6.2 Elements of Transport Protocols

The transport service is implemented by a transport protocol used between the two transport entities. In some ways, transport protocols resemble the data link protocols we studied in detail in Chap. 3. Both have to deal with error control, sequencing, and flow control, among other issues.

However, significant differences between the two also exist. These differences are due to major dissimilarities between the environments in which the two protocols operate, as shown in Fig. 6-7. At the data link layer, two routers communicate directly via a physical channel, whereas at the transport layer, this physical channel is replaced by the entire subnet. This difference has many important implications for the protocols, as we shall see in this chapter.

Figure 6-7. (a) Environment of the data link layer. (b) Environment of the transport layer.

For one thing, in the data link layer, it is not necessary for a router to specify which router it wants to talk to—each outgoing line uniquely specifies a particular router. In the transport layer, explicit addressing of destinations is required.

For another thing, the process of establishing a connection over the wire of Fig. 6-7(a) is simple: the other end is always there (unless it has crashed, in which case it is not there). Either way, there is not much to do. In the transport layer, initial connection establishment is more complicated, as we will see.

Another, exceedingly annoying, difference between the data link layer and the transport layer is the potential existence of storage capacity in the subnet. When a router sends a frame, it may arrive...
or be lost, but it cannot bounce around for a while, go into hiding in a far corner of the world, and then suddenly emerge at an inopportune moment 30 sec later. If the subnet uses datagrams and adaptive routing inside, there is a nonnegligible probability that a packet may be stored for a number of seconds and then delivered later. The consequences of the subnet's ability to store packets can sometimes be disastrous and can require the use of special protocols.

A final difference between the data link and transport layers is one of amount rather than of kind. Buffering and flow control are needed in both layers, but the presence of a large and dynamically varying number of connections in the transport layer may require a different approach than we used in the data link layer. In Chap. 3, some of the protocols allocate a fixed number of buffers to each line, so that when a frame arrives a buffer is always available. In the transport layer, the larger number of connections that must be managed make the idea of dedicating many buffers to each one less attractive. In the following sections, we will examine all of these important issues and others.

6.2.1 Addressing

When an application (e.g., a user) process wishes to set up a connection to a remote application process, it must specify which one to connect to. (Connectionless transport has the same problem: To whom should each message be sent?) The method normally used is to define transport addresses to which processes can listen for connection requests. In the Internet, these end points are called ports. In ATM networks, they are called AAL-SAPs. We will use the generic term TSAP, (Transport Service Access Point). The analogous end points in the network layer (i.e., network layer addresses) are then called NSAPs. IP addresses are examples of NSAPs.

Figure 6-8 illustrates the relationship between the NSAP, TSAP and transport connection. Application processes, both clients and servers, can attach themselves to a TSAP to establish a connection to a remote TSAP. These connections run through NSAPs on each host, as shown. The purpose of having TSAPs is that in some networks, each computer has a single NSAP, so some way is needed to distinguish multiple transport end points that share that NSAP.
A possible scenario for a transport connection is as follows.

1. A time of day server process on host 2 attaches itself to TSAP 1522 to wait for an incoming call. How a process attaches itself to a TSAP is outside the networking model and depends entirely on the local operating system. A call such as our LISTEN might be used, for example.

2. An application process on host 1 wants to find out the time of day, so it issues a CONNECT request specifying TSAP 1208 as the source and TSAP 1522 as the destination. This action ultimately results in a transport connection being established between the application process on host 1 and server 1 on host 2.

3. The application process then sends over a request for the time.

4. The time server process responds with the current time.

5. The transport connection is then released.

Note that there may well be other servers on host 2 that are attached to other TSAPs and waiting for incoming connections that arrive over the same NSAP.

The picture painted above is fine, except we have swept one little problem under the rug: How does the user process on host 1 know that the time-of-day server is attached to TSAP 1522? One
possibility is that the time-of-day server has been attaching itself to TSAP 1522 for years and gradually all the network users have learned this. In this model, services have stable TSAP addresses that are listed in files in well-known places, such as the /etc/services file on UNIX systems, which lists which servers are permanently attached to which ports.

While stable TSAP addresses work for a small number of key services that never change (e.g. the Web server), user processes, in general, often want to talk to other user processes that only exist for a short time and do not have a TSAP address that is known in advance. Furthermore, if there are potentially many server processes, most of which are rarely used, it is wasteful to have each of them active and listening to a stable TSAP address all day long. In short, a better scheme is needed.

One such scheme is shown in Fig. 6-9 in a simplified form. It is known as the initial connection protocol. Instead of every conceivable server listening at a well-known TSAP, each machine that wishes to offer services to remote users has a special process server that acts as a proxy for less heavily used servers. It listens to a set of ports at the same time, waiting for a connection request. Potential users of a service begin by doing a CONNECT request, specifying the TSAP address of the service they want. If no server is waiting for them, they get a connection to the process server, as shown in Fig. 6-9(a).

Figure 6-9. How a user process in host 1 establishes a connection with a time-of-day server in host 2.
After it gets the incoming request, the process server spawns the requested server, allowing it to inherit the existing connection with the user. The new server then does the requested work, while the process server goes back to listening for new requests, as shown in Fig. 6-9(b).

While the initial connection protocol works fine for those servers that can be created as they are needed, there are many situations in which services do exist independently of the process server. A file server, for example, needs to run on special hardware (a machine with a disk) and cannot just be created on-the-fly when someone wants to talk to it.

To handle this situation, an alternative scheme is often used. In this model, there exists a special process called a name server or sometimes a directory server. To find the TSAP address corresponding to a given service name, such as "time of day," a user sets up a connection to the name server (which listens to a well-known TSAP). The user then sends a message specifying the service name, and the name server sends back the TSAP address. Then the user releases the connection with the name server and establishes a new one with the desired service.

In this model, when a new service is created, it must register itself with the name server, giving both its service name (typically, an ASCII string) and its TSAP. The name server records this information in its internal database so that when queries come in later, it will know the answers.
The function of the name server is analogous to the directory assistance operator in the telephone system—it provides a mapping of names onto numbers. Just as in the telephone system, it is essential that the address of the well-known TSAP used by the name server (or the process server in the initial connection protocol) is indeed well known. If you do not know the number of the information operator, you cannot call the information operator to find it out. If you think the number you dial for information is obvious, try it in a foreign country sometime.

6.2.2 Connection Establishment

Establishing a connection sounds easy, but it is actually surprisingly tricky. At first glance, it would seem sufficient for one transport entity to just send a CONNECTION REQUEST TPDU to the destination and wait for a CONNECTION ACCEPTED reply. The problem occurs when the network can lose, store, and duplicate packets. This behavior causes serious complications.

Imagine a subnet that is so congested that acknowledgements hardly ever get back in time and each packet times out and is retransmitted two or three times. Suppose that the subnet uses datagrams inside and that every packet follows a different route. Some of the packets might get stuck in a traffic jam inside the subnet and take a long time to arrive, that is, they are stored in the subnet and pop out much later.

The worst possible nightmare is as follows. A user establishes a connection with a bank, sends messages telling the bank to transfer a large amount of money to the account of a not-entirely-trustworthy person, and then releases the connection. Unfortunately, each packet in the scenario is duplicated and stored in the subnet. After the connection has been released, all the packets pop out of the subnet and arrive at the destination in order, asking the bank to establish a new connection, transfer money (again), and release the connection. The bank has no way of telling that these are duplicates. It must assume that this is a second, independent transaction, and transfers the money again. For the remainder of this section we will study the problem of delayed duplicates, with special emphasis on algorithms for establishing connections in a reliable way, so that nightmares like the one above cannot happen.
The crux of the problem is the existence of delayed duplicates. It can be attacked in various ways, none of them very satisfactory. One way is to use throw-away transport addresses. In this approach, each time a transport address is needed, a new one is generated. When a connection is released, the address is discarded and never used again. This strategy makes the process server model of Fig. 6.9 impossible.

Another possibility is to give each connection a connection identifier (i.e., a sequence number incremented for each connection established) chosen by the initiating party and put in each TPDU, including the one requesting the connection. After each connection is released, each transport entity could update a table listing obsolete connections as (peer transport entity, connection identifier) pairs. Whenever a connection request comes in, it could be checked against the table, to see if it belonged to a previously-released connection.

Unfortunately, this scheme has a basic flaw: it requires each transport entity to maintain a certain amount of history information indefinitely. If a machine crashes and loses its memory, it will no longer know which connection identifiers have already been used.

Instead, we need to take a different tack. Rather than allowing packets to live forever within the subnet, we must devise a mechanism to kill off aged packets that are still hobbling about. If we can ensure that no packet lives longer than some known time, the problem becomes somewhat more manageable.

Packet lifetime can be restricted to a known maximum using one (or more) of the following techniques:

1. Restricted subnet design.
2. Putting a hop counter in each packet.
3. Timestamping each packet.

The first method includes any method that prevents packets from looping, combined with some way of bounding congestion delay over the (now known) longest possible path. The second method consists of having the hop count initialized to some appropriate value and decremented each time the packet is forwarded. The network protocol simply discards any packet whose hop counter becomes
zero. The third method requires each packet to bear the time it was created, with the routers agreeing to discard any packet older than some agreed-upon time. This latter method requires the router clocks to be synchronized, which itself is a nontrivial task unless synchronization is achieved external to the network, for example by using GPS or some radio station that broadcasts the precise time periodically.

In practice, we will need to guarantee not only that a packet is dead, but also that all acknowledgements to it are also dead, so we will now introduce $T$, which is some small multiple of the true maximum packet lifetime. The multiple is protocol dependent and simply has the effect of making $T$ longer. If we wait a time $T$ after a packet has been sent, we can be sure that all traces of it are now gone and that neither it nor its acknowledgements will suddenly appear out of the blue to complicate matters.

With packet lifetimes bounded, it is possible to devise a foolproof way to establish connections safely. The method described below is due to Tomlinson (1975). It solves the problem but introduces some peculiarities of its own. The method was further refined by Sunshine and Dalal (1978). Variants of it are widely used in practice, including in TCP.

To get around the problem of a machine losing all memory of where it was after a crash, Tomlinson proposed equipping each host with a time-of-day clock. The clocks at different hosts need not be synchronized. Each clock is assumed to take the form of a binary counter that increments itself at uniform intervals. Furthermore, the number of bits in the counter must equal or exceed the number of bits in the sequence numbers. Last, and most important, the clock is assumed to continue running even if the host goes down.

The basic idea is to ensure that two identically numbered TPDUs are never outstanding at the same time. When a connection is set up, the low-order $k$ bits of the clock are used as the initial sequence number (also $k$ bits). Thus, unlike our protocols of Chap. 3, each connection starts numbering its TPDUs with a different initial sequence number. The sequence space should be so large that by the time sequence numbers wrap around, old TPDUs with the same sequence number are long gone. This linear relation between time and initial sequence numbers is shown in Fig. 6-10.
Figure 6-10. (a) TPDUs may not enter the forbidden region. (b) The resynchronization problem.

Once both transport entities have agreed on the initial sequence number, any sliding window protocol can be used for data flow control. In reality, the initial sequence number curve (shown by the heavy line) is not linear, but a staircase, since the clock advances in discrete steps. For simplicity we will ignore this detail.

A problem occurs when a host crashes. When it comes up again, its transport entity does not know where it was in the sequence space. One solution is to require transport entities to be idle for $T$ sec after a recovery to let all old TPDUs die off. However, in a complex internetwork, $T$ may be large, so this strategy is unattractive.

To avoid requiring $T$ sec of dead time after a crash, it is necessary to introduce a new restriction on the use of sequence numbers. We can best see the need for this restriction by means of an example. Let $T$, the maximum packet lifetime, be 60 sec and let the clock tick once per second. As shown by the heavy line in Fig. 6-10(a), the initial sequence number for a connection opened at time $x$ will be $x$.

Imagine that at $t = 30$ sec, an ordinary data TPDU being sent on (a previously opened) connection 5 is given sequence number 80. Call this TPDU $X$. Immediately after sending TPDU $X$, the host crashes and then quickly restarts. At $t = 60$, it begins reopening connections 0 through 4. At $t = 70$, it reopens connection 5, using initial sequence number 70 as required. Within the next 15 sec it sends data TPDUs 70 through 80. Thus, at $t = 85$ a new TPDU with sequence number 80 and connection 5 has been injected into the subnet. Unfortunately, TPDU $X$ still exists. If it should arrive at the receiver before the new TPDU 80, TPDU $X$ will be accepted and the correct TPDU 80 will be rejected as a duplicate.
To prevent such problems, we must prevent sequence numbers from being used (i.e., assigned to new TPDUs) for a time $T$ before their potential use as initial sequence numbers. The illegal combinations of time and sequence number are shown as the forbidden region in Fig. 6-10(a). Before sending any TPDU on any connection, the transport entity must read the clock and check to see that it is not in the forbidden region.

The protocol can get itself into trouble in two distinct ways. If a host sends too much data too fast on a newly-opened connection, the actual sequence number versus time curve may rise more steeply than the initial sequence number versus time curve. This means that the maximum data rate on any connection is one TPDU per clock tick. It also means that the transport entity must wait until the clock ticks before opening a new connection after a crash restart, lest the same number be used twice. Both of these points argue in favor of a short clock tick (a few µsec or less).

Unfortunately, entering the forbidden region from underneath by sending too fast is not the only way to get into trouble. From Fig. 6-10(b), we see that at any data rate less than the clock rate, the curve of actual sequence numbers used versus time will eventually run into the forbidden region from the left. The greater the slope of the actual sequence number curve, the longer this event will be delayed. As we stated above, just before sending every TPDU, the transport entity must check to see if it is about to enter the forbidden region, and if so, either delay the TPDU for $T$ sec or resynchronize the sequence numbers.

The clock-based method solves the delayed duplicate problem for data TPDUs, but for this method to be useful, a connection must first be established. Since control TPDUs may also be delayed, there is a potential problem in getting both sides to agree on the initial sequence number. Suppose, for example, that connections are established by having host 1 send a CONNECTION REQUEST TPDU containing the proposed initial sequence number and destination port number to a remote peer, host 2. The receiver, host 2, then acknowledges this request by sending a CONNECTION ACCEPTED TPDU back. If the CONNECTION REQUEST TPDU is lost but a delayed duplicate CONNECTION REQUEST suddenly shows up at host 2, the connection will be established incorrectly.

To solve this problem, Tomlinson (1975) introduced the three-way
handshake. This establishment protocol does not require both sides to begin sending with the same sequence number, so it can be used with synchronization methods other than the global clock method. The normal setup procedure when host 1 initiates is shown in Fig. 6-11(a). Host 1 chooses a sequence number, \( x \), and sends a CONNECTION REQUEST TPDU containing it to host 2. Host 2 replies with an ACK TPDU acknowledging \( x \) and announcing its own initial sequence number, \( y \). Finally, host 1 acknowledges host 2's choice of an initial sequence number in the first data TPDU that it sends.

**Figure 6-11. Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. (a) Normal operation. (b) Old duplicate CONNECTION REQUEST appearing out of nowhere. (c) Duplicate CONNECTION REQUEST and duplicate ACK.**

Now let us see how the three-way handshake works in the presence of delayed duplicate control TPDU. In Fig. 6-11(b), the first TPDU is a delayed duplicate CONNECTION REQUEST from an old connection.
This TPDU arrives at host 2 without host 1's knowledge. Host 2 reacts to this TPDU by sending host 1 an ACK TPDU, in effect asking for verification that host 1 was indeed trying to set up a new connection. When host 1 rejects host 2's attempt to establish a connection, host 2 realizes that it was tricked by a delayed duplicate and abandons the connection. In this way, a delayed duplicate does no damage.

The worst case is when both a delayed CONNECTION REQUEST and an ACK are floating around in the subnet. This case is shown in Fig. 6-11(c). As in the previous example, host 2 gets a delayed CONNECTION REQUEST and replies to it. At this point it is crucial to realize that host 2 has proposed using $y$ as the initial sequence number for host 2 to host 1 traffic, knowing full well that no TPDU 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 TPDU 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 acknowledgement got through, and if it did not, he knows that blue army #2 will not attack. We could now make a four-way handshake protocol, but that does not help either.

In fact, it can be proven that no protocol exists that works. Suppose that some protocol did exist. Either the last message of the protocol is essential or it is not. If it is not, remove it (and any other unessential messages) until we are left with a protocol in which every message is essential. What happens if the final message does not get through? We just said that it was essential, so if it is lost, the attack does not take place. Since the sender of the final message can never be sure of its arrival, he will not risk attacking. Worse yet, the other blue army knows this, so it will not attack.
either.

To see the relevance of the two-army problem to releasing connections, just substitute "disconnect" for "attack." If neither side is prepared to disconnect until it is convinced that the other side is prepared to disconnect too, the disconnection will never happen.

In practice, one is usually prepared to take more risks when releasing connections than when attacking white armies, so the situation is not entirely hopeless. Figure 6-14 illustrates four scenarios of releasing using a three-way handshake. While this protocol is not infallible, it is usually adequate.

**Figure 6-14. Four protocol scenarios for releasing a connection.** (a) Normal case of three-way handshake. (b) Final ACK lost. (c) Response lost. (d) Response lost and subsequent DRs lost.

In Fig. 6-14(a), we see the normal case in which one of the users...
sends a DR (DISCONNECTION REQUEST) TPDU to initiate the connection release. When it arrives, the recipient sends back a DR TPDU, too, and starts a timer, just in case its DR is lost. When this DR arrives, the original sender sends back an ACK TPDU and releases the connection. Finally, when the ACK TPDU arrives, the receiver also releases the connection. Releasing a connection means that the transport entity removes the information about the connection from its table of currently open connections and signals the connection's owner (the transport user) somehow. This action is different from a transport user issuing a DISCONNECT primitive.

If the final ACK TPDU is lost, as shown in Fig. 6-14(b), the situation is saved by the timer. When the timer expires, the connection is released anyway.

Now consider the case of the second DR being lost. The user initiating the disconnection will not receive the expected response, will time out, and will start all over again. In Fig. 6-14(c) we see how this works, assuming that the second time no TPDUs are lost and all TPDUs are delivered correctly and on time.

Our last scenario, Fig. 6-14(d), is the same as Fig. 6-14(c) except that now we assume all the repeated attempts to retransmit the DR also fail due to lost TPDUs. After $N$ retries, the sender just gives up and releases the connection. Meanwhile, the receiver times out and also exits.

While this protocol usually suffices, in theory it can fail if the initial DR and $N$ retransmissions are all lost. The sender will give up and release the connection, while the other side knows nothing at all about the attempts to disconnect and is still fully active. This situation results in a half-open connection.

We could have avoided this problem by not allowing the sender to give up after $N$ retries but forcing it to go on forever until it gets a response. However, if the other side is allowed to time out, then the sender will indeed go on forever, because no response will ever be forthcoming. If we do not allow the receiving side to time out, then the protocol hangs in Fig. 6-14(d).

One way to kill off half-open connections is to have a rule saying that if no TPDUs have arrived for a certain number of seconds, the connection is then automatically disconnected. That way, if one side ever disconnects, the other side 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 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 $\text{MAX}_\text{SEQ} + 1$ 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 in 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 TPDUs lost by the subnet, no harm is done by having the receiver drop TPDUs, 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 TPDUs 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 TPDUs 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 TPDUs 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 TPDUs, 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 dedicate any buffers, but rather to acquire them dynamically at both ends. Since the sender cannot be sure the receiver will be able to acquire a buffer, the sender must retain a copy of the TPDU until it is acknowledged. On the other hand, for file transfer and other high-bandwidth traffic, it is better if the receiver does dedicate a full window of buffers, to allow the data to flow at maximum speed. Thus, for low-bandwidth bursty traffic, it is better to buffer at the sender, and for high-bandwidth smooth traffic, it is better to buffer at the receiver.

As connections are opened and closed and as the traffic pattern changes, the sender and receiver need to dynamically adjust their buffer allocations. Consequently, the transport protocol should allow a sending host to request buffer space at the other end. Buffers could be allocated per connection, or collectively, for all the connections running between the two hosts. Alternatively, the receiver, knowing its buffer situation (but not knowing the offered traffic) could tell the sender "I have reserved X buffers for you." If the number of open connections should increase, it may be
necessary for an allocation to be reduced, so the protocol should provide for this possibility.

A reasonably general way to manage dynamic buffer allocation is to decouple the buffering from the acknowledgements, in contrast to the sliding window protocols of Chap. 3. Dynamic buffer management means, in effect, a variable-sized window. Initially, the sender requests a certain number of buffers, based on its perceived needs. The receiver then grants as many of these as it can afford. Every time the sender transmits a TPDU, it must decrement its allocation, stopping altogether when the allocation reaches zero. The receiver then separately piggybacks both acknowledgements and buffer allocations onto the reverse traffic.

Figure 6-16 shows an example of how dynamic window management might work in a datagram subnet with 4-bit sequence numbers. Assume that buffer allocation information travels in separate TPDU, as shown, and is not piggybacked onto reverse traffic. Initially, A wants eight buffers, but is granted only four of these. It then sends three TPDU, of which the third is lost. TPDU 6 acknowledges receipt of all TPDU up to and including sequence number 1, thus allowing A to release those buffers, and furthermore informs A that it has permission to send three more TPDU starting beyond 1 (i.e., TPDU 2, 3, and 4). A knows that it has already sent number 2, so it thinks that it may send TPDU 3 and 4, which it proceeds to do. At this point it is blocked and must wait for more buffer allocation. Timeout-induced retransmissions (line 9), however, may occur while blocked, since they use buffers that have already been allocated. In line 10, B acknowledges receipt of all TPDU up to and including 4 but refuses to let A continue. Such a situation is impossible with the fixed window protocols of Chap. 3.

The next TPDU from B to A allocates another buffer and allows A to continue.

**Figure 6-16. Dynamic buffer allocation.** The arrows show the direction of transmission. An ellipsis (...) indicates a lost TPDU.
Potential problems with buffer allocation schemes of this kind can arise in datagram networks if control TPDUs can get lost. Look at line 16. B has now allocated more buffers to A, but the allocation TPDPU was lost. Since control TPDUs are not sequenced or timed out, A is now deadlocked. To prevent this situation, each host should periodically send control TPDUs giving the acknowledgement and buffer status on each connection. That way, the deadlock will be broken, sooner or later.

Until now we have tacitly assumed that the only limit imposed on the sender's data rate is the amount of buffer space available in the receiver. As memory prices continue to fall dramatically, it may become feasible to equip hosts with so much memory that lack of buffers is rarely, if ever, a problem.

When buffer space no longer limits the maximum flow, another bottleneck will appear: the carrying capacity of the subnet. If adjacent routers can exchange at most $x$ packets/sec and there are $k$ disjoint paths between a pair of hosts, there is no way that those hosts can exchange more than $kx$ TPDUs/sec, no matter how much buffer space is available at each end. If the sender pushes too hard (i.e., sends more than $kx$ TPDUs/sec), the subnet will become congested because it will be unable to deliver TPDUs as fast as they are coming in.

What is needed is a mechanism based on the subnet's carrying capacity rather than on the receiver's buffering capacity. Clearly, the flow control mechanism must be applied at the sender to prevent it from having too many unacknowledged TPDUs outstanding at once. Belsnes (1975) proposed using a sliding window flow control scheme...
in which the sender dynamically adjusts the window size to match the network's carrying capacity. If the network can handle \( c \) TPDUs/sec and the cycle time (including transmission, propagation, queueing, processing at the receiver, and return of the acknowledgement) is \( r \), then the sender's window should be \( cr \). With a window of this size the sender normally operates with the pipeline full. Any small decrease in network performance will cause it to block.

In order to adjust the window size periodically, the sender could monitor both parameters and then compute the desired window size. The carrying capacity can be determined by simply counting the number of TPDUs acknowledged during some time period and then dividing by the time period. During the measurement, the sender should send as fast as it can, to make sure that the network's carrying capacity, and not the low input rate, is the factor limiting the acknowledgement rate. The time required for a transmitted TPDU to be acknowledged can be measured exactly and a running mean maintained. Since the network capacity available to any given flow varies in time, the window size should be adjusted frequently, to track changes in the carrying capacity. As we will see later, the Internet uses a similar scheme.

### 6.2.5 Multiplexing

Multiplexing several conversations onto connections, virtual circuits, and physical links plays a role in several layers of the network architecture. In the transport layer the need for multiplexing can arise in a number of ways. For example, if only one network address is available on a host, all transport connections on that machine have to use it. When a TPDU comes in, some way is needed to tell which process to give it to. This situation, called **upward multiplexing**, is shown in Fig. 6-17(a). In this figure, four distinct transport connections all use the same network connection (e.g., IP address) to the remote host.

**Figure 6-17. (a) Upward multiplexing. (b) Downward multiplexing.**
Multiplexing can also be useful in the transport layer for another reason. Suppose, for example, that a subnet uses virtual circuits internally and imposes a maximum data rate on each one. If a user needs more bandwidth than one virtual circuit can provide, a way out is to open multiple network connections and distribute the traffic among them on a round-robin basis, as indicated in Fig. 6-17(b). This modus operandi is called **downward multiplexing**. With \( k \) network connections open, the effective bandwidth is increased by a factor of \( k \). A common example of downward multiplexing occurs with home users who have an ISDN line. This line provides for two separate connections of 64 kbps each. Using both of them to call an Internet provider and dividing the traffic over both lines makes it possible to achieve an effective bandwidth of 128 kbps.

### 6.2.6 Crash Recovery

If hosts and routers are subject to crashes, recovery from these crashes becomes an issue. If the transport entity is entirely within the hosts, recovery from network and router crashes is straightforward. If the network layer provides datagram service, the transport entities expect lost TPDUs all the time and know how to cope with them. If the network layer provides connection-oriented service, then loss of a virtual circuit is handled by establishing a new one and then probing the remote transport entity to ask it which TPDUs it has received and which ones it has not received. The latter ones can be retransmitted.

A more troublesome problem is how to recover from host crashes. In particular, it may be desirable for clients to be able to continue working when servers crash and then quickly reboot. To illustrate the difficulty, let us assume that one host, the client, is sending a long file to another host, the file server, using a simple stop-and-
wait protocol. The transport layer on the server simply passes the incoming TPDUs to the transport user, one by one. Partway through the transmission, the server crashes. When it comes back up, its tables are reinitialized, so it no longer knows precisely where it was.

In an attempt to recover its previous status, the server might send a broadcast TPDU to all other hosts, announcing that it had just crashed and requesting that its clients inform it of the status of all open connections. Each client can be in one of two states: one TPDU outstanding, $S1$, or no TPDUs outstanding, $S0$. Based on only this state information, the client must decide whether to retransmit the most recent TPDU.

At first glance it would seem obvious: the client should retransmit only if and only if it has an unacknowledged TPDU outstanding (i.e., is in state $S1$) when it learns of the crash. However, a closer inspection reveals difficulties with this naive approach. Consider, for example, the situation in which the server's transport entity first sends an acknowledgement, and then, when the acknowledgement has been sent, writes to the application process. Writing a TPDU onto the output stream and sending an acknowledgement are two distinct events that cannot be done simultaneously. If a crash occurs after the acknowledgement has been sent but before the write has been done, the client will receive the acknowledgement and thus be in state $S0$ when the crash recovery announcement arrives. The client will therefore not retransmit, (incorrectly) thinking that the TPDU has arrived. This decision by the client leads to a missing TPDU.

At this point you may be thinking: ''That problem can be solved easily. All you have to do is reprogram the transport entity to first do the write and then send the acknowledgement.'' Try again. Imagine that the write has been done but the crash occurs before the acknowledgement can be sent. The client will be in state $S1$ and thus retransmit, leading to an undetected duplicate TPDU in the output stream to the server application process.

No matter how the client and server are programmed, there are always situations where the protocol fails to recover properly. The server can be programmed in one of two ways: acknowledge first or write first. The client can be programmed in one of four ways: always retransmit the last TPDU, never retransmit the last TPDU, retransmit only in state $S0$, or retransmit only in state $S1$. This
gives eight combinations, but as we shall see, for each combination there is some set of events that makes the protocol fail.

Three events are possible at the server: sending an acknowledgement ($A$), writing to the output process ($W$), and crashing ($C$). The three events can occur in six different orderings: $AC(W)$, $AWC$, $C(AW)$, $C(WA)$, $WAC$, and $WC(A)$, where the parentheses are used to indicate that neither $A$ nor $W$ can follow $C$ (i.e., once it has crashed, it has crashed). Figure 6-18 shows all eight combinations of client and server strategy and the valid event sequences for each one. Notice that for each strategy there is some sequence of events that causes the protocol to fail. For example, if the client always retransmits, the $AWC$ event will generate an undetected duplicate, even though the other two events work properly.

Figure 6-18. Different combinations of client and server strategy.

Making the protocol more elaborate does not help. Even if the client and server exchange several TPDUs before the server attempts to write, so that the client knows exactly what is about to happen, the client has no way of knowing whether a crash occurred just before or just after the write. The conclusion is inescapable: under our ground rules of no simultaneous events, host crash and recovery cannot be made transparent to higher layers.

Put in more general terms, this result can be restated as recovery from a layer $N$ crash can only be done by layer $N + 1$, and then only if the higher layer retains enough status information. As mentioned above, the transport layer can recover from failures in the network layer, provided that each end of a connection keeps track of where
This problem gets us into the issue of what a so-called end-to-end acknowledgement really means. In principle, the transport protocol is end-to-end and not chained like the lower layers. Now consider the case of a user entering requests for transactions against a remote database. Suppose that the remote transport entity is programmed to first pass TPDUs to the next layer up and then acknowledge. Even in this case, the receipt of an acknowledgement back at the user's machine does not necessarily mean that the remote host stayed up long enough to actually update the database. A truly end-to-end acknowledgement, whose receipt means that the work has actually been done and lack thereof means that it has not, is probably impossible to achieve. This point is discussed in more detail by Saltzer et al. (1984).