6

THE TRANSPORT LAYER

The transport layer is not just another layer. It is really the heart of the whole protocol hierarchy. Its task is to provide reliable, cost-effective data transport from the source machine to the destination machine, independent of the physical network or networks currently in use. Without the transport layer, the whole concept of layered protocols would make little sense. In this chapter we will study the transport layer in detail, including its design, services, and protocols.

Many networking applications need only a method to reliably transmit a stream of bits from one machine to another. For example, pipes between machines in a distributed UNIX system just need bit transport. They do not need or want any session or presentation services to get in the way. In fact, there are a surprisingly large number of applications that do not need any services from the session layer or above, but it is considered in poor taste to point this out publicly. The ARPANET does not even have session or presentation layers, and few complaints have been voiced about their absence.

This chapter will rely on the OSI model and its terminology more than any of the previous ones. This change in emphasis is due to the fact that many networks were already in operation before the OSI model was designed. These networks generally were well thought out up through the network layer, but began to get excessively fuzzy around the transport layer. As a result, a substantial body of terminology, literature, and operational experience for layers 1 through 3 was well established before the OSI model came along, and is likely to continue developing

for years to come. Most of the previous chapters have drawn heavily on this experience (e.g., the term "packet" had been long established before the OSI term "NPDU" came along). This chapter will also emphasize connection-oriented transport, since that is by far the most common type.

Starting with the transport layer, however, the pre-OSI influence has been much weaker. Only one pre-OSI transport protocol (the ARPANET's TCP) is well-established, and even that one may eventually be replaced by its OSI equivalent. Relatively little pre-OSI terminology relating to the transport layer (and almost none relating to the session and presentation layers) is in common use. Considering all these factors, this chapter will have much more of an OSI flavor than its predecessors, a harbinger of things to come. However, we will still treat the general principles first and the details of the various example protocols in a separate section at the end of the chapter, as before.

6.1. TRANSPORT LAYER DESIGN ISSUES

In this section we will provide an introduction to some of the issues that the designers of the transport layer must grapple with. They include the kind of service provided to the session layer, the quality of this service, and the transport layer primitives provided to invoke the service. Finally, we will conclude the section with an initial discussion of the protocols needed to realize the transport service.

6.1.1. Services Provided to the Session Layer

The ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective service to its users, normally entities (e.g., processes) in the session layer. To achieve this goal, the transport layer makes use of the services provided by the network layer. The hardware and/or software within the transport layer that does the work is called the **transport entity**. The relationship of the network, transport, and session layers is illustrated in Fig. 6-1.

Just as there are two types of network service, there are also two types of transport service: connection-oriented and connectionless. The connection-oriented transport service is similar to the connection-oriented network service in many ways. In both cases, connections have three phases: establishment, data transfer, and release. Addressing and flow control are also similar in both layers. Furthermore, the connectionless transport service is also very similar to the connectionless network service.

The obvious question is then: "If the transport layer service is so similar to the network layer service, why are there two distinct layers? Why is one layer not adequate?" The answer is subtle, but crucial, and goes back to Fig. 1-7. In this figure we can see that the network layer is part of the communication subnet and is run by the carrier (at least for WANs). What happens if the network layer offers

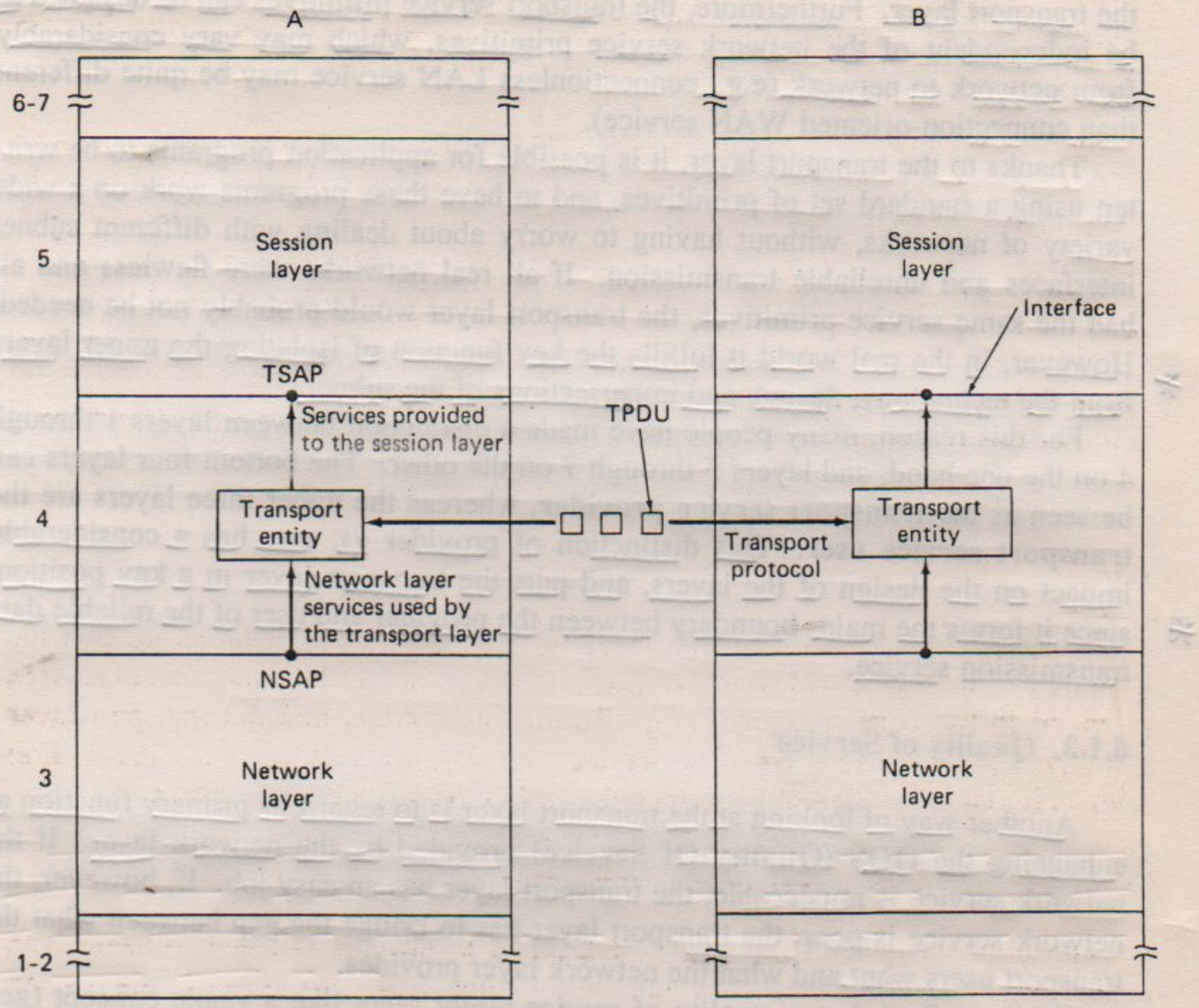


Fig. 6-1. The network, transport, and session layers.

connection-oriented service, but is unreliable? Suppose it frequently loses packets? What happens if it crashes or issues *N-RESET*s all the time?

Since the users have no control over the subnet, they cannot solve the problem of poor service by using better IMPs or putting more error handling in the data link layer. The only possibility is to put another layer on top of the network layer that improves the quality of the service. If a transport entity is informed halfway through a long transmission that its network connection has been abruptly terminated, with no indication of what has happened to the data currently in transit, it can set up a new network connection to the remote transport entity. Using this new network connection, it can send a query to its peer asking which data arrived and which did not, and then pick up from where it left off.

In essence, the existence of the transport layer makes it possible for the transport service to be more reliable than the underlying network service. Lost packets, mangled data, and even network *N-RESET*s can be detected and compensated for by

the transport layer. Furthermore, the transport service primitives can be designed to be independent of the network service primitives, which may vary considerably from network to network (e.g., connectionless LAN service may be quite different than connection-oriented WAN service).

Thanks to the transport layer, it is possible for application programs to be written using a standard set of primitives, and to have these programs work on a wide variety of networks, without having to worry about dealing with different subnet interfaces and unreliable transmission. If all real networks were flawless and all had the same service primitives, the transport layer would probably not be needed. However, in the real world it fulfills the key function of isolating the upper layers from the technology, design, and imperfections of the subnet.

For this reason, many people have made a distinction between layers 1 through 4 on the one hand, and layers 5 through 7 on the other. The bottom four layers can be seen as the **transport service provider**, whereas the upper three layers are the **transport service user**. This distinction of provider vs. user has a considerable impact on the design of the layers, and puts the transport layer in a key position, since it forms the major boundary between the provider and user of the reliable data transmission service.

6.1.2. Quality of Service

Another way of looking at the transport layer is to regard its primary function as enhancing the **QOS (Quality Of Service)** provided by the network layer. If the network service is impeccable, the transport layer has an easy job. If, however, the network service is poor, the transport layer has to bridge the gap between what the transport users want and what the network layer provides.

While at first glance, quality of service might seem like a vague concept (getting everyone to agree what constitutes "good" service is a nontrivial exercise), QOS can be characterized by a number of specific parameters. The OSI transport service allows the user to specify preferred, acceptable, and unacceptable values for these parameters at the time a connection is set up. Some of the parameters also apply to connectionless transport. It is up to the transport layer to examine these parameters, and depending on the kind of network service or services available to it, determine whether it can provide the required service. In the remainder of this section we will discuss the QOS parameters. They are summarized in Fig. 6-2.

The **connection establishment delay** is the amount of time elapsing between a transport connection being requested and the confirm being received by the user of the transport service. It includes the processing delay in the remote transport entity. As with all parameters measuring a delay, the shorter the delay, the better the service.

The **connection establishment failure probability** is the chance of a connection not being established within the maximum establishment delay time, for example, due to network congestion, lack of table space, or other internal problems.

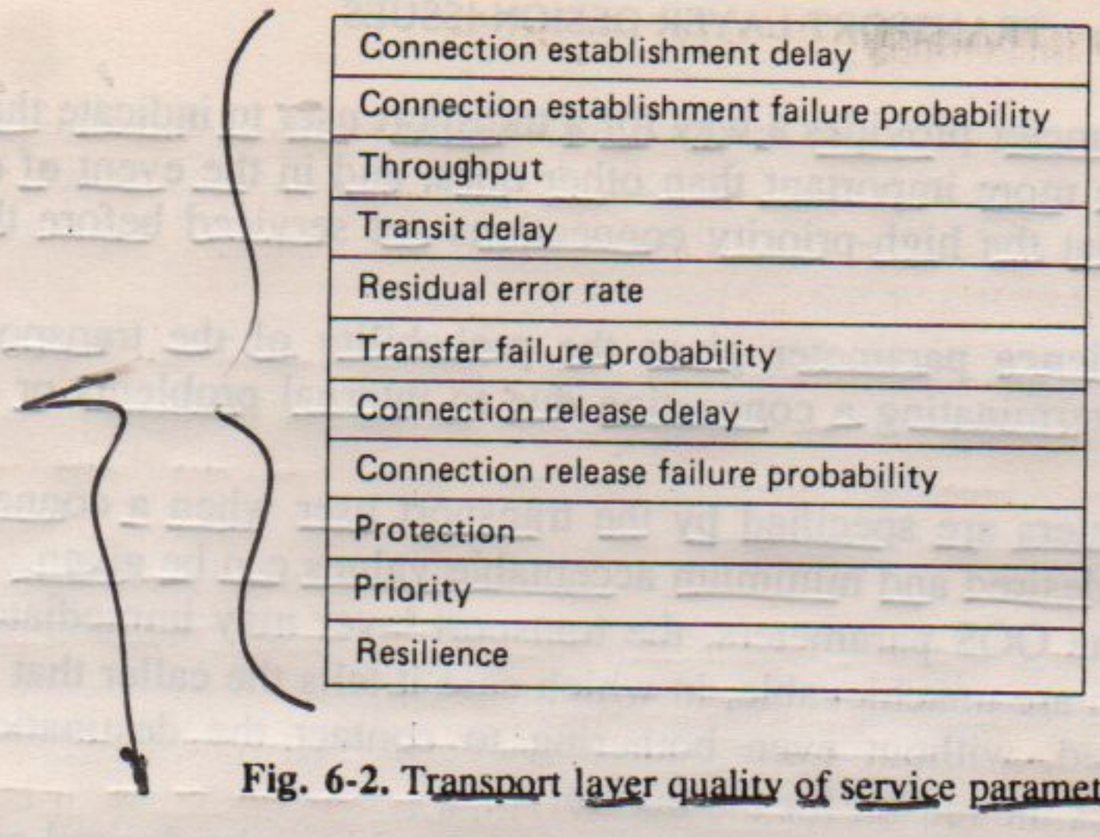


Fig. 6-2. Transport layer quality of service parameters.

The **throughput** parameter measures the number of bytes of user data transferred per second, measured over some recent time interval. The throughput is measured separately for each direction. Actually, there are two kinds of throughput: the actual measured throughput and the throughput that the network is capable of providing. The actual throughput may be lower than the network's capacity because the user has not been sending data as fast as the network is willing to accept it.

The **transit delay** measures the time between a message being sent by the transport user on the source machine and its being received by the transport user on the destination machine. As with throughput, each direction is handled separately.

The **residual error rate** measures the number of lost or garbled messages as a fraction of the total sent in the sampling period. In theory, the residual error rate should be zero, since it is the job of the transport layer to hide all network layer errors. In practice it may have some (small) finite value.

The **transfer failure probability** measures how well the transport service is living up to its promises. When a transport connection is established, a given level of throughput, transit delay, and residual error rate are agreed upon. The transfer failure probability gives the fraction of times that these agreed upon goals were not met during some observation period.

The **connection release delay** is the amount of time elapsing between a transport user initiating a release of a connection, and the actual release happening at the other end.

The **connection release failure probability** is the fraction of connection release attempts that did not complete within the agreed upon connection release delay interval.

The **protection** parameter provides a way for the transport user to specify interest in having the transport layer provide protection against unauthorized third parties (wiretappers) reading or modifying the transmitted data.

The **priority** parameter provides a way for a transport user to indicate that some of its connections are more important than other ones, and in the event of congestion to make sure that the high-priority connections get serviced before the low-priority ones.

Finally, the **resilience** parameter gives the probability of the transport layer itself spontaneously terminating a connection due to internal problems or congestion.

The QOS parameters are specified by the transport user when a connection is requested. Both the desired and minimum acceptable values can be given. In some cases, upon seeing the QOS parameters, the transport layer may immediately realize that some of them are unachievable, in which case it tells the caller that the connection attempt failed, without even bothering to contact the destination. The failure report specifies the reason for the failure.

In other cases, the transport layer knows it cannot achieve the desired goal (e.g., 1200 bytes/sec throughput), but it can achieve a lower, but still acceptable rate (e.g., 600 bytes/sec). It then sends the lower rate and the minimum acceptable rate to the remote machine, asking to establish a connection. If the remote machine cannot handle the proposed value, but it can handle a value above the minimum, it may lower the parameter to its value. If it cannot handle any value above the minimum, it rejects the connection attempt. Finally, the originating transport user is informed of whether the connection was established or rejected, and if it was established, the values of the parameters agreed upon.

This process is called **option negotiation**. Once the options have been negotiated, they remain that way throughout the life of the connection. The OSI Transport Service Definition (ISO 8072) does not give the encoding or allowed values for the QOS parameters. These are normally agreed upon between the carrier and the customer at the time the customer subscribes to the network service. To keep customers from being too greedy, most carriers have the tendency to charge more money for better quality service.

6.1.3. The OSI Transport Service Primitives

The OSI transport service primitives provide for both connection-oriented and connectionless service. The transport primitives are listed in Fig. 6-3. A comparison of Fig. 6-3 with Fig. 5-4 will show that the transport and network services are (intentionally) similar.

Despite the similarities with the network service, there are also some important differences. The main difference is that the network service is intended to model the service offered by real networks (e.g., X.25 networks), warts and all. These networks can lose packets and can spontaneously issue *N-RESET*s due to internal network problems. Thus the network service provides a way for its users to deal with acknowledgements and *N-RESET*s.

The transport service, in contrast, does not mention either acknowledgements or

T-CONNECT.request(callee, caller, exp_wanted, qos, user_data) T-CONNECT.indication(callee, caller, exp_wanted, qos, user_data) T-CONNECT.response(qos, responder, exp_wanted, user_data) T-CONNECT.confirm(qos, responder, exp_wanted, user_data)
T-DISCONNECT.request(user_data) T-DISCONNECT.indication(reason, user_data)
T-DATA.request(user_data) T-DATA.indication(user–data) T-EXPEDITED-DATA.request(user_data) T-EXPEDITED-DATA.indication(user_data)

(a)

T-UNITDATA.request(callee, caller, qos, user_data) T-UNITDATA.indication(callee, caller, qos, user_data)

(b)

Notes on terminology:
- Callee: Transport address (TSAP) to be called
- Caller: Transport address (NSAP) used by calling transport entity
- Exp_wanted: Boolean flag specifying whether expedited data will be sent
- Qos: Quality of service desired
- User_data: 0 or more bytes of data transmitted but not examined
- Reason: Why did it happen
- Responder: Transport address connected to at the destination

Fig. 6-3. (a) OSI connection-oriented transport service primitives. (b) OSI connectionless transport service primitives.

*N-RESET*s. From the transport user's point of view, the service is error-free. Of course real networks are not error-free, but that is precisely the purpose of the transport layer—to provide a reliable service on top of an unreliable network. The acknowledgements and *N-RESET*s that come from the network service are intercepted by the transport entities and the errors are recovered from by the transport protocol. If a network connection is reset, the transport layer can just establish a new one, and continue from where it left off with the old one.

Another important difference between the network service and transport service is who the services are intended for. The network service is used by the transport entities, which normally are part of the operating system or located on a special hardware board or chip. Few users write their own transport entities, and thus few users or programs ever see the bare network service.

In contrast, many users have no use for the session and presentation layers, and do see the transport primitives. As we mentioned earlier, the ARPANET does not even have session or presentation layers, so all programs that use the network interact with the transport primitives (which are different from the OSI transport primitives, but are roughly comparable). To illustrate this point, consider processes connected by pipes in UNIX. They assume the connection between them is perfect.

They do not want to know about acknowledgements or *N-RESET*s or network congestion or anything like that. What they want is a perfect connection. Process *A* puts data into one end of the pipe, and process *B* takes it out of the other. This is what the connection-oriented transport service is all about—hiding the imperfections of the network service so that user processes can just assume the existence of an error-free bit stream.

Just for the record, the situation is not quite so black-and-white as we have just sketched it. The quality of service parameter provides a large number of gray values between perfect service and perfectly awful service. Still, the basic point we have made remains: the transport service is designed to relieve programs that use it from having to worry about dealing with network errors.

Figure 6-4 shows the relationship among the OSI primitives. In each of the eight parts of the figure, one transport user is shown to the left of the double lines, the other transport user is shown to the right of the double lines, and the transport service provider (i.e., the transport layer itself) is shown between the double lines. Furthermore, time runs downward, so that events at the top occur before events at the bottom.

The normal connection setup is illustrated in Fig. 6-4(a). Four primitives are used. One of the transport entities executes a *T-CONNECT.request* primitive to signal its desire to establish a connection with the transport user attached to the **transport service access point** (**TSAP**) address named in the *CONNECT.request* primitive. This primitive results in a *T-CONNECT.indication* occurring at the destination. The transport user attached to the TSAP addressed gets the indication and can either accept it with a *T-CONNECT.response*, as shown in Fig. 6-4(a), or reject it with a *T-DISCONNECT.request*, as shown in Fig. 6-4(b). The result of an acceptance comes back to the initiator as a *T-CONNECT.confirm*. The result of a rejection comes back as a *T-DISCONNECT.indication*.

One other scenario is also possible for an attempt to establish a connection. This scenario is illustrated in Fig. 6-4(c), and occurs when the transport service provider itself rejects the connection. Such a rejection may be the transport user's fault (e.g., a bad parameter in the *T-CONNECT.request* primitive) or the transport provider's fault (e.g., the transport provider has run out of internal table space). In this scenario, nothing is transmitted across the network, so the remote site does not even hear about the failed attempt.

In Fig. 6-4(d)-(f) we see three ways a connection can be released. The normal way is that one of the parties issues a *T-DISCONNECT.request*, which is signaled to the other party as a *T-DISCONNECT.indication*. Either the calling or called party may initiate the release of the connection. If both parties simultaneously issue a *T-DISCONNECT.request* primitive, the connection is released without either side getting an indication.

Finally, the transport provider itself can terminate a connection by issuing *T-DISCONNECT.indication* primitives on both ends, as shown in Fig. 6-4(f). In a way, the last scenario is a little like the network layer issuing an *N-RESET.indication*.

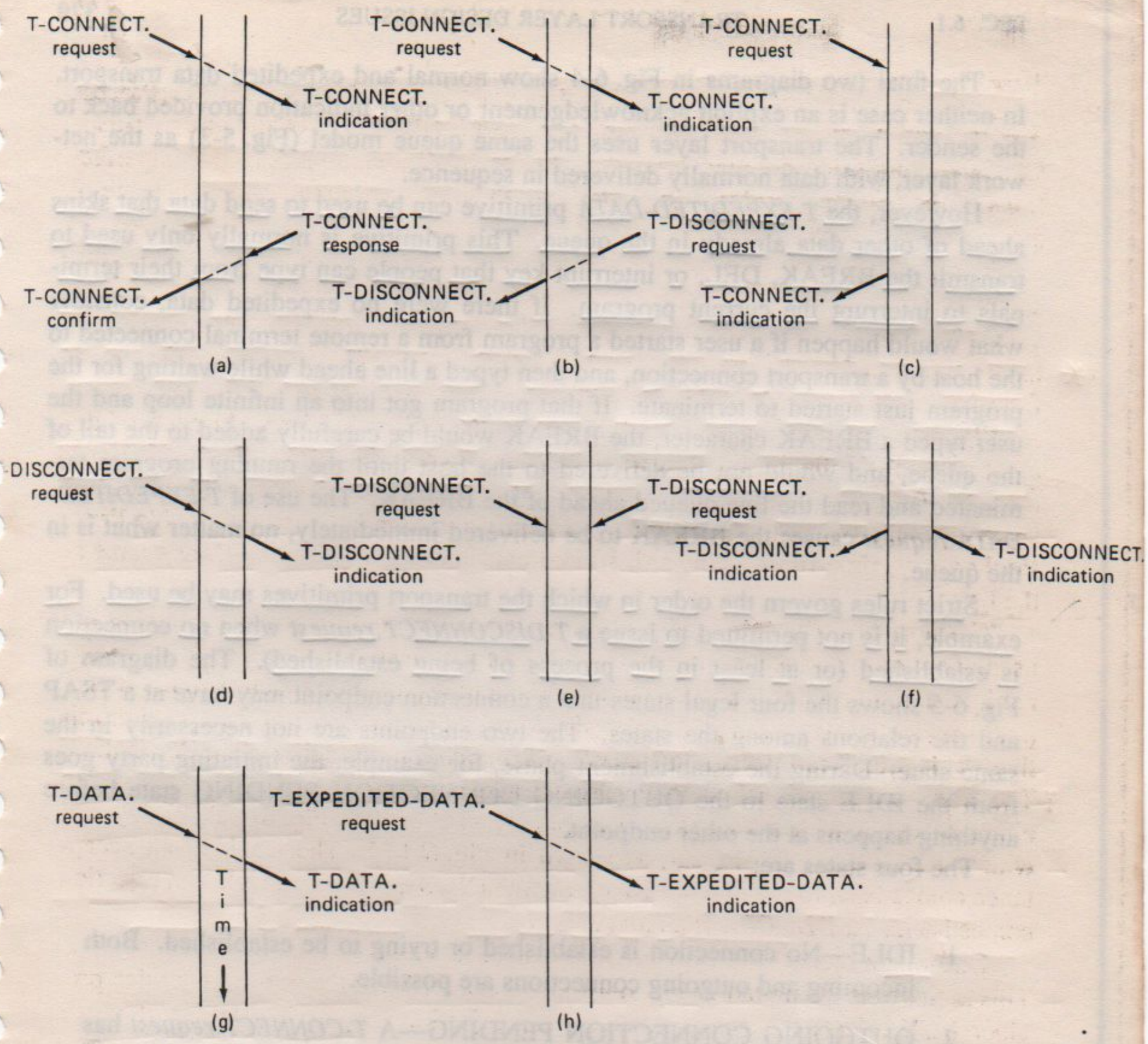


Fig. 6-4. Some valid sequences of OSI transport primitives. (a) Connection setup. (b) Connection rejected by called user. (c) Connection rejected by transport layer. (d) Normal connection release. (e) Simultaneous release by both sides. (f) Transport layer initiated release. (g) Normal data transfer. (h) Expedited data transfer.

The connection is terminated. Clearly, a well-designed transport service provider should not issue spontaneous *T-DISCONNECT.indication* primitives too lightly, but there are circumstances in which no other course of action is possible. For example, if the underlying network crashes and refuses to react to repeated attempts to communicate with it, there is little else the transport service provider can do than break all the connections. Unless the session layer has taken special precautions against this problem, the failure will have to be reported back to the highest level, and may require human intervention to retry the failed command.

The final two diagrams in Fig. 6-4 show normal and expedited data transport. In neither case is an explicit acknowledgement or other indication provided back to the sender. The transport layer uses the same queue model (Fig. 5-3) as the network layer, with data normally delivered in sequence.

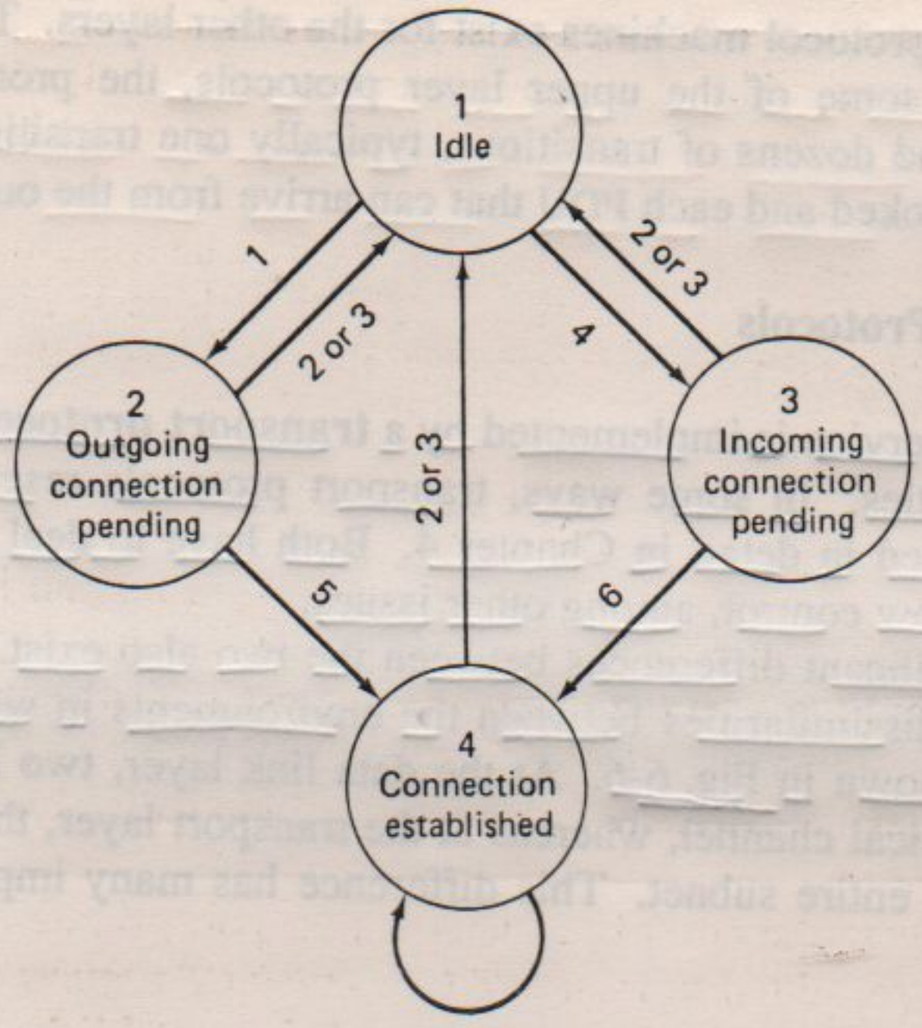
However, the *T-EXPEDITED-DATA* primitive can be used to send data that skips ahead of other data already in the queue. This primitive is normally only used to transmit the BREAK, DEL, or interrupt key that people can type from their terminals to interrupt the current program. If there were no expedited data, consider what would happen if a user started a program from a remote terminal connected to the host by a transport connection, and then typed a line ahead while waiting for the program just started to terminate. If that program got into an infinite loop and the user typed a BREAK character, the BREAK would be carefully added to the tail of the queue, and would not be delivered to the host until the running program terminated and read the line queued ahead of the BREAK. The use of *T-EXPEDITED-DATA.request* causes the BREAK to be delivered immediately, no matter what is in the queue.

Strict rules govern the order in which the transport primitives may be used. For example, it is not permitted to issue a *T-DISCONNECT.request* when no connection is established (or at least in the process of being established). The diagram of Fig. 6-5 shows the four legal states that a connection endpoint may have at a TSAP and the relations among the states. The two endpoints are not necessarily in the same state. During the establishment phase, for example, the initiating party goes from the IDLE state to the OUTGOING CONNECTION PENDING state before anything happens at the other endpoint.

The four states are:

1. IDLE—No connection is established or trying to be established. Both incoming and outgoing connections are possible.
2. OUTGOING CONNECTION PENDING—A *T-CONNECT.request* has been done. The reply from the remote peer has not yet been received.
3. INCOMING CONNECTION PENDING—A *T-CONNECT.indication* has come in. It has not yet been accepted or rejected.
4. CONNECTION ESTABLISHED—A valid connection has been established. The establishment phase is completed and data transfer can begin.

Two ways exist to get from the IDLE state to the CONNECTION ESTABLISHED state. The left-hand route (1-2-4) is used when an outgoing call is being placed. The intermediate state (2) is entered after the *T-CONNECT.request* has been issued, and holds until the *T-CONNECT.confirm* comes in. The remote side rejects the



Transitions:
1. T-CONNECT.request received from transport user
2. T-DISCONNECT.indication received from transport service provider
3. T-DISCONNECT.request received from transport user
4. T-CONNECT.indication received from transport service provider
5. T-CONNECT.confirm received from transport service provider
6. T-CONNECT.response received from transport user
7. T-DATA.request received from transport user
8. T-DATA.indication received from transport service provider
9. T-EXPEDITED-DATA.request received from transport user
10. T-EXPEDITED-DATA.indication received from transport service provider

Fig. 6-5. Transport connection endpoint states and transport primitives.

connection or the caller changes its mind and issues a *T-DISCONNECT.request*, the IDLE state is entered again.

The right-hand route (1-3-4) is used when an incoming call is received. The intermediate state is entered after the *T-CONNECT.indication* comes in. If the connection is intentionally rejected or a *T-DISCONNECT.indication* comes in (either from a fickle caller or from the transport layer itself), we return to the IDLE state. However, if the connection is accepted, state 4 is reached.

From state 4 there are two ways back to the IDLE state. These correspond to the local transport user releasing the connection and the transport layer releasing the connection. The latter event may happen either upon request of the remote transport user or of necessity by the transport service provider.

The set of states and transitions governing connection endpoints shown in Fig. 6-5 is a finite state machine. This one is called the **transport protocol**

machine. Similar protocol machines exist for the other layers. This one is particularly simple. For some of the upper layer protocols, the protocol machine has dozens of states and dozens of transitions, typically one transition for each primitive that can be invoked and each PDU that can arrive from the outside.

6.1.4. 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 Chapter 4. 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. 6-6. At the data link layer, two IMPs 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.

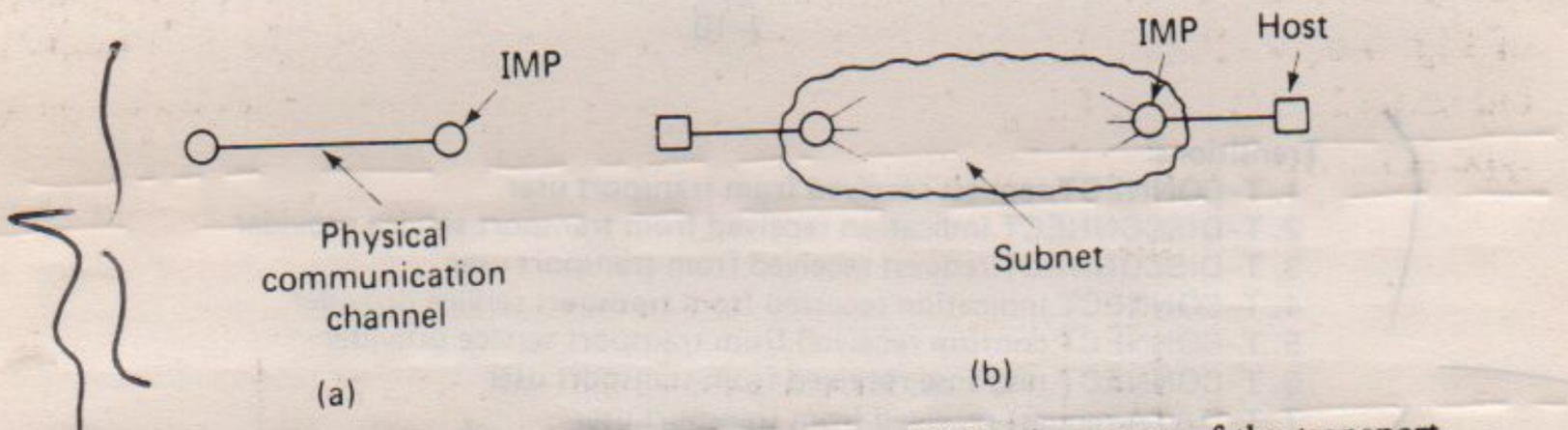


Fig. 6-6. (a) Environment of the data link layer. (b) Environment of the transport layer.

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

For another thing, the process of establishing a connection over the wire of Fig. 6-6(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 an IMP 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 this ability of the subnet to store packets can be disastrous, and 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 Chapter 4, some of the protocols allocate a fixed number of buffers to each line, so that when a frame arrives there is always a buffer 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.

From the transport protocol designer's point of view, the actual properties of the subnet (Fig. 6-6) are less important than the *service* offered by the network layer, although the latter is, of course, strongly influenced by the former. To some extent, however, the network layer service may mask the least desirable aspects of the subnet and provide a better interface.

For the purposes of studying transport protocols, we will group the various kinds of network services into three categories as shown in Fig. 6-7. The first category, type A, consists of service that is essentially perfect. The fraction of packets lost, duplicated, or garbled is negligible. *N-RESETs* are so rare that they can be ignored. In effect, the environment in which the transport layer operates is that of Fig. 6-6(a). Public WANs offering type A service are scarcer than hen's teeth, but some LANs come fairly close. The transport protocols needed to operate over a type A network are straightforward.

Network type	Description
A	Flawless, error-free service with no N-RESETS
B	Perfect packet delivery, but with N-RESETS
C	Unreliable service with lost and duplicated packets and possibly N-RESETS

Fig. 6-7. Types of service that can be offered by the network layer.

A more common situation for WANs is type B service. Individual packets are rarely, if ever, lost (because the network and data link layer protocols recover from these losses transparently) but from time to time the network layer issues *N-RESETs*, either due to internal congestion, hardware problems, or software bugs. It is then up to the transport protocol to pick up the pieces, establish a new network connection, resynchronize, and continue, so that the *N-RESET* is completely hidden from the transport user. Most public X.25 networks are type B. Transport protocols for type B networks are more complex than those for type A.

The third type is the network service that is not reliable enough to be trusted at all. WANs offering pure connectionless (datagram) service, packet radio networks, and many internetworks fall into this category. Transport protocols that must live

with type C service are the most complex of all, and must solve all the problems that we saw in the data link layer, and many more as well.

Thus different transport protocols will be needed for different situations. The worse the network service, the more complex the transport protocol. OSI has recognized this problem, and devised a transport protocol with five variants, as listed in Fig. 6-8.

Protocol class	Network type	Name
0	A	Simple class
1	B	Basic error recovery class
2	A	Multiplexing class
3	B	Error recovery and multiplexing class
4	C	Error detection and recovery class

Fig. 6-8. Transport protocol classes.

Class 0 is the simplest class. It sets up a network connection for each transport connection requested and assumes the network connection does not make errors. The transport protocol does no sequencing or flow control, relying on the underlying network layer to get everything right. It does however, provide mechanisms for establishing and releasing transport connections.

Class 1 is like class 0 except that it has been designed to recover from *N-RESET*s. If the network connection being used for a given transport connection is ever subject to an *N-RESET*, the two transport entities resynchronize and continue from where they left off. To achieve resynchronization, they must use and keep track of sequence numbers, something not required with class 0. Other than the ability to recover from *N-RESET*s, class 1 does not provide any error control or flow control on top of what the network layer itself provides.

Class 2, like class 0 is designed to be used with reliable networks (type A). It differs from class 0 in that two or more transport connections may be sent (multiplexed) over the same network connection. This feature is useful when there are many transport connections open, each with relatively little traffic, and the carrier has a high charge for connect time for each open network connection. For example, in an office full of airline reservation terminals, each terminal might have a separate transport connection to a remote computer, with all the transport connections going over one (or a few) network connections, to reduce networking costs. We will look at multiplexing in more detail later in this chapter.

Class 3 combines the features of classes 1 and 2. It allows multiplexing and can also recover from *N-RESET*s. It also uses explicit flow control.

Class 4 is designed for type C network service. It is completely paranoic and takes Murphy's Law (If something can go wrong, it will) as a given. It must therefore be able to handle lost, duplicate, and garbled packets. *N-RESET*s, and everything else the network can throw at it. Needless to say, class 4 protocols are much more complex than the other ones. We will study some of the problems that class 4 protocols must deal with later in this chapter.

It should be pointed out that having a simple-minded connectionless network service and putting all the complexity in the transport protocol is not necessarily a bad idea. Doing a lot of work to make the service almost reliable in the data link and network layers, and then discovering that it is not quite good enough to satisfy the transport layer, so that class 4 must be used there anyway, may be inefficient.

The choice of which protocol class will be used on any given connection is determined by the transport entities at the time the connection is established. The initiator can propose a preferred class, and zero or more alternative classes. The responder then chooses the protocol class to use from the list supplied. If none of the choices offered are acceptable, the connection is rejected.

This negotiation is required because different users may have different concepts of what "reliable" means. To a casual user behind a terminal who wants to access a remote computer to see if any electronic mail has arrived, a network that loses an average of one packet per week may well be considered a type A (perfect) network, and the class 0 protocol may be sufficient. A bank doing multimillion dollar funds transfers all day long may view the same network as definitely type C, and may insist upon the heavy-duty class 4 protocol. Empirical studies of the transport protocols have been made by Meister (1987) and Cole and Lloyd (1986). *

6.1.5. Elements of Transport Protocols

The exact features provided by a transport protocol depend on the environment in which it operates (e.g., the type of network service available) and the type of service it must provide. Nevertheless, it is possible to give a list of basic elements which are common to many transport protocols. Figure 6-9 gives such a list, and furthermore shows which features are applicable to each of the five OSI protocol classes. This list should not be taken too literally because the details of the features sometimes differ among the various protocol classes, and not all the alternatives and variants are mentioned in the list.

In the following paragraphs we will briefly discuss each of the protocol elements listed in Fig. 6-9. All connection-oriented protocols must provide a mechanism for establishing connections. Furthermore all of them provide a way for the called party to accept or refuse a requested connection. Establishing and releasing connections will be discussed in detail in the next section.

To actually move bits across the network, the transport entities normally establish a network connection, and keep track of the mapping between transport

Protocol element	Class 0	1	2	3	4
Connection establishment	x	x	x	x	x
Connection refusal	x	x	x	x	x
Assignment to network connection	x	x	x	x	x
Splitting long messages into TPDUs	x	x	x	x	x
Association of TPDUs with connection	x	x	x	x	x
TPDU transfer	x	x	x	x	x
Normal release	x	x	x	x	x
Treatment of protocol errors	x	x	x	x	x
Concatenation of TPDUs to the user		x	x	x	x
Error release	x		x		
TPDU numbering		x	o	x	x
Expedited data transfer		o	o	x	x
Transport layer flow control			o	x	x
Resynchronization after a RESET		x		x	x
Retention of TPDUs until ack		x		x	x
Reassignment after network disconnect		x		x	x
Frozen references		x		x	x
Multiplexing			x	x	x
Use of multiple network connections					x
Retransmission upon timeout					x
Resequencing of TPDUs					x
Inactivity timer					x
Transport layer checksum					o

x = present o = optional (blank) = absent

Fig. 6-9. Elements of transport protocols and their relationships to the five OSI connection-oriented transport protocol classes.

connections and network connections. However, it is also possible for the transport entities to use a connectionless network protocol for data transport, provided that the transport protocol is class 4 (or a non-OSI protocol with the same functionality).

Before going on to the next item, let us first say a few words about terminology. When discussing the data link layer we called the units exchanged "frames." In the network layer we called them "packets." Both of these terms are widely used (e.g., in the CCITT X.25 recommendation). For the "transport packet" there is no comparable word, so we will use the OSI term **TPDU** (**Transport Protocol Data Unit**). We will call the item of information passed by the transport user to the transport provider a **message** since the OSI term, **TSDU** (**Transport Service Data Unit**), is rarely used. In some cases, the distinction between a message and a TPDU is not important, in which case we will use whichever term seems most appropriate in the context.

The messages to be transmitted may be of any length, so it is up to the transport

layer to split them into TPDUs for transport. If a TPDU does not fit in a single packet, each TPDU may have to be split as well. One of the tasks of all five OSI protocol classes is to split long messages into the TPDU size used by the protocol, and then reassemble the pieces transparently at the other end.

If multiple connections are open on a machine, the transport entities will have to give each connection a number, and put the connection number in each TPDU, so that when a TPDU arrives at the other end, the receiving transport entity will know which connection to associate it with. Needless to say, transport of TPDUs is also a feature of all transport protocols.

Normal release of a connection is also found in all protocols, although it works slightly differently in class 0. In this class, there is always a one-to-one mapping between transport connections and network connections. The transport connection is released implicitly by just releasing the underlying network connection. With the other classes, releasing is explicit, by exchange of control TPDUs.

All protocols must deal with protocol errors. If an invalid TPDU arrives, there must be rules governing what to do. In some cases the action may be to ignore it; in others it may be to release some connection (or all connections). Protocol errors are not supposed to happen, of course, but allowing a transport entity just to crash if one occurs is not a good idea.

The remaining items on the list do not apply to all five OSI protocol classes. For example, concatenation of TPDUs applies to classes 1 through 4, but not 0. This feature allows the transport entity to collect several TPDUs and send them together as a single packet, thus reducing the number of calls to the network layer.

Error release refers to the fact that for protocol classes 0 and 2, an *N-RESET* or *N-DISCONNECT* terminates the transport connection(s) using that network connection. No attempt is made to recover.

TPDU numbering is used to keep track of the TPDUs. By assigning successive TPDUs on a connection successively higher sequence numbers, explicit acknowledgements and flow control are possible, and there is a way to figure out after an *N-RESET* which TPDU was the last one received. Class 0 (and optionally class 2) do not use sequence numbers.

Expedited data transfer is a possibility in the four upper protocol classes, but is not available in class 0.

Transport layer flow control consists of having an explicit part of the transport protocol dealing with how many TPDUs may be sent at any instant. A sliding window scheme can be used, but there are also other possibilities. If no explicit flow control scheme is used at the transport layer, the underlying flow control of the network connection is used.

Resynchronization after an *N-RESET* is done in classes 1, 3, and 4 to allow each side to discover which of the TPDUs it sent have arrived. Closely related to resynchronization is the necessity for transport entities to retain copies of TPDUs sent until they have been acknowledged, so that they can be retransmitted in the event of an *N-RESET*. Since classes 0 and 2 give an error release after an *N-RESET*, rather

than attempting to resynchronize, they do not have to handle retention of TPDUs until they are acknowledged.

Reassignment after a network disconnect is related to the above problems. If the network breaks the connection altogether, rather than just resetting it, then it is up to the transport layer to establish a new connection on which to work.

Frozen references are important in networks that can store packets for a non-negligible time. The idea here is to refrain from giving a TPDU an identification that is identical to that of an older TPDU that is still in existence, just in case the old one pops up in an unexpected moment. We will discuss this subtle matter in detail later.

Next come multiplexing and use of multiple network connections. Both of these have to do with having several transport connections on one network connection or vice versa. We will also study these later.

Retransmission upon timeout is only needed in class 4, because this is the only class in which lost packets are common enough to require error control in the transport layer. We saw how the timer mechanism worked in the data link layer. It is essentially the same in the transport layer, although choosing good timeout values is more difficult (Karn and Partridge, 1987).

If TPDUs may be lost (class 4 again), the destination may have received TPDUs in the wrong order, and has to put them together again. This mechanism was also present in protocol 6 in Chapter 4.

The inactivity timer is different from the timer used to detect lost TPDUs. It is used to detect a dead network connection. If there is no sign of life from the network layer for the period of time governed by this timer, despite repeated attempts to communicate with it, the transport layer must assume something is radically wrong. The normal recovery procedure is to try to establish a new network connection.

Finally, the last item is the use of software checksums. TPDUs can be checksummed in software, to guard against networks whose own lower layer checksumming is inadequate. The checksum algorithm (Fletcher, 1982) has been designed to be easy to compute in software (basically, adding up the bytes modulo 256).

6.2. CONNECTION MANAGEMENT

As we mentioned before, in many ways transport protocols resemble data link protocols. Sliding window protocols, for example, can be used in both layers. One significant difference, however, is how connections are managed. In the data link layer, there is little connection management. The lines between the IMPs are always there and always ready for use. In the transport layer, the establishment, release, and general management of connections is much more complex. In this section we will look at some of the issues surrounding connection management in detail. We will see that special protocols are sometimes needed.

6.2.1. Addressing

When a (transport) user wishes to set up a connection to another user, he must specify which remote user to connect to. (Connectionless transport has the same problem: to whom should each message be sent?) The method normally used is to define transport service access points (TSAPs) to which processes can attach themselves and wait for connection requests to arrive. TSAPs are completely analogous to the NSAPs we saw in the previous chapter, only they exist at the top of the transport layer, rather than at the top of the network layer.

Figure 6-10 illustrates the relationship between the NSAP, TSAP, network connection, and transport connection. A possible connection scenario is as follows.

1. A time-of-day server process on machine *B* attaches itself to TSAP 122 to wait for a *T-CONNECT.indication*. How a process attaches itself to a TSAP is outside the OSI model and depends entirely on the local operating system.
2. A process on machine *A* wants to find out the time-of-day, so it issues a *T-CONNECT.request* specifying TSAP 6 as the source and TSAP 122 as the destination.
3. The transport entity on *A* selects an NSAP on its machine and on the destination machine, and sets up a network connection (e.g., an X.25 virtual circuit) between them. Using this network connection, it can talk to the transport entity on *B*.
4. The first thing the transport entity on *A* says to its peer on *B* is: "Good morning. I would like to establish a transport connection between my TSAP 6 and your TSAP 122. What do you say?"
5. The transport entity on *B* then issues the *T-CONNECT.indication*, and if the time-of-day server at TSAP 122 is agreeable, the transport connection is established.

Note that the transport connection goes from TSAP to TSAP, whereas the network connection only goes part way, from NSAP to NSAP (e.g., from X.121 address to X.121 address).

The picture painted above is fine, except we have swept one little problem under the rug: How does the user process on *A* know that the time-of-day server is attached to TSAP 122? One possibility is that the time-of-day server has been attaching itself to TSAP 122 for years, and gradually all the network users have learned this. In this model, services have stable TSAP addresses which can be printed on paper and given to new users when they join the network.

While stable TSAP addresses might work for a small number of key services

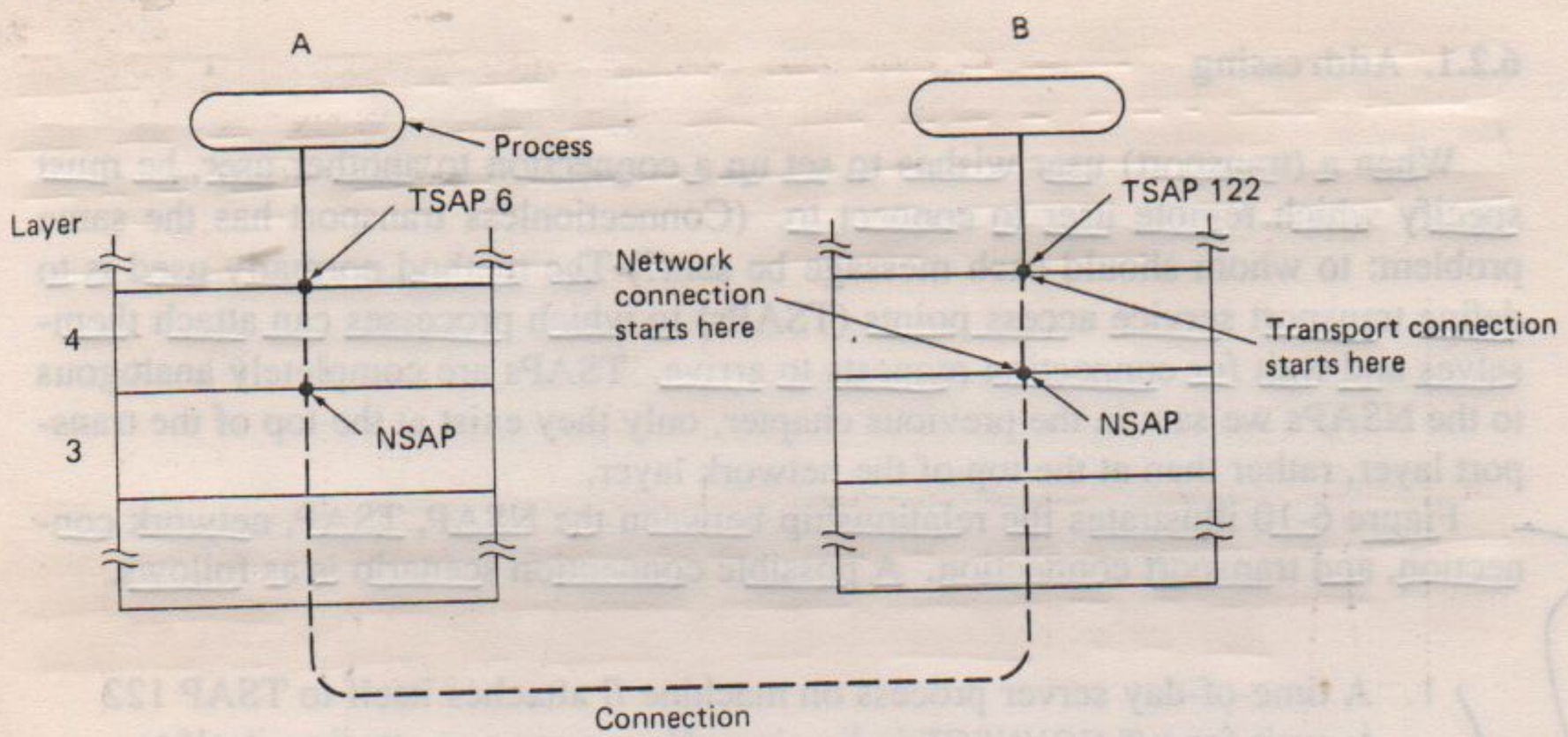


Fig. 6-10. TSAPs, NSAPs, and connections.

that never change, in general, user processes 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, pioneered in the ARPANET, is shown in Fig. 6-11 in a simplified form. It is known as the ARPANET **initial connection protocol**. Instead of every conceivable server listening at a well-known TSAP, each machine that wishes to offer service to remote users has a special **process server** (or **logger**) through which all services must be requested. Whenever the process server is idle, it listens to a well-known TSAP. Potential users of any service must begin by doing a *T-CONNECT.request*, specifying the TSAP address of the process server.

Once the connection has been established, the user sends the process server a message telling which program it wants to run (e.g., the time-of-day program). The process server then chooses an idle TSAP and spawns a new process, telling the new process to listen to the chosen TSAP. Finally, the process server sends the remote user the address of the chosen TSAP, terminates the connection, and goes back to listening on its well-known TSAP.

At this point the new process is listening on a TSAP that the user now knows, so it is possible for the user to release the connection to the process server and connect to the new process. Once this connection has been set up, the new process executes the desired program, the name of which was passed to it by the process server, together with address of the TSAP to listen to. When the server has performed its job, it releases the connection and terminates itself.

While the ARPANET 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

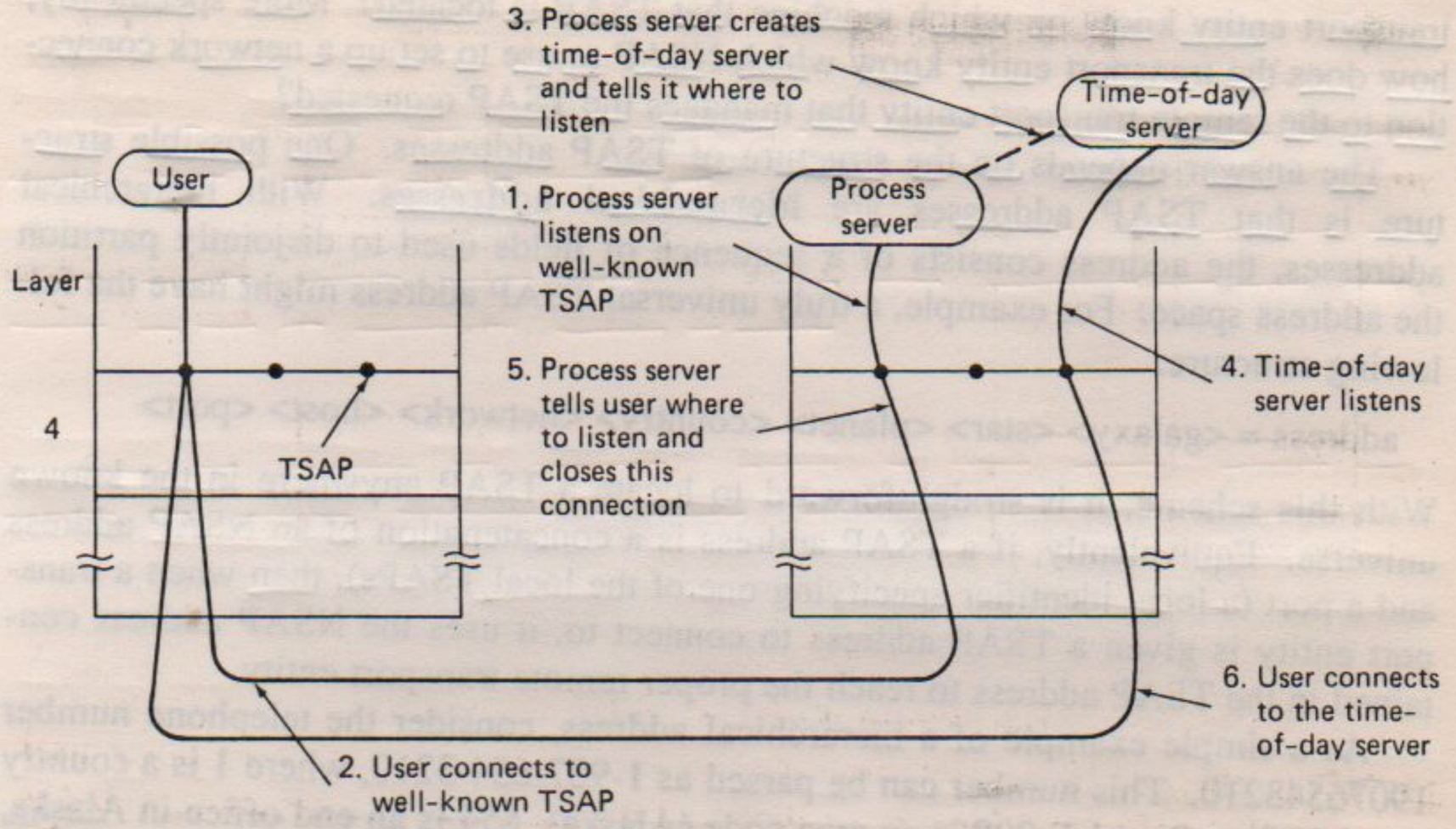


Fig. 6-11. How a user process in host *A* establishes a connection with a time-of-day server in host *B*.

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 the address of its TSAP. The name server records this information in its internal data base, 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 some time.

Now let us suppose that the user has successfully located the address of the TSAP to be connected to. Another interesting question is how does the local

transport entity know on which machine that TSAP is located? More specifically, how does the transport entity know which NSAP to use to set up a network connection to the remote transport entity that manages the TSAP requested?

The answer depends on the structure of TSAP addresses. One possible structure is that TSAP addresses are **hierarchical addresses**. With hierarchical addresses, the address consists of a sequence of fields used to disjointly partition the address space. For example, a truly universal TSAP address might have the following structure:

address = <galaxy> <star> <planet> <country> <network> <host> <port>

With this scheme, it is straightforward to locate a TSAP anywhere in the known universe. Equivalently, if a TSAP address is a concatenation of an NSAP address and a port (a local identifier specifying one of the local TSAPs), then when a transport entity is given a TSAP address to connect to, it uses the NSAP address contained in the TSAP address to reach the proper remote transport entity.

As a simple example of a hierarchical address, consider the telephone number 19076543210. This number can be parsed as 1-907-654-3210, where 1 is a country code (U.S. + Canada), 907 is an area code (Alaska), 654 is an end office in Alaska, and 3210 is one of the "ports" (subscriber lines) in that end office.

The alternative to a hierarchical address space is a **flat address space**. If the TSAP addresses are not hierarchical, a second level of mapping is needed to locate the proper machine. There would have to be a name server that took TSAP addresses as input and returned NSAP addresses as output. Alternatively, in some situations, it might be possible to broadcast a query asking the destination machine to please identify itself.

6.2.2. Establishing a Connection

Establishing a connection sounds easy, but it is actually surprisingly tricky, especially in a type C network. At first glance, it would seem sufficient for one transport entity to just send a *CR* (*CONNECTION REQUEST*) TPDU to the destination and wait for a *CC* (*CONNECTION CONFIRM*) reply. The problem occurs when the network can lose, store, and duplicate packets.

Imagine a subnet that is so congested that acknowledgements never get back in time, and each packet times out and is retransmitted two or three times. Suppose the subnet uses datagrams inside, and every packet follows a different route. Some of the packets might get stuck in traffic jams 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, assumes 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 throwaway TSAP addresses. In this approach, each time a TSAP address is needed, a new, unique address is generated, typically based on the current time. When a connection is released, the addresses are discarded forever. This strategy makes the process server model of Fig. 6-11 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 came 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 very old packets that are still wandering 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 of the following techniques:

1. Restricted subnet design.
2. Putting a hop counter in each packet.
3. Time stamping 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 incremented each time the packet is forwarded. The data link protocol simply discards any packet whose hop counter has exceeded a certain value. The third method requires each packet to bear the time it was created, with the IMPs agreeing to discard any packet older than some agreed upon time. This latter method requires the IMP

clocks to be synchronized, which itself is a nontrivial task unless synchronization is achieved external to the network, for example by listening to WWV or some other radio station that broadcasts the exact 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).

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 Chapter 4, each connection starts numbering its TPDUs with a different sequence number. The sequence space should be so large (e.g., 32 bits) 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. 6-12.

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 really 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 in Fig. 6-12, the initial sequence number for a connection opened at time x will be x. Imagine that at $t = 30$ sec, an

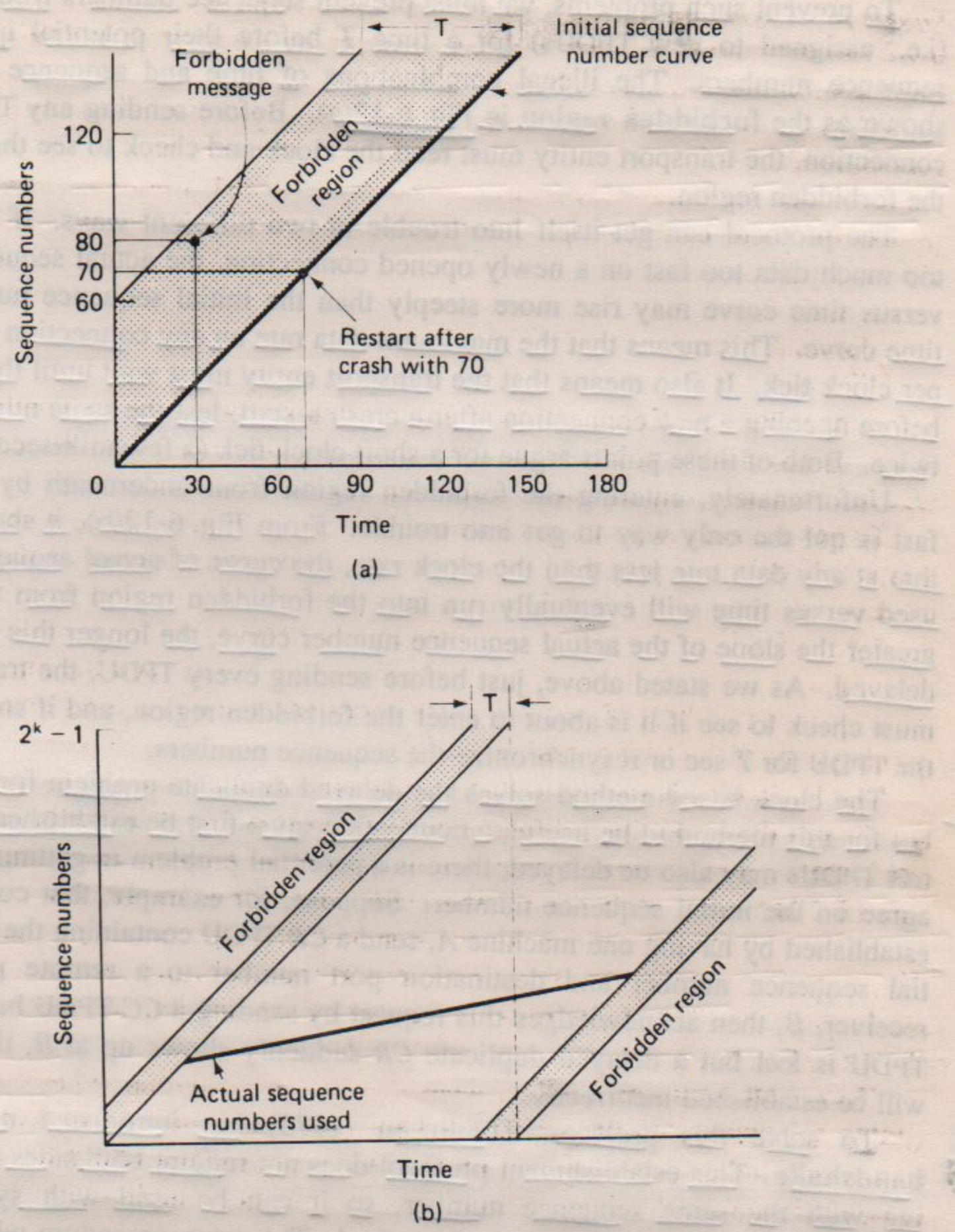


Fig. 6-12. (a) TPDUs may not enter the forbidden region. (b) The resynchronization problem.

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, it will be accepted and the correct TPDU rejected.

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. 6-12(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 different 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 for a short clock tick (a few milliseconds).

Unfortunately, entering the forbidden region from underneath by sending too fast is not the only way to get into trouble. From Fig. 6-12(b), it should be clear 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 one machine A, send a *CR* TPDU containing the proposed initial sequence number and destination port number to a remote peer, B. The receiver, B, then acknowledges this request by sending a *CC* TPDU back. If the *CR* TPDU is lost but a delayed duplicate *CR* suddenly shows up at B, the connection will be established incorrectly.

To solve this problem, Tomlinson (1975) has 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 setup procedure when A initiates is shown in Fig. 6-13. In this figure, A is the transport entity (not transport user) to the left of the vertical lines, and B is the transport entity to the right. The arrows denote TPDUs sent and received, not transport user primitives, as was the case when we were discussing the service rather than the protocol. A chooses a sequence number somehow, say x, and sends it to B. B replies with a *CC* TPDU acknowledging x and announcing its own initial sequence number, y (which may be equal to x, of course). Finally, A acknowledges B's choice of an initial sequence number in its first data TPDU.

Now let us see how the three-way handshake works in the presence of delayed

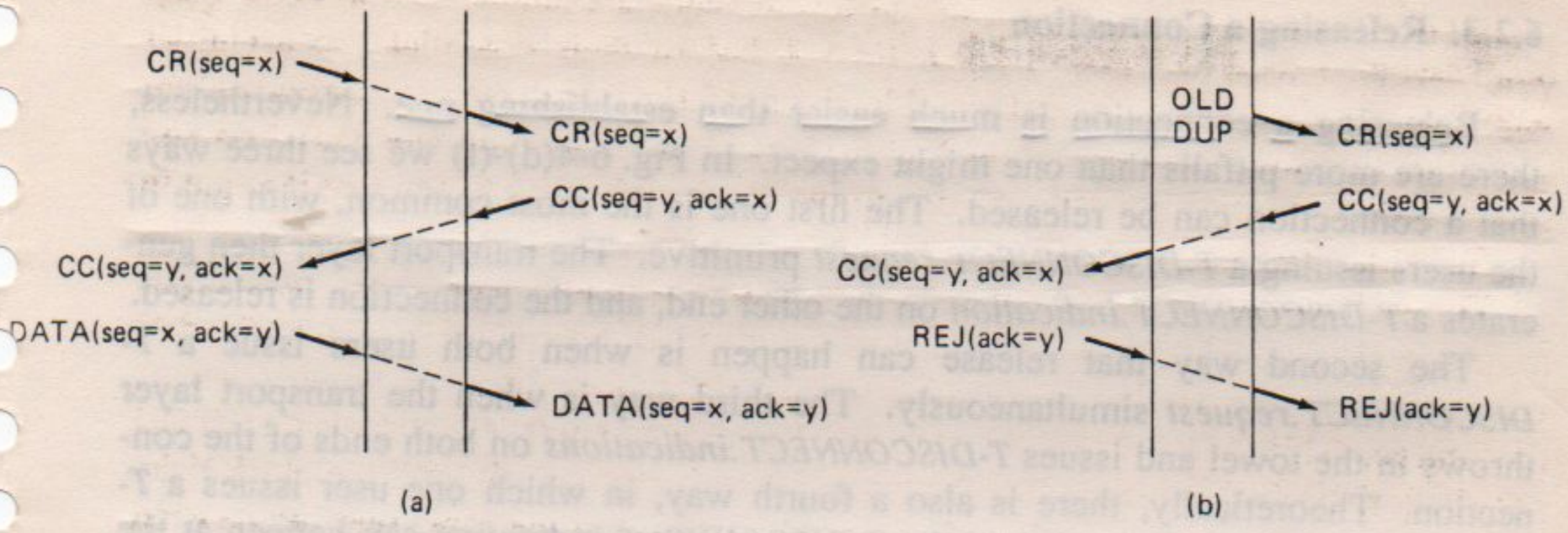


Fig. 6-13. Three protocol scenarios for establishing a connection using a three-way handshake. (a) Normal operation. (b) Old duplicate *CR* appearing out of nowhere. (c) Duplicate *CR* and duplicate ACK.

duplicate control TPDUs. In Fig. 6-13(b), the first TPDU is a delayed duplicate *CR* from a connection since released. This TPDU arrives at *B* without *A*'s knowledge. *B* reacts to this TPDU by sending *A* a *CC* TPDU, in effect asking for verification that *A* was indeed trying to set up a new connection. When *A* rejects *B*'s attempt to establish, *B* realizes that it was tricked by a delayed duplicate and abandons the connection.

The worst case is when both a delayed *CR* and an acknowledgement to a *CC* are floating around in the subnet. This case is shown in Fig. 6-13(c). As in the previous example, *B* gets a delayed *CR* and replies to it. At this point it is crucial to realize that *B* has proposed using *y* as the initial sequence number for *B* to *A* 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 *B*, the fact that *z* has been acknowledged rather than *y* tells *B* that this, too, is an old duplicate.

6.2.3. Releasing a Connection

Releasing a connection is much easier than establishing one. Nevertheless, there are more pitfalls than one might expect. In Fig. 6-4(d)-(f) we see three ways that a connection can be released. The first one is the most common, with one of the users issuing a *T-DISCONNECT.request* primitive. The transport layer then generates a *T-DISCONNECT.indication* on the other end, and the connection is released.

The second way that release can happen is when both users issue a *T-DISCONNECT.request* simultaneously. The third way is when the transport layer throws in the towel and issues *T-DISCONNECT.indications* on both ends of the connection. Theoretically, there is also a fourth way, in which one user issues a *T-DISCONNECT.request* and before the *T-DISCONNECT.indication* can happen at the other end, the transport layer itself disconnects the other end.

All of these forms of disconnect have one feature in common: they are abrupt and may result in data loss. Consider the scenario of Fig. 6-14. After the connection is established, *A* sends a TPDU that arrives properly at *B*. Then *A* sends another TPDU and disconnects. Unfortunately, *B* issues a *T-DISCONNECT.request* before the second TPDU arrives. The result is that the connection is released and data are lost.

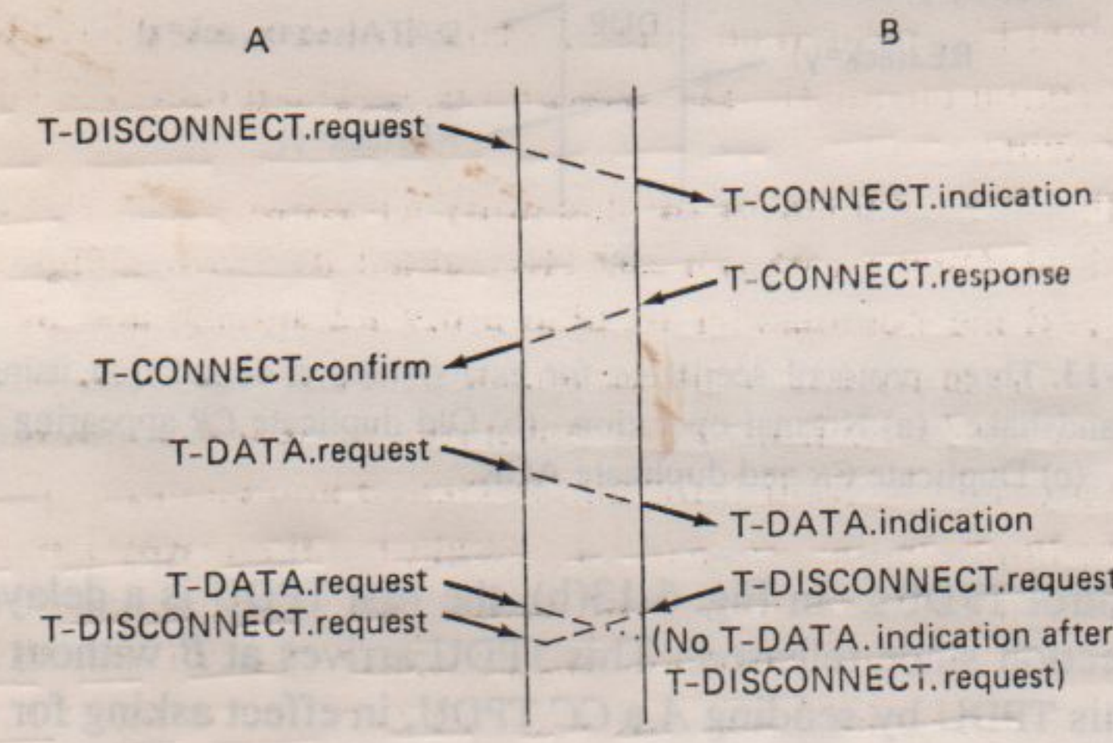


Fig. 6-14. Abrupt disconnection with loss of data.

Clearly a more sophisticated release protocol is required to avoid data loss. An obvious way to handle release is not to have either side issue a *T-DISCONNECT.request* until it is sure that the other side has received all data that has been sent, and has no more data to send itself. In other words, the protocol could be something like *A* saying: "I am done. Are you done too?" If *B* 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 deals with this issue. It is called the **two-army problem.** Imagine that a white army is encamped in a valley, as shown in Fig. 6-15. On both of the surrounding

hillsides are blue armies. The white army is larger than either of the blue armies alone, but together they 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.

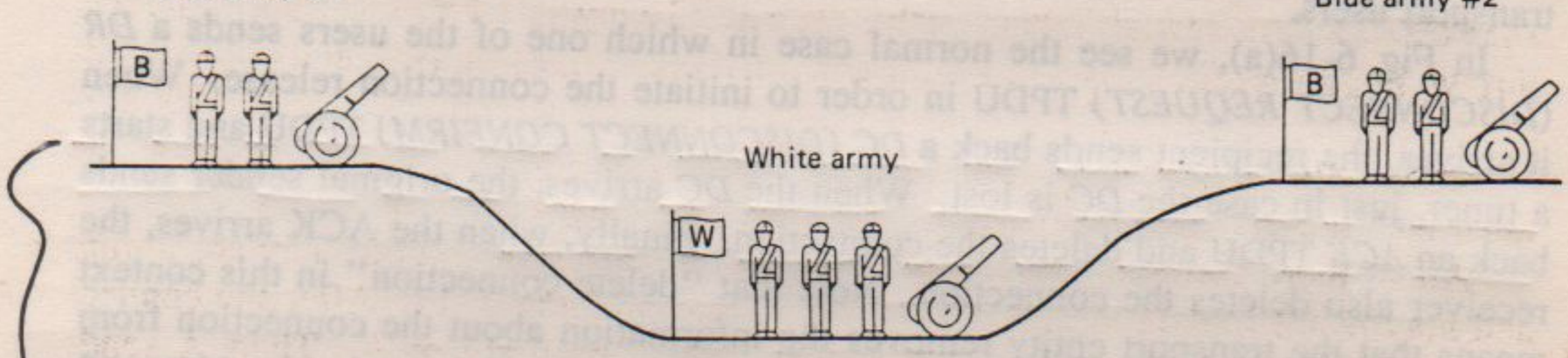


Fig. 6-15. The two-army problem.

The blue armies obviously 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, and the commander of blue army #2 agrees, and that 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 easily 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 6-16 illustrates four scenarios of releasing using a three-way handshake. While this protocol is not infallible, it is usually adequate. Note that it shows the TPDUs sent and received by the transport entities, not the primitives seen by the transport users.

In Fig. 6-16(a), we see the normal case in which one of the users sends a *DR* (*DISCONNECT REQUEST*) TPDU in order to initiate the connection release. When it arrives, the recipient sends back a *DC* (*DISCONNECT CONFIRM*) TPDU and starts a timer, just in case the *DC* is lost. When the *DC* arrives, the original sender sends back an *ACK* TPDU and deletes the connection. Finally, when the ACK arrives, the receiver also deletes the connection. Note that "delete connection" in this context means that the transport entity removes the information about the connection from its table of open connections, and signals the connection's owner (the transport user) somehow. This action is completely different from a transport user issuing a *T-DISCONNECT.request* primitive.

If the final ACK is lost, as shown in Fig. 6-16(b), the situation is saved by the timer. When the timer expires, the connection is deleted anyway.

Now consider the case of the *DC* (or *DR*) being lost. The user initiating the disconnect will not receive the *DC*, will time out, and will start all over again. In Fig. 6-16(c) we see how this works, assuming that the second time no TPDUs are lost.

Our last scenario, Fig. 6-16(d), is the same as Fig. 6-16(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 deletes 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 delete 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. 6-16(b).

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 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 re-

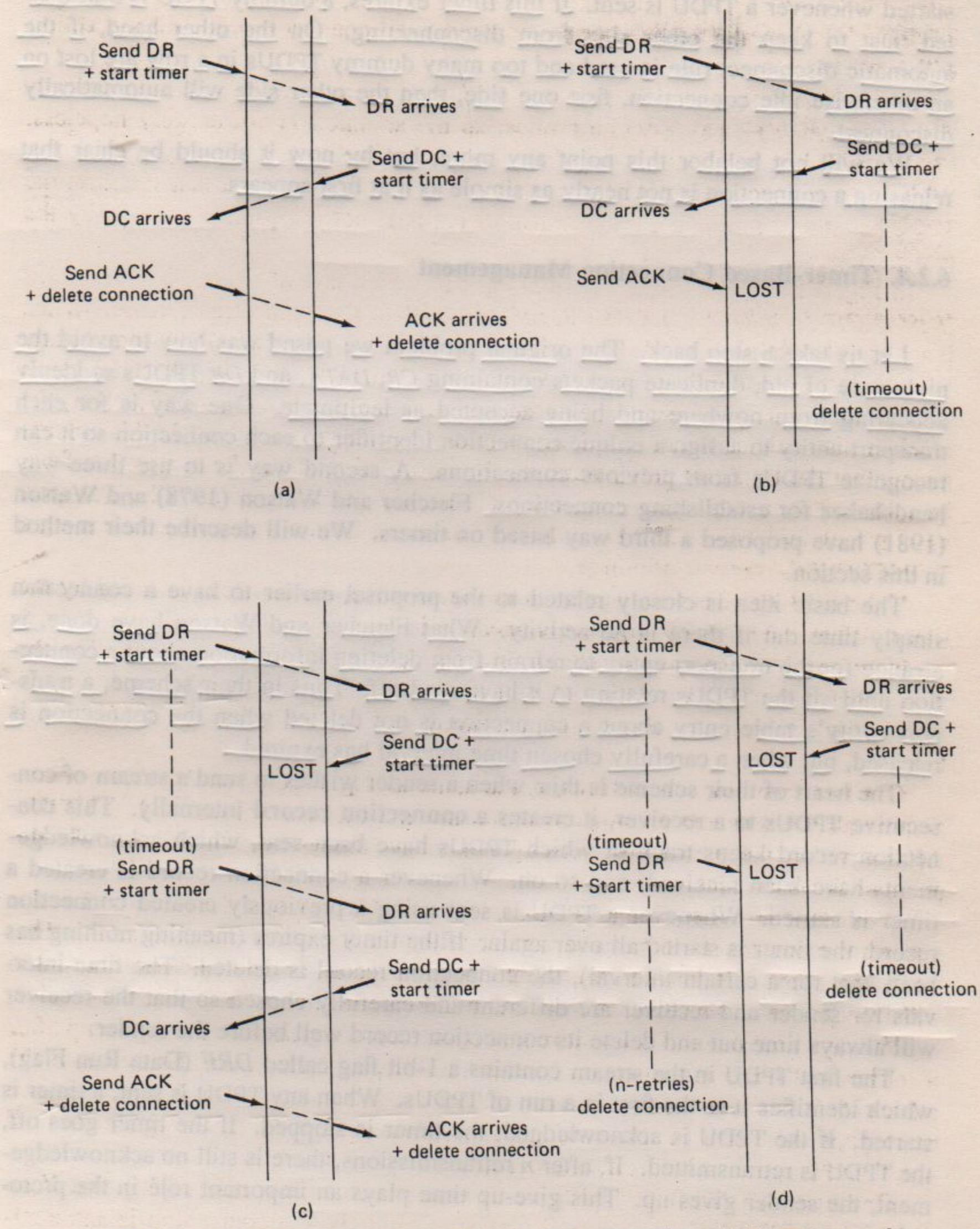


Fig. 6-16. Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final *ACK* lost. (c) *DC* lost. (d) *DC* lost and subsequent *DRs* lost.

started 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 TPDUs 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 is not nearly as simple as it at first appears.

6.2.4. Timer-Based Connection Management

Let us take a step back. The original problem we posed was how to avoid the nightmare of old, duplicate packets containing *CR*, *DATA*, and *DR* TPDUs suddenly appearing from nowhere and being accepted as legitimate. One way is for each transport entity to assign a unique connection identifier to each connection so it can recognize TPDUs from previous connections. A second way is to use three-way handshakes for establishing connections. Fletcher and Watson (1978) and Watson (1981) have proposed a third way based on timers. We will describe their method in this section.

The basic idea is closely related to the proposal earlier to have a connection simply time out if there is no activity. What Fletcher and Watson have done, is arrange for the transport entity to refrain from deleting information about a connection until all the TPDUs relating to it have died off. Thus in their scheme, a transport entity's table entry about a connection is not deleted when the connection is released, but when a carefully chosen time interval has expired.

The heart of their scheme is this: when a sender wishes to send a stream of consecutive TPDUs to a receiver, it creates a **connection record** internally. This connection record keeps track of which TPDUs have been sent, which acknowledgements have been received, and so on. Whenever a connection record is created a timer is started. Whenever a TPDU is sent using a previously created connection record, the timer is started all over again. If the timer expires (meaning nothing has been sent for a certain interval), the connection record is deleted. The time intervals for sender and receiver are different and carefully chosen so that the receiver will always time out and delete its connection record well before the sender.

The first TPDU in the stream contains a 1-bit flag called *DRF* (Data Run Flag), which identifies it as the first in a run of TPDUs. When any TPDU is sent, a timer is started. If the TPDU is acknowledged, the timer is stopped. If the timer goes off, the TPDU is retransmitted. If, after n retransmissions, there is still no acknowledgement, the sender gives up. This give-up time plays an important role in the protocol.

When a TPDU with the *DRF* flag set arrives at the receiver, the receiver notes its sequence number and creates a connection record. Subsequent TPDUs will only be accepted if they are in sequence. If the first TPDU to arrive at the receiver does

not have the *DRF* flag set, it is discarded. Eventually the first TPDU (original or retransmission) will arrive and the connection record can be created. In other words, a connection record is only created when a TPDU with *DRF* set arrives.

Whenever a TPDU arrives in sequence, it can be passed to the transport user and an acknowledgement returned to the sender. If a TPDU arrives out of sequence when a connection record exists, it may be buffered (like protocol 6 in Chapter 4). It may also be acknowledged. However, an acknowledgement does not imply that all the previous TPDUs have been received as well, unless a flag, *ARF* (Acknowledgment Run Flag), is set.

Let us first consider a simple case of how this protocol works. A sequence of TPDUs is sent and all are received in order and acknowledged. When the sender gets the acknowledgement of the final TDPU sent and sees the *ARF* flag, it stops all the retransmission timers. If no more data are forthcoming from the transport user, eventually the receiver's connection record times out and then later the sender's does too. No explicit establishment or release TPDUs are needed (although the *DRF* flag is somewhat analogous to a *CR* TPDU).

Now consider what happens if the TPDU bearing the *DRF* flag is lost. The receiver will not create a connection record. Eventually the sender will time out and retransmit the TPDU, repeatedly if necessary, until it is acknowledged. Once a connection record has been created by the receiver, a gap in the TPDU stream is easily detected and repaired by sender timeouts and retransmissions.

Suppose a stream of TPDUs is sent and correctly received, but some of the acknowledgements are lost. Subsequent retransmissions are also lost, so the receiver's connection record times out and is deleted. What is to prevent an old duplicate of the TPDU with the *DRF* flag from now appearing at the receiver and triggering a new connection record? The trick is to make the receiver's connection record timer much longer than the sender's give-up timer plus the maximum TPDU lifetime. Thus once the initial TPDU is accepted, the connection record will be kept in existence until the receiver is certain that the sender has stopped sending the TPDU with the *DRF* flag (either because it got an acknowledgement or it gave up). In this way, once the initial TPDU has been accepted, there is no danger that it will appear after the connection record has been deleted—all copies of it are definitely gone. The other TPDUs in the run are harmless because the receiver will not accept them (only TPDUs with the *DRF* flag are acceptable when the receiver has no connection record).

Furthermore, if some TPDUs remain unacknowledged, the sender will keep retransmitting them, thus keeping its connection record alive. As long as the connection record shows unacknowledged TPDUs outstanding, no new TPDU with *DRF* set will be sent.

Now let us see what happens if the transport user sends messages in bursts. Suppose a burst of TPDUs is sent and all are acknowledged. When the transport user finally gets around to sending a new message, one of three situations must hold:

1. Both sender and receiver still have their connection records.
2. The sender has its connection record but the receiver does not.
3. Both connection records have been deleted.

It cannot happen that the sender's record has been deleted while the receiver's is still around because the timer for the sender's record is intentionally longer to prevent just this case. In case 1, the next TPDU will carry a *DRF* flag and will start a new run; it will be numbered one higher than the previous TPDU because the connection record still exists. The receiver will accept it without problems. In case 2, the receiver will create a new connection record and regard it as the start of a new run, since it has already forgotten the previous run. In case 3, the sender will create a new connection record, which means that the TPDU will not be numbered in sequence with the previous ones. However, the receiver no longer knows what the previous ones were, so it does not matter. It should be clear now that the fourth case (receiver knowing which TPDU to expect but sender not knowing which one to send) has been carefully forbidden for good reason.

In summary, this protocol has an interesting mixture of connectionless and connection-oriented properties. The minimum exchange is two TPDUs, which makes it as efficient as a connectionless protocol for query-response applications. Unlike a connectionless protocol, however, if it turns out that a sequence of TPDUs must be sent, they are guaranteed to be delivered in order. Finally, connections are released automatically by the clever use of timers.

6.2.5. 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 the IMP usually has relatively few lines whereas the 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 Chapter 4, frames were buffered at both the sending IMP and at the receiving IMP. In protocol 6, for example, both sender and receiver are required to dedicate $MaxSeq + 1$ 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 TPDUs 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. The sender just keeps trying until it gets an acknowledgement.

In summary, if the network service is unreliable (i.e., type B or C), the sender must buffer all TPDUs sent, just as in the data link layer. However, with reliable (type A) 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 identical size buffers, with one TPDU per buffer, as in Fig. 6-17(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.

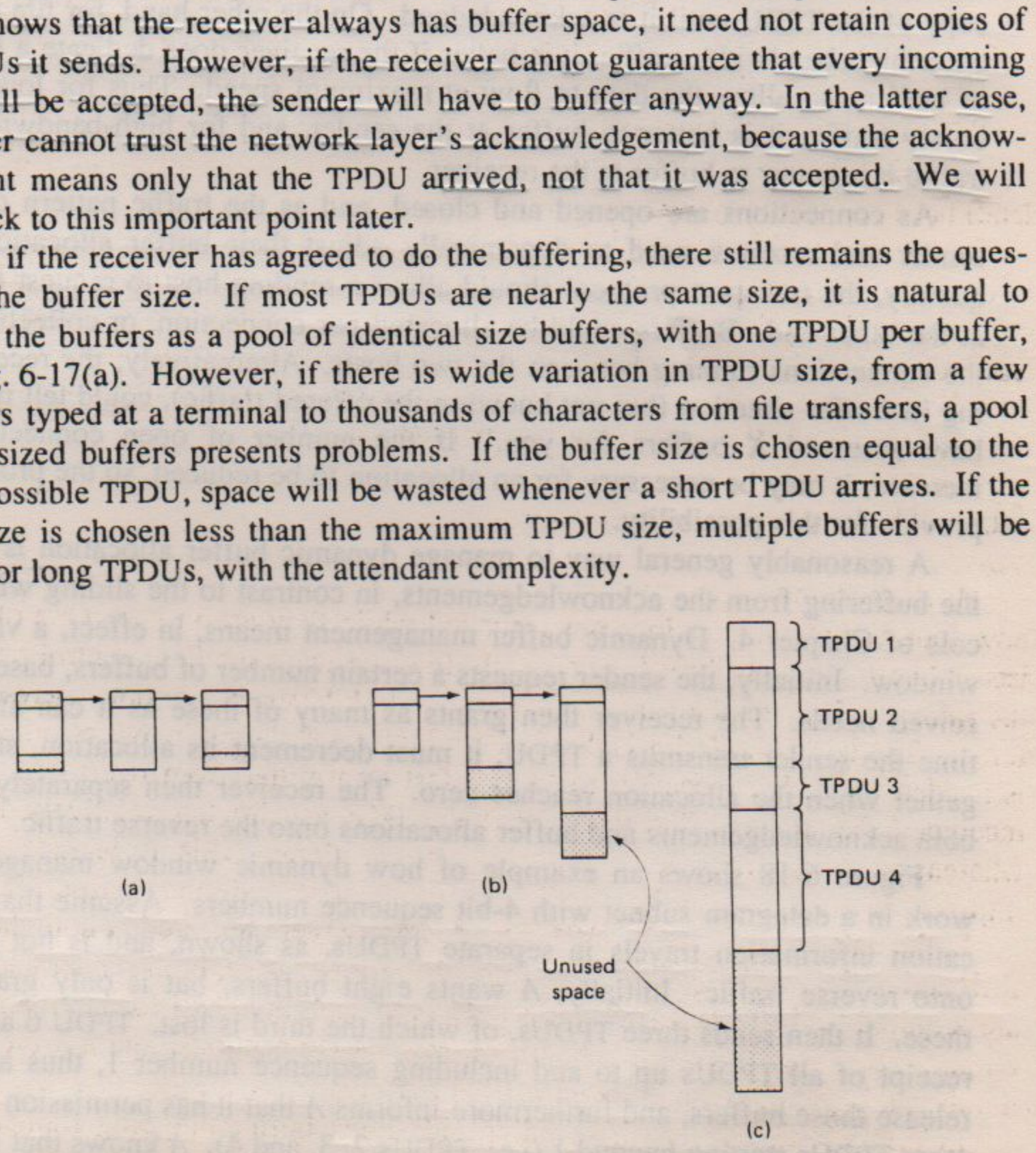


Fig. 6-17. (a) Chained fixed-size buffers. (b) Chained variable-size buffers. (c) One large circular buffer per connection.

Another approach to the buffer size problem is to use variable-size buffers, as in Fig. 6-17(b). The advantage here is better memory utilization, at the price of far more complicated buffer management. A third possibility is to dedicate a single large circular buffer per connection, as in Fig. 6-17(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 Chapter 4. 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 6-18 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 TPDUs, as shown, and is not piggybacked onto reverse traffic. Initially, *A* wants eight buffers, but is only granted four of these. It then sends three TPDUs, of which the third is lost. TPDU 6 acknowledges receipt of all TPDUs 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 TPDUs starting beyond 1 (i.e., TPDUs 2, 3, and 4). *A* knows that it has already sent number 2, so it thinks that it may send TPDUs 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 TPDUs up to and including 4, but refuses to let *A* continue. Such a situation is impossible with the fixed window protocols of Chapter 4. The next TPDU from *B* to *A* allocates another buffer and allows *A* to continue.

	A	Message	B	Comments
1	→	< request 8 buffers >	→	A wants 8 buffers
2	←	< ack = 15, buf = 4 >	←	B grants messages 0-3 only
3	→	< seq = 0, data = m0 >	→	A has 3 buffers left now
4	→	< seq = 1, data = m1 >	→	A has 2 buffers left now
5	→	< seq = 2, data = m2 >	...	Message lost but A thinks it has 1 left
6	←	< ack = 1, buf = 3 >	←	B acknowledges 0 and 1, permits 2-4
7	→	< seq = 3, data = m3 >	→	A has 1 buffer left
8	→	< seq = 4, data = m4 >	→	A has 0 buffers left, and must stop
9	→	< seq = 2, data = m2 >	→	A times out and retransmits
10	←	< ack = 4, buf = 0 >	←	Everything acknowledged, but A still blocked
11	←	< ack = 4, buf = 1 >	←	A may now send 5
12	←	< ack = 4, buf = 2 >	←	B found a new buffer somewhere
13	→	< seq = 5, data m5>	→	A has 1 buffer left
14	→	< seq = 6, data = m6 >	→	A is now blocked again
15	←	< ack = 6, buf = 0 >	←	A is still blocked
16	...	< ack = 6, buf = 4 >	←	Potential deadlock

Fig. 6-18. 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 TPDU 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.

Up 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 IMPs can exchange at most x frames/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) has 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 capacity of the network depends on the amount of traffic in it, the window size should be adjusted frequently, to track changes in the carrying capacity.

6.2.6. 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, in networks that use virtual circuits within the subnet, each open connection consumes some table space in the IMPs for the entire duration of the connection. If buffers are dedicated to the virtual circuit in each IMP as well, a user who left his terminal logged into a remote machine during a coffee break is nevertheless consuming expensive resources. Although this implementation of packet switching defeats one of the main reasons for having packet switching in the first place—to bill the user based on the amount of data sent, not the connect time—a number of PTTs have chosen this approach, presumably because it so closely resembles the circuit switching model to which they have grown accustomed over the decades.

The consequence of a price structure that heavily penalizes installations for having many virtual circuits open for long periods of time is to make multiplexing of different transport connections onto the same network connection attractive. This form of multiplexing, called **upward multiplexing**, is shown in Fig. 6-19(a). In this figure, four distinct transport connections all use the same network connection (e.g., X.25 virtual circuit) to the remote host. When connect time forms the major component of the carrier's bill, it is up to the transport layer to group transport connections according to their destination and map each group onto the minimum number of network connections. If too many transport connections are

mapped onto one network connection, the performance will be poor, because the window will usually be full, and users will have to wait their turn to send one message. If too few transport connections are mapped onto one network connection, the service will be expensive. When upward multiplexing is used with X.25, we have the ironic (tragic?) situation of having to identify the connection using a field in the transport header, even though X.25 provides more than 4000 virtual circuit numbers expressly for that purpose.

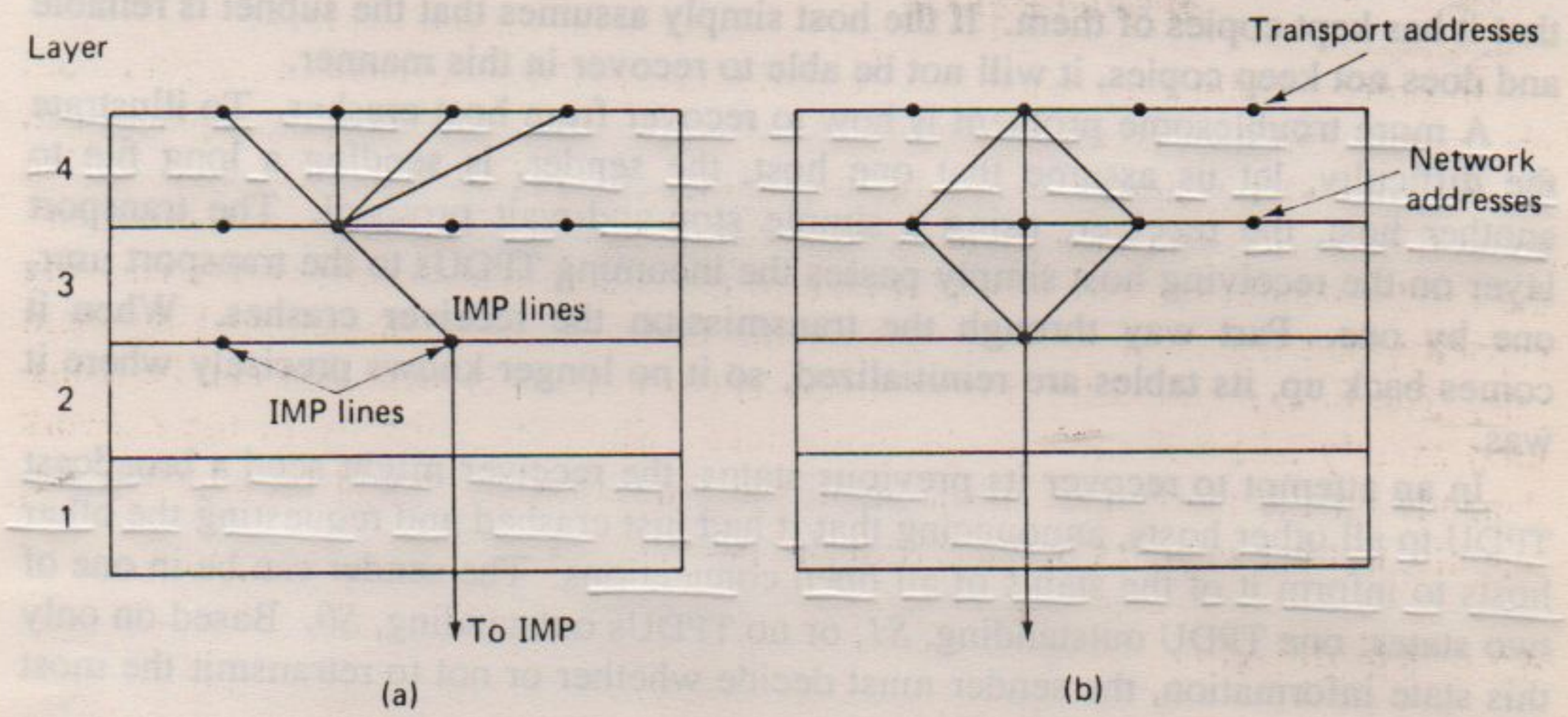


Fig. 6-19. (a) Upward multiplexing. (b) Downward multiplexing.

Multiplexing can also be useful in the transport layer for another reason, related to carrier technical decisions rather than carrier pricing decisions. Suppose, for example, that a certain important user needs a high-bandwidth connection from time to time. If the subnet enforces a sliding window flow control with a 3-bit sequence number, the user must stop sending as soon as seven packets are outstanding, and must wait for the packets to propagate to the remote host and be acknowledged. If the physical connection is via a satellite, the user is effectively limited to seven packets every 540 msec. With 128-byte packets, the usable bandwidth is about 13 kbps, even though the physical channel bandwidth is more than 1000 times higher.

One possible solution is to have the transport layer open multiple network connections, and distribute the traffic among them on a round-robin basis, as indicated in Fig. 6-19(b). This modus operandi is called **downward multiplexing**. With k network connections open, the effective bandwidth is increased by a factor of k. With 4095 X.25 virtual circuits, 128-byte packets, and a 3-bit sequence number, it is theoretically possible to achieve data rates in excess of 50 Mbps. Of course, this performance can be achieved only if the host-IMP line can support 50 Mbps, because all 4095 virtual circuits are still being sent out over one physical line, at least in Fig. 6-19(b). If multiple host-IMP lines are available, downward multiplexing can also be used to increase the performance even more.

6.2.7. Crash Recovery

If hosts and IMPs are subject to crashes, recovery from these crashes becomes an issue. If the network layer issues an *N-RESET*, for example, the transport entities must exchange information after the crash to determine which TPDUs were received and which were not. In effect, after a crash host *A* can ask host *B*: "I have four unacknowledged TPDUs outstanding, 2, 3, 4, and 5; have you received any of them?" Based on the answer, *A* can retransmit the appropriate TPDUs, provided that it has kept copies of them. If the host simply assumes that the subnet is reliable and does not keep copies, it will not be able to recover in this manner.

A more troublesome problem is how to recover from host crashes. To illustrate the difficulty, let us assume that one host, the sender, is sending a long file to another host, the receiver, using a simple stop-and-wait protocol. The transport layer on the receiving host simply passes the incoming TPDUs to the transport user, one by one. Part way through the transmission the receiver 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 receiver might send a broadcast TPDU to all other hosts, announcing that it had just crashed and requesting the other hosts to inform it of the status of all open connections. The sender can be in one of two states: one TPDU outstanding, *S1*, or no TPDUs outstanding, *S0*. Based on only this state information, the sender must decide whether or not to retransmit the most recent TPDU.

At first glance it would seem obvious that the sender should retransmit 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 when the receiving host first sends an acknowledgement, and then, when the acknowledgement has been sent, performs the write. Writing a TPDU onto the output stream and sending an acknowledgement are considered as two distinct indivisible events that cannot be done simultaneously. If a crash occurs after the acknowledgement has been sent, but before the write has been done, the other host will receive the acknowledgement and thus be in state *S0* when the crash recovery announcement arrives. The sender will therefore not retransmit, thinking the TPDU has arrived correctly, leading 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 sender will be in state *S1* and thus retransmit, leading to an undetected duplicate TPDU in the output stream.

No matter how the sender and receiver are programmed, there are always situations where the protocol fails to recover properly. The receiver can be programmed in one of two ways: acknowledge first or write first. The sender 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 receiver: 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* may follow *C* (i.e., once it has crashed, it has crashed). Figure 6-20 shows all eight combinations of sender and receiver 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 behave incorrectly. For example, if the sender always retransmits, the *AWC* event will generate an undetected duplicate, even though the other two events work properly.

	Strategy used by receiving host					
	First ACK, then write			First write, then ACK		
Strategy used by sending host	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

Fig. 6-20. Different combinations of sender and receiver strategy.

Making the protocol more elaborate does not help. Even if the sender and receiver exchange several TPDUs before the receiver attempts to write, so that the sender knows exactly what is about to happen, the sender 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/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 it is.

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 data base. 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 data base. 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).

6.3. A SIMPLE TRANSPORT PROTOCOL ON TOP OF X.25

To make the ideas discussed so far more concrete, in this section we will study an example transport layer in detail. The example has been carefully chosen to be reasonably realistic, yet still simple enough to be easy to understand. The abstract service primitives are the OSI connection-oriented primitives, with the exception of the expedited data feature, which just adds complexity without providing any new insight into how the transport layer works.

6.3.1. The Example Service Primitives

Our first problem is how to express the OSI transport primitives in Pascal. *CONNECT.request* is easy: we will just have a library procedure *connect*, that can be called with the appropriate parameters necessary to establish a connection. However, *CONNECT.indication* is much harder. How do we signal the called transport user that there is an incoming call? In essence, an incoming call is an interrupt, a difficult concept to deal with in a high-level language and poor programming practice as well. The *CONNECT.indication* primitive is an excellent way of modeling how telephones work (telephones really do generate interrupts, by ringing), but is a not a good way of modeling how computers work.

To provide a reasonable interface to our transport layer, we will have to do what all real networks do, and invent a different, and much more computer-oriented, model for connection establishment. In our model, there are two procedures available, *listen* and *connect*. When a process (i.e., a transport user) wants to be able to accept incoming calls, it calls *listen*, specifying a particular TSAP to listen to. The process then blocks (i.e., goes to sleep) until some remote process attempts to establish a connection to its TSAP.

The other procedure, *connect*, can be used when a process wants to initiate the establishment of a connection. The caller specifies the local and remote TSAPs, and is blocked while the transport layer tries to set up the connection. If the connection succeeds, both parties are unblocked, and can start exchanging data.

Note that this model is highly asymmetric. One side is passive, executing a *listen* and waiting until something happens. The other side is active and initiates the connection. An interesting question arises of what to do if the active side begins first. One strategy is to have the connection attempt fail if there is no

listener at the remote TSAP. Another strategy is to have the initiator block (possibly forever) until a listener appears.

A compromise, used in our example, is to hold the connection request at the receiving end for a certain time interval. If a process on that host calls *listen* before the timer goes off, the connection is established; otherwise, it is rejected and the caller is unblocked.

To release a connection, we will use a procedure *disconnect*. When both sides have disconnected, the connection is released.

Data transmission has precisely the same problem as connection establishment: although *T-DATA.request* can be implemented directly with a call to a library procedure, *T-DATA.indication* cannot be. We will use the same solution for data transmission as for connection establishment, an active call *send* that transmits data, and a passive call *receive* that blocks until a message has arrived.

Our concrete service definition thus consists of five primitives: *CONNECT*, *LISTEN*, *DISCONNECT*, *SEND*, and *RECEIVE*. Each primitive corresponds exactly with a library procedure that executes the primitive (unlike the OSI model, in which there is barely any correspondence at all between the primitives and the library procedures). The parameters for the service primitives and library procedures are as follows:

```
connum = CONNECT(local, remote)
connum = LISTEN(local)
status = DISCONNECT(connum)
status = SEND(connum, buffer, bytes)
status = RECEIVE(connum, buffer, bytes)
```

The *CONNECT* primitive takes two parameters, a local TSAP (i.e., transport address), *local*, and a remote TSAP, *remote*, and tries to establish a transport connection between the two. If it succeeds, it returns in *connum* a nonnegative number used to identify the connection on subsequent calls. If it fails, the reason for failure is put in *connum* as a negative number. In our simple model, each TSAP may participate in only one transport connection, so a possible reason for failure is that one of the transport addresses is currently in use. Some other reasons are: remote host down, illegal local address, and illegal remote address.

The *LISTEN* primitive announces the caller's willingness to accept connection requests directed at the indicated TSAP. The user of the primitive is blocked until an attempt is made to connect to it. There is no timeout.

The *DISCONNECT* primitive terminates a transport connection. The parameter *connum* tells which one. Possible errors are: *connum* belongs to another process, or *connum* is not a valid connection identifier. The error code, or 0 for success, is returned in *status*.

The *SEND* primitive transmits the contents of the buffer as a message on the indicated transport connection, possibly in several units if it is too big. Possible

errors, returned in *status*, are: no connection, illegal buffer address, or negative count.

The *RECEIVE* primitive indicates the caller's desire to accept data. The size of the incoming message is placed in *bytes*. If the remote process has released the connection or the buffer address is illegal (e.g., outside the user's program), *status* is set accordingly to an error code.

6.3.2. The Example Transport Entity

Before looking at the code of the example transport entity, please be sure you realize that this example is analogous to the early examples presented in Chapter 4: it is more for pedagogical purposes than a serious proposal. Many of the technical details (such as extensive error checking) that would be needed in a production system have been omitted for the sake of simplicity. Nevertheless, most of the basic ideas found in the transport entity for the class 0 OSI protocol are present in our example.

The transport layer makes use of the network service primitives to send and receive TPDUs. Just as the OSI transport service primitives cannot be mapped directly onto library procedures, neither can the network service procedures. In this example we get around this problem by using X.25 as the network layer interface. Each TPDU will be carried in one packet and each packet will correspond to one TPDU. We will call these units "packets" below. In this example we will assume that X.25 is completely reliable (type A), neither losing packets nor resetting the circuit. Figure 6-21 gives an example program for implementing our transport service. Such a program (effectively the code of the transport entity) may be part of the host's operating system or it may be a package of library routines running within the user's address space. It may also be contained on a co-processor chip or network board plugged into the host's backplane. For simplicity, the example of Fig. 6-21 has been programmed as though it were a library package, but the changes needed to make it part of the operating system are minimal (primarily how user buffers are accessed).

It is worth noting, however, that in this example, the "transport entity" is not really a separate entity at all, but part of the user process. In particular, when the user executes a primitive that blocks, such as *LISTEN*, the entire transport entity blocks as well. While this design is fine for a host with only a single user process, on a host with multiple users, it would be more natural to have the transport entity be a separate process, distinct from all the user processes.

The interface to the network layer (X.25) is via the procedures *ToNet* and *FromNet* (not shown). Each has six parameters: the connection identifier, which maps one-to-one onto network virtual circuits; the X.25 *Q* and *M* bits, which indicate control message and more data from this message follows in the next packet, respectively; the packet type, chosen from the set *CALL REQUEST*, *CALL*

```
const MaxConn = ... ;  MaxMsg = ... ;  MaxPkt = ... ;
      TimeOut = ... ; cred = ... ;
      q0 = 0; q1 = 1; m0 = 0; m1 = 1; ok = 0;
      ErrFull = -1;  ErrReject = -2;  ErrClosed = -3;  LowErr = -3;

type bit = 0..1;
     TransportAddress = integer;
     ConnId = 0 .. MaxConn;            {connection identifier}
     PktType = (CallReq, CallAcc, ClearReq, ClearConf, DataPkt, credit);
     cstate = (idle, waiting, queued, established, sending, receiving, disconnecting);
     message = array [0 .. MaxMsg] of 0 .. 255;
     msgptr = ↑message;        {pointer to a message}
     ErrorCode = LowErr .. 0;
     ConnIdOrErr = LowErr .. MaxConn;
     PktLength = 0 .. MaxPkt;
     packet = array[PktLength] of 0 .. 255;

var ListenAddress: TransportAddress; {local address being listened to}
    ListenConn: ConnId;              {connection identifier for listen}
    data: packet;                    {scratch area for packet data}
    conn: array[ConnId] of record
      LocalAddress, RemoteAddress: TransportAddress;
      state: cstate;                 {state of this connection}
      UserBufferAddress: msgptr;     {pointer to receive buffer}
      ByteCount: 0 .. MaxMsg;        {send/receive count}
      ClrReqReceived: boolean;       {set when CLEAR REQUEST packet received}
      timer: integer;                {used to time out CALL REQUEST packets}
      credits: integer               {number of messages that may be sent}
    end;

function listen(t: TransportAddress): ConnIdOrErr;
{User wants to listen for a connection.  See if CALLREQ has already arrived.}
var i: integer; found: boolean;
begin
  i := 1;
  found := false;
  while (i <= MaxConn) and not found do
    if (conn[i].state = queued) and (conn[i].LocalAddress = t)
      then found := true
      else i := i + 1;
    if not found then
      begin   {no CALLREQ is waiting.  Go to sleep until arrival or timeout.}
        ListenAddress := t; sleep; i := ListenConn
      end;
    conn[i].state := established;      {connection is established}
    conn[i].timer := 0;                {timer is not used}
    listen := i;                       {return connection identifier}
    ListenConn := 0;                   {0 is assumed to be an invalid address}
    ToNet(i, q0, m0, CallAcc, data, 0)      {tell net to accept connection}
end; {listen}
```

Fig. 6-21. A sample transport entity.

```
function connect(l, r: TransportAddress): ConnIdOrErr;
{User wants to connect to a remote process. Send CALLREQ packet.}
var i: integer;
begin i := MaxConn;                    {search table backwards}
  data[0] := r;  data[1] := l;         {CALL REQUEST packet needs these}
  while (conn[i].state <> idle) and (i > 1) do i := i - 1;
  if conn[i].state = idle then
    with conn[i] do
      begin                            {make a table entry that CALLREQ has been sent}
        LocalAddress := l;  RemoteAddress := r;  state := waiting;
        ClrReqReceived := false;  credits := 0;  timer := 0;
        ToNet(i, q0, m0, CallReq, data, 2);
        sleep;                         {wait for CALLACC or CLEARREQ}
        if state = established then connect := i;
        if ClrReqReceived then
          begin                        {other side refused call}
            connect := ErrReject;
            state := idle;             {back to idle state}
            ToNet(i, q0, m0, ClearConf, data, 0)
          end
      end
  else connect := ErrFull              {reject CONNECT: no table space}
end; {connect}

function send(cid: ConnId; bufptr: msgptr; bytes: integer): ErrorCode;
{User wants to send a message.}
var i, count: integer;  m: bit;
begin
  with conn[cid] do
    begin                              {enter sending state}
      state := sending;
      ByteCount := 0;
      if (not ClrReqReceived) and (credits = 0) then sleep;
      if not ClrReqReceived then
        begin {credit available; split message into packets}
          repeat
            if bytes - ByteCount > MaxPkt
              then begin count := MaxPkt; m := 1 end
              else begin count := bytes - ByteCount; m := 0 end;
            for i := 0 to count - 1 do data[i] := bufptr↑[ByteCount + i];
            ToNet(cid, q0, m, DataPkt, data, count);
            ByteCount := ByteCount + count;
          until ByteCount = bytes;     {loop until whole message sent}
          credits := credits - 1;      {one credit used up}
          send := ok
        end
      else send := ErrClosed;          {SEND failed: peer wants to disconnect}
      state := established
    end
end; {send}
```

```
function receive(cid: ConnId; bufptr: msgptr; var bytes: integer): ErrorCode;
{User is prepared to receive a message.}
begin
  with conn[cid] do
    begin
      if not ClrReqReceived then
        begin                    {connection still established; try to receive}
          state := receiving;
          UserBufferAddress := bufptr;   ByteCount := 0;
          data[0] := cred;   data[1] := 1;
          ToNet(cid, q1, m0, credit, data, 2);        {send credit}
          sleep;                 {block awaiting data}
          bytes := ByteCount
        end;
      if ClrReqReceived then receive := ErrClosed else receive := ok;
      state := established
    end
end; {receive}

function disconnect(cid: ConnId): ErrorCode;
{User wants to release a connection.}
begin
  with conn[cid] do
    if ClrReqReceived
      then begin state := idle; ToNet(cid, q0, m0, ClearConf, data, 0) end
      else begin state := disconnecting; ToNet(cid, q0, m0, ClearReq, data, 0) end;
  disconnect := ok
end; {disconnect}

procedure PacketArrival;
{A packet has arrived, get and process it.}
var cid: ConnId;                 {connection on which packet arrived}
    q, m: bit;
    ptype: PktType;              {CallReq, CallAcc, ClearReq, ClearConf, DataPkt, credit}
    data: packet;                {data portion of the incoming packet}
    count: PktLength;            {number of data bytes in packet}
    i: integer;                  {scratch variable}

begin
  FromNet(cid, q, m, ptype, data, count);  {go get it}
  with conn[cid] do
  case ptype of
  CallReq:                       {remote user wants to establish connection}
    begin
      LocalAddress := data[0]; RemoteAddress := data[1];
      if LocalAddress = ListenAddress
        then begin ListenConn := cid; state := established; wakeup end
        else begin state := queued; timer := TimeOut end;
      ClrReqReceived := false;  credits := 0
    end;
```

```
CallAcc:                          {remote user has accepted our CALL REQUEST}
  begin
    state := established;
    wakeup
  end;

ClearReq:                         {remote user wants to disconnect or reject call}
  begin
    ClrReqReceived := true;
    if state = disconnecting then state := idle;     {clear collision}
    if state in [waiting, receiving, sending] then wakeup
  end;

ClearConf:                        {remote user agrees to disconnect}
  state := idle;

credit:                           {remote user is waiting for data}
  begin
    credits := credits + data[1];
    if state = sending then wakeup
  end;

Datapkt:                          {remote user has sent data}
  begin
    for i := 0 to count - 1 do UserBufferAddress↑[ByteCount + i] := data[i];
    ByteCount := ByteCount + count;
    if m = 0 then wakeup
  end
 end;
end; {PacketArrival}

procedure clock;
{The clock has ticked, check for timeouts of queued connect requests.}
var i: ConnId;
begin
  for i := 1 to MaxConn do
    with conn[i] do
      if timer > 0 then
        begin {timer was running}
          timer := timer - 1;
          if timer = 0 then
            begin     {timer has expired}
              state := idle;
              ToNet(i, q0, m0, ClearReq, data, 0)
            end
        end
end; {clock}
```

ACCEPTED, *CLEAR REQUEST*, *CLEAR CONFIRMATION*, *DATA*, and *CREDIT*; a pointer to the data itself; and the number of bytes of data.

On calls to *ToNet*, the transport entity (i.e., some procedure in Fig. 6-21) fills in all the parameters for the network layer to read; on calls to *FromNet*, the network layer dismembers an incoming packet for the transport entity. By passing information as procedure parameters rather than passing the actual outgoing or incoming packet itself, the transport layer is shielded from the details of the network layer protocol. If the transport entity should attempt to send a packet when the underlying virtual circuit's sliding window is full, it is suspended within *ToNet* until there is room in the window. This mechanism is transparent to the transport entity and is controlled by the network layer using commands like *EnableTransportLayer* and *DisableTransportLayer* analogous to those described in the protocols of Chapter 4. The management of the X.25 packet layer window is also done by the network layer.

In addition to this transparent suspension mechanism, there are also explicit *sleep* and *wakeup* procedures (not shown) called by the transport entity. The procedure *sleep* is called when the transport entity is logically blocked waiting for an external event to happen, generally the arrival of a packet. After *sleep* has been called, the transport entity (and user process, of course) stop executing.

Each connection maintained by the transport entity of Fig. 6-21 is always in one of seven states, as follows:

1. Idle—Connection not established yet.
2. Waiting—*CONNECT* has been executed and *CALL REQUEST* sent.
3. Queued—A *CALL REQUEST* has arrived; *LISTEN* has not been done.
4. Established—The connection has been established.
5. Sending—The user is waiting for permission to transmit a packet.
6. Receiving—A *RECEIVE* has been done.
7. Disconnecting—A *DISCONNECT* has been done locally.

Transitions between states can occur when primitives are executed, when packets arrive, or when the timer expires.

The collection of procedures shown in Fig. 6-21 are of two types. Most are directly callable by user programs. *PacketArrival* and *clock* are different, however. They are spontaneously triggered by external events: the arrival of a packet and the clock ticking, respectively. In effect, they are interrupt routines. We will assume that they are never invoked while a transport entity procedure is running. Only when the user process is sleeping or executing outside the transport entity may they be called. This property is crucial to the correct functioning of the transport entity.

The existence of the *Q* (Qualifier) bit in the X.25 header allows us to avoid the

overhead of a transport protocol header. Ordinary data messages are sent as X.25 data packets with $Q = 0$. Transport protocol control messages, of which there is only one (*CREDIT*) in our example, are sent as X.25 data packets with $Q = 1$. These control messages are detected and processed by the receiving transport entity, of course.

The main data structure used by the transport entity is the array *conn*, which has one record for each potential connection. The record maintains the state of the connection, including the transport addresses at either end, the number of messages sent and received on the connection, the current state, the user buffer pointer, the number of bytes of the current messages sent or received so far, a bit indicating that the remote user has issued a *DISCONNECT*, a timer, and a permission counter used to enable sending of messages. Not all of these fields are used in our simple example, but a complete transport entity would need all of them, and perhaps more. Each *conn* entry is assumed initialized to the *idle* state.

When the user calls *CONNECT*, the network layer is instructed to send a *CALL REQUEST* packet to the remote machine, and the user is put to sleep. When the *CALL REQUEST* packet arrives at the other side, the transport entity is interrupted to run *PacketArrival* to check if the local user is listening on the specified address. If so, a *CALL ACCEPTED* packet is sent back and the remote user is awakened; if not, the *CALL REQUEST* is queued for *TimeOut* clock ticks. If a *LISTEN* is done within this period, the connection is established; otherwise, it times out and is rejected with a *CLEAR REQUEST* packet. This mechanism is needed to prevent the initiator from blocking forever in the event that the remote process does not want to connect to it.

Although we have eliminated the transport protocol header, we still need a way to keep track of which packet belongs to which transport connection, since multiple connections may exist simultaneously. The simplest approach is to use the X.25 virtual circuit number as the transport connection number as well. Furthermore, the virtual circuit number can also be used as the index into the *conn* array. When a packet comes in on X.25 virtual circuit k, it belongs to transport connection k, whose state is in the record *conn*[k]. For connections initiated at a host, the connection number is chosen by the originating transport entity. For incoming calls, the X.25 network makes the choice, choosing any unused virtual circuit number.

To avoid having to provide and manage buffers within the transport entity, a flow control mechanism different from the traditional sliding window is used here. Instead, when a user calls *RECEIVE*, a special **credit message** is sent to the transport entity on the sending machine and is recorded in the *conn* array. When *SEND* is called, the transport entity checks to see if a credit has arrived on the specified connection. If so, the message is sent (in multiple packets if need be) and the credit decremented; if not, the transport entity puts itself to sleep until a credit arrives. This mechanism guarantees that no message is ever sent unless the other side has already done a *RECEIVE*. As a result, whenever a message arrives there is guaranteed to be a buffer available into which it can be put. The scheme can easily

be generalized to allow receivers to provide multiple buffers and request multiple messages.

You should keep the simplicity of Fig. 6-21 in mind. A realistic transport entity would normally check all user supplied parameters for validity, handle recovery from network resets, deal with call collisions, and support a more general transport service including such facilities as interrupts, datagrams, and nonblocking versions of the *SEND* and *RECEIVE* primitives.

6.3.3. The Example as a Finite State Machine

Writing a transport entity is difficult and exacting work, especially for the higher transport protocol classes. To reduce the chance of making an error, it is often useful to represent the state of the protocol as a finite state machine.

We have already seen that our example protocol has seven states per connection. It is also possible to isolate 12 events that can happen to move a connection from one state to another. Five of these events are the five service primitives. Another six are the arrivals of the six legal packet types. The last one is the expiration of the timer. Figure 6-22 shows the main protocol actions in matrix form. The columns are the states and the rows are the 12 events.

Each entry in the matrix (i.e., the finite state machine) of Fig. 6-22 has up to three fields: a predicate, an action, and a new state. The predicate indicates under what conditions the action is taken. For example, in the upper left-hand entry, if a *LISTEN* is executed and there is no more table space (predicate *P1*), the *LISTEN* fails and the state does not change. On the other hand, if a *CALL REQUEST* packet has already arrived for the transport address being listened to (predicate *P2*), the connection is established immediately. Another possibility is that *P2* is false, that is, no *CALL REQUEST* has come in, in which case the connection remains in the *Idle* state, awaiting a *CALL REQUEST* packet.

It is worth pointing out that the choice of states to use in the matrix is not entirely fixed by the protocol itself. In this example, there is no state *listening*, which might have been a reasonable thing to have following a *LISTEN*. There is no *listening* state because a state is associated with a connection record entry, and no connection record is created by *LISTEN*. Why not? Because we have decided to use the X.25 virtual circuit numbers as the connection identifiers, and for a *LISTEN*, the virtual circuit number is ultimately chosen by the X.25 network when the *CALL REQUEST* packet arrives.

The actions *A1* through *A12* are the major actions, such as sending packets and starting timers. Not all the minor actions, such as initializing the fields of a connection record are listed. If an action involves waking up a sleeping process, the actions following the wakeup also count. For example, if a *CALL REQUEST* packet comes in and a process was asleep waiting for it, the transmission of the *CALL ACCEPT* packet following the wakeup counts as part of the action for *CALL*

Primitives / Incoming packets / Clock	Idle	Waiting	Queued	Established	Sending	Receiving	Disconnecting
LISTEN	P1: ~/Idle P2: A1/Estab $\overline{P2}$: A2/Idle		~/Estab				
CONNECT	P1: ~/Idle $\overline{P1}$: A3/Wait						
DISCONNECT				P4: A5/Idle $\overline{P4}$: A6/Disc			
SEND				P5: A7/Estab $\overline{P5}$: A8/Send			
RECEIVE				A9/Receiving			
CallReq	P3:A1/Estab $\overline{P3}$:A4/Queu'd						
CallAcc		~/Estab					
ClearReq		~/Idle		A10/Estab	A10/Estab	A10/Estab	~/Idle
ClearConf							~/Idle
DataPkt						A12/Estab	
Credit				A11/Estab	A7/Estab		
Timeout			~/Idle				

Predicates

P1: Connection table full
P2: CallReq pending
P3: LISTEN pending
P4: ClearReq pending
P5: Credit available

Actions

A1: Send CallAcc
A2: Wait for CallReq
A3: Send CallReq
A4: Start timer
A5: Send ClearConf
A6: Send ClearReq
A7: Send message
A8: Wait for credit
A9: Send credit
A10: Set ClrReqReceived flag
A11: Record credit
A12: Accept message

Fig. 6-22. The example protocol as a finite state machine. Each entry has an optional predicate, and optional action, and the new state. The tilde indicates that no major action is taken. An overbar above a predicate indicates the negation of the predicate. Blank entries correspond to impossible or invalid events.

REQUEST. After each action is performed, the connection may move to a new state, as shown in Fig. 6-22.

The advantage of representing the protocol as a matrix is threefold. First, in this form it is much easier for the programmer to systematically check each combination of state and event to see if an action is required. In production implementations, some of the combinations would be used for error handling. In Fig. 6-22 no distinction is made between impossible situations and illegal ones. For example, if a connection is in *waiting* state, the *DISCONNECT* event is impossible because the user is blocked and cannot execute any primitives at all. On the other hand, in *sending* state, data packets are not expected because no credit has been issued. The arrival of a data packet is a protocol error that should be checked for.

The second advantage of the matrix representation of the protocol is in implementing it. One could envision a two-dimensional array in which element $a[i][j]$ was a pointer or index to the procedure that handled the occurrence of event i when in state j. One possible implementation is to write the transport entity as a short loop, waiting for an event at the top of the loop. When an event happens, the relevant connection is located and its state is extracted. With the event and state now known, the transport entity just indexes into the array a and calls the proper procedure. This approach gives a much more regular and systematic design than our transport entity.

The third advantage of the finite state machine approach is for protocol description. In some standards documents, including the OSI connection-oriented transport protocol standard (ISO 8073), the protocols are given as finite state machines of the type of Fig. 6-22. Going from this kind of description to a working transport entity is much easier if the transport entity is also driven by a finite state machine based on the one in the standard.

The primary disadvantage of the finite state machine approach is that it may be more difficult to understand than the straight programming example we used initially. However, this problem may be partially solved by drawing the finite state machine as a graph, as is done in Fig. 6-23.

6.4. EXAMPLES OF THE TRANSPORT LAYER

In this section we will once again examine our running examples to see what their transport layers are like. As usual, we will emphasize the protocol aspects in these examples.

6.4.1. The Transport Layer in Public Networks

Nearly all public networks use the connection-oriented OSI transport service (ISO 8072) and OSI transport protocols (ISO 8073). We have already looked at the OSI transport service; let us now look at the protocols.

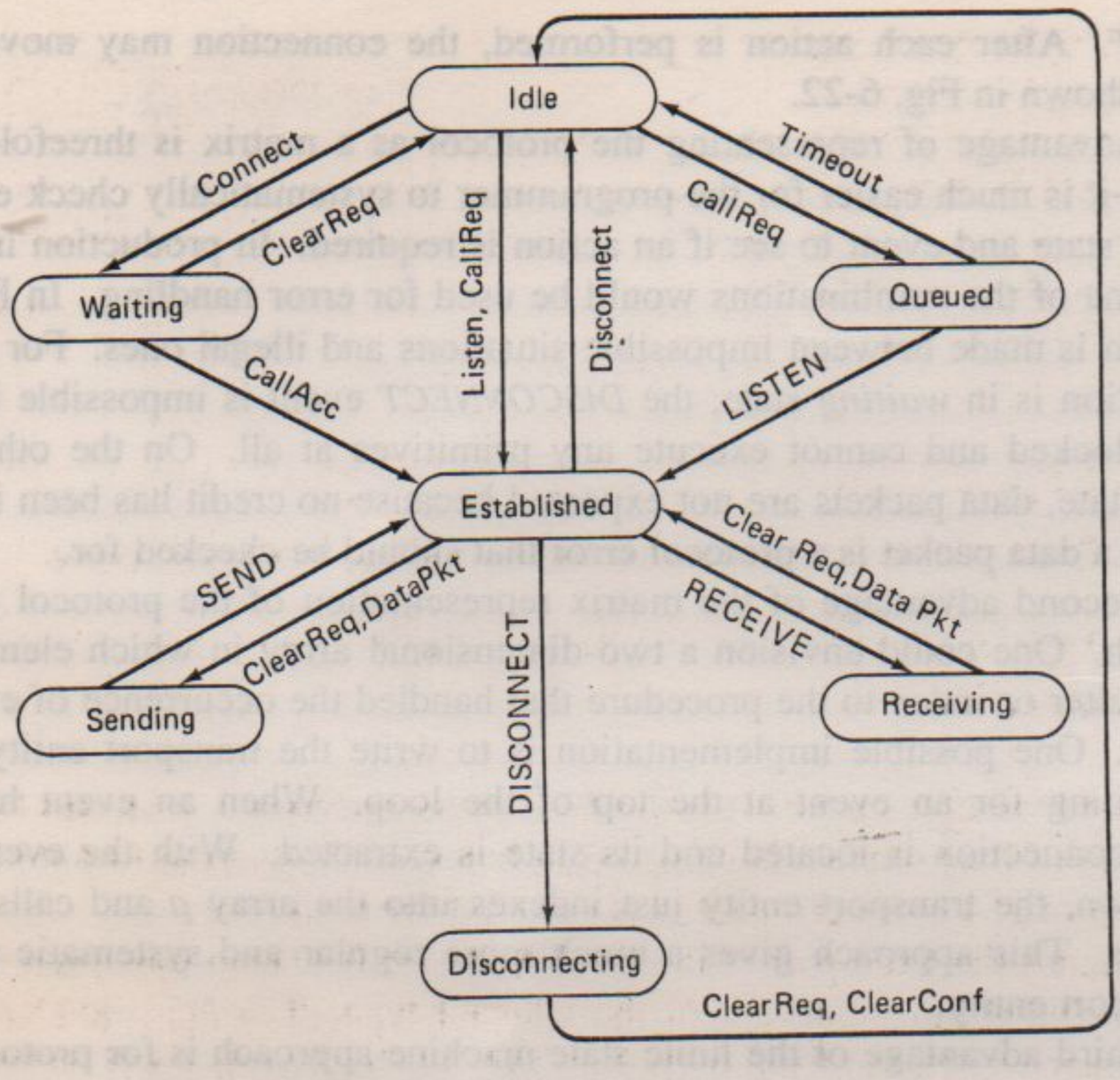


Fig. 6-23. The example protocol in graphical form. Transitions that leave the connection state unchanged have been omitted for simplicity.

As mentioned earlier, the OSI transport protocol has five variations, Classes 0 through 4. Each variation is intended for a specific type of network reliability, ranging to perfect to completely unreliable. We will briefly review the five protocol classes below. Class 0 (Simple class) is intended for use with type A (perfect) networks. It is primarily concerned with connection establishment and release, data transfer, and breaking large messages up into smaller TPDUs, if necessary. It uses the underlying network connection to do all the rest of the work (flow control, error control, and so on).

Class 1 (Basic error recovery class) differs from Class 0 in its ability to recover from *N-RESET*s generated by the network layer. After a network layer connection is broken, the transport entities establish a new network layer connection and continue from where they left off. In order to accomplish this recovery, the transport entities number the TPDUs, a feature not present in Class 0. Class 0 just gives up if the network connection breaks.

Class 2 (Multiplexing class) is the same as Class 0, except that it also supports multiplexing of multiple transport connections onto one network connection. Also, it permits explicit flow control using a credit scheme analogous to the one used in the example of Sec. 6-3. Finally, Class 2 also permits the use of expedited data.

Class 3 (Error recovery and multiplexing class) includes the features of Classes

1 and 2, namely, it can recover from network layer failures and it can also support the multiplexing of multiple transport connections onto one network connection.

Class 4 (Error detection and recovery class) is the most sophisticated and most interesting class. It has been designed to deal with unreliable network service, for example, datagram subnets that can lose, store, and duplicate packets. It is this class that we will primarily examine below.

The OSI transport protocol has 10 different TPDU types. Each TPDU has up to four parts:

1. A 1-byte field giving the length of the fixed plus variable headers.
2. A fixed part of the header (length depends on the TPDU type).
3. A variable part of the header (length depends on the parameters).
4. A user data.

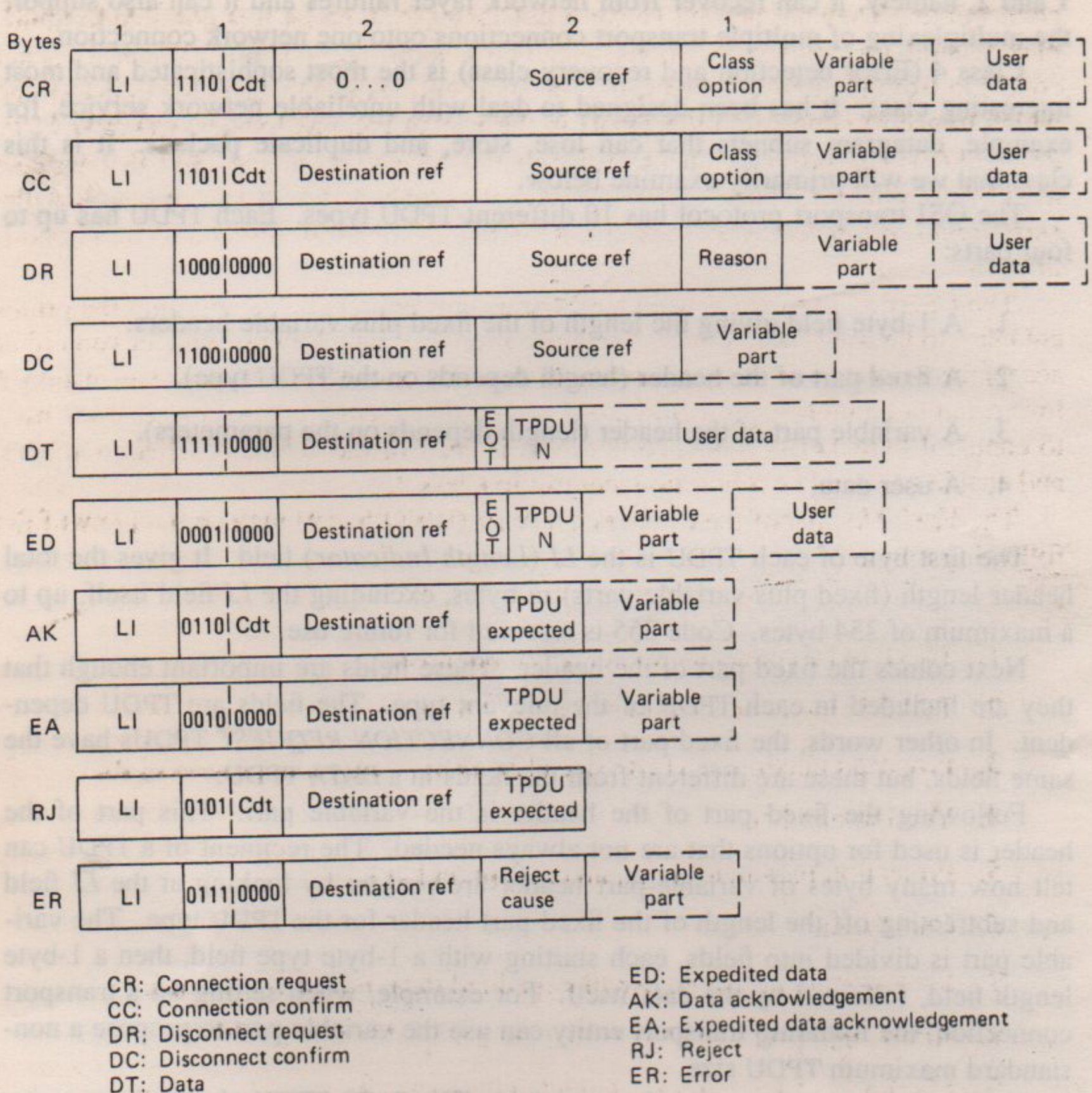
The first byte of each TPDU is the *LI* (*Length Indicator*) field. It gives the total header length (fixed plus variable parts) in bytes, excluding the *LI* field itself, up to a maximum of 254 bytes. Code 255 is reserved for future use.

Next comes the fixed part of the header. These fields are important enough that they are included in each TPDU of the relevant type. The fields are TPDU dependent. In other words, the fixed part of all *CONNECTION REQUEST* TPDUs have the same fields, but these are different from the fields in a *DATA* TPDU.

Following the fixed part of the header is the variable part. This part of the header is used for options that are not always needed. The recipient of a TPDU can tell how many bytes of variable-part header are present by looking at the *LI* field and subtracting off the length of the fixed-part header for the TPDU type. The variable part is divided into fields, each starting with a 1-byte type field, then a 1-byte length field, followed by the data itself. For example, when setting up a transport connection, the initiating transport entity can use the variable part to propose a non-standard maximum TPDU size.

Following the header come the user data. The *DATA* TPDU obviously contains user data, but some of the other TPDUs also contain a limited amount too. The formats of the 10 TPDU types are shown in Fig. 6-24. Some of these have minor variants that are not shown.

The *CONNECTION REQUEST*, *CONNECTION CONFIRM*, *DISCONNECT REQUEST*, and *DISCONNECT CONFIRM* TPDUs are completely analogous to the *CALL REQUEST*, *CALL ACCEPTED*, *CLEAR REQUEST*, and *CLEAR CONFIRM* packets used in X.25. To establish a connection, the initiating transport entity sends a *CONNECTION REQUEST* TPDU and the peer replies with *CONNECTION CONFIRM*. Similarly, to release a connection, either one of the transport entities sends a *DISCONNECT REQUEST* and the peer replies with *DISCONNECT CONFIRM*. The *DATA* and *EXPEDITED DATA* TPDUs are used for regular and expedited data,



CR: Connection request
CC: Connection confirm
DR: Disconnect request
DC: Disconnect confirm
DT: Data
ED: Expedited data
AK: Data acknowledgement
EA: Expedited data acknowledgement
RJ: Reject
ER: Error

Fig. 6-24. The OSI transport protocol TPDUs.

respectively. These two types are acknowledged by the *DATA ACKNOWLEDGEMENT* and *EXPEDITED DATA ACKNOWLEDGEMENT* TPDUs, respectively. Finally, the *REJECT* and *ERROR* TPDUs are used for error handling.

Let us now look at the various TPDU types one at a time in more detail. *CONNECTION REQUEST* is used to establish a connection. Like all TPDUs, it contains a 1-byte *LI* field giving the total header length (excluding the *LI* field itself).

Next comes a byte containing a 4-bit TPDU type and the *cdt* (credit) field. The Class 4 protocol uses a credit scheme for flow control, rather than a sliding window scheme, and this field tells the remote transport entity how many TPDU's it may initially send.

The *Destination reference* and *Source reference* fields identify transport

connections. They are needed because in Classes 2, 3, and 4 it is possible to multiplex several transport connections over one network connection. When a packet comes in from the network layer, the transport entities use these fields to determine which transport connection the TPDU in the packet belongs to. The *CONNECTION REQUEST* TPDU provides an identifier in the *Source reference* field that will be used by the initiator. The *CONNECTION CONFIRM* TPDU adds to that identifier the *Destination reference*, which is used by the destination for connection identification.

The *Class option* field is used by the transport entities for negotiating the protocol class to be used. The initiator makes a proposal, which the responder can either accept or reject. The responder can also make a counterproposal, suggesting a lower, but not a higher, protocol class. The field also contains two bits that are used to enable 4-byte TPDU sequence numbers instead of the standard 1-byte numbers, and enable or disable explicit flow control in Class 2.

The *Variable part* of the *CONNECTION REQUEST* TPDU may contain any of the following options:

1. TSAP to be connected to at the remote host.
2. TSAP being connected to at the local host.
3. Proposed maximum TPDU size (128 to 8192 bytes, in powers of 2).
4. Version number.
5. Protection parameter (e.g., an encryption key).
6. Checksum.
7. Some option bits (e.g., use of expedited data, use of checksum).
8. Alternative protocol classes that are acceptable to the initiator.
9. Maximum delay before acknowledging a TPDU, in milliseconds.
10. Throughput expected (average desired and minimum acceptable).
11. Residual error rate (average desired and maximum acceptable).
12. Priority (0 to 65535, with 0 being the highest priority).
13. Transit delay (average and maximum acceptable, in milliseconds).
14. How long to keep trying to recover after an *N-RESET*.

Some of the parameters, such as the alternative protocol classes, are intended for the remote peer. However, others, such as the quality of service parameters (throughput, residual error rate, etc.), are aimed at the network layer. If the

network layer is unable to provide at least the minimum service required, then the network layer itself rejects the connection with a *DISCONNECT REQUEST*, rather than putting it through.

The *User data* field may contain up to 32 bytes of data in Classes 1 through 4. It is not permitted in Class 0. This field may be used for any purpose the users wish (e.g., a password for remote login).

When the *CONNECTION REQUEST* TPDU arrives at the remote host, the transport entity there causes a *CONNECT.indication* primitive to the transport user. If the user decides to accept the incoming call, the remote transport layer replies to the initiator with a *CONNECTION CONFIRM* TPDU. The format and options in the *CONNECTION CONFIRM* TPDU are the same as those in the *CONNECTION REQUEST* TPDU.

In type A and B networks, this establishment procedure is sufficient, but in type C networks it is not, due to the possibility of delayed duplicate packets. To eliminate the possibility of old packets interfering with a connection establishment, a three-way handshake is used, with the *CONNECTION CONFIRM* TPDU itself being acknowledged with an *ACK* TPDU. Furthermore, after a connection is released, the *Source reference* and *Destination reference* are considered **frozen references** and not reused for an interval long enough to guarantee that all old duplicates have died out. Unlike Tomlinson's clock scheme, the OSI transport protocol does not describe how to deal with crashes. The safest way is just to wait for the maximum packet lifetime before rebooting.

To release a connection, a transport entity sends a *DISCONNECT REQUEST* TPDU to its peer. The format of the fixed part of the header is the same as for the *CONNECTION REQUEST* TPDU, except that the *Class option* field is replaced with the *Reason* field telling why the connection is being released. Among other possibilities are: the transport user executed a *DISCONNECT.request* primitive, there was a bad parameter in a TPDU, the TSAP to be connected to does not exist, or the network is congested. The *Variable part* of the header and the *User data* field (up to 64 bytes) can provide additional explanations.

The required response to a *DISCONNECT REQUEST* TPDU is a *DISCONNECT CONFIRM* TPDU. The only field that may be present in the *Variable part* is the checksum, if checksums are being used on the connection.

The formats of the *DATA* TPDU and *EXPEDITED DATA* TPDU shown in Fig. 6-24 are the normal types for Class 4. An additional format with a 4-byte *TPDU Nr* is also permitted to make sure that TPDU sequence numbers will not wrap around for a very long time. The *EOT* flag is set to 1 to indicate End Of Transport message. This flag is needed so the remote transport entity knows when to stop reassembling TPDUs and pass the resulting message to the remote transport user. The only *Variable part* field is the checksum, when it is in use.

The *DATA ACKNOWLEDGEMENT* and *EXPEDITED DATA ACKNOWLEDGEMENT* TPDUs acknowledge the receipt of TPDUs up to, but not including, the one whose sequence number is given in the acknowledgement. Thus that one is the one

expected next. The *Variable part* of both contain the checksum, if checksums are being used.

In addition, the *DATA ACKNOWLEDGEMENT* TPDU also contains information about the flow control window. In particular, the receiver can explicitly specify the lower edge and size of the window of sequence numbers that the sender is permitted to send. It is explicitly permitted for the receiver to reduce the window size (i.e., take back credits). For example, suppose the receiver has acknowledged TPDU 3 and given a window size of 8, thus permitting TPDUs 4 through 11. Shortly thereafter, the receiver notices that it is short on buffer space and decides to reduce the window to 3 by sending a new acknowledgement for 3 with a window of 2. This action permits only TPDUs 4 and 5 and forbids 6 through 11, even though these were previously permitted.

With a type C network, a problem arises if the two acknowledgements are delivered in the wrong order. The sender will wrongfully conclude that it may send TPDUs up to 11. To eliminate this problem, the protocol allows a *subsequence* field in the *Variable part*. Normally this field is not used, but when a second acknowledgement is sent for the same TPDU, a *subsequence* field is included with the value 1. A third acknowledgement carries a *subsequence* value of 2, and so on. This method allows the sender to deduce the order in which the acknowledgements were sent, and only accept the last one as valid.

The *REJECT* TPDU is only used in Classes 1 and 3, and is primarily used when resynchronizing after an *N-RESET*. Its function is to signal a problem and invite the peer to retransmit all TPDUs starting at the sequence number indicated. The credit value is also reset.

The final TPDU type is used to report protocol errors. The *Reject cause* field tells what was wrong. Possibilities include: invalid parameter code in the *Variable part*, invalid TPDU type, and invalid TPDU value.

As we mentioned earlier, the OSI transport layer defines a connectionless service and protocol, in addition to the connection-oriented service and protocol we have been describing at length. The purpose of the OSI connectionless transport service is to allow its users to send messages with the *T-UNITDATA* primitive without the overhead of first establishing and later releasing a connection.

The OSI connectionless transport service can work using either connection-oriented network service or connectionless network service. To a large extent, putting connectionless transport service on top of connection-oriented network service defeats the purpose of having connectionless transport service, but in some cases the only network service available is connection-oriented (e.g., a public X.25 network), so that case has been provided for.

When operating the connectionless transport service on top of a connectionless network service, each TPDU goes into a single packet. These TPDUs are not acknowledged and are not guaranteed to be reliably delivered. No promise is given about the order in which TPDUs are delivered.

The OSI connectionless service uses a connectionless protocol. Only one TPDU

format is used, as shown in Fig. 6-25. This TPDU is similar to those used in the connection-oriented service. The *Variable part* contains the source and destination TSAP addresses, and optionally a checksum.

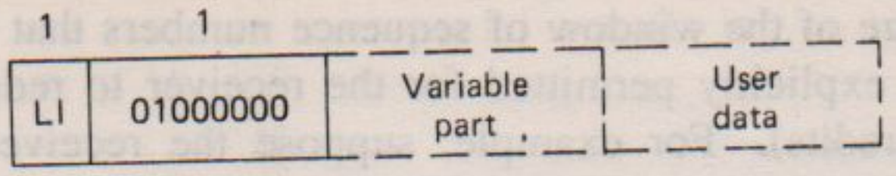


Fig. 6-25. The OSI connectionless TPDU format.

Work on the implementation of OSI and related transport protocols is discussed in (Chong, 1986; and Watson and Mamrak, 1987).

6.4.2. The Transport Layer in the ARPANET (TCP)

In the original ARPANET design, the subnet was assumed to offer virtual circuit service (i.e., be perfectly reliable). The first transport layer protocol, **NCP** (**Network Control Protocol**), was designed with a perfect subnet in mind. It just passed TPDUs (called messages) to the network layer and assumed that they would all be delivered in order to the destination. Experience showed that the ARPANET was indeed reliable enough for this protocol to be completely satisfactory for traffic within the ARPANET itself.

However, as time went on, and the ARPANET grew into the ARPA Internet, which included many LANs, a packet radio subnet, and several satellite channels, the end-to-end reliability of the subnet decreased. This development forced a major change in the transport layer and led to the gradual introduction of a new transport layer protocol, **TCP** (**Transmission Control Protocol**), which was specifically designed to tolerate an unreliable subnet (type C in OSI terms). Associated with TCP was a new network layer protocol, **IP**, which we studied in the previous chapter. Today TCP/IP is not only used in the ARPANET and ARPA Internet, but in many commercial systems as well.

A TCP transport entity accepts arbitrarily long messages from user processes, breaks them up into pieces not exceeding 64K bytes, and sends each piece as a separate datagram. The network layer gives no guarantee that datagrams will be delivered properly, so it is up to TCP to time out and retransmit them as need be. Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence.

Every byte of data transmitted by TCP has its own private sequence number. The sequence number space is 32 bits wide to make sure that old duplicates have long since vanished by the time the sequence numbers have wrapped around. TCP does, however, explicitly deal with the problem of delayed duplicates when attempting to establish a connection, using the three-way handshake for this purpose.

Figure 6-26 shows the header used by TCP. The first thing that strikes one is

that the minimum TCP header is 20 bytes. Unlike OSI Class 4, with which it is roughly comparable, TCP has only one TPDU header format. Let us dissect this large header field by field. The *Source port* and *Destination port* fields identify the end points of the connection (the TSAP addresses in OSI terminology). Each host may decide for itself how to allocate its ports.

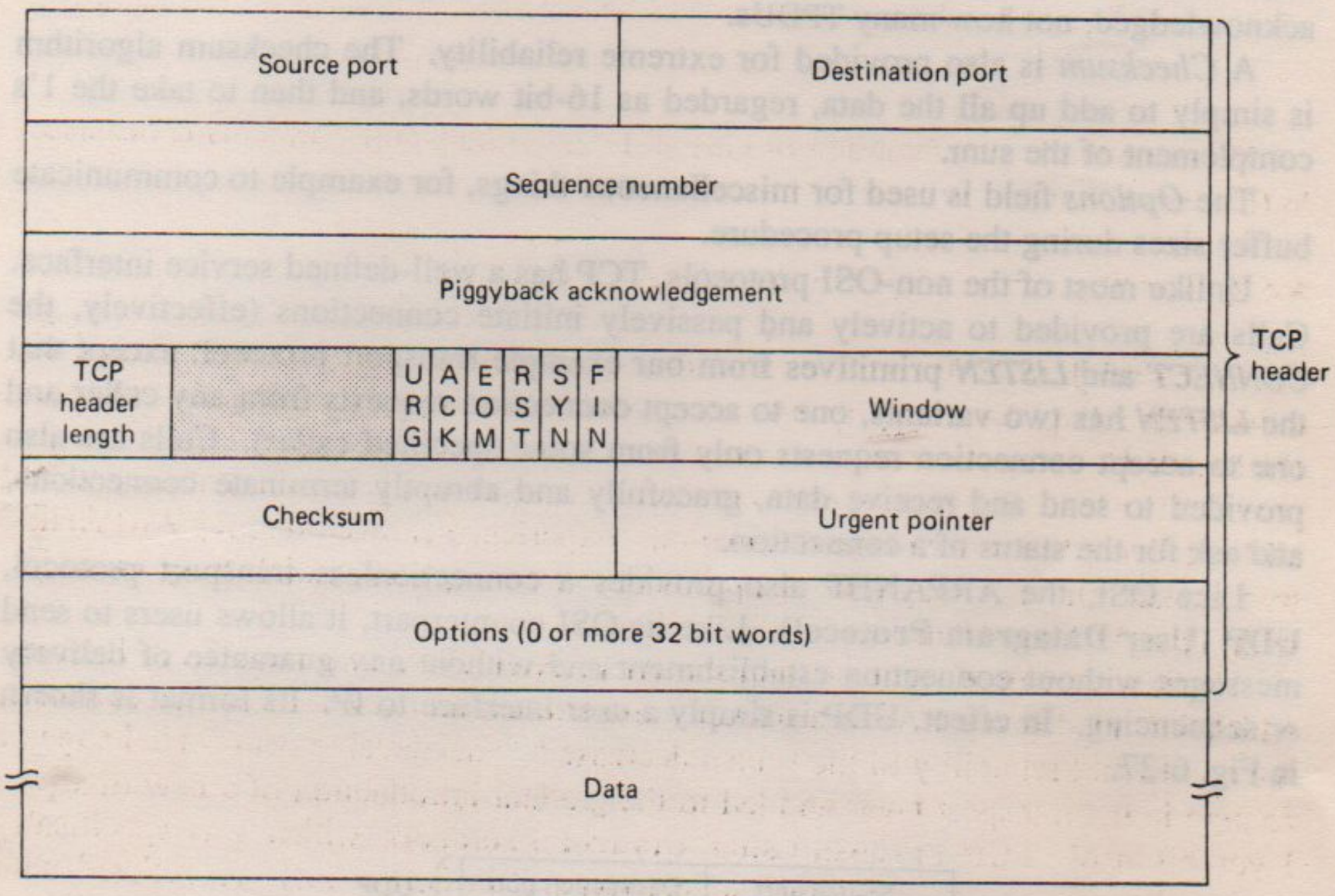


Fig. 6-26. The TCP TPDU structure.

The *Sequence number* and *Piggyback acknowledgement* fields perform their usual functions. They are 32 bits long because every byte of data is numbered in TCP.

The *TCP header length* tells how many 32-bit words are contained in the TCP header. This information is needed because the *Options* field is variable length, so the header is too.

Next come six 1-bit flags. *URG* is set to 1 if the *Urgent pointer* is in use. The *Urgent pointer* is used to indicate a byte offset from the current sequence number at which urgent data are to be found. This facility is in lieu of interrupt messages. The *SYN* bit is used to establish connections. The connection request has $SYN = 1$ and $ACK = 0$ to indicate that the piggyback acknowledgement field is not in use. The connection reply does bear an acknowledgement, so it has $SYN = 1$ and $ACK = 1$. In essence the *SYN* bit is used to denote *CONNECTION REQUEST* and *CONNECTION CONFIRM*, with the *ACK* bit used to distinguish between those two possibilities. The *FIN* bit is used to release a connection. It specifies that the

sender has no more data. After closing a connection, a process may continue to receive data indefinitely. The *RST* bit is used to reset a connection that has become confused due to delayed duplicate *SYN*s or host crashes. The *EOM* bit indicates End Of Message.

Flow control in TCP is handled using a variable-size sliding window. A 16-bit field is needed, because *Window* tells how many bytes may be sent beyond the byte acknowledged, not how many TPDUs.

A *Checksum* is also provided for extreme reliability. The checksum algorithm is simply to add up all the data, regarded as 16-bit words, and then to take the 1's complement of the sum.

The *Options* field is used for miscellaneous things, for example to communicate buffer sizes during the setup procedure.

Unlike most of the non-OSI protocols, TCP has a well-defined service interface. Calls are provided to actively and passively initiate connections (effectively, the *CONNECT* and *LISTEN* primitives from our example transport protocol, except that the *LISTEN* has two variants, one to accept connection requests from any caller and one to accept connection requests only from some specified caller). Calls are also provided to send and receive data, gracefully and abruptly terminate connections, and ask for the status of a connection.

Like OSI, the ARPANET also provides a connectionless transport protocol, **UDP (User Datagram Protocol)**. Like its OSI counterpart, it allows users to send messages without connection establishment and without any guarantee of delivery or sequencing. In effect, UDP is simply a user interface to IP. Its format is shown in Fig. 6-27.

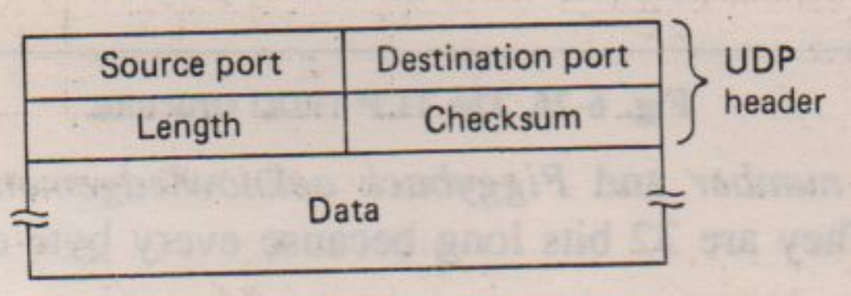


Fig. 6-27. The ARPANET UDP format.

Comparison of OSI Class 4 and TCP

The OSI Class 4 transport protocol (often called **TP4**), and TCP have numerous similarities but also some differences. In this section we will examine these similarities and differences (Groenbaek, 1986). Let us start with the points on which the two protocols are the same. Both protocols are designed for providing a reliable, connection-oriented, end-to-end transport service on top of an unreliable network that can lose, garble, store, and duplicate packets. Both must deal with the

worst case problems, such as a subnet that can store a valid sequence of packets and then "play them back" later.

The two protocols are also alike in that both have a connection establishment phase, a data transfer phase, and then a connection release phase. The general concepts of establishing, using, and releasing connections are also similar, although some of the details differ. In particular, both TP4 and TCP use three-way handshakes to eliminate potential difficulties caused by old packets suddenly emerging and causing trouble.

However, the two protocols also have some notable differences, which are listed in Fig. 6-28. First, TP4 uses nine different TPDU types, whereas TCP only has one. This difference results in TCP being simpler, but it also needs a larger header, because all fields must be present in all TPDUs. The minimum size of the TCP header is 20 bytes; the minimum size of the TP4 header is 5 bytes. Both protocols allow optional fields that can increase the header size above the minimum.

Feature	OSI TP4	TCP
Number of TPDU types	9	1
Connection collision	2 connections	1 connection
Addressing format	Not defined	32 bits
Quality of service	Open ended	Specific options
User data in CR	Permitted	Not permitted
Stream	Messages	Bytes
Important data	Expedited	Urgent
Piggybacking	No	Yes
Explicit flow control	Sometimes	Always
Subsequence numbers	Permitted	Not permitted
Release	Abrupt	Graceful

Fig. 6-28. Differences between the OSI TP4 protocol and TCP.

A second difference is what happens if two processes simultaneously attempt to set up connections between the same two TSAPs (i.e., a connection collision). With TP4, two independent, full-duplex connections are established. With TCP, a connection is identified by a pair of TSAPs, so only one connection is established.

A third difference is in the addressing format used. TP4 does not specify the exact format of a TSAP address; TCP uses 32-bit numbers as TSAPs.

The issue of quality of service is also handled differently in the two protocols and forms the fourth difference. TP4 has a rather elaborate and open-ended mechanism for a three-way negotiation of the quality of service. This negotiation involves the calling process, the called process, and the transport service itself. Many parameters can be specified, and both target and minimum acceptable values can be given. TCP, in contrast, does not have a quality of service field at all, but the underlying IP service has an 8-bit field that allows a choice to be made out of a limited number of combinations of speed and reliability.

A fifth difference is that TP4 allows user data to be carried in the *CR* TPDU, but TCP does not allow user data in the initial TPDU. The initial data (e.g., a password) might be necessary to decide whether or not a connection should be established. With TCP, making establishment depend on user data is not possible.

The previous four differences relate to the connection establishment phase. The next five concern the data transfer phase. A very basic difference is the model of data transport. The TP4 model is that of an ordered series of messages (TSDUs in OSI terminology). The TCP model is that of a continuous stream of bytes, without any explicit message boundaries. In practice, however, the TCP model is not really a pure byte stream because a library procedure, *push*, can be called to flush out any data buffered but not yet sent. When the remote user does a read, data from before and after the *push* will not be combined, so in a sense *push* can be thought of as defining a sort of message boundary.

The seventh difference concerns how important data that need special processing (such as BREAK characters) are dealt with. TP4 has two independent message streams, regular data and expedited data, multiplexed together. Only one expedited message may be outstanding at any instant. TCP uses the *Urgent* field to indicate that some number of bytes within the current TPDU are special and should be processed out of order.

The eighth difference is the absence of piggybacking in TP4 and its presence in TCP. This difference is not quite as significant as it may first appear since it is possible for a transport entity to put two TPDUs, for example, DT and AK in a single network packet.

The ninth difference is the way flow control is handled. TP4 can use a credit scheme, but it can also rely on the window scheme of the network layer to regulate the flow. TCP always uses an explicit flow control mechanism with the window size specified in each TPDU.

The tenth difference relates to this window scheme. In both protocols the receiver is entitled to reduce the window at will. This possibility potentially gives rise to problems if the grant of a large window and its subsequent retraction arrive in the wrong order. In TCP there is no solution to this problem. In TP4 it is solved by the subsequence number that is included in the retraction, thus allowing the sender to determine that the small window followed, rather then preceded, the large one.

Finally, our eleventh and last difference between the two protocols is in the way connections are released. TP4 uses an abrupt disconnection in which a series of data TPDUs may be followed directly by a *DR* TPDU. If the data TPDUs are lost, they will not be recovered by the protocol and information will be lost. TCP uses a three-way handshake to avoid data loss upon disconnection. The OSI model handles this problem in the session layer. It is worth noting that the U.S. National Bureau of Standards was so displeased with this property of TP4 that it introduced extra TPDUs into the transport protocol to allow disconnection without data loss. As a consequence, the U.S. and international versions of TP4 are not identical.

NETWORKING IN UNIX

A large number of machines on the ARPANET and the ARPA Internet run Berkeley UNIX. For this reason, it is worth taking a short look at how it handles networking. Berkeley UNIX supports TCP/IP, which is accessed through a set of primitives.

In contrast to the OSI primitives, which are highly abstract, the Berkeley primitives are much more specific. They are implemented as a set of system calls that allow users to access the transport service. The major system calls are listed in Fig. 6-29. Several minor calls and some variants of the major calls are not shown, for simplicity.

Socket	Create a TSAP of a given type
Bind	Associate an ASCII name to a previously created socket
Listen	Create a queue to store incoming connection requests
Accept	Remove a connection request from the queue or wait for one
Connect	Initiate a connection with a remote socket
Shutdown	Terminate the connection on a socket
Send	Send a message through a given socket
Recv	Receive a message on a given socket
Select	Check a set of sockets to see if any can be read or written

Fig. 6-29. The principal transport service calls in Berkeley UNIX.

Central to the service interface is the concept of a **socket**, which is similar to an OSI TSAP. Sockets are end points to which connections can be attached from the bottom (the operating system side) and to which processes can be attached from the top (the user side). The *socket* system call creates a socket (a data structure within the operating system). The parameters to the call specify the address format (e.g., an ARPA Internet name), the socket type (e.g., connection-oriented or connectionless), and the protocol (e.g., TCP/IP).

Once a socket has been created, buffer space can be allocated to it for storing incoming connection requests. This space is allocated using the *listen* call. A socket specified in a *listen* call then becomes a passive end point waiting for a connection request to arrive from outside.

In order for a remote user to send a connection request to a socket, the socket has to have a name (TSAP address). Names can be attached to sockets by the *bind* system call. Once bound, a name can be published or distributed in some way, so that remote processes can address the socket.

The way a user process attaches itself to a socket to passively await the arrival of a connection request is the *accept* call (essentially the same as the *listen* call in our example). If a request has already arrived, it will be taken from the socket's queue; otherwise, the process will block until a request comes in (unless the socket

has been specified as nonblocking). Either way, when a request is available, a new socket is created and the new socket is used for the connection end point. In this manner, a single well-known port can be used to establish many connections.

To initiate a connection to a remote socket, a process can make the *connect* system call specifying a local socket and a remote name as parameters. This call establishes a connection between the two sockets. Alternatively, if the sockets are of the connectionless type, the operating system records an association between the two, so that *send*s on the local socket result in messages being sent to the remote one, even though no formal connection exists.

To terminate a connection or association the *shutdown* call is used. The two directions of a full duplex connection can be independently shut down.

The calls *send* and *recv* are used to send and receive messages. Several variations of the basic call are present.

Finally, we have the *select* call. This system call is useful for processes that have several connections established. In many cases such a process wants to do a *recv* on any socket that has a message waiting for it. Unfortunately it does not know which sockets have messages pending and which do not. If it picks one socket at random, it may end up blocking for a long time waiting for a message, while messages are waiting on several other sockets. The *select* call allows it to block until reads (or writes) are possible on some set of sockets specified by the parameters. For example, a process can say that it wants to block until a message is available on any one of a given set of sockets. When the call terminates, the caller is told which sockets have messages pending and which do not.

6.4.3. The Transport Layer in MAP and TOP

MAP and TOP use the OSI transport protocol. Since they use a connectionless protocol (ISO 8473) in the network layer, they are forced to use the TP4 variant in the transport layer.

TP4 contains several options that can be negotiated at the time a connection is established. Specific choices have been made for some of them. To start with, the computer initiating the connection is required to specify the use of Class 4 in the *CR* TPDU, and the responder is required to accept it in the *CC* TPDU. If either side is unable or unwilling to do this, the connection cannot be established.

Both the 7-bit and 31-bit sequence number options must be supported. Implementations are encouraged to use the 31-bit option all the time, except in those circumstances in which the underlying network cannot support it.

Expedited data must be supported by all implementations, even though the OSI standard says that it is optional. If during the connection establishment either peer rejects this option, the connection must be rejected because some of the upper layers use this facility.

Finally, all MAP and TOP implementations must be prepared to use the

software checksum option, in which transport entities checksum each TPDU before sending it or after receiving it. Doing the checksum in software is very expensive in terms of CPU time, but guards against the possibility of losing data due to memory errors. Use of this option for any given connection is optional, but if it is desired, it must be available.

6.4.4. The Transport Layer in USENET

USENET does not have an official transport protocol. Each pair of communicating machines can negotiate the use of any desired transport protocol (or none at all). Many pairs of machines use TCP/IP, but X.25 and the *uucp* protocol described in Chap. 4 are also widely used.

The upper layers do not require any specific transport protocol, although they do need a way to establish a reliable connection between machines for the purpose of logging in. If this goal can be achieved, each pair of machines can use any mutually agreeable transport protocol, or none at all, if the underlying communication is reliable enough (e.g., over a LAN).

6.5. SUMMARY

The purpose of the transport layer is to bridge the gap between what the network layer offers and what the transport user wants. It also serves to isolate the upper layers from the technology of the network layer by providing a standardized service definition. In this way, changes in the network technology will not require changes to software in the higher layers.

The OSI transport service definition views a connection as having three phases: establishment, use, and release. For each phase, service primitives are available to perform the required actions. Connectionless service is also provided for.

Network layer service can be categorized as A, B, or C, depending on how reliable it is. For type A (reliable) networks, simple protocols can be used. For type B (almost reliable) networks, the transport protocol must be able to recover from *N-RESET*s. For type C networks, the transport protocol must use complex mechanisms to deal with many subtle errors that may occur.

Connection management is a key responsibility of the transport layer. For type C networks, connection establishment needs to be quite elaborate, usually involving a three-way handshake. Connection release can also be a problem, as our example of the two-army problem demonstrated. One possible solution is to use timer-based connections.

After finishing with connection management, we studied a sample transport layer using X.25 as the network service. In this example we saw one possible way of relating the abstract OSI service primitives to executable procedures. We also saw how the protocol can be represented as a finite state machine.

Finally, we examined the transport layer in our usual running examples and looked at the two main transport protocols currently in use, the OSI transport protocol and TCP. The similarities and differences between these two protocols were pointed out.

PROBLEMS

1. The dynamic buffer allocation scheme of Fig. 6-18 tells the sender how many buffers it has beyond the acknowledged message. An alternative way of conveying the same information would be for the buffer field to simply tell how many additional buffers, if any, had been allocated. In this method the sender maintains a counter that is incremented by the contents of the buffer field in arriving messages, and decremented when a message is sent for the first time. Are the two methods equally good?

2. A user process sends a stream of 128-byte messages to another user process over a connection. The receiver's main loop consists of two actions, fetch message and process message. The time required to fetch and process a message has an exponential probability density, with mean 10 msec. The window mechanism allows up to 16 outstanding messages at any instant. All communication lines in the subnet are 230 kbps, but due to delays in the subnet, the arrival pattern at the receiver is approximately Poisson. Measurements show that the time for an acknowledgement to get back to the sender, measured from the first bit of transmission, is 200 msec. Use queueing theory to determine the mean number of bytes of buffer space required at the receiving host.

3. A group of N users located in the same building are all using the same remote computer via an X.25 network. The average user generates L lines of traffic (input + output) per hour, on the average, with the mean line length being P bytes, excluding the X.25 headers. The packet carrier charges C cents per byte of user data transported, plus X cents per hour for each X.25 virtual circuit open. Under what conditions is it cost effective to multiplex all N transport connections onto the same X.25 virtual circuit, if such multiplexing adds 2 bytes of data to each packet? Assume that even one X.25 virtual circuit has enough bandwidth for all the users.

4. Class 0 of the OSI transport protocol does not have any explicit flow control procedure. Does this mean that a fast sender can transmit data can drown a slow receiver in data?

5. Imagine a generalized n-army problem, in which the agreement of any two of the armies is sufficient for victory. Does a protocol exist that allows blue to win?

6. In a network that has a maximum packet size of 128 bytes, a maximum packet lifetime

of 30 sec, and an 8-bit packet sequence number, what is the maximum data rate per connection?

7. Suppose that the clock-driven scheme for generating initial sequence numbers is used with a 15-bit wide clock counter. The clock ticks once every 100 msec, and the maximum packet lifetime is 60 sec. How often need resynchronization take place
(a) in the worst case?
(b) when the data consumes 240 sequence numbers/min?

8. Why does the maximum packet lifetime, T, have to be large enough to ensure that not only the packet, but also its acknowledgements have vanished?

9. Imagine that a two-way handshake rather than a three-way handshake were used to set up connections. In other words, the third message was not required. Are deadlocks now possible? Give an example or show that none exist.

10. Consider the problem of recovering from host crashes (i.e., Fig. 6-20). If the interval between writing and sending an acknowledgement, or vice versa, can be made relatively small, what are the two best sender-receiver strategies for minimizing the chance of a protocol failure?

11. Are deadlocks possible with the transport entity described in the text?

12. What happens when the user of the transport entity given in Fig. 6-21 sends a zero length message? Discuss the significance of your answer.

13. Out of curiosity, the implementer of the transport entity of Fig. 6-21 has decided to put counters inside the *sleep* procedure to collect statistics about the *conn* array. Among these are the number of connections in each of the seven possible states, n_i ($i = 1, \ldots, 7$). After writing a massive FORTRAN program to analyze the data, our implementer discovered that the relation $\Sigma n_i = MaxConn$ appears to always be true. Are there any other invariants involving only these seven variables?

14. For each event that can potentially occur in the transport entity of Fig. 6-21, tell whether it is legal or not when the user is sleeping in *sending* state.

15. The X.25 protocol does not use subsequence numbers like TP4. Is this simply because X.25 was invented years ago, before the idea of subsequence numbers had been thought of, or is there a different reason?

16. Consider the problem of internetworking with type C connectionless subnets. If a TPDU passes through a network whose maximum packet size is smaller than the standard TPDU size, will this cause problems with TP4? How about with TCP?

17. TCP only allows one connection to exist between any pair of TSAPs. Do you think this is also true of TP4? Discuss your answer.

18. Discuss the advantages and disadvantages of credits versus sliding window protocols.

19. TCP uses a single transport protocol header, whereas OSI has many of them. Discuss the advantages and disadvantages of each method.

20. Datagram fragmentation and reassembly is handled by IP, and is invisible to TCP. Does this mean that TCP does not have to worry about data arriving in the wrong order?

21. Modify the program of Fig. 6-21 to do error recovery. Add a new packet type, *reset*, that can arrive after a connection has been opened by both sides but closed by neither. This event, which happens simultaneously on both ends of the connection, means that any packets that were in transit have either been delivered or destroyed, but in either case no longer are in the subnet.

22. Modify the program of Fig. 6-21 to multiplex all transport connections onto a single X.25 virtual circuit. This change will probably require you to create and manage an explicit transport header to keep track of which packet belongs to which connection.

23. Write a program that simulates buffer management in a transport entity using a sliding window for flow control rather than the credit system of Fig. 6-21. Let higher layer processes randomly open connections, send data, and close connections. To keep it simple, have all the data travel from machine *A* to machine *B*, and none the other way. Experiment with different buffer allocation strategies at *B*, such as dedicating buffers to specific connections versus a common buffer pool, and measure the total throughput achieved by each one.