*"What immortal hand or eye
Could frame that fearful symmetry?"*

— William Blake, 'The Tiger'

Chapter 6

Effects of Asymmetry

This chapter investigates the problems to TCP performance that arise because of asymmetric effects. These problems arise in several wireless networks including bandwidth-asymmetric networks and latency-asymmetric packet radio networks, for different underlying reasons. However, the effect on TCP performance is the same in both cases: performance degrades significantly because of imperfections and variabilities in the acknowledgment (ACK) feedback from the receiver to the sender. We investigate these problems in both bandwidth-asymmetric and packet radio networks and solve them. Our solutions to these problems use a combination of link-layer and end-to-end algorithms in a two-pronged approach: first, we devise techniques to reduce the frequency of TCP ACKs; then, we develop techniques to handle reduced ACK frequency.

6.1 Motivation

Our work is motivated by two very different network technologies: bandwidth-asymmetric networks like wireless cable modem networks and direct broadcast satellite networks on the one hand, and multihop wireless packet radio networks on the other. In the first case, the reverse path used by TCP ACKs is constrained in bandwidth, with slow or loss-prone ACK feedback degrading TCP performance in the forward direction. In packet radio networks, traffic flowing simultaneously in different directions adversely affects performance. This is because most packet radio networks use half-duplex radio units, which cannot transmit and receive data frames at the same time. The interactions between TCP and the media-access protocols in these networks cause significant ACK queues to build up, leading to highly variable communication latencies and round-trip times. TCP does not adapt well to this variability.

Despite the technological differences between asymmetric-bandwidth and packet radio networks, TCP performance suffers in both these networks because of slow, imperfect and variable ACK feedback. We address this root problem and solve it in general. The end result is a set of solutions that improve performance in different asymmetric situations, independent of the specific technology in question.

Asymmetry is inherent in the design of many wireless networks. For example, many systems have a central transmitter (or base station) transmitting at high power to receiving mobile units. However, to reduce power consumption, the portable units transmit to the base station at much lower power levels. In addition, they often have to contend with other mobile units to gain access to the channel. Thus, while packets from the base station reach the mobile hosts promptly, the converse is not always true.

Asymmetric network technologies are common in mobile ad hoc environments too. For example, consider an ad hoc network of mobile hosts communicating with each other internally via a packet radio network, whose external link to the Internet is over an asymmetric-bandwidth wireless cable modem network. Motivated by this scenario, we combine these two wireless access technologies in our study—wireless cable and packet radio—to investigate the problems to transport performance that then arise.

We generalize the various phenomena and examples described above to the following general, technology-independent definition of asymmetry: *a network is said to exhibit 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 also depends significantly on those of the reverse direction*. Here, *forward* refers to the direction from the sender to the receiver and *reverse* the opposite. This general definition immediately permits a classification of different types of asymmetry. In addition to *bandwidth asymmetry* described above, this definition extends to *latency*, *media-access* and *error-rate* asymmetries.

In this chapter, we study bandwidth and latency asymmetry, both in isolation and in combination, using Hybrid's wireless cable modem network and Metricom's Ricochet packet radio network as motivating technologies. We use measurements on a real testbed as well as simulations experiments with different topologies and workloads to identify performance problems. Based on these results, we design and evaluate several techniques to improve performance.

Fundamentally, network asymmetry affects the performance of reliable transport protocols such as TCP because these protocols rely on feedback in the form of ACKs from the receiver to ensure reliability and smooth transmissions. Because TCP transmissions are ACK-clocked (Section 2.2.3), any disruption in the feedback process can impair the performance of the forward data transfer. For example, a low bandwidth ACK path could significantly impede the growth of the TCP sender window during slow start, *independent* of the link bandwidth in the direction of data transfer. A second example occurs in packet radio networks, where variable latencies in the presence of bidirectional traffic (caused by ACKs 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, significantly impairing prompt loss recovery.

We investigate the problems caused by asymmetry on TCP performance and design solutions to overcome these problems. These include both end-to-end and link-layer techniques such as *ACK filtering*, *TCP sender adaptation,* and *ACK reconstruction*. These solutions improve individual connection throughputs, distribute throughputs fairly amongst connections and improve the scaling properties of asymmetric systems by being able to support more connections for a fixed available bandwidth.

6.2 Asymmetric Bandwidth Networks

In this section, we discuss the problems that degrade unidirectional transfer performance in bandwidth asymmetric networks, shown in Figure 6.1. Depending on the characteristics of the reverse path, two types of situations arise for unidirectional traffic over asymmetric-bandwidth networks:

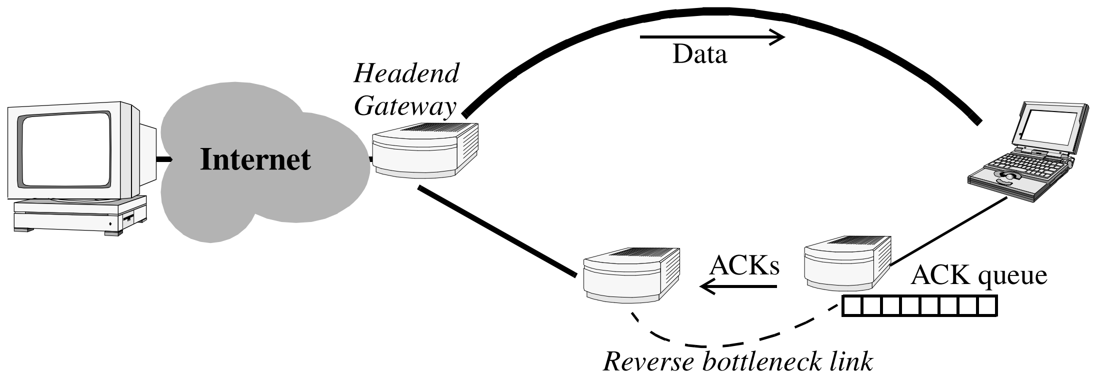

Figure 6.1 Schematic representation of a bandwidth-asymmetric network. If the reverse bottleneck queue is large, ACKs are queued there and reach the sender slowly, which reduces the forward throughput.

when the reverse bottleneck link has sufficient queueing to prevent packet (ACK) losses, and when the reverse bottleneck link has a small buffer. We consider each situation in turn.

6.2.1 Deep Reverse Queues

If the reverse bottleneck link has deep queues (Figure 6.1) so that ACKs do not get dropped on the reverse path, then performance is a strong function of the *normalized bandwidth ratio*, k, defined by Lakshman and Madhow [78]. k is 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 50 Kbps reverse channel, the raw bandwidth ratio is 200. With 1000-byte data packets and 40-byte ACKs, the ratio of the packet sizes is 25. This implies that k is 200/25 = 8. Thus, if the receiver acknowledges more frequently than one ACK every $k = 8$ data packets, the reverse bottleneck link will get saturated before the forward bottleneck link does, limiting the throughput in the forward direction.

If $k > 1$ and ACKs are not delayed or dropped, TCP ACK-clocking breaks down. Consider two data packets transmitted by the sender in quick succession. En route 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 temporal spacing all the way back to the sender, enabling it to transmit new data packets that maintain the same spacing. However, the limited reverse bandwidth and queuing at the reverse bottleneck router alters the inter-ACK spacing observed at the sender. When ACKs arrive at the bottleneck link in the reverse direction at a faster rate than the link can support, they get queued behind one another. The spacing between them when they emerge from the link is *dilated* with respect to their original

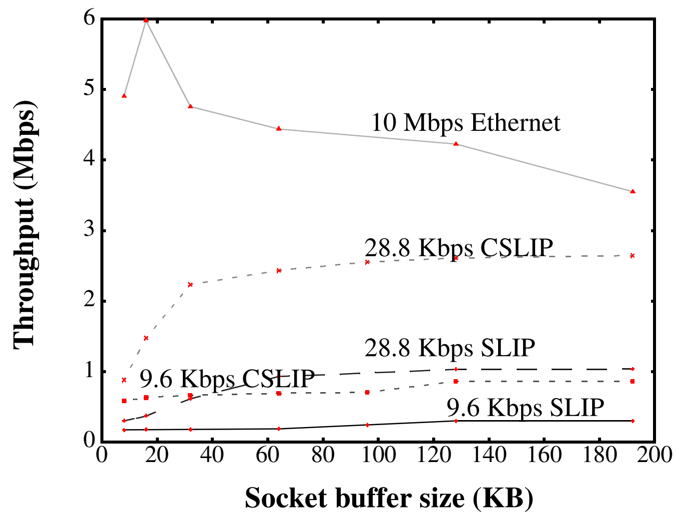

Figure 6.2 Measured performance of the Hybrid wireless cable network using different return channels, across a range of socket buffer sizes. Each run of the experiment involved the transfer of 1 MB of data in the forward direction between two BSD/OS hosts.

spacing, and is a function of the reverse bottleneck bandwidth. Thus the sender clocks out new data at a slower rate than if there had been no queuing of ACKs. No longer is the performance of the connection dependent on the forward bottleneck link alone; instead, it is throttled by the rate of arriving ACKs. As a side-effect, the sender's rate of congestion window growth slows down too.

Figure 6.2 shows the measured performance of a bulk TCP transfer over an asymmetric wireless cable networks using different reverse ACK channels, over reverse bottleneck links that have sufficient queueing. For each return bandwidth, SLIP header compression (CSLIP) [60] reduces the sizes of ACKs and reduces k, thus improving performance over the uncompressed case. 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 uncompressed SLIP and 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 6.2. Thus, for unidirectional traffic, CSLIP results in significant improvements in performance. In comparison, the performance with an Ethernet return channel is much better because of the absence of bandwidth asymmetry ($k = 0.052 << 1$) and a much smaller link delay than the dialup lines.

6.2.2 Shallow Reverse Queues

The second situation arises when the reverse bottleneck link has a relatively small amount of buffer space to accommodate ACKs. As the transmission window grows, this queue fills and ACKs are dropped. If the receiver acknowledges every packet, only one of every *k* ACKs get through to the sender, and the remaining *(k-1)* are dropped due to buffer overflow at the reverse channel buffer (here *k* is the normalized bandwidth ratio). In this case, the reverse bottleneck link capacity and slow ACK arrival are not *directly* responsible for any degraded performance. However, there are three important reasons for degraded performance in this case because ACKs are infrequent:

1. First, the sender transmits data in large bursts. If the sender receives only one ACK in *k*, it transmits data in bursts of *k* (or more) segments because each ACK shifts the sliding window by at least *k* (acknowledged) segments. This increases the likelihood of data loss along the forward path especially when *k* is large, because routers do not handle large bursts of packets well.

2. Second, TCP sender implementations increase their congestion window by counting the *number* of ACKs they receive and not on *how much* data is actually acknowledged by each ACK. Thus fewer ACKs implies a slower rate of growth of the congestion window, which degrades performance over long-delay connections.

3. Third, the sender's fast retransmission and recovery algorithms are less effective when ACKs are lost. The sender may not receive the threshold number of duplicate ACKs even if the receiver transmits more than the required number. Furthermore, the sender may not receive enough duplicate ACKs to adequately inflate its window during fast recovery.

In summary, bandwidth asymmetric networks suffer from degraded performance due to slow or imperfect ACK feedback.

6.3 Latency Asymmetry in Packet Radio Networks

We now look at the subtle effects that manifest themselves when TCP connections run over a packet radio network. While we use a network modeled after Metricom's Ricochet network in this section, our general results and conclusions are applicable to other packet radio networks as well. We start by describing the underlying packet radio network technology.

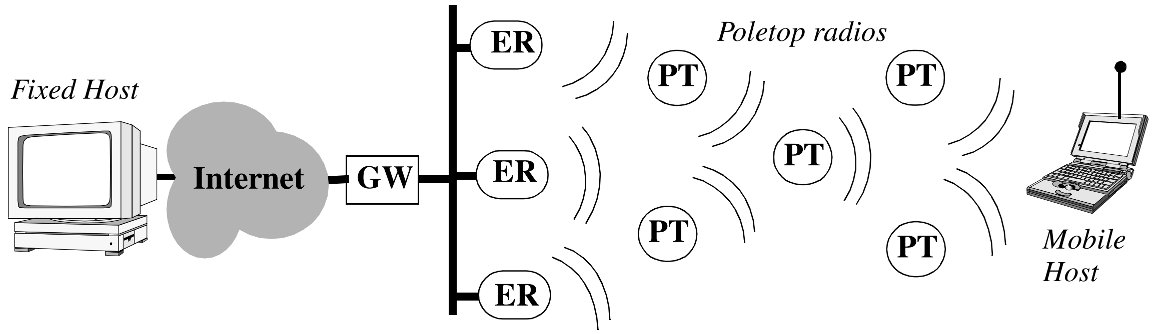

Figure 6.3 Topology of the Ricochet packet radio network. The Mobile Host (MH) has a modem attached to it, which communicates with a Fixed Host (FH) on the Internet through the Poletop Radios (PT) and Ethernet Radios (ER). The Gateway (GW) routes packets between the packet radio network and the Internet.

6.3.1 Technology

In this section, we describe the network topology, the characteristics of the radio units, and the media-access (MAC) and link-layer protocols in the packet radio network that displays latency asymmetry.

Topology: The network topology of the packet radio network is shown in Figure 6.3. The maximum link speed between any two nodes in the wireless cloud is 100 Kbps. Packets from a fixed host (FH) on the Internet are routed via a gateway (GW) and through the poletop radios (PT), to the end mobile host (MH). Similarly, packets from the MH are routed in the opposite direction. The Ethernet Radios (ER) interface between the wired Internet and the wireless links. Typically, there are between one and three wireless hops before the packet reaches the MH from the GW and vice versa.

Half-duplex radios: Because the power of a transmission drowns incoming receptions in the same frequency band, the radio units in the network cannot simultaneously transmit and receive data—i.e., they are *half-duplex.* After transmitting (receiving) a packet, a sender (receiver) wishing to receive (transmit) a packet needs to *turnaround* and change modes. Moving from transmitting to receiving mode takes a non-trivial amount of time, called the transmit-to-receive turn-around time, T_{TR}. Similarly, going from receiving to transmitting mode takes a time equal to the receive-to-transmit turnaround time, T_{RT}. Typical values of these times are shown in Table 6.2 on page 153.

Media-access: The radios in this network are frequency-hopping, spread-spectrum units operating in the 915 MHz ISM band. The details of the frequency-hopping protocol are not relevant to transport performance, since the predominant reason for poor performance is the interaction between the MAC protocol and TCP. The MAC protocol for contention resolution is based on a polling scheme, similar (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 and is willing to receive this transmission, 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 reschedule the 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 peers in the network.

Error-control: The reliable link-layer protocol used in this network for error-control is a simple frame-by-frame protocol with a window size of one. 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 it after a timeout interval. Such simple link-layer protocols are the norm in several packet radio networks (see, e.g., [68]).

6.3.2 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.

Variable delays and ACK queueing: The need for the communicating peers to first synchronize via the RTS/CTS protocol before communication and the significant turn-around time for the radios result in a high per-packet overhead. Furthermore, since the RTS/CTS exchange needs to

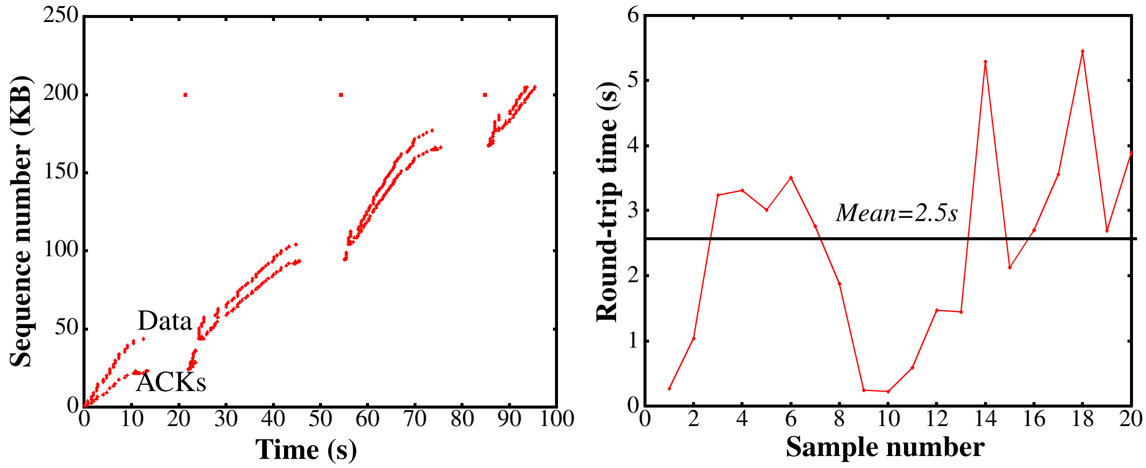

Figure 6.4 (a) Packet and ACK sequence trace of a 200 KB TCP bulk transfer measured over one wireless hop in the Ricochet network. The three pauses are sender timeouts, lasting between 9 and 12 seconds each. This happens because the retransmission timeout estimate is large. (b) Twenty successive round-trip 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. The combination of the 2.5 s mean round-trip and large deviation leads to long timeouts.

back off when the polled radio is busy (for example, engaged in a conversation with a different peer), this overhead is variable. This leads to large and variable communication latencies in packet-radio networks. In addition, with an asymmetric workload with most data flowing in one direction to clients, ACKs tend to get queued in certain radio units (especially in the client modems), exacerbating the variable communication latencies.

These variable latencies and queueing of ACKs adversely affect smooth data flow. In particular, TCP ACK traffic interferes with the flow of data and increases the traffic load on the system. Figure 6.4(a) shows the packet sequence trace of a measured 200 KB TCP transfer over an unloaded wired path and one wireless hop in the Ricochet network. This clearly shows the effect of the radio turnarounds and increased variability affecting performance. The connection progresses well for the most part, except for the three timeouts that occur during the transfer. These timeouts last between 9 and 12 seconds each, keeping the connection idle for a total of 35% of the total transfer time! These timeouts clearly under-utilize the available bandwidth for the connection.

Why are these timeouts so long in duration? Ideally, the round-trip time estimate (*srtt*) of a TCP data transfer will be relatively constant (i.e., have a low linear deviation, *rttvar*). Then the TCP retransmission timeout, set to *srtt + 4*rttvar* (Section 2.2.2), will track the smoothed round-trip time estimate and respond well when multiple losses occur in a window. Unfortunately, this is not true for connections in this network, as shown in Figure 6.4(b). This figure plots the individual round-trip time estimates for 20 successive samples during the same connection over the Ricochet network. The mean value of these samples is about 2.5 seconds and the standard deviation is large—about 1.5 seconds. Because of the high variation in the individual samples, the retransmission timer is on the order of 10 seconds, leading to the long idle timeout 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. If only the mean or median round-trip estimates were taken into account, the potential for spurious retransmissions of segments still in transit is large. Thus, our challenge is to devise solutions to this problem where the increased ACK flow and ACK queueing lead to variable latencies and long timeout periods.

An optimization to the link-layer error-control protocol that piggybacks link-layer ACKs with data reduces TCP round-trip variations to some extent. This scheme is motivated by the observation that the radios turn-around both for data frames as well as for link-layer ACKs. The presence of traffic in both directions, even when caused by TCP ACKs, already causes turnarounds to happen. So, if link-layer ACKs were piggybacked with data frames, some extra radio turnarounds can be eliminated. Recent releases of the radio software in the Ricochet network attempt this optimization whenever possible.

However, despite this optimization the fundamental problem of additional traffic and the underlying MAC protocols affecting round-trip time estimates and causing variabilities in performance still persists. Connections traversing multiple wireless hops are especially vulnerable to this effect, because it is now more likely that the radio units may already be engaged in conversation with other peers. Figure 6.5 shows this by plotting the round-trip samples and TCP estimates of the smoothed round-trip estimate and mean linear deviation during a bulk 1 MB TCP transfer in single-hop wireless network with this optimization (the best-case topology for the optimization). The average smoothed estimate maintained by TCP is about 2.2 seconds and the linear deviation is about 1 second, still rather large.

Our simulations and results (Section 6.6) of the multi-hop wireless network assume that the radio units perform this optimization and piggyback link-layer ACKs with data whenever possible. We

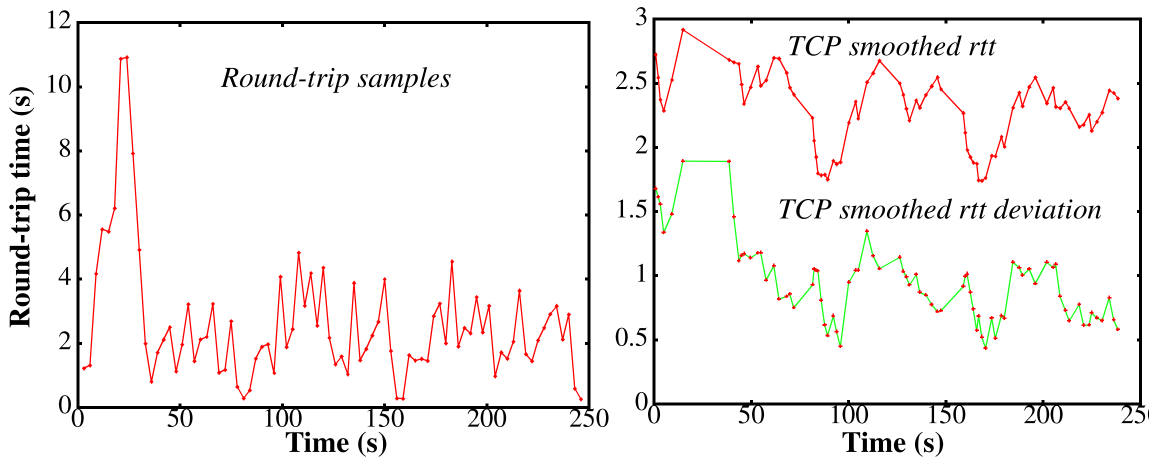

Figure 6.5 Round-trip time estimate for a 1 MB bulk transfer in a one-hop Ricochet wireless network using an optimized link-layer protocol that piggybacks link-layer ACKs with data. The figure on the left shows the raw round-trip samples, while the figure on the right shows the TCP smoothed round-trip estimate and mean linear deviation using the EWMA filters discussed in Section 2.2.2. The fundamental problems with large round-trip time variations still persist despite the piggybacking optimization.

also note that the basic reliable link-layer protocols in many packet radio networks do not piggyback ACKs with data.

6.4 Solutions

Our solutions to the problems described in the previous two sections are inspired by the following observation. In all these networks, regardless of technology, the root problems arise because of imperfections, slowness and variability in the ACK feedback. This immediately suggests the first prong of our two-pronged solution: *reduce the frequency of ACKs* because of the unfavorable interactions that arise when there are too many ACKs in the network. However, TCP does not handle infrequent ACKs well, as explained in Section 6.3.2. This motivates the second prong of our solution: devise techniques to *handle infrequent ACK feedback.* Together, these techniques lead to significant improvements in performance over different asymmetric networks.

We use a link-layer technique called *ACK filtering* to reduce ACK frequency. To handle infrequent ACKs, we develop two techniques: *sender adaptation,* an end-to-end scheme and *ACK reconstruction,* a link-layer scheme. In our experiments, we also evaluate the performance of ACK congestion control (ACC), described in Section 2.6.4.

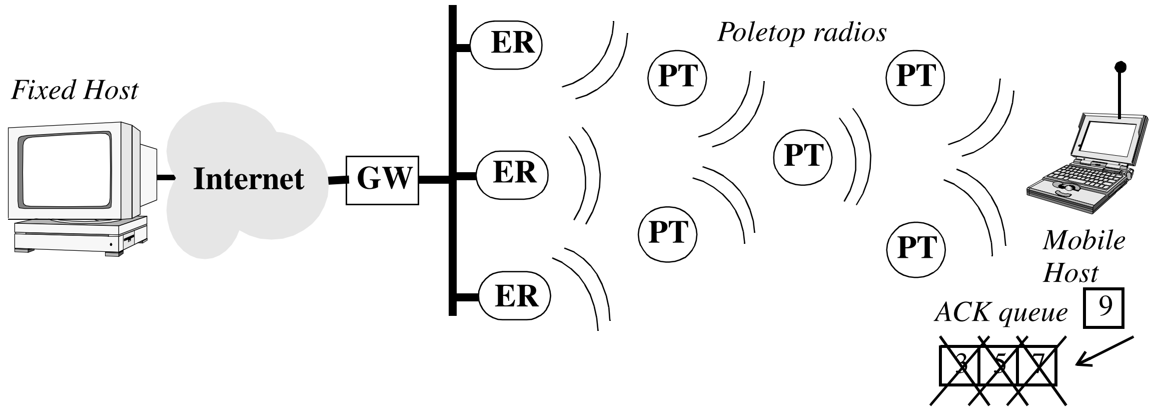

Figure 6.6 ACK filtering in the packet radio network. Both the MH's portable modem and the poletop radios implement AF. When cumulative ACK 9 is about to be enqueued, previously enqueued ACKs 3, 5 and 7 are purged without affecting the semantics of ACKs.

6.4.1 ACK Filtering (AF)

AF, shown in Figure 6.6, is a TCP-aware link-layer technique that reduces the number of TCP ACKs sent on the reverse channel. The challenge is to ensure that the sender does not stall waiting for ACKs, which can happen if ACKs are indiscriminately removed on the reverse path. AF removes only certain ACKs without starving the sender by taking advantage of the fact that TCP ACKs are cumulative. As far as the sender's error control mechanism is concerned, the information contained in an ACK with a later sequence number subsumes the information contained in any earlier ACK.

When an ACK from the receiver is about to be enqueued at a reverse direction router, the router or the end-host's link layer (if the host is directly connected to the constrained link) traverses its queues to check if any previous ACKs belonging to the same connection are already in the queue. If so, it removes them from the queue, reducing the number of ACKs that go back to the sender. The removal of these "redundant" ACKs frees up space for other data and ACK packets. In particular, it ensures that later ACKs do not get queued behind earlier ones waiting to be sent on the reverse path.

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 implicitly present in the packets in the queue.

In the experiments reported in this chapter, AF deterministically purges all preceding ACKs belonging to a connection whenever a new ACK for the same connection with a larger cumulative value enters the queue.

6.4.2 TCP Sender Adaptation (SA)

AF alleviates the problem of congestion on the reverse path by decreasing the frequency of ACKs, with each ACK potentially acknowledging several data packets. However, this is only one part of the complete solution: unilaterally reducing ACK frequency, while reducing the level of contention along the reverse path, actually leads to worse performance. In particular, as explained in Section 6.2, it causes three problems: the propensity of the sender to transmit large bursts of data, a slowdown in window growth, and reduced effectiveness of the fast retransmission algorithm. SA addresses these problems by modifying the TCP sender algorithm and making it adapt to infrequent ACKs.

First, it reduces the sender bursts 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, it schedules transmissions of data for later points in time based on its estimate of the connection's data rate. It estimates the data rate as the ratio *win*/*srtt*, where *win* is the current TCP window size and *srtt* is the smoothed RTT estimate. Thus, large bursts of data are broken up into smaller bursts spread out evenly over time.

Second, SA takes into account the amount of data acknowledged by each ACK, rather than the number of ACKs while incrementing the congestion window, *cwnd*. So, if an ACK acknowledges *s* segments, the window grows as if *s* separate ACKs had arrived. This solves the window growth problem.

Third, an SA-enhanced sender does not just count the number of duplicate ACKs, to account for their removal by an ACK filter. With AF, when the reverse channel router filters some of the enqueued duplicate ACKs, sets a field in the TCP header of the remaining ACK informing the

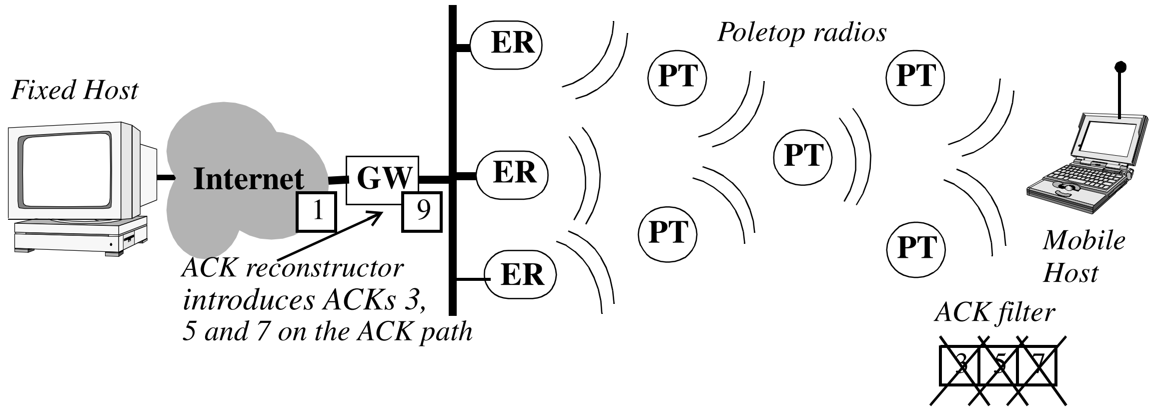

Figure 6.7 ACK reconstruction in the packet radio network, implemented at the Gateway node. When ACK 9 arrives, the previous ACK seen was 1. Rather than forward 9 at once, AR generates ACKs 3, 5 and 7 and sends them to the sender before forwarding 9.

sender of the number of purged duplicates. This helps the sender in the fast retransmission and recovery phases.

SA is generally applicable when the ACK path is loss-prone or if ACC (Section 2.6.4) is deployed in the system.

6.4.3 ACK Reconstruction (AR)

AR is a TCP-aware link-layer technique that reconstructs a sporadic ACK stream after it has traversed the reverse bottleneck. It prevents the reduced ACK frequency from adversely affecting the performance of standard TCP sender implementations on the Internet (i.e., those that do not implement SA). This enables us to use schemes such as AF (and ACC) without modifying TCP senders on the Internet. This solution can be easily deployed by Internet Service Providers (ISPs) of asymmetric access technologies in conjunction with AF, to achieve good performance without changing any TCP implementations. It is schematically shown for a packet radio network in Figure 6.7.

AR deploys a soft-state agent called the ACK reconstructor at the other end of the constrained ACK bottleneck. For example, it may be deployed at the gateway (GW in Figure 6.7) to the packet radio cloud. The reconstructor carefully fills in the gaps in the ACK sequence and introduces ACKs to smooth out the ACK stream seen by the sender. However, it does so without violating the end-to-end semantics of TCP ACKs, as explained below.

Suppose two ACKs, *a1* and *a2,* arrive at the reconstructor after traversing the constrained reverse link at times *t1* and *t2* respectively. Let $a2\text{-}a1 = \Delta a > 1$. If *a2* were to reach the sender soon after *a1* with no intervening ACKs, then at least Δa segments are sent by the sender in one burst (if the flow control window is large enough) and the congestion window increases by at most one, independent of Δa. AR remedies this problematic situation by interspersing ACKs to provide the sender with a larger number of ACKs at a consistent rate, which reduces bursts and allows the congestion window to increase faster.

How is this done? 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, when successive ACKs separated by Δa arrive at the reconstructor, it interposes $ceil(\Delta a/\delta a) - 2$ 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. To do this, the reconstructor measures the rate at which ACKs arrive at its input. 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 temporal spacing 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 the reconstructor sets δt equal to Δt, then the connection would 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 $\delta t = (\delta a/\Delta a)*\Delta t$. Therefore, the latest ACK in current sequence, *a2*, is held back for a time roughly equal to Δt, and $ceil(\Delta a/\delta a) - 2$ ACKs are evenly interposed in this time.

Thus, by carefully controlling the number and spacing between ACKs, standard TCP senders can be made to increase their congestion window at the right rate and avoid burst transmissions. 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 ACKs are fully preserved. The trade-off in AR is between obtaining smoother data transmissions, a better rate of congestion window increase and a reduction in the round-trip variation, versus a modest increase in the round-trip time estimate at the sender. We believe that this is a good trade-off in the asymmetric environments we are concerned with.

6.5 Implementation

In this section, we discuss the implementation details of AF and SA in the BSD/OS 3.0 system.

6.5.1 ACK Filtering

The goal of AF is to remove all redundant ACKs from the queue of the reverse channel when a newer cumulative ACK arrives there. We describe below our experiences with implementing this scheme, and also discuss some practical problems in the context of the packet radio network modems.

Since both the reverse connection for the bandwidth asymmetry case and the network connectivity for the latency asymmetry case use Hayes-compatible modems running the Point-to-Point Protocol (PPP) [60], we implemented this scheme in the PPP layer of the BSD/OS 3.0 kernel. When a new ACK arrives, we traverse 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 [60], 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 (the location of the ACK filter), which makes it expensive and hard to implement.

2. Explicitly maintain two queues at the PPP layer, one for header-compressed packets and the other for their uncompressed 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 (not just PPP), and more importantly, because we would need to constantly maintain the consistency between the two queue data structures.

3. Modify the PPP queue data structure to maintain not just pointers to kernel `mbufs` [97] (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 `mbufs`. This is the simplest efficient alternative and is the one we implemented.

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 adaptation implementation (Section 6.5.2) to recognize and react to this option.

When we implemented this mechanism and deployed it, we found that the queueing of ACKs occurs inside the modem and not in the kernel's PPP queue, with both the Ricochet and standard telephone modems. Thus, the effectiveness of the implementation in the PPP layer of the kernel is limited. We need to modify the modem firmware to implement AF or control queue lengths on the modem and cause queueing to happen in the kernel's PPP queue—we could not do this because we had no access to the modem's internal firmware.

Fortunately, since we have implemented accurate simulation models of the wireless cable and packet radio networks, we could perform detailed simulation experiments and obtain insights into the reasons for performance improvement with AF in these networks.

6.5.2 Sender Adaptation

The goal of SA is to enable the TCP sender to operate successfully even when the frequency of ACKs is reduced by AF or ACC. There are two aspects to sender adaptation:

1. Incrementing the congestion window based on the amount of data acknowledged, rather than on the count of ACKs.
2. Avoiding large bursts of data when each ACK acknowledges several data segments.

Our current implementation does not include the fast retransmission solution described in Section 6.4.2.

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 twice the maximum segment size, `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). Scaling down by `2*mss` makes it equivalent to TCP Reno with standard delayed ACKs, which involves one window increase operation for every two acknowledged data segments [20].

To avoid large bursts of data, the sender limits the maximum number of segment that can be transmitted in a single burst to a configurable parameter called `maxburst`. If 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 current window size to the round trip time. The other segments are transmitted clocked by this software timer, one segment at a time. We set `maxburst` to three in our implementation.

To compute the rate at which the software timer must fire, we need to measure the round trip time more accurately than is done using the timestamp option in standard TCP, which has a coarse 500 ms timer granularity. While a coarse granularity is acceptable for the retransmission timer, it leads to highly conservative behavior when used to rate control transmissions. We solve this problem by having the sender insert the current time that has a microsecond granularity in the timestamp option [61], which is then echoed by the receiver. Thus, every time an ACK arrives, we know the accurate time at which it was originally sent and can calculate an accurate difference to update `t_exact_srtt`, the “exact” round-trip time estimate. This is a 4-byte option; we do not run into wrap-around problems as long as the RTT is under 4000 seconds (many other things would break in TCP before this if the true round-trip were indeed that large!). The quantity `t_exact_srtt` thus computed is used in the rate calculation described above.

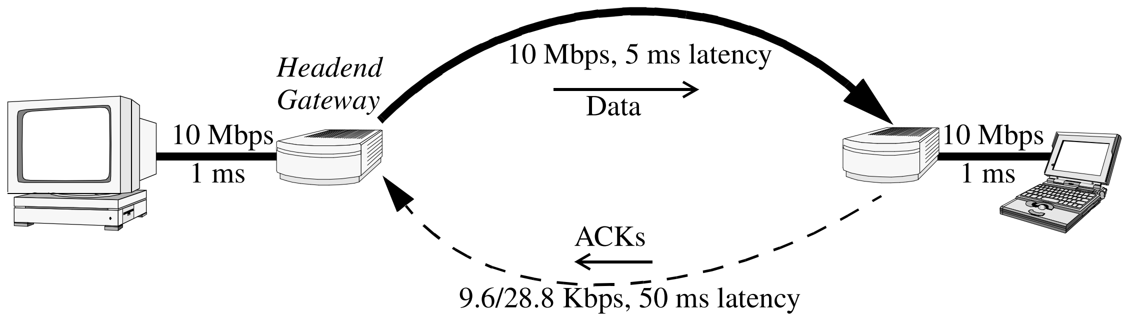

Figure 6.8 The simulation topology used to model a network with bandwidth asymmetry. The bandwidth and delay parameters have been chosen to closely model the Hybrid wireless cable modem network.

6.6 Simulation Results

In this section, we present and discuss the results of detailed *ns* simulations of bandwidth and latency asymmetry, modeled after the Hybrid wireless cable and Ricochet packet radio networks respectively.

6.6.1 Bandwidth Asymmetry

We first present the results of several simulations of unidirectional TCP transfers on a network that exhibits bandwidth asymmetry and discuss the reasons for observed performance.

The simulation topology we used to investigate the effects of bandwidth asymmetry is shown in Figure 6.8. 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 (50 ms). Although reverse channels ranging from slow to high speed dialup lines to ISDN have different latencies in reality, keeping the delay constant in the simulation experiments helps us focus on the bandwidth asymmetry alone.

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 no data losses occurred in the forward direction in most cases. The second set of experiments (B) was designed to investigate how the different protocols performed when losses occurred in the forward

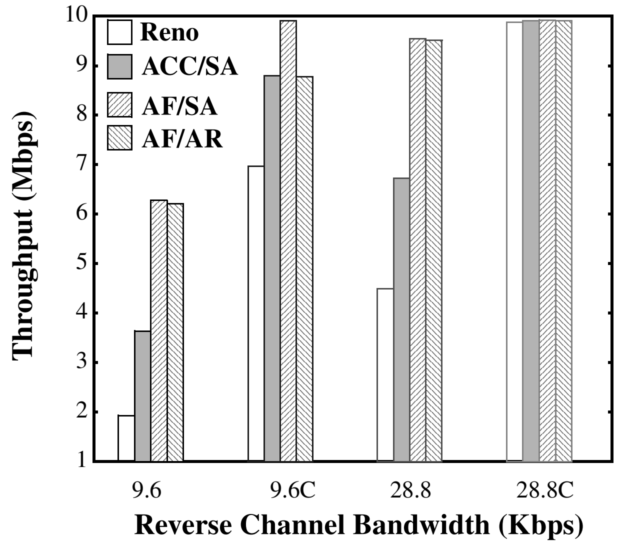

Figure 6.9 Throughputs from the simulation of a single one-way transfer with no data losses (experiment (A)). "C" indicates the use of SLIP header compression.

direction due to insufficient forward buffer capacity, and in particular to highlight the benefits of using SA or AR on a loss-prone forward path when the ACK stream is prone to losses.

6.6.1.1 Loss-free transfers

Figure 6.9 shows the throughputs obtained for four protocols—TCP Reno, Reno with AF/SA (i.e., AF with SA at the sender), Reno with AF/AR and Reno with ACC/SA—with different types of return channels with no data losses in the forward direction. 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 to model situations with and without header compression. The motivation is to understand how the bandwidth and queue length of the return channel affect forward throughput when there is no data loss.

Since the transfers are long, the reverse buffer fills early for TCP Reno. Beyond that point, only one ACK in k gets through on average, causing the sender to transmit bursts of k packets on average. As long as k does not exceed the bottleneck buffer size in the forward direction, burst transmissions from the sender do not lead to losses.

The normalized bandwidth ratio (Section 2.6.2 & Section 6.2.1) 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) because losses do indeed occur on the forward path. SA or AR used in conjunction with AF breaks up potential bursts, thereby avoiding performance degradation suffered by 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 uncompressed SLIP 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 gain, and in some cases (when there are no losses in the forward direction) it entirely eliminates the problems caused by bandwidth asymmetry.
2. AF and ACC lead to significant improvements when the reverse buffer is small, especially when k is large.
3. Large reverse buffers cause TCP Reno performance to severely degrade, and cause ACC to perform poorer than AF. This is because Reno now progresses at the rate at which ACKs leave the reverse queue, and ACC is benign until the number of reverse ACKs is a substantial fraction of the reverse queue.

6.6.1.2 Loss-prone 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, AF and ACC alter the inter-ACK spacing and how SA and AR prevent burst transmissions. Table 6.1 shows the results of these experiments for the different protocols. The maximum window size was set to 120 packets (to ensure that the receiver window is not the performance bottleneck) and all queue sizes were set to 10 packets. We used a 28.8 Kbps header-compressed reverse channel.

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

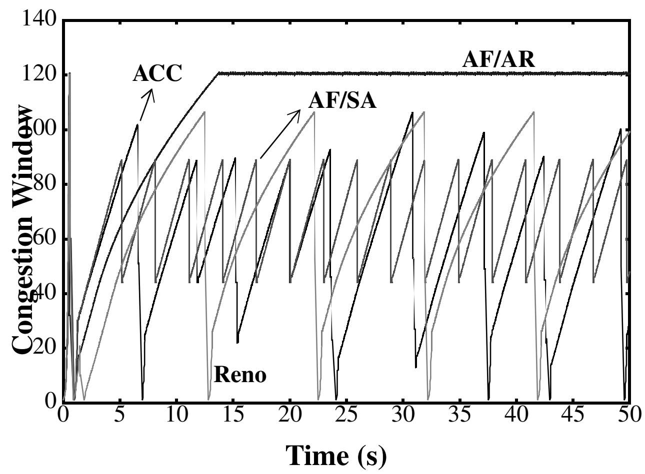

Figure 6.10 Congestion window evolution for the different schemes. AF/AR, AF/SA and ACC don't fluctuate as much as Reno, achieving better performance.

AF/AR and AF/SA perform the best, achieving throughputs between 15% and 21% better than Reno. The degree of burstiness reduces significantly and the reverse router queue is no longer perpetually full. This is shown in Figure 6.10, which depicts the time-evolution of congestion windows for the different protocols. Table 6.1 shows the time-averaged TCP congestion window and round-trip times for the different protocols. It is clear from these results that reducing the frequency of ACKs alone is not sufficient—AF alone performs even worse than Reno and techniques like SA or AR are essential to achieve good performance.

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 (Section 2.6.4). 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 loss-prone ACK stream.

6.6.2 Latency Asymmetry

Based on experimental measurements of the Ricochet network, we modeled the system in *ns* (Section 3.1). The simulation parameters used to obtain the results described in this section are shown in Table 6.2. 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. Link-layer retransmissions of corrupted packets only add to the latency 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 KB

Table 6.2 Simulation parameters of the multi-hop packet radio network. The number of wireless hops varies between one and three.

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 one and three, similar to the topology of the commercial Ricochet network. Then, we experiment with multiple simultaneous TCP transfers through the wireless cloud to investigate the issues of fairness and scale in this wireless network.

6.6.2.1 Single Bulk Transfers

The workload in these experiments consists of a 100-second TCP transfer, with no other competing traffic and a maximum receiver window size of 32 KB. The choice of long transfer time lets us investigate the steady state performance of the different schemes. Congestion losses occur as a result of buffer overflow and lead to sender timeouts if multiple packets are lost in a transmission window. We investigate the performance of TCP Reno, AF/SA and ACC/SA.

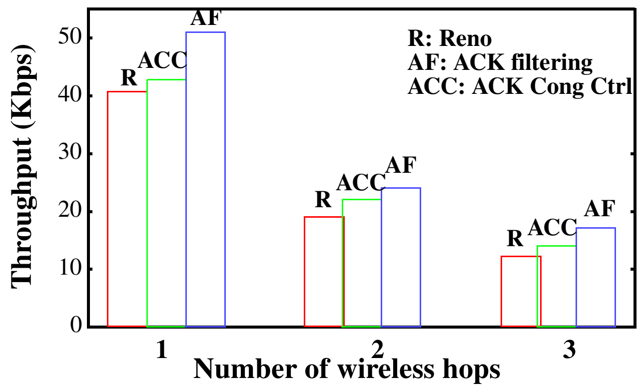

Figure 6.11 Simulated TCP throughputs of Reno, ACC and AF, as a function of the number of wireless hops.

Figure 6.11 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 6.3—the improvement in throughput varies from 25% (one wireless hop) to 41% (three wireless hops). Similar improvements are observed when AR is used at the gateway to the wireless cloud instead of SA at the sender.

# hops	Reno (Kbps)	ACC/SA (Kbps (%))	AF/SA (Kbps (%))
1	40.7	42.8 (5%)	51.0 (25%)
2	19.0	22.0 (16%)	24.0 (26%)
3	12.1	14.5 (20%)	17.1 (41%)

Table 6.3 Simulated performance of Reno, ACC/SA and AF/SA as a function of the number of wireless hops in the multihop wireless network. Throughputs are in Kbps; the numbers in parentheses are percentage improvements compared to Reno.

The main reasons for this improvement are the reduced number of packets and reduced round-trip variability of the two enhanced protocols compared to Reno. Figure 6.12 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 in the network. These factors lead to a 26% improvement in end-to-end throughput, from 19 Kbps (Reno) to 24 Kbps

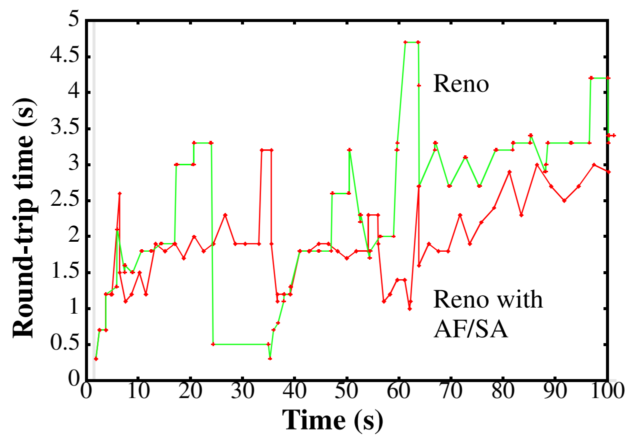

Figure 6.12 Round-trip times obtained from simulations of TCP Reno and TCP with AF/SA for a connection over two wireless hops. The round-trip times for the Reno connection are much more variable, with a mean of 2.67 s and a standard deviation of 1 s, whereas AF reduces the mean to 1.85 s and the standard deviation to 0.6 s.

over two 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 41%. While the exact values of the round-trip time and the deviation are a function of the window size and the amount of competing traffic, similar improvements 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 ACKs to a larger extent than ACC does.

6.6.2.2 Multiple Simultaneous Transfers

We now experiment with multiple simultaneous transfers in the multihop wireless network to understand performance as a function of the number of connections traversing the network. These concurrent connections compete for resources in the network. We focus on simulations of connections over one wireless hop in this section for Reno+AF/SA, the best performing protocol, 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,* as defined in Section 3.3. We are interested in obtaining large values of both the network utilization as well as the fairness index for any configuration.

# 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 6.4 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.

Table 6.4 shows the 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 is a measure of the degree of variation in the throughputs observed by the concurrent connections. The aggregate performance varies between 44 and 47 Kbps for TCP Reno and between 48 and 52 Kbps with AF, as the number of connections varies, implying that there is little change in utilization as a function of the number of connections for both protocols. However, the standard deviation of the throughputs of the concurrent connections is much higher for Reno than for AF/SA. The distribution of throughputs is more spread out and less equal across any set of connections, especially as the offered load on the network scales to larger numbers of connections.

We quantify this effect in Figure 6.13, where the fairness index (Section 3.3) of throughput is plotted as a function of the number of concurrent connections, n. 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 substantially improves the fairness index of the throughput distribution by over 25%, 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. The duration of TCP timeouts are smaller because of the reduced variation in round-trip times. Another

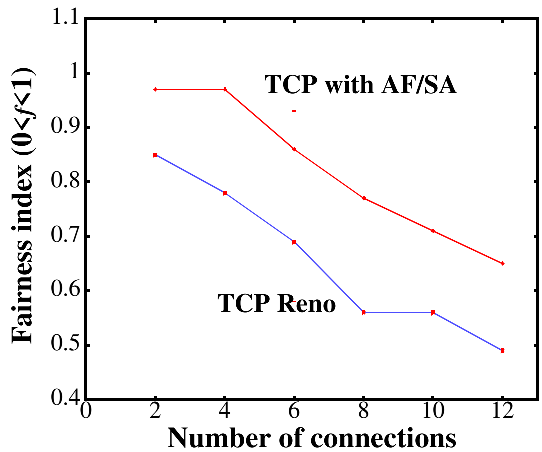

Figure 6.13 Simulation results for the fairness index of TCP Reno and TCP with AF, as a function of the number of connections traversing one hop of the packet radio network.

way of interpreting this result is that the performance of each TCP connection becomes more predictable.

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

6.6.3 Combining Bandwidth and Latency Asymmetry

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 important 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 6.14 shows the measured performance of 1 MB 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 measured performance (20 runs each). These controlled measurements were performed in the absence of any cross-traffic; the inherent variability of the return path manifests itself in the significant error-bars on the graph.

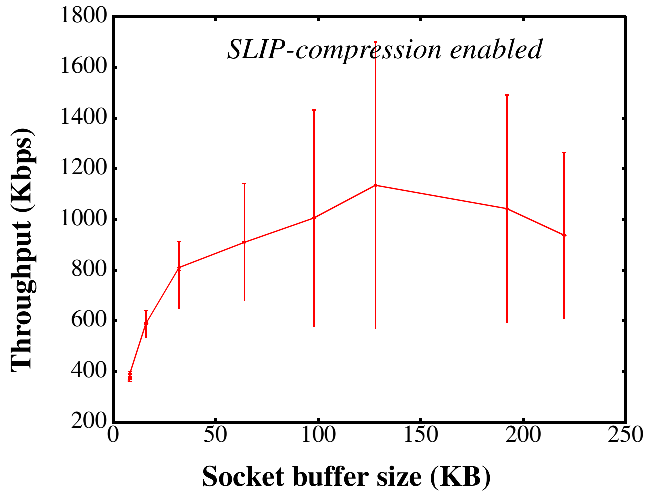

Figure 6.14 Measured performance of a 1 MB TCP transfer across a Hybrid™ wireless cable downlink and Ricochet return channel. The large errorbars (standard deviations) are a consequence of the high variability of the Ricochet return channel.

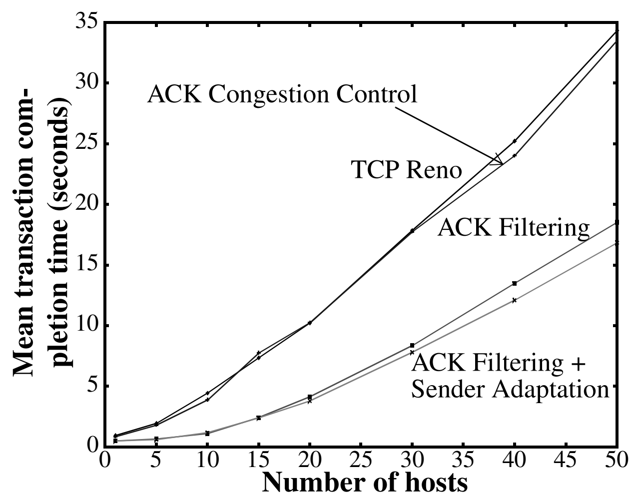

Figure 6.15 Simulated scaling behavior of *TCP Reno*, *AF*, and *ACC* as a function of the number of connections for a Web-like micro-benchmark over a packet radio return path. The graph shows average completion time vs. *n*, the number of hosts (4 concurrent connections per host). AF performs the best as *n* increases.

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 link-layer protocols to 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 four concurrent TCP transfers of 10 KB each to the client. This is not intended to be an accurate model of reality, but 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 affects performance in the packet radio network, we implicitly assume that the Web requests use pipelining [108], with one 500-byte request resulting in four 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 completion time for the entire Web transaction (i.e., all four concurrent connections).

Figure 6.15 shows the mean completion time for a transaction as the number of hosts varies for TCP Reno, Reno with AF and Reno with ACC. Two curves are shown for the AF case: with sender adaptation enabled and without.

These results indicate that AF significantly reduces the response time and improves the throughput of the network as the system scales. The reason ACC is not particularly beneficial here is the shorter transfer lengths in this workload compared to the bulk transfer experiments. No connection exceeds 10 packets in length, so the sender's window is never large. This limits the extent to which ACC can improve performance because it takes a longer duration to adapt to the state of congestion on the reverse path. The other reason is the larger number of ACKs traversing the network with ACC compared to AF. A critical factor in this network is the latency and variability associated with each packet on the wireless network. AF is a significant benefit because it purges *all* redundant ACKs from the queue independent of the state of congestion, thus reducing the number of packets traversing the wireless cloud.

The lack of sender adaptation does not significantly impair 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. Finally, we note that with persistent-connection HTTP [108], recommended by HTTP/1.1 [44], connections will tend to be longer. This is likely to help ACC perform better than it currently does for this workload.

6.7 Summary

In this chapter, we discussed the adverse effects of asymmetry on TCP performance. These problems arise in several wireless networks, including bandwidth-asymmetric networks and packet radio networks for different underlying reasons. However, the net effect on TCP performance is the same in both cases: performance degrades significantly because of imperfections and variabilities in the ACK feedback from the receiver to the sender. Our solutions to these problems use a combination of link-layer and end-to-end algorithms in a two-pronged approach: we devise techniques to reduce the frequency of TCP ACKs, and develop techniques to handle this reduced ACK frequency. Our detailed simulations show that these techniques significantly improve end-to-end performance in both bandwidth-asymmetric wireless cable and latency-asymmetric packet radio networks. ACK filtering combined with TCP sender adaptation improves the scaling properties of a network that includes both bandwidth and latency asymmetry.