There are more things in heaven and earth, Horatio,
Than are dreamt of in your philosophy.

— Hamlet [I. v. 166], William Shakespeare

Chapter 4

The Berkeley Snoop Protocol

This chapter describes the design and implementation of the *Berkeley Snoop protocol*, which significantly improves the end-to-end performance of TCP in error-prone cellular wireless networks. It achieves this by exploiting the idea of *cross-layer protocol optimizations* for enhanced end-to-end performance.

4.1 Introduction

In Chapter 2, we described the major problems that arise when a TCP connection traverses wireless links, communicating with hosts that are mobile. An important characteristic of wireless links is the occurrence of bit-errors due to physical and environmental factors. This is often coupled with periods of intermittent connectivity and packet losses that could occur during cellular handoffs, as described in Section 2.5. Protocols and applications have to adapt to these channel errors to achieve good performance.

In this chapter, we will be concerned primarily with cellular network topologies (Section 2.4.1). In particular, we will focus on topologies similar to the one shown in Figure 4.1, that are composed of base stations and single-hop wireless links. Here, an end-host is directly connected via a wireless

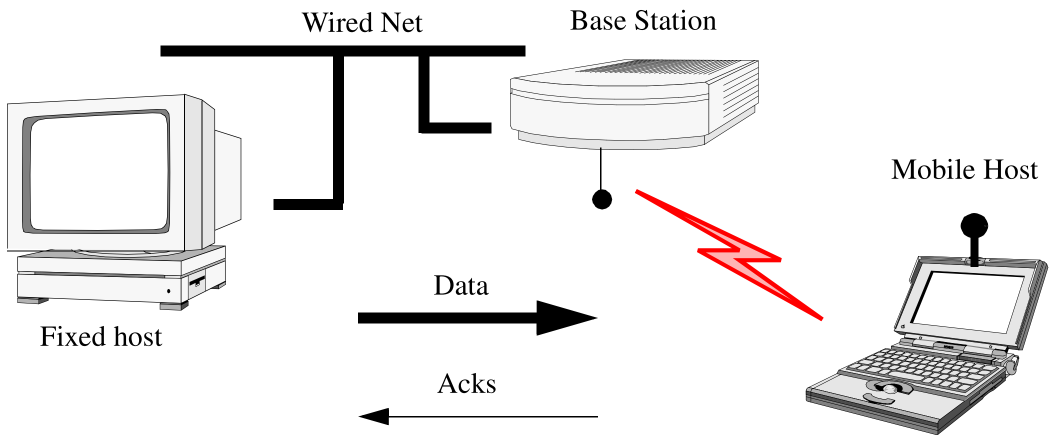

Figure 4.1 Wireless topology for the Snoop protocol.

link to a *base station (BS)*, which is then connected over a number of wired hops to the rest of the Internet. We call the end-host connected via the wireless link, the *mobile host (MH)*, and any host on the wired Internet communicating with it, the *fixed host (FH)*. Our goal is to achieve good end-to-end performance for reliable data transfers in both directions.

For data transfer *to* a mobile host, the Snoop protocol improves TCP performance by deploying an agent at the base station, called the snoop agent. This agent aids in the quick, local recovery of data loss by taking advantage of the information conveyed in TCP acknowledgments (ACKs) from the receiver. The internal state it maintains is soft, in that it is not absolutely necessary for the correct end-to-end functioning of the protocol. Furthermore, it is very easy to regenerate and periodically refresh if it is lost or corrupted.

For data transfer *from* a mobile host, the protocol works by using a mechanism called *Explicit Loss Notification*. These are notifications generated by the base station and sent to the mobile sender as part of the ACK stream, which reacts to them by performing retransmissions, but *not* congestion control. In the first case (transfer to a mobile host), the Snoop protocol is an example of a *transport-aware* link protocol, while in the second case (transfer from a mobile host) it is a *link-aware* transport protocol.

The rest of this chapter is organized as follows. In Section 4.2, we specify the design goals and present an overview of the key ideas in the protocol. In Section 4.3 we describe the details of data transfer to a mobile host and in Section 4.4 the details of data transfer from a mobile host. In

Section 4.5, we describe how the Snoop protocol achieves good performance in the face of user mobility using a multicast-based low-latency handoff protocol. We discuss the implementation details in Section 4.6, performance results from our implementation in Section 4.7, and the results of detailed simulations on different workloads using an empirical error model in Section 4.8. We conclude this chapter with a summary in Section 4.9.

The next chapter contains a detailed performance evaluation of the Snoop protocol and quantitative comparisons with other protocols. There, we discuss the relative benefits of each of several techniques used in this and other protocols, and argue that exploiting cross-layer protocol optimizations is key to achieving good performance in error-prone wireless environments.

4.2 Overview

We start by presenting some preliminary measurements of TCP over a wireless network to illustrate the problem and understand the reasons for poor performance. We then discuss our design goals and present an overview of the Snoop protocol.

4.2.1 Preliminary Measurements

Figure 4.2 shows the packet sequence trace of a 2 MByte bulk TCP transfer between a fixed host connected to the base station via 16 Internet hops, communicating with a mobile host over a 2 Mbps WaveLAN link. These experiments were performed when there was little network congestion over the wide-area wired Internet, so any losses that occurred were due to wireless corruption. In the first case, we placed the base station and receiver radio units close to each other and ensured that no wireless losses occurred, while in the second case the radio units were in their normal locations in our testbed. We find that when wireless errors occur, TCP performance degrades significantly—in these experiments, a coarse packet error rate of about 5% led to a performance degradation of a factor of 4.5 from ideal.

A closer analysis of the packet sequence trace reveals the reasons for this. During the course of the transfer, packets are lost over the wireless link and need to be retransmitted. Every time the TCP sender detects the loss of a packet, it retransmits it, but also reduces its congestion window, ascribing the loss to be due to network congestion. This is the correct interpretation for wired networks,

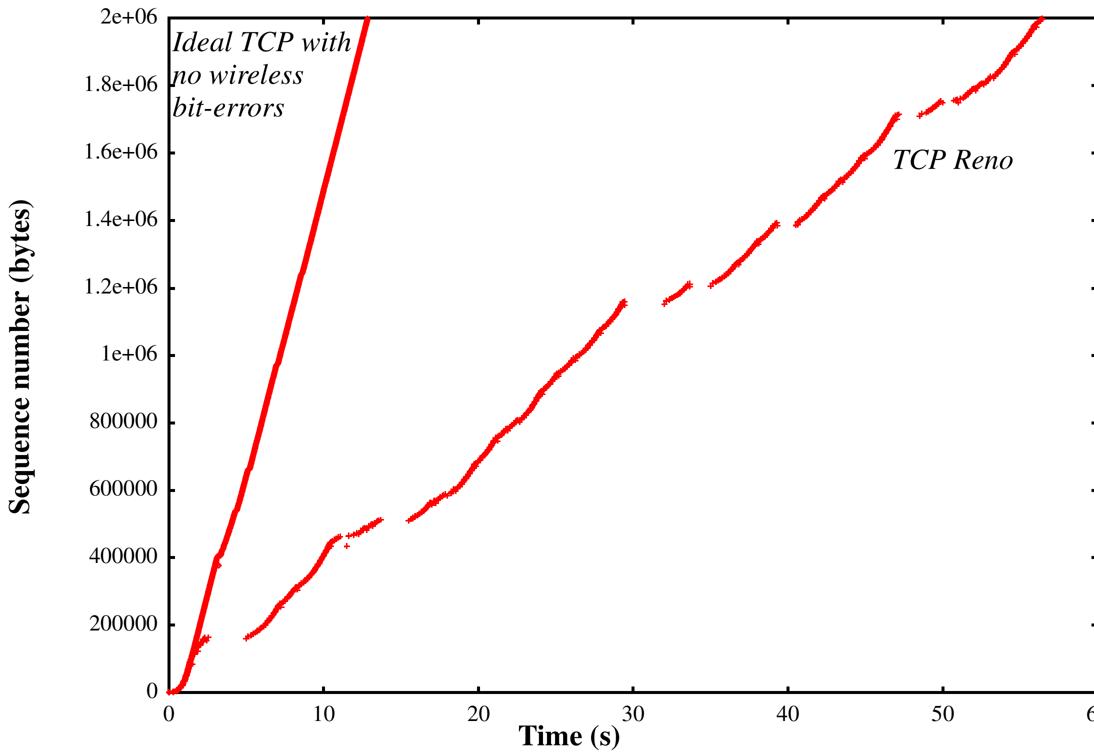

Figure 4.2 Sequence trace of TCP transfers over an error-free wireless link and a normal, error-prone link. The performance degradation compared to the ideal case is a factor of 4.5 in this experiment, because the sender misinterprets corruption losses as a sign of network congestion and incurs expensive timeouts that cause long idle times.

but is usually the incorrect response in networks that have wireless links. As a result, average windows are small and timeouts are frequent. Figure 4.2 shows the stalls that occur in the progress of the connection due to timeouts, leading to degraded performance.

Our goal is to improve the performance of TCP in such networks where non-congestion losses are common, in both local-area and wide-area network topologies.

4.2.2 Design Goals

In this section, we articulate our key design goals in developing a solution to the problems described above.

- *Develop a local solution:* When data is transferred from a fixed Internet host to a mobile host across both wired and wireless links, the problems caused by wireless links are *local* problems. A local solution to these problems is better than a sender-based one because loss recovery is faster in the former case and wired bandwidth is conserved. The Snoop protocol is one such mechanism—a localized wireless link protocol that improves end-to-end performance.

- *Eliminate adverse interactions between protocol layers:* In Section 2.5, we described the adverse interactions between conventional link-layer and end-to-end transport protocols. Our design explicitly avoids these interactions.

- *Enable incremental deployment:* Most current network applications that require reliable unicast transmission use TCP as their transport protocol. Therefore, it is desirable to achieve our goal of improving its performance in our network without changing existing TCP implementations in the fixed network. This will enable incremental deployment of our solutions.

- *Preserve end-to-end semantics:* We use the end-to-end argument in system design [127] as a guiding principle in our design. It states that a service should be implemented in a subsystem only if the service can be completely implemented in that subsystem, or if by *partially* implementing the service in the subsystem, *it substantially improves end-to-end performance.* We design our link-layer solution carefully and do not change the service model or semantics provided by TCP.

- *Use only soft state:* Another key design goal is the use of only *soft state* [31] in the protocol. The state maintained by the protocol must be *soft,* so that its loss does not impact the correct end-to-end functioning of the connection. If the state associated with the protocol is lost, it can be rebuilt easily upon the arrival of the next packet and ACK.

4.2.3 Protocol Description

The Snoop protocol modifies the networking software only at the base station and the mobile host connected over the wireless link. In particular, no modifications are required to TCP implementations elsewhere in the Internet. The protocol works by deploying a *snoop agent* at the base station. The agent mainly performs the functions of loss detection and loss recovery via retransmissions.

For transfer of data from a fixed host to a mobile host, we make modifications only at the base station. These modifications include caching unacknowledged TCP data and performing local retransmissions based on policies that depend on TCP ACKs from the mobile host and timeouts of locally-maintained timers. By using duplicate ACKs to identify packet loss and performing local retransmissions as soon as this loss is detected, the snoop agent shields the sender from the vagaries of the wireless link. In particular, transient situations of very low communication quality and temporary disconnectivity are hidden from the sender.

For transfer of data from a mobile host to a fixed host, the snoop agent detects segments corrupted over the wireless link at the base station by snooping on packets arriving from the mobile host and identifying holes in the transmission sequence. Then, when it detects duplicate ACKs from the receiver signifying a loss of data due to corruption, it sets a bit in their TCP headers and forwards them to the mobile sender. The sender uses this *Explicit Loss Notification (ELN)* information to identify that the loss was unrelated to congestion, and retransmits the packet *without* invoking any congestion-control actions. In this case, the protocol requires modifications to both the base station and mobile hosts.

Together, these mechanisms improve the performance of the connection in both directions, without sacrificing the end-to-end semantics of TCP or modifying host TCP code in the fixed Internet. This combination of greatly improved end-to-end performance, preservation of end-to-end semantics, and local recovery techniques is the main contribution of this work. In Chapter 5, we analyze the detailed reasons for improved performance and arrive at some general conclusions about the design of simple, yet highly-effective, link-layer protocols that do not interact adversely with TCP.

4.3 Data Transfer from a Fixed Host

We first describe the protocol for transfer of data from a fixed host to a mobile host through a base station. The base station's networking software is modified by adding the snoop agent, which monitors every packet that passes through the connection in either direction. Packets that arrive at the base station and destined for the mobile host now pass through the snoop agent between the IP and link layers. We emphasize that no transport layer code runs at the base station; rather, there is transport-aware link-level software that operates on all packets and ACKs.

The snoop agent maintains a cache of TCP packets[1] sent from the FH that have been forwarded to, but not yet been acknowledged by the MH. This is easy to do since TCP has a cumulative ACK policy (Section 2.2.1). When a new packet arrives from the FH, the agent adds it to its cache and passes the packet on to the forwarding code which performs the normal packet forwarding functions. The snoop agent also monitors TCP ACKs sent from the mobile host, using new ACKs to clean its cache and maintains only unacknowledged packets there. It detects the loss of a packet by the arrival of duplicate ACKs or by a timeout based on locally maintained round-trip timers. When it does, it retransmits the lost packet to the MH if it has the packet cached, since these are packets that are lost due to wireless corruption. Additionally, it *suppresses* duplicate ACKs corresponding to wireless losses from the TCP sender, thus ensuring that false congestion control invocations associated with fast retransmissions are avoided. If the agent does not have the packet cached, then it was lost earlier in the path due to congestion. The corresponding duplicate ACKs are forwarded to the TCP sender, allowing fast retransmissions and congestion control to occur as usual. Thus, the snoop agent at the base station hides non-congestion-related wireless packet losses from the FH sender using local retransmissions and by suppressing certain duplicate ACKs; at the same time, it ensures that the sender continues to react appropriately to network congestion.

The snoop agent has two main components: *data processing* and *ACK processing*. The flowcharts summarizing the salient features of the algorithms for these steps are shown in Figures 4.3 and 4.4. The remainder of this section describes these algorithms in detail.

4.3.1 Data Processing

In this phase, the snoop agent processes incoming data segments from the fixed host. A TCP segment is identified by the sequence number of its first byte of data and its size[2]. At the BS, the snoop agent keeps track of the last sequence number seen for the connection. At any stage in the protocol, one of several kinds of packets can arrive at the BS from the FH, and the snoop agent processes them in different ways:

1. We use the terms “packet” and “segment” interchangeably. Where the distinction is important, we explicitly point it out.
2. On a sender retransmission, TCP implementations may re-segment data. We handle this case correctly in the implementation of the snoop agent.

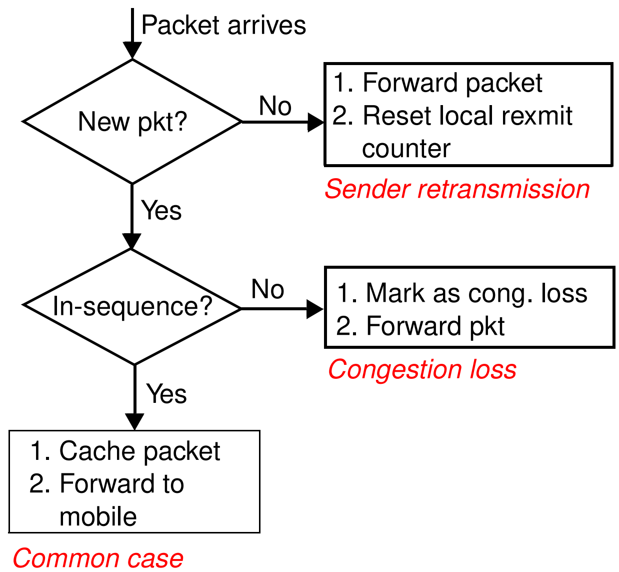

Figure 4.3 Flowchart for data processing.

1. *A new packet in the increasing TCP sequence:* This is the common case, when a new packet in the normal increasing sequence arrives at the BS. In this case the packet is added to the snoop cache and forwarded on to the MH. The agent does not perform any extra copying of data while doing this. In addition, it logs the current transmission time to later estimate the round-trip time of the connection over the wireless link.

2. *An out-of-sequence packet that has been cached earlier:* Although this is an uncommon case, it can and does happen. First, if there have been multiple losses in a single window due to congestion on the wired path, there are times when a timeout and slow start occur after a fast retransmission. This could lead to a sender retransmission of a previously cached segment. Second, if there is persistent congestion over the wireless link, local retransmissions are not triggered from the base station's snoop agent. This implies that the TCP sender recovers from such losses by retransmitting packets. Finally, if the wireless link has been error-prone for several hundred milliseconds, the sender could time out and retransmit the missing segment, leading to this situation. In our experience, the first two situations are the common reasons for the occurrence of this condition.

   If the sequence number is greater than the last ACK seen, it is very likely that this packet didn't reach the MH earlier, and so it is forwarded on. If, on the other hand, the sequence number is less than the last ACK, this packet has probably already been received by the MH. At this point, there are several possible actions the agent could take. One possibility would be to discard this packet and continue, but this is not always the best thing to do. The reason for this is that the original ACK with the same sequence number could have been lost due to congestion while going back to the FH. The second possibility, to facilitate the sender getting to the current state of the connection as fast as possible, is for the snoop agent to send a TCP ACK corresponding to the last ACK seen at the BS (with the source address and port corresponding to the MH) to the FH. However, the disadvantage of this is that if the information in the snoop agent's state is wrong for any reason, the correctness of the end-to-end protocol is compromised. The third option is to simply forward the packet to the MH, and await information from subsequent ACKs to refresh the state at the agent. This is the option the agent uses, in keeping with our soft-state design philosophy. The agent also resets the number of local retransmissions to zero, and updates the transmission time of this segment to correctly estimate the round-trip time when its ACK arrives.

3. *An out-of-sequence packet that has not been cached earlier:* In this case the packet was either lost earlier due to congestion on the wired network, or has been delivered out-of-order by the network. The former is more likely, especially if the sequence number of the packet (i.e, the sequence number of its first data byte) is more than one or two packets away from the last one seen so far by the snoop agent. This packet is cached by the agent and then forwarded to the MH. It is also marked as having been retransmitted by the sender. The ACK processing algorithm uses this information to process duplicate ACKs that arrive for this packet from the MH.

4.3.2 ACK Processing

In this phase, the snoop agent monitors and processes the ACKs sent back by the MH and performs various operations depending on the type and number of ACKs it receives. These ACKs fall into one of four categories:

1. A *new*, *expected* ACK: This is the common case that occurs when no recent segments have been lost, and signifies an increase in the packet sequence received at the MH. This ACK initiates the cleaning of the snoop cache and all acknowledged segments are freed. The round-trip time estimate for the wireless link is also updated at this time. Finally, the ACK is forwarded to the FH.

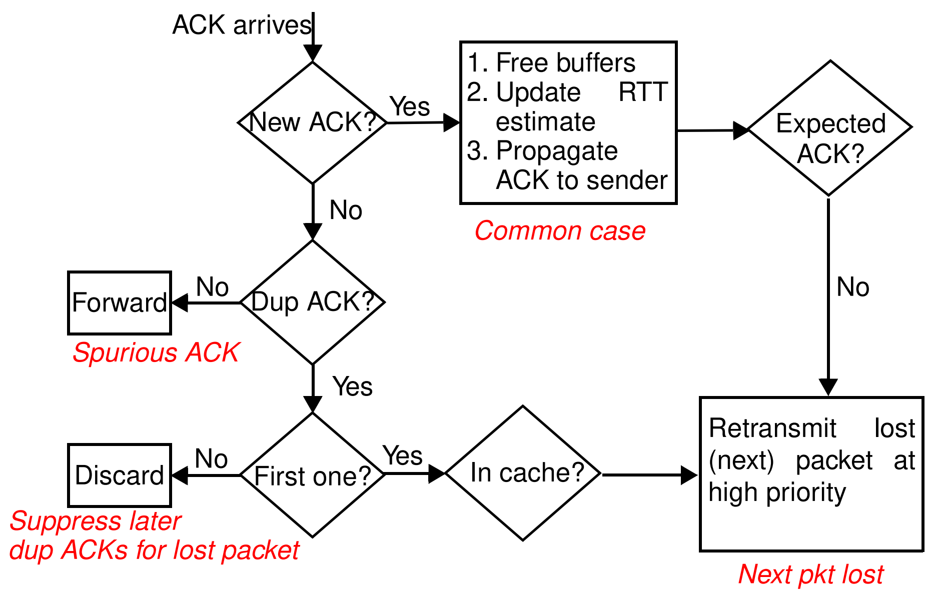

Figure 4.4 Flowchart for ACK processing.

2. A *spurious* ACK: This is an ACK less than the last ACK seen by the snoop agent, and is a situation that rarely occurs. Just as in case (2) of the data processing algorithm, it could be discarded. Instead, we forward it to the FH, to guard against the possibility that the internal state of the snoop agent may be incorrect.

3. A *duplicate* ACK (DUPACK): This is an ACK that is identical to a previously received one. In particular, it is the same as the highest cumulative ACK seen so far, and does not advertise a different receiver window from the previous one. In this case the next packet in sequence from the DUPACK has not yet been received by the MH. However, some subsequent packets in the sequence have been received, since the MH generates a DUPACK for each TCP segment received out of sequence. One of several actions is taken depending on the type of DUPACK and the current state of the snoop agent:

- The first case occurs when the DUPACK is for a packet that is either not in the snoop cache or has been marked during data processing as having been retransmitted by the sender. If the packet is not in the cache, it needs to be resent from the FH, perhaps after invoking the necessary congestion control mechanisms at the sender. If the packet was marked as a sender-retransmitted packet, the DUPACK needs to be propagated to the FH because the TCP sender maintains state based on the number of duplicate ACKs it receives when it retransmits a packet. Therefore, both these situations require the DUPACK to be routed to the FH. This is especially important to help the sender perform fast recovery (Section 2.2.2) correctly.

- The second case occurs when the snoop agent gets a DUPACK that it does not expect to receive for the packet. This typically happens when the first DUPACK arrives for the packet after a subsequent packet in the stream reaches the MH, following a packet loss. The arrival of each successive packet in the window causes a DUPACK to be generated for the lost packet. To make the number of such DUPACKs as small as possible, the agent retransmits the lost packet as soon as the loss is detected at a *higher priority* than normal packets (how this is done is described in Section 4.6). In addition, the snoop agent estimates the maximum number of DUPACKs that can arrive for this packet. This is done by counting the number of packets that were transmitted after the lost packet prior to its retransmission. The snoop agent also updates its estimate of the next expected new ACK, once the losses in this window are recovered. This is equal to one more than the last data byte sent prior to the loss.

- The third case occurs when an "expected" DUPACK arrives, based on the above maximum estimate. The missing packet would have already been retransmitted when the first DUPACK arrived (and the estimate was zero), so no retransmissions occur at this stage. In practice, the retransmitted packet reaches the MH before most of the later packets do (because it was retransmitted at a higher priority) and the BS sees an increase in the ACK sequence before all the expected DUPACKs arrive.

  In addition, we *suppress* this DUPACK by discarding it, thereby preventing the sender from receiving and reacting to it. This is because we have already ensured a retransmission attempt for the corresponding segment, and do not want the sender to perform a fast retransmission and invoke congestion-control actions in response to this perceived loss.

4. A *new, unexpected* ACK: During the processing of DUPACKs following a set of losses in a window, the snoop agent keeps track of the first new ACK it expects to see when all losses are recovered from. If the new ACK is smaller than this estimate, we call it "unexpected." The most likely cause for this is that the next segment was lost as well, for otherwise the new ACK would

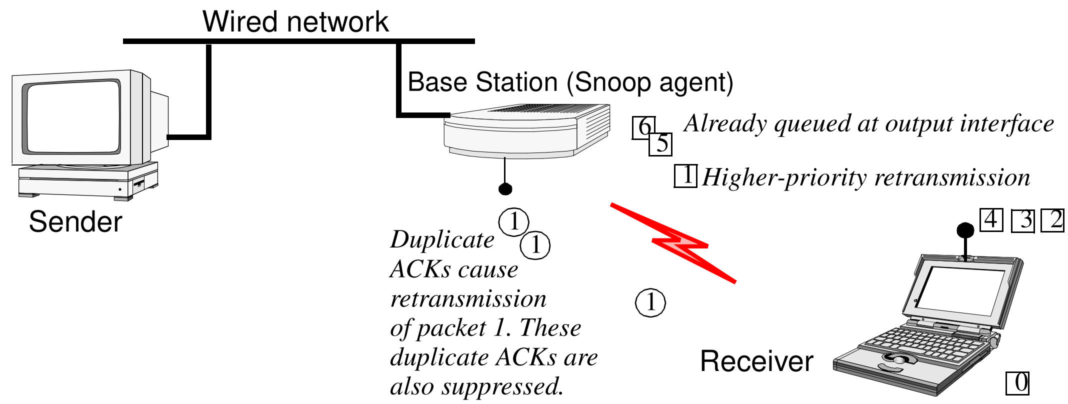

Figure 4.5 Higher-priority retransmissions from the Snoop agent ensure that the new ACK generated is smaller than if retransmissions were queued at the tail of the interface queue. In this example, packet 1 was lost and is being retransmitted at higher priority. When it reaches, the new cumulative ACK would be 4. If the retransmission were not done at higher priority, two more duplicate ACKs would be generated (when 5 and 6 reach), and the new ACK generated when 1 eventually reaches would be 6 (compared to 4).

have equalled the next expected one. In addition, no DUPACKs may arrive for this loss because all the DUPACKs in the window arrived for the previous loss. Thus, after cleaning the cache and freeing all acknowledged segments, we retransmit the next missing one.

Retransmitting packets at a higher priority improves performance at all error rates. This enables retransmitted packets to reach the mobile host sooner, reducing the number of duplicate ACKs. It reduces the sensitivity of the protocol to accurately maintaining a count of the number of expected duplicate ACKs, since ACKs could get lost during loss periods. In the common case when the wireless link is the connection's bandwidth bottleneck, the first new ACK generated after a higher-priority retransmission is less than what it would be without such prioritization, as shown in Figure 4.5. This implies that the number of segments transmitted by the sender in a burst is smaller in the former case, since the sender's sliding window would shift by a smaller amount. This in turn reduces the degree of congestion in the network by reducing the occurrence of bursts.

The snoop agent also keeps track of the number of local retransmissions for a packet, and resets this number to zero if the packet arrives again from the sender following a timeout or a fast retransmission. In addition to retransmitting packets depending on the number and type of ACKs (*data-driven retransmissions*), the snoop agent also performs retransmissions driven by timeouts. This is described in more detail in Section 4.6.

4.4 Data Transfer from a Mobile Host

We now proceed to discuss the case of data transfer from a mobile host. While this has conventionally been a less-common scenario for wireless information access, it is rapidly gaining in importance. There are an increasing number of Web and other information servers that serve pages from behind wireless links [85, 64].

It might be tempting to come to the conclusion that a protocol very similar to the one described in earlier sections for data transfer to the mobile host can be used in this case as well. In particular, one of the important features of the previous case was that it required changes only to the base station and not to any other entities in the network. However, it is unlikely, if not impossible, that a protocol with modifications made *only* at the base station can substantially improve the end-to-end performance of reliable bulk data transfers from the MH to other hosts on the network, while preserving the precise semantics of TCP ACKs. For example, simply caching packets at the base station and retransmitting them as necessary will not be particularly useful, since packet corruption is likely to occur en route from the mobile host to the base station. Running a snoop agent at the mobile host will be inappropriate because the same duplicate ACKs signify both corruption and congestion losses. There is no way for the mobile sender to know if the loss of a packet happened on the wireless link or elsewhere in the network due to congestion.

There are at least two ways in which the basic snoop scheme can be enhanced to achieve good performance for this case—the first involves the use of negative acknowledgments and the second the use of Explicit Loss Notifications (ELN). We first describe the first approach, and then the second, which we believe to be an elegant solution to the problem that has the advantage of simplicity. In the next chapter, we show how the ELN-based algorithm naturally generalizes to the case when the wireless link is a transit link bridging two or more wired networks.

4.4.1 Using Negative Acknowledgments

An agent at the base station keeps track of the packets that were lost in any transmitted window and generates negative acknowledgments (NAKs) for those packets back to the mobile sender. This is especially useful if several packets are lost in a single transmission window, a situation that happens often under high interference or in fades where the strength and quality of the signal are low. These NAKs are sent when either a threshold number of packets (from a single window) have reached the base station or when a certain amount of time has expired without any new packets from the mobile. Encoding these NAKs as a bit vector can ensure that the fraction of the sparse wireless bandwidth consumed by NAKs is relatively low. The mobile host used these NAKs to selectively retransmit lost packets.

NAKs can be implemented using TCP's Selective Acknowledgment (SACK) option [92, Section 2.3.1]. The snoop agent could use SACKs to enable the mobile host to quickly (relative to the round-trip time of the connection) retransmit missing packets. The only change required at the mobile host is to enable SACK processing. No changes of any sort are required in any of the fixed hosts. Also, SACKs are generated by the snoop agent when it detects a gap corresponding to missing packets; we emphasize again that no transport-layer code runs at the base station to do this.

There is a danger of a possible violation of end-to-end semantics in this scheme, because the snoop agent would send SACK information about segments it receives, but they may not yet have been received by the intended receiver. This will lead to problems if the corresponding segments get lost later in the path. However, this is not strictly true because of the semantics of TCP SACKs: they are *advisory* in nature, and only cumulative ACKs are binding. What this means is that even if a TCP sender receives SACKs for certain data blocks, it must not clean those segments from it's transmission buffer or assume successful reception, until cumulative ACKs arrive for them.

However, this approach based on SACKs is inelegant because it relies on a relatively obscure feature of the SACK specification. Of course, we could implement an explicit NAK scheme, but that would require complex modifications at the sender, which would anyway implement SACKs in the future. In addition, we would like to avoid explicit protocol messaging as far as possible, and both SACKs and NAKs generated from the base station do not provide this. All these issues led us to investigate alternate protocols; what follows is one such scheme.

4.4.2 Using Explicit Loss Notifications

Explicit Loss Notification (ELN) is a mechanism by which the reason for the loss of a packet can be communicated to the TCP sender. In particular, it provides a way by which senders can be informed that a loss happened because of reasons unrelated to network congestion (e.g., due to wireless bit errors), so that sender retransmissions can be decoupled from congestion control. If the receiver or a base station knows for sure that the loss of a segment was not due to congestion, it sets the ELN bit in the TCP header and propagate it to the source. In the situation at hand, this ELN message is sent as part of the same connection (and not in a separate way, using ICMP for instance). This simplifies the sender implementation as it receives messages in-band. ELN is a general concept that has applications in a wide variety of wireless topologies. In this section, we describe it in the context of data transfer from a mobile host connected to the rest of the Internet via a cellular wireless link. The next chapter discusses how it helps improve end-to-end performance over wireless transit links.

The snoop agent running at the base station monitors all TCP segments that arrive over the wireless link. However, unlike in the previous case, it does not cache any TCP segments since it does not perform any retransmissions. Rather, it keeps track of *holes* in the sequence space as it receives data segments, where a hole is a missing interval in the sequence space. These holes correspond to segments that have been lost over the wireless link. However, it could also be the case that the packet was lost due to congestion at the base station. To avoid against wrongly marking a congestion hole as having been due to a wireless loss, it only adds a hole to the list of holes when the number of packets queued on the base station's input interface is not close to the maximum queue length.

When ACKs, especially duplicate ACKs, arrive from the receiver, the agent at the base station consults its list of holes. It sets the ELN bit on the ACK if it corresponds to a segment in the list before forwarding it to the data sender. It also cleans up all holes with sequence numbers smaller than the current ACK, since they correspond to segments that have been successfully received by the receiver. When the sender receives an ACK with ELN information in it, it retransmits the next segment, but does not take any congestion control actions. The sender also makes sure that each segment is retransmitted at most once during the course of a single round-trip, as the snoop agent would flag an ELN for *each* duplicate ACK following a loss.

4.5 Mobility

As a mobile host moves, it may get out of the range of its current base station but still be within the range of other neighboring base stations. To maintain the mobile host's connectivity, a *handoff* is performed to re-route traffic to and from the mobile host via the new base station. However, depending on the details of the handoff algorithms, this procedure could lead to packet losses and reordering, which in turn could cause significant deterioration in the performance of ongoing TCP transfers [25].

In Section 2.5.5, we discussed the details of some schemes to improve TCP performance when users are mobile [25, 8, 131]. In this section, we describe how the Snoop protocol is integrated with a low-latency handoff protocol that leads to good TCP performance in the presence of both bit-errors and user mobility.

A small amount of buffering at and retransmission from base stations prevents packet loss during the short handoff period. In [26], the buffering happens at the mobile host's old base station, which forwards packets to the new base station at the time of handoff. We use a multicast-based scheme, where one or more base stations in the vicinity of the current one join a multicast group corresponding to the mobile host and receive all packets destined to it [131]. When the handoff occurs, the new base station is readily able to forward the buffered and the newly arriving packets without incurring any losses or introducing any delay or reordering. This prevents unnecessary invocations of TCP fast retransmissions.

In contrast to these schemes that operate at the network layer, handoffs in a split-connection context (e.g., I-TCP [10]), involve the transfer of transport-layer state from the old base station to the new one. This results in significantly higher latency; for example, [8] reports I-TCP handoff latencies of the order of hundreds of milliseconds in a WaveLAN-based network. As we show in Section 4.7.2, this is over an order of magnitude slower than our scheme.

4.5.1 Multicast-based Low Latency Handoffs

As in Mobile IP [112], all packets arriving from a fixed host are intercepted by an entity called a home agent. Corresponding to each mobile host is a pre-assigned multicast address, which the home agent knows. The home agent encapsulates every packet intended for the mobile host using this multicast address and forwards it as shown in Figure 4.6.

At the same time, the mobile host chooses a small number of base stations in its current neighborhood (based on metrics like the received signal strength, error rates, etc. computed from periodic beacon signals) to join this multicast group on its behalf. One of these base stations is set up as the primary and is the *forwarding* base station. The other secondary base stations are *buffering* base

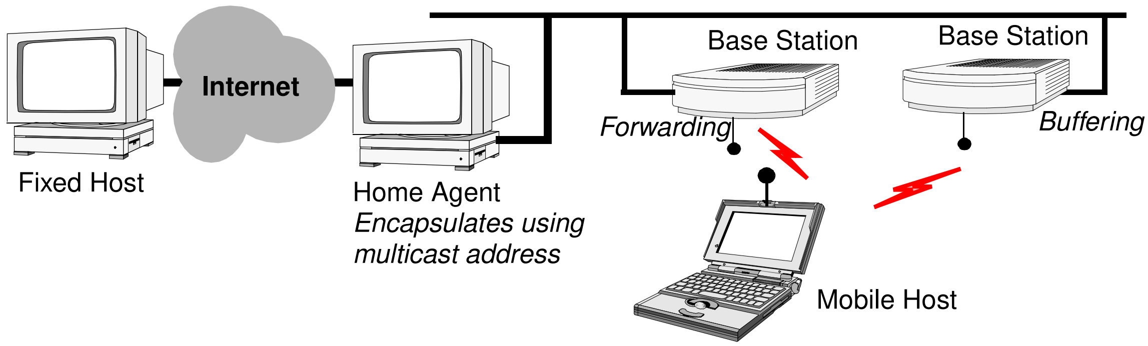

Figure 4.6 Multicast-based handoffs.

stations, storing a small number of recent packets in anticipation of a handoff. In our system, handoffs are initiated by the mobile host, which chooses one of the other base stations (typically a buffering base station) to become the primary forwarding one and sends a message to the current primary base station informing it to stop forwarding packets.

The snoop agent relies on a cache of unacknowledged packets at the base station to improve end-to-end TCP performance. After a handoff, the new base station must have the current set of unacknowledged packets in its cache to provide improved performance based on local retransmissions, when losses occur due to packet corruption. It is clear how this is achieved by using our multicast-based scheme.

Multicast allows the snoop agents at the buffering base stations to *mirror* the state present in the primary base station for the different TCP connections. Using these packets, the agents build up their caches for various connections to the mobile host. However, during this period ACKs *from* the MH continue to pass only through the *primary BS*, the mobile host's default router. Therefore, the nearby buffering base stations cannot snoop on any ACKs for the caches they are mirroring. This prevents them from promptly freeing up packets that have safely reached the MH. The caches at these base stations are maintained using a "discard-oldest" policy, since we would like to have the most recent packets at a new base station as soon as a handoff occurs.

Once the handoff occurs, ACKs begin to pass through the new forwarding BS. The first ACK the BS receives identifies which packets have been received by the MH on that connection. The BS uses this information to clean and synchronize the contents of the snoop cache to the last state of the previous primary BS.

The state of the new and previous BSs may not be identical after the synchronization. The new primary BS may be missing several packets from its snoop cache. This is because it may have either missed several packets while it was in buffering mode (due to congestion) or buffered packets for only a short period of time before the handoff. However, none of this affects the correct functioning of the connection because this state is soft. Since the snoop agent does not change any of the end-to-end semantics of TCP, the protocol is resilient to these gaps in its state. As the connection progresses the gaps in the agent's state are resolved. Our experience has been that it takes only a very small number of packets (usually less than four) before the Snoop agent at the new base station reaches the current state.

The duration of a handoff is short since no state is explicitly transferred between BSs. The state transfer is avoided because the new primary BS mirrors the state of the previous primary BS prior to handoff. This handoff scheme attains low-latency for TCP streams since multicast provides a simple, lightweight way to mirror state changes at the Snoop agents.

4.6 Implementation

We have implemented the Snoop protocol on a testbed consisting of IBM ThinkPad laptops and Pentium base stations running BSD/OS 3.0 from BSDI, communicating over a Lucent WaveLAN. The implementation currently supports bulk transfers to and from mobile hosts and is integrated with mobility software that provides smooth, low-latency handoffs described in the previous section.

Recall from Section 4.2.2 that one of the key design goals in the Snoop protocol was the use of *soft state*. This makes crash recovery and state management during mobility easy, unlike in approaches involving hard state, where a crash might compromise an existing connection or user mobility might entail explicit state transfer. This design goal was faithfully followed in our implementation, where care was taken to ensure that there was little permanence or criticality to the state maintained by the snoop agent, as far as the *correct* working of the end-to-end connection was concerned. The state maintained by the snoop agent can be reconstructed from scratch by snooping on a few packets and ACKs.

Although the Snoop protocol is conceptually a transport-aware *link* protocol when the direction of data transfer is from a FH to a MH, we implement it in the network (IP) layer of the BSD kernel. This is done primarily for reasons of simplicity, because we have ready access to the IP and TCP headers of the packet without any Ethernet-level headers at this point. This also enables easy integration with handoff modules that reside at the network layer of the stack.

When a TCP segment arrives from or is intended for a MH, it passes on to the Snoop protocol control function. This function determines which specific agent to invoke, depending on the connection, and if necessary, instantiates one.

4.6.1 State Management

When a TCP segment arrives from or destined for a MH, the snoop control code attempts to obtain the per-connection data structure from its hash table of connections. If this attempt fails, a new agent is created and initialized for the connection. A new agent is also created if the attempt succeeds, but the segment that just arrived is a SYN segment, because this signifies the start of a new connection. At the same time, the old agent (if there is one) is freed and its space reclaimed. We do not constrain the creation of a new segment to happen only when a SYN segment arrives, in keeping with our soft state design philosophy. By doing this, we ensure that agent setup is spontaneous and automatic upon the arrival of any segment.

The state maintained by the snoop agent is cleaned, and any cached packets freed, when a segment with the FIN or RST flags arrive. However, clean-up of agent state is not dependent on the arrival of a FIN—state is also cleaned if there has been no activity on the connection for a time specified by the *garbage collection timer,* set to 20 seconds in our implementation. If the end-to-end connection has not terminated and is in fact continuing after a long idle time, nothing untoward happens— agent state is implicitly instantiated upon the arrival of the first segment on the connection. In contrast, hard state implementations like split connections are constrained to keep the TCP connection open and consume control block state at the BS even when the connection is idle for long periods of time. Furthermore, they are vulnerable to mistakes in BS state or to BS crashes, a drawback not present in the soft state implementation that the Snoop protocol enables.

Finally, we note that state management is complicated by the bi-directional nature of TCP connections. Data can flow in both directions of the same connection, and ACKs are sometimes piggybacked on data segments. When a FIN segment arrives at the BS, only the state in the agent for that direction of the connection is cleaned; in general, the other direction will still be active. The garbage collection timer cleans state in both directions, since it is triggered only when both directions have been idle[1]. Another complication arises because of temporary routing pathologies when a MH moves between cells. There could be periods of time during which data packets go through one BS (via an active snoop agent), but ACKs may go through another BS. This situation can be detected, since the difference between the last sequence number and the last ACK seen at the BS would be larger than the maximum transmission window size. When this happens, the agent ensures that the replacement policy in the snoop cache is LRU; i.e., recently arriving segments are cached by replacing the oldest ones. The agent also disables the local retransmission timer until the routing inconsistency is resolved.

4.6.2 Data Transfer from a Fixed Host

To achieve good TCP performance in this case, we need to modify only the software at the BS. The mobile host and fixed hosts on the Internet are unchanged.

Data structures: The data used by the snoop agent are hierarchically organized. At the highest level is a hash table that uses the source and destination ports as a key and hashes into a pointer to the per-connection state. The per-connection state consists mainly of the snoop packet cache and some other associated information. The snoop cache is maintained as a circular (ring) buffer of packets, consisting mainly of pointers to kernel mbufs [97]. The state maintained for each packet includes the packet sequence number, its size, the number of attempted local retransmissions, and a flag set if the packet was thought to have been retransmitted by the sender. For each connection, we also maintain some other associated information, as depicted in Figure 4.7. Since this soft state takes up only 54 bytes per active connection, most of the used memory is due to the packets themselves. In general, the size of the cache at a forwarding base station needs to be large enough to handle the maximum transmission window size.

---

1. We could have maintained independent garbage collection timers for each direction, but decided against it because we wanted to minimize the number of timers and timer interrupts in the protocol.

```
typedef struct wi_conn_state { /* FH --> MH */
	u_short  wi_state;	/* current state of snoop (wired sender) */
	u_short  wl_state;	/* current state of snoop (wireless sender) */
	u_long   addr;		/* address of FH (wired side) */
	u_long   wladdr;	/* address of MH (wless side) */
	u_short  port;		/* wired side port */
	u_short  wlport;	/* wireless side port */
	tcp_seq  last_seen;	/* first byte of last packet buffered */
	tcp_seq  last_ack;	/* last byte recd. by MH for sure */
	u_short  last_win;	/* last flow control window size from receiver */
	tcp_seq  expected_next_ack;  /* expected next ACK on connection */
	short    expected_dacks;	/* expected number of duplicate ACKs */
	tcp_seq  iss;		/* initial seq # for this connection (used for debugging) */
	long     srtt;		/* smoothed rtt estimate */
	long     rttdev;	/* smoothed rtt linear deviation */
	short    bufhead;	/* next pkt goes here in ring buffer */
	short    buftail;	/* first unack'd pkt in ring buffer */
	short timeout_pending;		/* timeout pending on this connection */
	packet_t *pkts[];	/* ring buffer of unack'd packets (snoop cache) */
} wi_conn_state_t;

typedef struct wl_conn_state { /* MH --> FH */
	tcp_seq  wl_last_seen;	/* last seq  number seen from wireless side (MH) */
	tcp_seq  wl_last_ack;	/* last ACK seen */
	short    wl_bufhead;	/* head of seq_t ring buffer */
	short    wl_buftail;	/* tail of ring buffer */
	seq_t    *wlseqs[];	/* ringbuf of seq #s and sizes */
} wl_conn_state_t;
```

Figure 4.7 Per-connection data structures used in the implementation of the Snoop protocol. For data transfer from a FH, the space used is 54 bytes. Thus, most of the space is used by the TCP segments themselves. For a transfer from a MH, typical space consumption is 16 bytes, plus 8 bytes per hole to store the hole's sequence number and size.

Data processing: The implementation of this phase consists of a caching step and a forwarding one. Several studies have shown that one of the predominant implementation costs of TCP arises due to the copying of data [34, 73]. We use the reference counting mechanism in kernel mbufs to avoid data copying in the snoop agent. Thus, we do not incur any extra overhead associated with copying at the base station. Thus, the overhead due to the protocol implementation is negligible in the common case when segments are arriving in increasing order: an incoming segment is added to the cache without copying it, and it is forwarded on to the mobile host. A small number of state variables (the last sequence number seen and a flag indicating that the connection is still active) are updated.

To make it easy to clean acknowledged data from the cache and also perform local retransmissions, the cache is maintained in increasing sequence order. Thus, when out-of-sequence data arrives at the snoop agent, we may need to reorganize the ring buffer. If the arriving segment is smaller in sequence than the lowest segment currently cached, then we prepend it to the cache and forward the segment on. If not, we traverse the cache and locate the position of the current segment, shift all the earlier segments to the left, and insert the current one at that position. Finally, the cache might be full, in which case we eliminate the oldest segment and cache the current recent one. Continuing to save old segments is inappropriate in general because the mobile host may have moved out of range of the current base station and be sending (and receiving) its packets via another BS.

ACK processing(): When a new ACK arrives at the base station, we forward it on to the fixed host and clean the snoop cache by freeing the packets corresponding to packets already acknowledged by the mobile. We also update the estimate of the wireless link's round-trip time and linear deviation, and restart the pending retransmission timer. When a duplicate ACK arrives, we perform the algorithm described in Section 4.3.2.

Retransmissions: In addition to retransmitting packets depending on the number and type of ACKs received as described in Section 4.3.2, the snoop agent also performs retransmissions triggered by timeouts. The retransmission timer is based on the estimate of the smoothed round-trip time (srtt) of the last link. We compute this using the standard exponential weighted moving average technique, $srtt = (1-\alpha)*old\_srtt + \alpha*curr\_rtt$, with α set to 1/8. We also estimate the mean linear deviation of the round trip time similar to TCP (Chapter 2) and retransmit a packet if we do not receive an ACK in a time equal to the smoothed estimate plus four times the linear deviation. To limit the amount of time spent processing timer interrupts, we do not timeout more frequently than a minimum threshold time, set to 200 ms in our implementation.

On both data-driven and timer-driven retransmissions, the segment retransmission function ensures that the number of retransmissions is less than a configurable maximum threshold (whose default value is 8). If so, then it sets the IP type-of-service (TOS) field to IPTOS_LOWDELAY in the IP header of the packet and forwards the packet on to the destination. This is an indicator to the lower level interface queues to transmit this packet at a higher priority than currently enqueued packets.

4.6.3 Data Transfer from a Mobile Host

To achieve good TCP performance in this case, we need to make modifications to both the BS and the MH, which is the TCP sender. Fixed hosts in the rest of the Internet are unchanged.

Data structures: At the BS, the snoop agent detects holes in the data transmission from the MH. However, unlike in the previous case, it does not have to cache any data segments, which makes its implementation simpler. The basic data structure used for this purpose is a list of blocks, maximal contiguous segments of data specified by a starting sequence number and size. Gaps between blocks correspond to missing segments in the transmission sequence. At the MH, a new variable is added to the connection's TCP control block to keep track of the last ELN-triggered retransmission.

Data processing: When new data arrives from the MH at the BS, the agent attempts to coalesce it with an existing block and checks if it bridges a previous hole. A new data segment that is out of the normal increasing sequence creates a hole, because segments in between have been lost. The agent ensures that every hole was the consequence of a corruption-induced loss, and not due to congestion. Congestion-induced losses can occur in two ways—due to the base station's input queue overflowing, or due to the mobile host's output queue overflowing. Piggybacked on each packet from the MH is its instantaneous output queue length, which the base station can use to classify each loss it detects. When a new, out-of-sequence packet arrives at the BS, the snoop agent checks the size of its input queue and gets the size of the MH's output queue from the packet. If either of the instantaneous queue sizes is above a certain threshold of the maximum (set to 75% in our implementation), then this loss is not considered as one due to corruption. Because the agent should only flag ELN information for those losses that were unrelated to congestion, the implicitly maintained list of holes should only include packets lost due to corruption.

ACK processing: Whenever an ACK arrives at the BS, the agent updates the list of blocks by cleaning up all blocks (or parts of blocks) that have been acknowledged so far. It also checks if the ACK corresponds to a hole, i.e., if the next segment in sequence was lost due to corruption on the wireless link. It does so by comparing the ACK to the sequence number of the earliest block currently in its list for that connection. If the value of the ACK is smaller, it sets the ELN bit in the TCP header, modifies the TCP checksum, and forwards the ACK on to the MH. Because there is currently no specific bit in the TCP header for ELN, we use one of the unreserved bits for this purpose. Note that while ELN marking happens most often for duplicate ACKs, it can (and does) also happen for new ones. However, it does not happen when the list of blocks is empty, because there is no way the agent at the BS can know if any further data has been transmitted by the mobile host.

When the MH receives an ACK with the ELN bit set, it retransmits the missing packet and updates a variable (eln_last_rxmit) that keeps track of the last ELN-induced retransmission. This ensures that the MH does not retransmit the same packet on every ACK with ELN that it receives via the base station; thus, the retransmission policy is left to the end-host, and the base station only conveys relevant information to it. The TCP stack at the MH uses a configurable variable, eln_rexmt_thresh, to determine when to retransmit a segment upon receiving an ACK with ELN set. Upon receiving eln_rexmt_thresh ACKs with ELN, it retransmits the missing segment. In our implementation and experiments, we set its value to 2.

When the MH retransmits a segment based on ELN information, it does not reduce its congestion window and perform congestion control. It also bypasses the fast recovery code that inflates the congestion window for every duplicate ACK. Finally, we note that these actions are taken only for duplicate ACKs with ELN, and not for other duplicate ACKs that signify network congestion.

4.7 Implementation Performance

We performed several experiments with the Snoop protocol in our WaveLAN-based wireless testbed and compared the resulting performance with TCP Reno. We present the results of these experiments in this section. In the absence of packet losses, the maximum throughput achieved by a TCP connection was about 1.5 Mbps.

4.7.1 Wireless Bit-Errors

Figure 4.8 shows sequence traces of TCP transfers over an error-free link, as well as TCP Reno and the Snoop protocol over error-prone links. The middle curve shows the progress of a transfer using the Snoop protocol, which is a curve close to the ideal, error-free case. The performance improvement in this transfer over TCP Reno is a factor of four. This is typical of the degree of improvement we obtain in our WaveLAN-based wireless testbed for reliable data transfers.

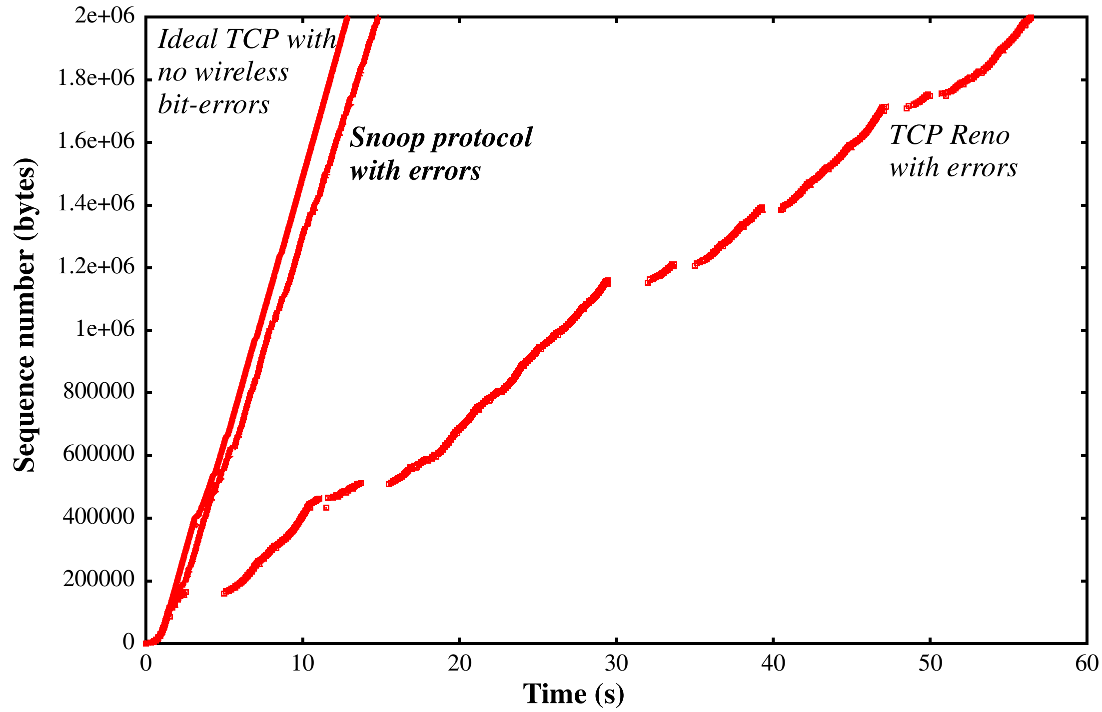

Figure 4.8 Sequence traces of wide-area TCP transfers over a last-hop WaveLAN wireless link, using an ideal error-free link, as well as TCP Reno and the Snoop protocol over error-prone links. The middle curve shows the performance improvement over TCP Reno and contrasts it to Figure 4.2.

We now discuss the results of measurements under controlled error conditions. We first present the results of data transfer from a fixed sender to a mobile receiver. The maximum possible window size for the connection was 64 KBytes and the maximum TCP segment size was 1440 bytes.

To measure the performance of the implementation under controlled conditions, we used an exponentially-distributed bit-error model. We emulated bit-errors using an exponential distribution for each bit-error rate. The emulator changed the IP checksum of the packet at the receiver if the error generator determined that the packet should be dropped. This was done on both sides of the wireless link, so both TCP data and ACKs were corrupted. The choice of exponential distribution was not intended to model reality, but to characterize the relative improvement in performance at each controlled bit-error rate. In Section 4.8, we discuss the performance of the Snoop protocol using traces of errors collected from a real testbed. Each run involved a 10 MByte transfer that was repeated twenty times at each error rate.

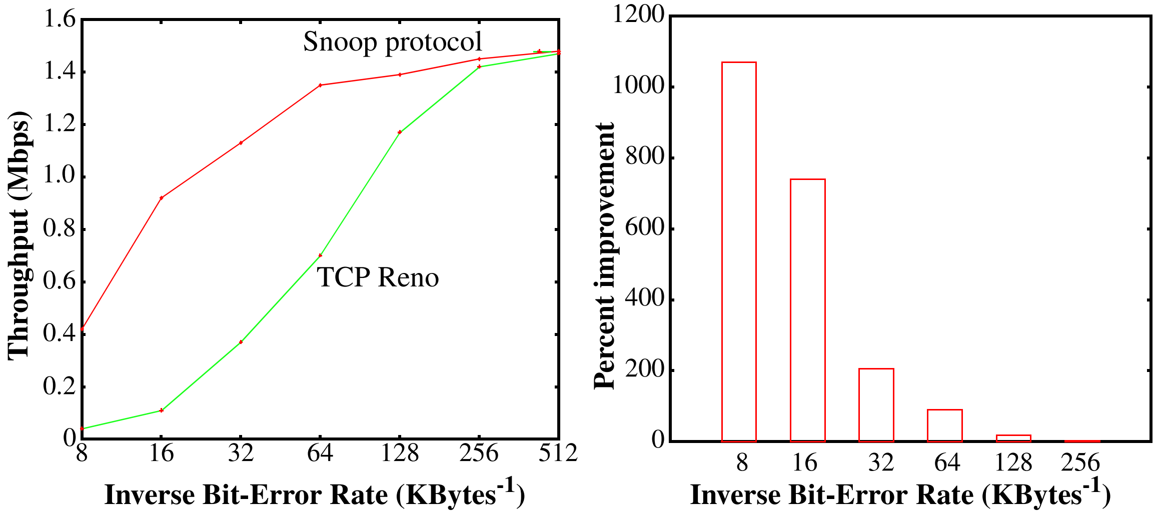

Figure 4.9 Performance of TCP Reno and the Snoop protocol for a bulk transfer across a range of bit-error rates. The figure on the left shows the throughput, while the one on the right shows the percentage improvement due to the Snoop protocol.

Figure 4.9 compares the throughput of a connection using the Snoop protocol with that of a connection using TCP Reno, for various exponentially-distributed bit-error rates shown on a log scale. We see that for bit-error rates of over $1\text{x}10^{-6}$ (close to the 64 KB point on the horizontal axis of the graph) the Snoop protocol performs significantly better than unmodified TCP, achieving a throughput improvement factor of 2 to 20 depending on the bit error rate. In fact, the Snoop protocol was robust and completed the run even when every other packet was being dropped over the last link, whereas the regular TCP connection did not make any significant progress. Under conditions of very low bit error rates ($< 5\text{x}10^{-7}$), there is little difference between the Snoop protocol and unmodified TCP. At such low bit errors there is typically much less than one error per transmitted window and unmodified TCP is quite robust at combating these error rates. At these error rates, the Snoop protocol behaves as is it were not present and incurs no overhead.

A more detailed picture of the behavior of the protocols can be seen can be seen in Figure 4.10, which plots the packet sequence trace for one of the experiments. The figure shows the comparison of sequence number progression in a connection using the Snoop protocol and a connection using TCP Reno for an exponentially-distributed bit-error rate of $3.9\text{x}10^{-6}$ (a bit error every 256 Kbits on average). We see that the Snoop protocol maintains a consistent level of progress, resulting in good throughput, while TCP Reno unnecessarily invokes congestion control procedures in response to

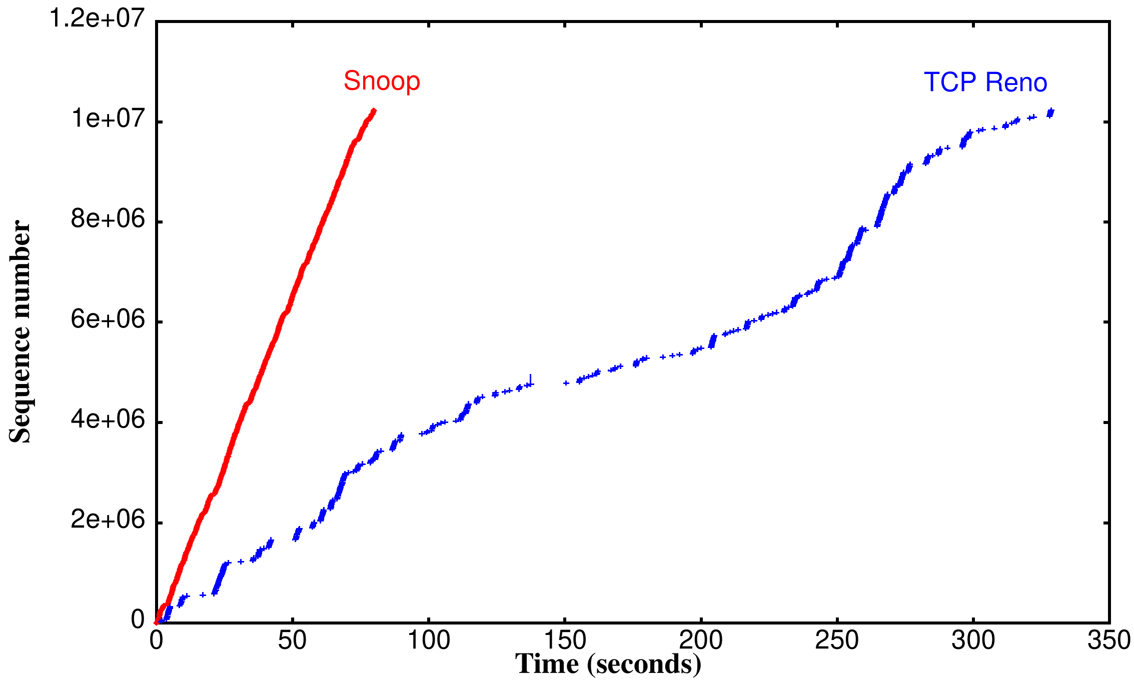

Figure 4.10 Sequence trace for transfer to MH over a channel with 3.8×10^{-6}(1/256 Kbits) BER.

bit-error losses several times during the course of the connection. This phenomenon leads to time-outs that appear as the flat and empty regions of the curve, significantly degrading the resulting throughput. For this particular transfer, the aggregate bandwidth with the Snoop protocol was about 1 Mbps, while it was only about 0.25 Mbps for TCP Reno.

We also performed several experiments to measure the performance of data transfer from the MH to the FH. These results, measured across a range of exponentially-distributed bit-error rates, are shown in Figure 4.11. As before, there are significant performance benefits of using the Snoop protocol coupled with ELN in this situation. These measurements were made for wide-area transfers between UC Berkeley and IBM Watson Research Center (NY), across one wireless WaveLAN hop and 16 Internet hops. At medium to high error rates, the performance improvement due to ELN is roughly 100%. At lower error rates, TCP Reno performs quite well as expected, and the benefits of ELN are not as pronounced. The main advantage of ELN is that it helps maintain a large TCP congestion window even when wireless error rates are high, reacting only to congestion.

While the relative performance of ELN is about 100% better than Reno at high error rates, it is not as high as the relative improvement for the FH to MH case using the Snoop protocol. The reason

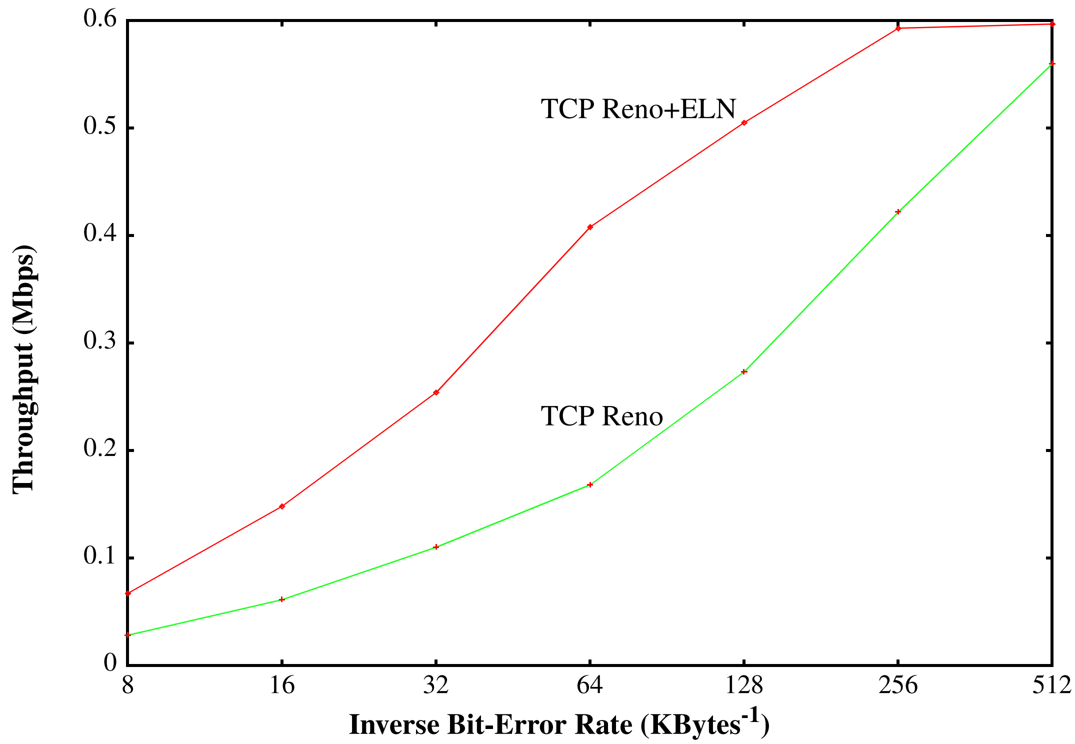

Figure 4.11 Throughput of TCP Reno and TCP Reno enhanced with ELN across a range of exponentially distributed bit-error rates, for transfers from a mobile host.

for this is that the Snoop protocol performs quick local recovery in addition to shielding the sender from congestion. On the other hand, loss recovery with ELN is due to TCP retransmissions alone. When multiple losses occur in a window TCP with ELN incurs a coarse timeout, leading to “only” a 100% improvement in throughput. A more detailed quantitative evaluation of this and enhancements to further improve performance are in Chapter 5.

4.7.2 Mobility

To isolate and measure the impact of handoffs on TCP performance, we examined the performance of a TCP connection to the mobile host with regularly spaced handoffs between two base stations. As mentioned in Section , each base station sends out a beacon signal once per second. The presence of these beacons reduces the peak TCP throughput in the absence of errors and handoffs to 1.45 Mbps. In this experiment, the arrival of a beacon at the mobile host does not trigger an analysis of the signal strengths of the different base stations; instead, the mobile host uses the time elapsed since the last handoff to determine if a handoff should occur. We performed several tests to stress the performance impact of the handoff scheme, varying the time between handoffs from 1 to

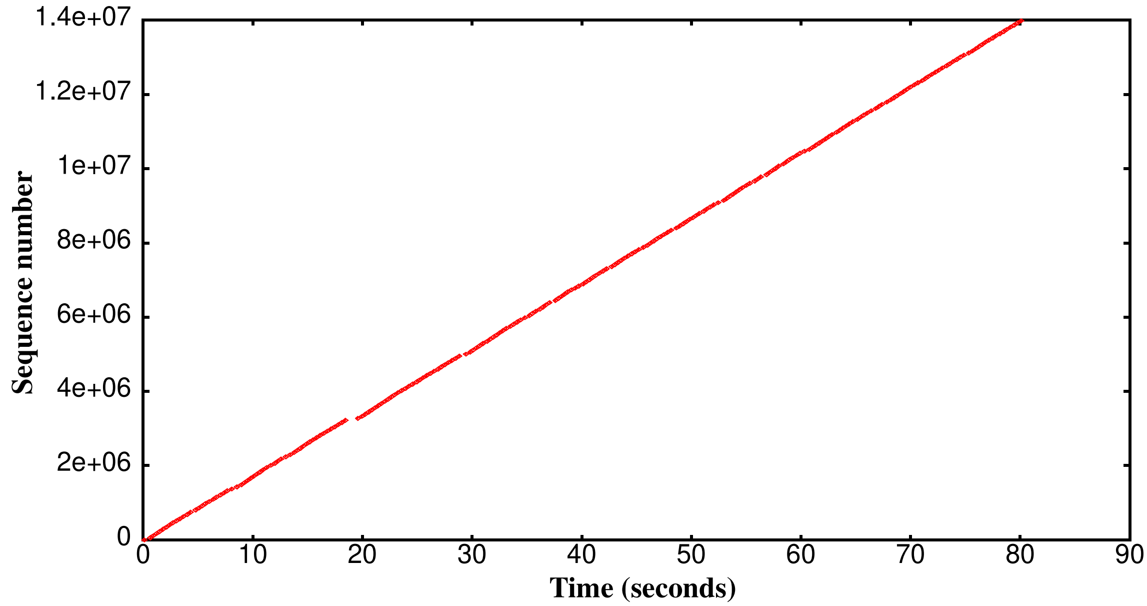

Figure 4.12 Sequence numbers for transfer to mobile host over channel with handoffs every 10 seconds.

10 seconds. The results for different handoff frequencies is shown in Table 4.1. These measure-

Time Between Handoffs (sec)	Throughput (Mbps)	Standard Deviation (Mbps)
1	1.42	.011
2	1.43	.016
3	1.43	.012
5	1.43	.014
8	1.44	.012
10	1.43	.012
Inf.	1.45	.011

Table 4.1 TCP Throughput received by the mobile host at different handoff frequencies. The last row, labeled "Inf." corresponds to the "no-handoff" case.

ments show that even frequent handoffs have very little impact on performance. In real environments, handoffs are usually likely to occur much less frequently than once per second.

The behavior of a connection experiencing handoff is shown in Figure 4.12. The figure plots the sequence numbers of a TCP connection to a mobile host with handoffs occurring every 10 sec-

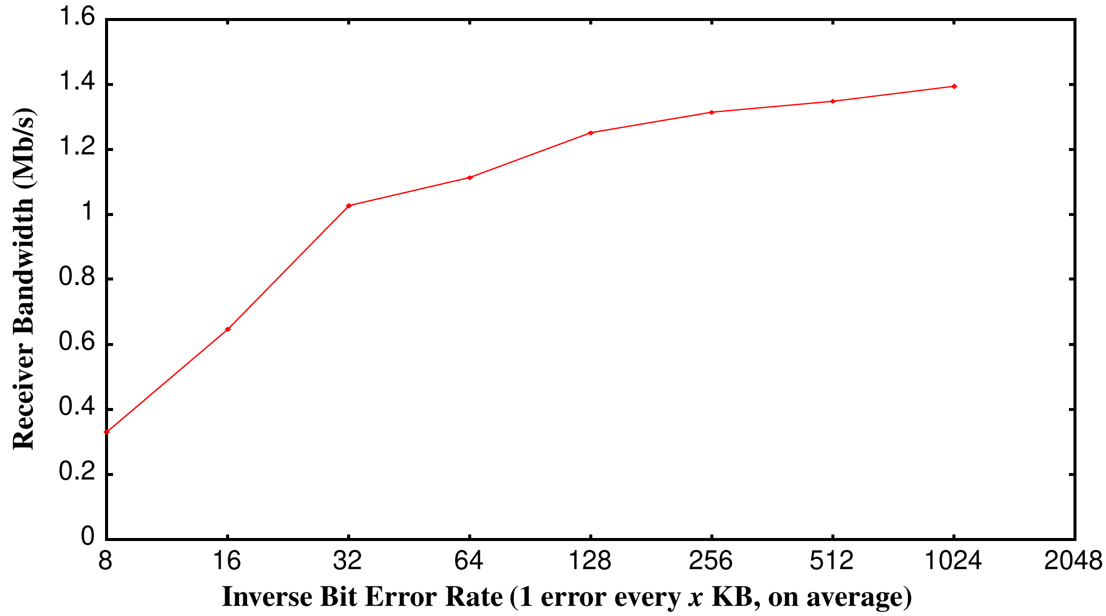

Figure 4.13 Throughput received by the mobile host at different bit-error rates ($\log_2$ scale). Handoffs occur every 5 seconds.

onds. We see that the data transfer progresses without any significant interruptions despite the presence of the handoffs. The throughput during this transfer was about 1.4 Mbps.

Figure 4.13 shows the performance obtained from the combination of the Snoop and handoff protocols. The figure plots the receiver throughput seen over a 10 Mbyte transfer. The connections experienced different error rates, shown on the x-axis, and handoffs at a fixed rate, once every 5 seconds. At low bit-error rates, $< 1\text{x}10^{-6}$ (one error every 1 Mbits of data), we obtain close to the peak performance available in the presence of beacons, 1.45 Mbps. At higher bit-error rates, we see performance comparable to transfers through Snoop to a mobile host not experiencing handoff. This indicates that the handoffs do not adversely affect the snoop processing and that the Snoop agent state is effectively mirrored at the buffering base stations.

4.8 Detailed Simulations

To thoroughly evaluate performance under different bandwidths, latencies, error rates and workloads, and to quantify the scaling behavior of the Snoop agent, we performed a number of other experiments in *ns* (Section 3.1). We now discuss these results.

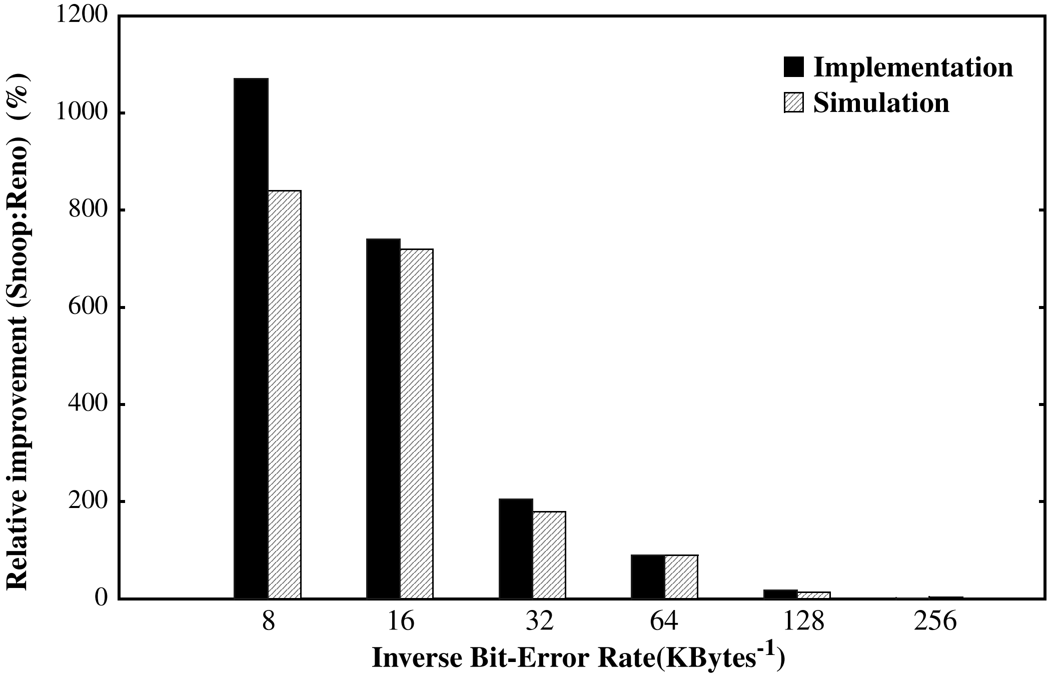

Figure 4.14 Comparison of implementation and simulation performance of the Snoop protocol relative to TCP Reno at different bit-error rates. The simulation performance closely tracks implementation measurements.

We first present some results that validate the simulation of the Snoop protocol and TCP in *ns*. Then, we present several results of bulk and Web transfers under different topologies, and a quantitative analysis of the scaling behavior of the Snoop protocol as a function of the memory consumed at the base station by the Snoop agent. In most of these experiments we use the empirical channel error model derived in Section 3.1.2.

We start by presenting a set of measurements to validate our simulation environment with real measurements. We do this by comparing the performance of a bulk TCP transfer in both environments as a function of an exponentially distributed bit-error distribution similar to the one used in the previous sections. This comparison is shown in Figure 4.14, which compares the percentage improvement of the Snoop protocol relative to TCP Reno for each error rate, in both the real implementation and the simulator. This comparison shows that the simulations track the real implementation quite closely. At high error rates, the performance of TCP Reno is especially dependent on the number and duration of timeouts that occur when multiple losses occur in a window, because losses are relatively frequent and window sizes are small. This explains why the rel-

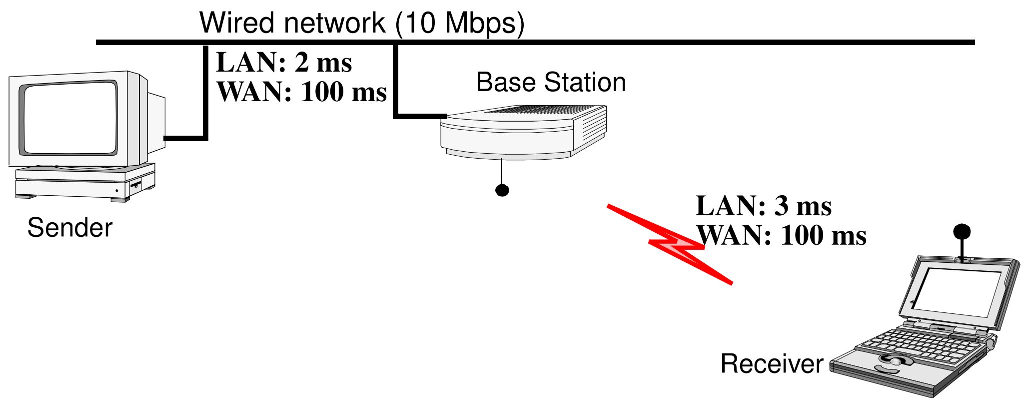

Figure 4.15 Network topologies used in simulation experiments. Wired and wireless link delays for the LAN and WAN cases are shown near the corresponding links.

ative improvement of the real implementation is larger than that in simulation in this realm—timers in the implementation are more conservative than in our simulations.

In the rest of this section, we use the empirical error models derived from traces collected in the Reinas outdoor wireless network. Section 3.2.1 describes this in more detail.

4.8.1 Bulk Transfers

We performed four kinds of experiments modeling different combinations of LAN and WAN scenarios to evaluate the Snoop protocol. These topologies are shown in Figure 4.15. We experimented with wireless link bandwidths from 128 Kbps to 4 Mbps, increasing in powers of two, in both LAN and WAN situations.

Each individual experiment, for a given combination of wired and wireless parameters, consisted of a 50-second TCP bulk transfer with 1000-byte TCP packets in the presence of a small amount of background cross-traffic. Errors on the wireless link were generated according to the empirical error model shown in Figure 3.3 and Figure 3.4. At the end of 50 seconds, we calculate the throughput achieved by the transfer; for each combination of parameters, we report the mean of 50 distinct experiments.

Figures 4.16 and 4.17 shows the results of these simulations for the wireless LAN and wireless WAN respectively. While the Snoop protocol achieves between 15% and 25% of the maximum possible link bandwidth even in the presence of significant error rates, both TCP Reno and TCP SACK achieve only between 3% and 10% of the maximum link speeds. Furthermore, the benefits

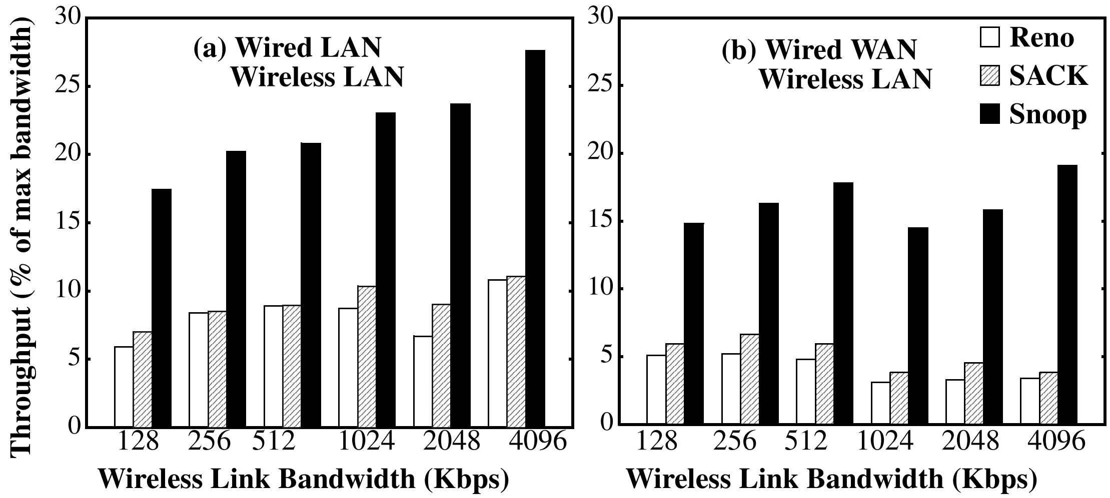

Figure 4.16 Simulated performance of TCP Reno, TCP SACK and the Snoop protocol as a function of link bandwidth of the wireless LAN. (a) shows the results of the TCP sender on the same LAN as the BS, while in (b), the sender is 100ms away from the BS.

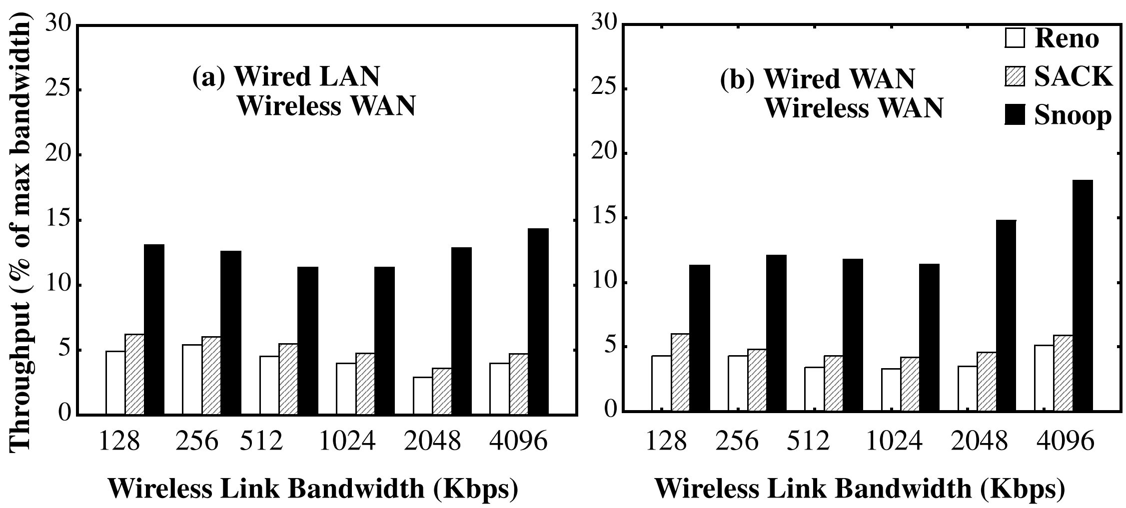

Figure 4.17 Simulated performance of TCP Reno, TCP SACK and the Snoop protocol as a function of link bandwidth of the wireless LAN. (a) shows the results of the TCP sender on the same LAN as the BS, while in (b), the sender is 100ms away from the BS.

of the Snoop protocol relative to TCP are more pronounced in the (wired or wireless) WAN case. In this case, the bandwidth-delay product of the connection is larger, making it more important for the sender to maintain as large a congestion window as possible. While the predominant benefits of the Snoop protocol in the wireless LAN case are due to its local recovery mechanisms and prevention of sender timeouts, the fact that TCP and TCP SACK frequently reduce their congestion window is not particularly problematic because of the inherently small bandwidth-delay product in this case. But this causes significant problems and results in worse performance in the WAN case.

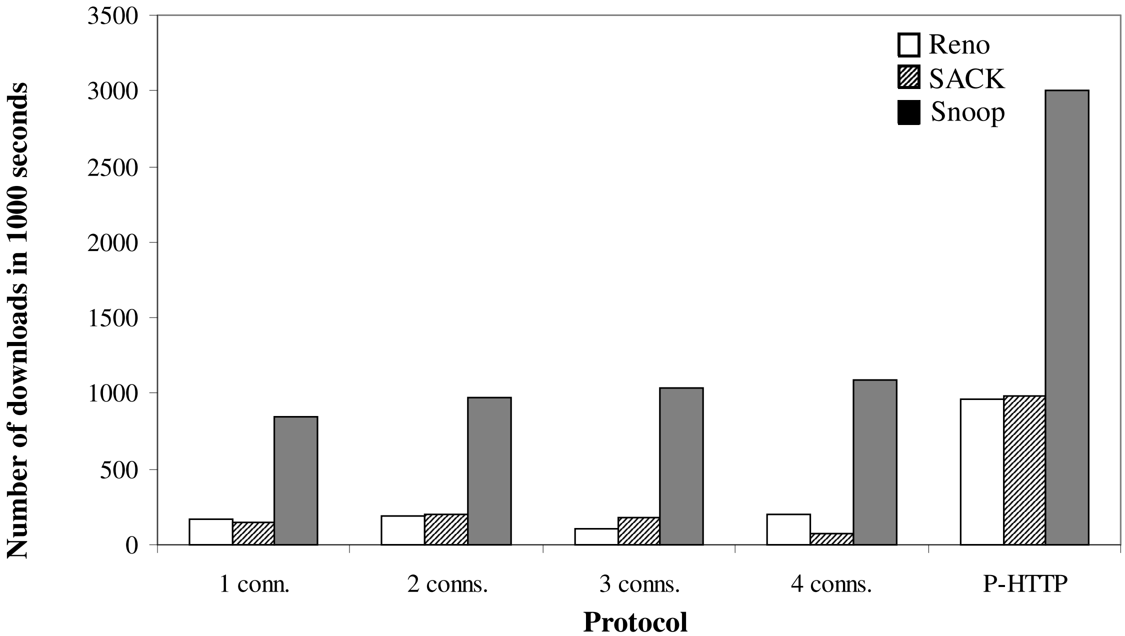

Figure 4.18 Simulated performance of the TCP Reno, TCP SACK, and the Snoop protocol on a Web workload in different protocol configurations. The graphs show the number of separate downloads (connections) using 1 to 4 concurrent connections per client, as well as the performance of persistent-HTTP [44] for the three protocols.

The same observations explain the better relative performance of the Snoop protocol at higher wireless link bandwidths.

4.8.2 Web Transfers

In the previous sections, we saw that the Snoop protocol significantly improved over TCP Reno and TCP SACK across a wide range of error rates and network topologies, for bulk transfer workloads. To investigate performance on shorter transfers, we performed a set of experiments with a Web-like access workload. We use Mah's empirical Web workload mentioned in Section 3.3 [88].

We conducted experiments using TCP Reno, TCP SACK and the Snoop protocol using the Web workload, varying the number of concurrent TCP connections from 1 to 4, as well as using persistent-HTTP [44]. The results of these experiments are shown in Figure 4.18, which depicts the number of successfully completed individual downloads (connections) in 1000 seconds. We see that the performance of the Snoop protocol is between three and six times better than the other protocols. We also note that the P-HTTP performance does not account for user think times between

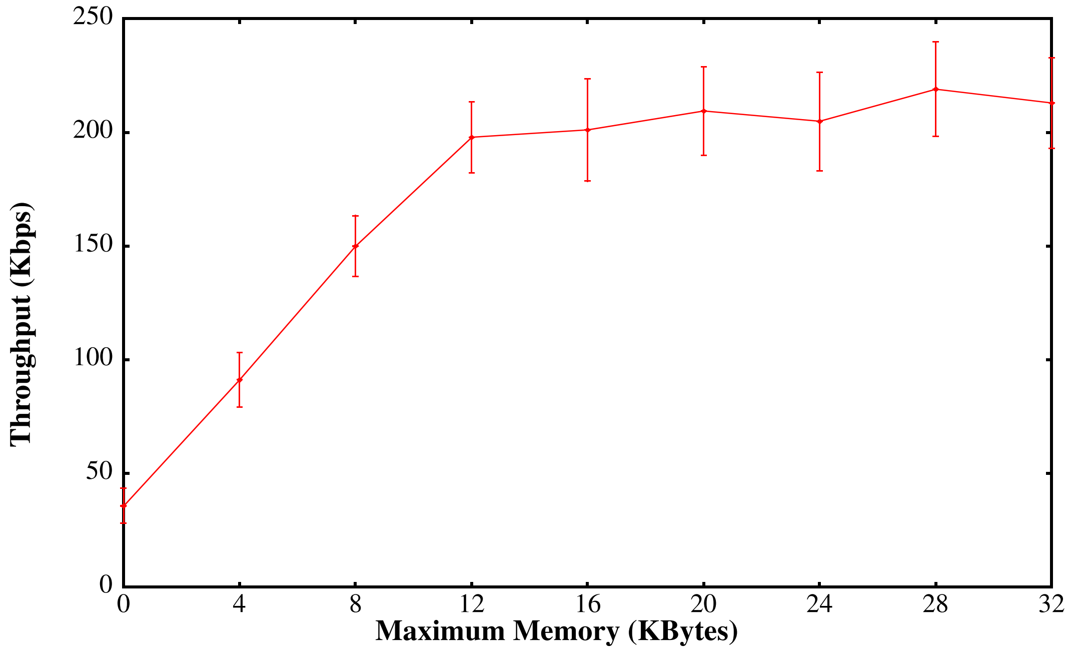

Figure 4.19 TCP connection throughput as a function of the maximum memory allocated to the snoop cache. The maximum window size of the connection is 32 KBytes. This graph shows the worst-case degradation because the bandwidth-delay product is large (50 KB).

successive downloads, and is hence an optimistic upper bound on the performance achievable in practice.

These results show that not only does the Snoop protocol perform well for large bulk transfers, but that it also results in significant performance improvements for short transfers (combined with occasional long ones) that characterize Web workloads today. The performance improvement is between a factor of three and six for this empirical Web workload under realistic wireless error conditions.

4.8.3 Scaling Behavior

In this section, we discuss the results of experiments to quantify the scaling behavior of the Snoop protocol as the number of connections grows. There are two potential sources of overhead associated with the Snoop protocol at the base station: processing overhead and memory consumption. We first show that the processing overhead is negligible and then evaluate the memory consumed by the snoop agent.

The per-packet processing required at the BS when no packet losses occur is not significantly more than without the Snoop protocol for transfers in either direction. For each incoming packet, the protocol performs a lookup in a hash table that maintains all active TCP connections keyed by source and destination address/port tuples and extract a pointer to the connection state. Then, depending on the direction, the packet is either added to a cache (without any extra memory copies) and forwarded on, or else forwarded on after a check is made for any holes in the transmission sequence. In both cases, this takes a negligible number of instructions compared to the (background) cost of routing and forwarding packets. Most of the protocol processing complexity occurs when losses occur and duplicate ACKs arrive, but our measurements show this to be a negligible component of the forwarding code time.

We now consider the question of memory consumption at the base station. For transfers originating from a mobile host, the amount of memory consumed is dependent on the number of holes in the transmission sequence and is largely independent of the transmission window size. It is also independent of the packet size, since no caching occurs in this case. For transfers to a mobile host, memory consumption is dominated by the per-packet storage cost, which we quantify below.

As the number of connections increases, the state maintained by the snoop agent increases roughly linearly. This state has two components—a small amount of control state and state required to store unacknowledged packets. The maximum amount of cache memory consumed by the snoop agent per connection is bounded by the maximum transmission window size of the connection. In our implementation, this space is not statically allocated, but is dynamic, reflecting the actual window size changes in the connection.

Thus, the key scaling question is: as the total amount of cache memory in the base station varies, how does the number of sustainable connections vary, at a given error rate? Here, the term "sustainable" is used to imply that all connections must have a noticeable performance improvement by having the Snoop protocol run at the base station, since we can always sustain as many connections as we want, limited only by the wireless link bandwidth, with the snoop agents providing no performance benefit.

We answer this question by varying the amount of per-connection cache memory at the base station, and by measuring the performance of the Snoop protocol for bulk TCP transfers as a function of the amount of cache memory. The results of this experiment, performed on 1KB sized packets, is shown in Figure 4.19. We see that the performance linearly increases with cache memory until it plateaus off at the maximum value when the cache size reaches a certain threshold value. This value is equal to the "effective" window size for the connection. We note that this curve highlights the worst-case performance degradation because the bandwidth-delay product of the connection is large (50 KB).

This linear performance degradation when cache memory is small is expected, as the following analysis shows. Clearly, the existence of the cache does not influence packets that were successfully received by the receiver. For any lost packet needing retransmission, let p be the probability of it existing in the Snoop cache. If W is the effective window size for the connection and m the available cache memory, then $p = m/W$. Now, for each lost packet in the cache, the corresponding connection throughput is B_h (cache hit contribution). Similarly, the throughput for a cache miss is $B_m < B_h$. Thus, the effective connection throughput is $pB_h + (1\text{-}p)B_m$. Since $p = m/W$, the performance is linear in m, and is constant for $m >= W$.

For the error rates observed in our experiments, performance was close to the best possible for snoop cache sizes over 12 KBytes per connection, corresponding to the effective window size. Given current technology trends and the inexpensiveness of memory, we do not expect base station memory to be a practical limitation to schemes like the Snoop protocol, especially considering the performance benefits obtained. A nice property of the soft state in the Snoop protocol is that the performance degradation is linear in the amount of memory used.

4.9 Summary

In this chapter, we presented the design and implementation of the Snoop protocol, a protocol that significantly improves the performance of TCP in single-hop cellular wireless networks, providing performance improvements up to 20 times, across a wide range of error rates.

For data transfer *to* a mobile host, the Snoop protocol improves TCP performance by deploying an agent at the base station, called the Snoop agent. This agent aids in the quick, local recovery of data loss by taking advantage of the information conveyed in TCP ACKs from the receiver. It retransmits packets locally based on duplicate ACKs and local timeouts, and suppresses certain duplicate ACKs from the sender to shield wireless losses from it. The internal state it maintains is soft, in the sense that it is not absolutely necessary for the correct end-to-end functioning of the protocol, and is very easy to regenerate and is periodically refreshed. For data transfer *from* a mobile host, the protocol uses the *Explicit Loss Notification* mechanism. These notifications are generated by the base station and sent to the mobile sender as part of the ACK stream, which reacts to them in an appropriate manner. While the Snoop protocol in the first case (transfer to a mobile host) is an example of a *transport-aware link protocol*, in the second case (transfer from a mobile host) it enhances the transport protocol to make it *link-aware*. We have also integrated the Snoop protocol with a multicast-based low-latency handoff protocol, achieving significant improvements in performance in the face of both wireless bit-errors and user mobility between cells.

In the next chapter, we present a detailed performance evaluation of the Snoop protocol and quantitatively compare it to many other alternatives. We discuss the relative benefits of several techniques used in this and other protocols, and argue that exploiting cross-layer protocol optimizations as the Snoop protocol does, is key to achieving good performance in error-prone wireless environments. We also discuss some other enhancements to the Snoop protocol in that chapter to handle wireless transit links and exploit selective acknowledgments for better loss recovery when losses occur in bursts. Finally, we discuss some key lessons we have learned from our experience with reliable data transport over error-prone wireless networks.