted by the HTTP/1.1 standard. Among the awards he has received are the National Talent Scholarship in India, the Charles Fish fellowship at Berkeley, and the best student paper award at the Usenix '95 conference. Venkat may be reached via e-mail at padmanab@microsoft.com and on the Web at http://www.research.microsoft.com/~padmanab.

Randy H. Katz (ACM F '96, IEEE F '96) is a professor of computer science at the University of California at Berkeley, and is a principal investigator in the Bay Area Research Wireless Access Network (BARWAN) project. He has taught at Berkeley since 1983, with the exception of 1993 and 1994 when he was a program manager and deputy director of the Computing Systems Technology Office at the Defense Department's Advanced Research Projects Agency. He has written over 130 technical publications on CAD, database management, multiprocessor architectures, high performance storage systems, video server architectures, and computer networks.

Dr. Katz received a B.S. degree at Cornell University, and an M.S. and a Ph.D. at the University of California at Berkeley, all in computer science. His e-mail address is randy@cs.berkeley.edu and his WWW home page is http://www.cs.berkeley.edu/~randy.