Appendix B

Implementing TCP Session

B.1 Introduction

Of the three components of TCP session — integrated congestion control, integrated loss recovery, and connection scheduling — integrated loss recovery is the most challenging to implement. So we focus on integrated loss recovery, with brief discussions of the other components where necessary.

B.1.1 Implementing the Loss Recovery Rules

The main challenge in implementing the loss recovery rules is determining whether an ack is a later ack for one or more segments, and keeping an updated count of later acks for each unacknowledged segment. For this purpose, we define a structure, *segment*, containing the following information for each unacknowledged segment:

1. Start and end sequence numbers (*startseq* and *endseq*).
2. Number of duplicate acks and later acks, both initialized to 0 (*dupacks* and *lateracks*).
3. Whether or not this corresponds to a retransmission (*retransflag*).
4. Pointers to the corresponding TCB and SCB (*tcb* and *scb*).

These structures, corresponding to each data segment, are linked together in a list using:

5. Pointer to the next segment in the list (*nextseg*).

The SCB contains a pointer, *seglist*, to this list. In the discussion that follows, we use *segment* (in italics) to denote the list element corresponding to a TCP data segment.

Each time a data segment is transmitted (either a fresh transmission or a retransmission), a corresponding *segment* is appended to the tail of the list. This ensures that the list is automatically maintained sorted by transmission time.

Much of the processing happens at the time an ack is received. We consider several cases for the received ack:

Case #1: *Ack for new data with no reordering across connections in the TCP session*

By "no reordering", we mean that the ack does not acknowledge any data segment that was sent more recently than the segments on other connections (if any) that have not yet been acknowledged.

Upon receipt of the ack, the sender identifies the corresponding TCB and SCB. Using the information contained in the TCB and in the ack just received, it computes *new_data_acked*, the amount of previously unacknowledged data that has now been acknowledged by this ack. It then starts scanning the *seglist* pointed to by the SCB, starting at the head. Each *segment* (belonging to the same connection as the ack) is examined to see if it has been acknowledged by this most recent ack. If yes, then the *segment* is removed from the *seglist*. It may be that the ack only acknowledges part of a segment. In such a case, the *startseq* field of the *segment* is updated to reflect the latest ack information, but the *segment* is retained in the *seglist*. In either case, a running count of the amount of acknowledged data accounted for (and removed from the *seglist*) is kept. The *session_ownd* is decremented by this amount to reflect the reduction in the amount of data that is still outstanding.

The sender continues to scan the *seglist* until the amount of acknowledged data accounted for equals *new_data_acked*. This would happen precisely when the sender has just scanned the *segment* corresponding the receiver's cumulative ack. There is no need to continue scanning because *segments* beyond this point correspond to data segments sent only after the one that triggered the ack. Since we assume that there is no reordering, the ack just received does not convey any information about these *segments*, so the scanning procedure is terminated at this point.

The amount of work to be done by the sender in this case (presumably the common case) is quite limited. At the time a segment is sent out, a small, constant number of operations are needed to append the corresponding *segment* to the tail of the *seglist*. At the time an ack arrives, the *seglist* needs to be scanned beginning at the head. However, since there is no reordering in this case, the scanning is confined to a small region close to the head of the list.

Case #2: *Ack for new data, but with reordering across connections in the TCP session*

Reordering of acks across connections refers to the situation where the sender gets back an ack for a data segment on a connection sooner than one for another segment sent prior to it on a different connection. This can happen either because of reordering of packets in the network or because of the delayed ack algorithm.

In either case, the processing done by the sender is much as in case #1. The sender scans the *seglist,* removing/updating *segments* to reflect the newly acknowledged data, until *new_data_acked* amount of data has been accounted for. In addition, however, each time a *segment* of a different connection is encountered, its *lateracks* count is incremented by 1. The chronological ordering of the *seglist* coupled with the scanning procedure ensures that such (unacknowledged) *segments* must indeed have been sent prior to the one(s) that just got acknowledged.

At the time the *lateracks* count for a *segment* is incremented, the sender checks to see if the *segment* is now eligible for retransmission according to the rules listed in Section 6.3.4.1. If it is, the *segment* is added to a separate list, *retranslist*, corresponding to data segments that are candidates for retransmission. The retransmission procedure is discussed in Section B.1.2.

Rule #3 from Section 6.3.4.1 requires checking whether at least two *segments* of the same connection have each received the threshold number of duplicate/later acks. However, such a determination can only be made once the rest of the *seglist* has been scanned. To avoid scanning the *seglist* a second time, a *segment* that satisfies all the clauses of rule #3 with the possible exception of this one (i.e., a *segment* that is the first unacknowledged one in its connection and in itself has the threshold number of duplicate/later acks) is tentatively added to the *retranslist*. At the time the retransmission procedure is invoked, a check is done to see if the segment indeed satisfies *all* of the clauses in rule #3. Only if this is so is the corresponding data segment retransmitted.

The amount of work to be done at the time a new data segment is transmitted is the same as in case #1. However, when an ack is received, more of the *seglist* may need to be scanned. The amount to be scanned depends on the extent of reordering. Studies such as [88] have shown that the extent of packet reordering (if at all any) tends to be small, so there is reason to believe that not much of the *seglist* would need to be scanned. Furthermore, this case is presumably less common than case #1.

Case #3: *Duplicate ack*

A duplicate ack[1] does not acknowledge any new data, so *new_data_acked* is set to 0. The *seglist* is scanned until the *segment* corresponding to the duplicate ack is reached. If this *segment* does not correspond to a retransmission (i.e., its *retransflag* is false), its *dupacks* count is incremented by 1. (We discuss the case where *retransflag* is true in Section B.1.2). The *lateracks* count of each *segment* (belonging to other connections) ahead of it in the *seglist* is also incremented by 1. As in case #2, if any segment becomes eligible for retransmission, it is added to the *retranslist*.

For each duplicate ack received, *session_ownd* is decremented by one MSS (maximum segment size) if the corresponding segment's *retransflag* is true. This emulates the window inflation that happens during fast recovery with standard TCP Reno.

Although the *segment* corresponding to the duplicate ack could in general be located at any place in the *seglist*, it would tend to be close to the head (especially if other connections continue to make progress, causing the corresponding *segments* to be removed from the list). So the portion of the *seglist* to be scanned would tend to be small. As we discuss in Section B.1.2, the only exception is when a duplicate ack is received for a data segment that has just been retransmitted, since in such a case, the corresponding *segment* may be close to the tail of the *seglist*.

1. Note that just as in standard TCP, a window update packet is not considered to be a duplicate ack.

B.1.2 Retransmitting Segments

After the *seglist* has been processed as discussed above, the sender scans the *retranslist* to determine if there are any data segments that should be retransmitted. As discussed in Section B.1.1, certain *segments* may been added to the *retranslist* on a tentative basis pending full compliance with rule #3. Therefore, as it scans the *retranslist*, the sender checks again to make sure that a *segment* is indeed eligible for retransmission. Since we only assume cumulative ack information from the receiver (and no selective acks), there can be no more than one *segment* eligible for retransmission in each connection.

Each eligible data segment is then retransmitted. The corresponding *segment* is removed from the *seglist*, and a new one is inserted at the tail. This helps maintain the chronological ordering of the *seglist*. The *retransflag* of the new *segment* is set to true and its *dupacks* and *lateracks* counts are set to 0. If the *dupacks* count of the original *segment* was non-zero, *session_ownd* is decremented by the corresponding multiple of MSS.

The sender also initiates congestion control action. It halves *session_cwnd* and sets *session_ssthresh* to this new value of *session_cwnd*. However, as in TCP NewReno, it is desirable to avoid initiating congestion control action repeatedly for multiple losses in a loss window. For this purpose, the sender stores a pointer, *losswin*, to the tail of the *seglist* at the time congestion control action is initiated. This information is used and updated as follows:

- Congestion control action is initiated only if the *segment* being retransmitted lies beyond[1] the *losswin* recorded the last time.
- The *losswin* is updated only if the sequence number of the segment being retransmitted lies beyond the current value of *losswin*.

Figure B.1 illustrates the importance of these steps. It depicts a TCP session with three connections. The three connections suffer a packet loss each, in turn. Figure B.1(a) shows the correct manner in which *losswin* is updated and congestion control action initiated. When *losswin* is not used at all, Figure B.1(b) shows how congestion control action could incorrectly be initiated multiple times for losses within a single loss window. Finally, Figure B.1(c) shows the danger of updat-

1. By "beyond the *losswin*", we mean being closer to the tail of the *seglist* than *losswin*.

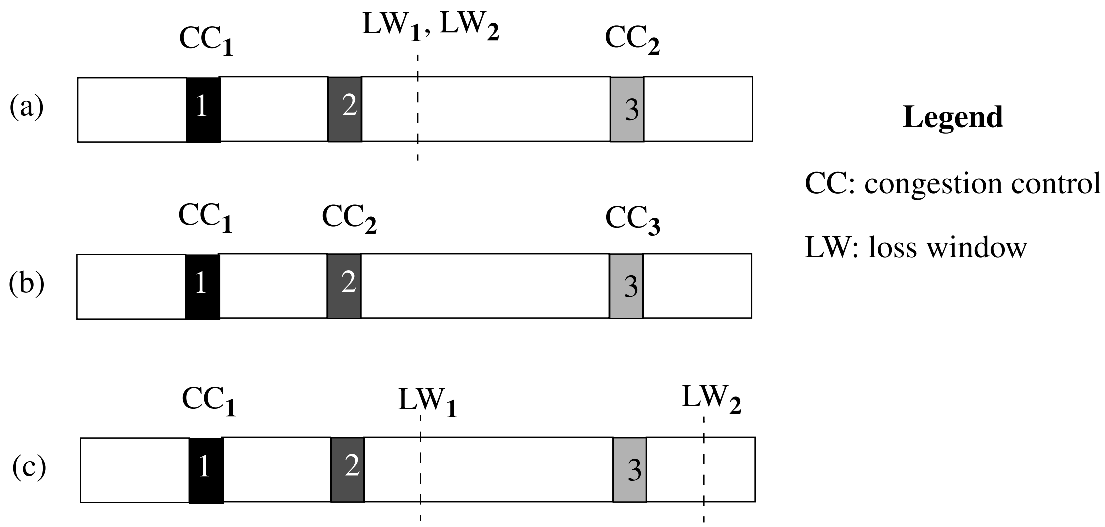

Figure B.1 An illustration of congestion control and loss window dynamics in the aftermath of three packet losses, one on each of three connections in a TCP session. (a) shows the correct set of actions — congestion control is initiated in response to loss #1 and #3, and the loss window is not updated at the time of loss #2. (b) shows the incorrect invocation congestion control in response to loss #2 because the sender does not keep track of the loss window. (c) shows the loss window being incorrectly updated in response to loss #2, causing congestion control action not to be invoked in response to loss #3.

ing *losswin* upon every retransmission, regardless of whether or not the segment being retransmitted lies beyond the old *losswin*. In such a case, congestion control action may not be initiated even when warranted.

Finally, as in TCP NewReno, a segment may become eligible for retransmission if a partial new ack is received for it. In this case, the loss recovery rules of Section 6.3.4.1 are bypassed.

B.1.3 Retransmission Timeout

A retransmission timeout is the last resort for loss recovery. A single retransmission timer is set whenever there is outstanding data on any of the connections constituting the TCP session. As in TCP, the timer is set to *srtt + 4*rttvar*. However, since *srtt* and *rttvar* are estimated by pooling together RTT samples obtained on all of the connections in the session, these are likely to be more accurate than the corresponding estimates individual connections.

Note that with TCP session, a timeout will occur only if the ack stream has dried up on *all* the connections. This coupled with the sharing of loss recovery information across connections (by way of integrated loss recovery) implies that a timeout is much less likely than with standard TCP.

Nevertheless, when a timeout does occur, *session_cwnd* is reset to 1 segment and *session_ssthresh* is set to half of the old value of *session_cwnd*. In addition, the entire *seglist* is deleted and *session_ownd* is reset to 0. Thus, the sender starts all over again with a clean slate. The first unacknowledged segment in each connection is eligible for retransmission. However, since *session_cwnd* has been reset to 1 segment, only one of the connections will be able to retransmit at the beginning. Further retransmissions happen in accordance with the connection scheduling policy.