

initial sequence number for host 2 to host 1 traffic, knowing full well that no TPDUs containing sequence number y or acknowledgements to y are still in existence. When the second delayed TPDU arrives at host 2, the fact that z has been acknowledged rather than y tells host 2 that this, too, is an old duplicate. The important thing to realize here is that there is no combination of old TPDUs that can cause the protocol to fail and have a connection set up by accident when no one wants it.

6.2.3 Connection Release

Releasing a connection is easier than establishing one. Nevertheless, there are more pitfalls than one might expect. As we mentioned earlier, there are two styles of terminating a connection: asymmetric release and symmetric release. Asymmetric release is the way the telephone system works: when one party hangs up, the connection is broken. Symmetric release treats the connection as two separate unidirectional connections and requires each one to be released separately.

Asymmetric release is abrupt and may result in data loss. Consider the scenario of Fig. 6-12. After the connection is established, host 1 sends a TPDU that arrives properly at host 2. Then host 1 sends another TPDU. Unfortunately, host 2 issues a DISCONNECT before the second TPDU arrives. The result is that the connection is released and data are lost.

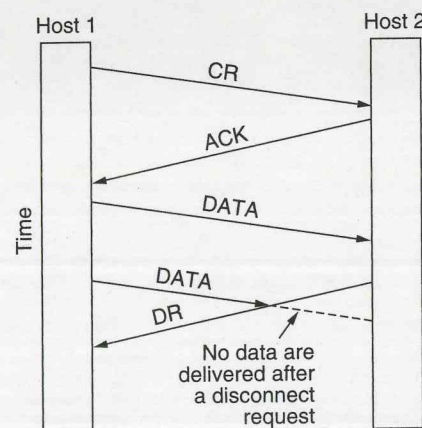


Figure 6-12. Abrupt disconnection with loss of data.

Clearly, a more sophisticated release protocol is needed to avoid data loss. One way is to use symmetric release, in which each direction is released independently of the other one. Here, a host can continue to receive data even after it has sent a DISCONNECT TPDU.

Symmetric release does the job when each process has a fixed amount of data to send and clearly knows when it has sent it. In other situations, determining that

all the work has been done and the connection should be terminated is not so obvious. One can envision a protocol in which host 1 says: I am done. Are you done too? If host 2 responds: I am done too. Goodbye, the connection can be safely released.

Unfortunately, this protocol does not always work. There is a famous problem that illustrates this issue. It is called the **two-army problem**. Imagine that a white army is encamped in a valley, as shown in Fig. 6-13. On both of the surrounding hillsides are blue armies. The white army is larger than either of the blue armies alone, but together the blue armies are larger than the white army. If either blue army attacks by itself, it will be defeated, but if the two blue armies attack simultaneously, they will be victorious.

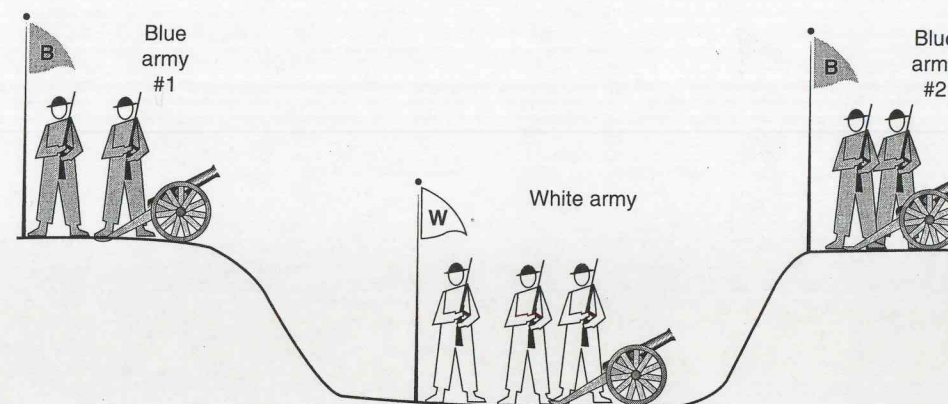


Figure 6-13. The two-army problem.

The blue armies want to synchronize their attacks. However, their only communication medium is to send messengers on foot down into the valley, where they might be captured and the message lost (i.e., they have to use an unreliable communication channel). The question is: Does a protocol exist that allows the blue armies to win?

Suppose that the commander of blue army #1 sends a message reading: "I propose we attack at dawn on March 29. How about it?" Now suppose that the message arrives, the commander of blue army #2 agrees, and his reply gets safely back to blue army #1. Will the attack happen? Probably not, because commander #2 does not know if his reply got through. If it did not, blue army #1 will not attack, so it would be foolish for him to charge into battle.

Now let us improve the protocol by making it a three-way handshake. The initiator of the original proposal must acknowledge the response. Assuming no messages are lost, blue army #2 will get the acknowledgement, but the commander of blue army #1 will now hesitate. After all, he does not know if his ac-

will detect the lack of activity and also disconnect. Of course, if this rule is introduced, it is necessary for each transport entity to have a timer that is stopped and then restarted whenever a TPDU is sent. If this timer expires, a dummy TPDU is transmitted, just to keep the other side from disconnecting. On the other hand, if the automatic disconnect rule is used and too many dummy TPDU's in a row are lost on an otherwise idle connection, first one side, then the other side will automatically disconnect.

We will not belabor this point any more, but by now it should be clear that releasing a connection without data loss is not nearly as simple as it at first appears.

6.2.4 Flow Control and Buffering

Having examined connection establishment and release in some detail, let us now look at how connections are managed while they are in use. One of the key issues has come up before: flow control. In some ways the flow control problem in the transport layer is the same as in the data link layer, but in other ways it is different. The basic similarity is that in both layers a sliding window or other scheme is needed on each connection to keep a fast transmitter from overrunning a slow receiver. The main difference is that a router usually has relatively few lines, whereas a host may have numerous connections. This difference makes it impractical to implement the data link buffering strategy in the transport layer.

In the data link protocols of Chap. 3, frames were buffered at both the sending router and at the receiving router. In protocol 6, for example, both sender and receiver are required to dedicate $MAX_SEQ + 1$ buffers to each line, half for input and half for output. For a host with a maximum of, say, 64 connections, and a 4-bit sequence number, this protocol would require 1024 buffers.

In the data link layer, the sending side must buffer outgoing frames because they might have to be retransmitted. If the subnet provides datagram service, the sending transport entity must also buffer, and for the same reason. If the receiver knows that the sender buffers all TPDU's until they are acknowledged, the receiver may or may not dedicate specific buffers to specific connections, as it sees fit. The receiver may, for example, maintain a single buffer pool shared by all connections. When a TPDU comes in, an attempt is made to dynamically acquire a new buffer. If one is available, the TPDU is accepted; otherwise, it is discarded. Since the sender is prepared to retransmit TPDU's lost by the subnet, no harm is done by having the receiver drop TPDU's, although some resources are wasted. The sender just keeps trying until it gets an acknowledgement.

In summary, if the network service is unreliable, the sender must buffer all TPDU's sent, just as in the data link layer. However, with reliable network service, other trade-offs become possible. In particular, if the sender knows that the receiver always has buffer space, it need not retain copies of the TPDU's it sends. However, if the receiver cannot guarantee that every incoming TPDU will be

accepted, the sender will have to buffer anyway. In the latter case, the sender cannot trust the network layer's acknowledgement, because the acknowledgement means only that the TPDU arrived, not that it was accepted. We will come back to this important point later.

Even if the receiver has agreed to do the buffering, there still remains the question of the buffer size. If most TPDU's are nearly the same size, it is natural to organize the buffers as a pool of identically-sized buffers, with one TPDU per buffer, as in Fig. 6-15(a). However, if there is wide variation in TPDU size, from a few characters typed at a terminal to thousands of characters from file transfers, a pool of fixed-sized buffers presents problems. If the buffer size is chosen equal to the largest possible TPDU, space will be wasted whenever a short TPDU arrives. If the buffer size is chosen less than the maximum TPDU size, multiple buffers will be needed for long TPDU's, with the attendant complexity.

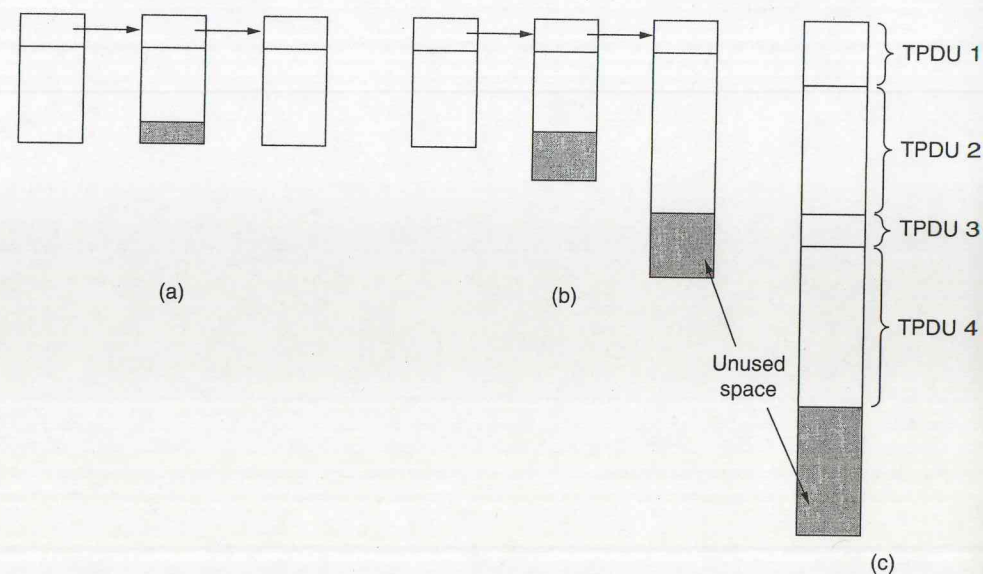


Figure 6-15. (a) Chained fixed-size buffers. (b) Chained variable-sized buffers. (c) One large circular buffer per connection.

Another approach to the buffer size problem is to use variable-sized buffers, as in Fig. 6-15(b). The advantage here is better memory utilization, at the price of more complicated buffer management. A third possibility is to dedicate a single large circular buffer per connection, as in Fig. 6-15(c). This system also makes good use of memory, provided that all connections are heavily loaded, but is poor if some connections are lightly loaded.

The optimum trade-off between source buffering and destination buffering depends on the type of traffic carried by the connection. For low-bandwidth bursty traffic, such as that produced by an interactive terminal, it is better not to