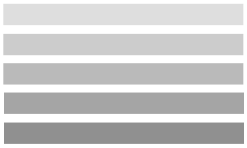

PART FOUR

MIDDLE LAYERS

12 THE TRANSPORT LAYER

The transport layer is the basic end-to-end building block of host networking. Everything above the transport layer is distributed-application–oriented; everything below the transport layer is transmission-network–oriented. In the work on OSI, the upper layers have tended to be the province of people with a computer systems (particularly operating systems) background; the lower layers, of people with an electrical engineering (particularly transmission systems) background. The imperfect alignment of the very different perspectives of the "computer people" and the "telecommunications people" has led to the definition of five different classes of transport protocol in OSI, each one tailored to a particular vision of the way in which hosts are properly interconnected by networks. The work on TCP/IP, on the other hand, was carried out by a relatively small group of people who managed to avoid being identified as "upper layer" or "lower layer" until long after the fact. This group agreed on a single model for network interconnection and defined a single transport protocol (TCP).

The OSI reference model explains the purpose of the transport layer in the following terms:

> "To provide transparent transfer of data between session entities and relieve them from any concern with the detailed way in which reliable and cost effective transfer of data is achieved."
> "To optimize the use of available network service to provide the performance required by each session entity at minimum cost."
> "To provide transmission of an independent, self-contained transport-service-data-unit from one transport-service-access-point to another in a single service access." (ISO/IEC 7498: 1993)

In plain-speak, this means that the OSI transport layer provides a

reliable data pipe for the upper layers as part of a connection-oriented service, and simple datagram delivery as part of a connectionless service. In the TCP/IP protocol suite, the reliable connection-oriented service is provided by the transmission control protocol [TCP] and the simple datagram service by the user datagram protocol UDP. During connection-oriented operation, the data stream submitted to the transport layer by transport user A must be delivered to transport user B without loss. There may be no duplication of any of the octets in the data stream, and the octets must be delivered in the same order as that in which they were submitted (end-to-end sequence control). The transport layer must also provide end-to-end error detection and recovery: the detection of (and recovery from) errors introduced into the data stream by the network (data corruption).

In both connection-oriented and connectionless modes, the transport layer must also do what it can to optimize the use of the network's resources, given quality of service objectives specified by the transport users. For example, if a local session entity requests a transport connection to a remote host, and the transport layer recognizes that a network connection already exists to that host, *and* the quality of service objectives for both the existing and newly requested transport connections can be satisfied, the local transport entity can decide to multiplex both transport connections onto a single network connection.

Some applications transfer large amounts of data and don't want any of those data lost; they want reliable transport connections. For some applications, however, loss of individual data elements is either irrelevant or annoying but not disruptive; and some applications prefer to provide reliability themselves. For these applications, the reliability provided by transport connections is redundant, and the connectionless transport service is the better choice. The first part of this chapter describes the connection-oriented transport services and protocols of OSI and TCP/IP in detail; their connectionless counterparts are described much more briefly (since they are much simpler) at the end of the chapter.

OSI's Connection-oriented Transport Service

The OSI transport service definition (ISO/IEC 8072: 1993) identifies the functions associated with the connection-oriented transport service (COTS), the transport service primitives and parameters used to define the service, and the parameters used to define transport quality of service.

The following *end-to-end* functions are elements of the connection-oriented transport service:

- *Multiplexing* of transport connections onto network connections (and demultiplexing of them at the destination).
- *Sequence control* to preserve the order of transport service data units submitted to the transport layer.
- *Segmenting* transport service data units into multiple transport protocol data units (and reassembling of the original transport service data units at the destination).
- *Blocking* multiple transport service data units into a single transport protocol data unit (and unblocking it into the original transport service data units at the destination).
- *Concatenating* multiple transport protocol data units into a single network service data unit (NSDU) (and separating it into the individual transport protocol data units at the destination).
- *Error detection* to ensure that any difference between the data submitted to the transport layer at the source and the data that arrive at the destination is detected.
- *Error recovery* to take appropriate action when errors are discovered by the "error-detection" function.
- *Flow control* to regulate the amount and pacing of data transferred between transport entities and between the adjacent session and transport layers.
- *Expedited data transfer* to permit certain transport service user data to bypass normal data flow control. (Similar to "urgent data" in TCP which is examined later in this chapter).

The primitives and parameters of the connection-oriented transport service are depicted in Table 12.1.

TCP/IP's Reliable Stream Service

TCP provides a reliable connection-oriented transport service. RFC 793 describes TCP as providing "robustness in spite of unreliable communications media" and "data transfer that is reliable, ordered, full-duplex, and flow controlled." The end-to-end functions of TCP include:

- *Multiplexing* of multiple pairs of processes within upper-layer protocols.
- *Sequence control* to preserve the order of octets submitted to TCP.
- *Flow control* to regulate the flow of data across the transport connection.

TABLE 12.1 OSI Transport Service Primitives and Parameters

Primitives		*Parameters*
T-CONNECT	request indication	Called address, calling address, expedited data option, quality of service, TS user data
T-CONNECT	response confirmation	Responding address, expedited data option, quality of service, TS user data
T-DATA	request indication	TS user data
T-EXPEDITED-DATA	request indication	TS user data
T-DISCONNECT	request indication	TS user data TS user data, disconnect reason

- *Push,* whereby a sending upper-layer protocol process can force both sending and receiving TCP processes to deliver data to the receiving upper-layer protocol process.
- *Urgent data,* an interrupt data service whereby a sending upper-layer protocol process may request that data marked "urgent" be processed quickly by the receiving upper-layer protocol process.

RFC 793 serves as both protocol specification and service definition, specifying the interaction between upper-layer protocol (ULP) processes and TCP (Tables 12.2 and 12.3), the format of the transport protocol, and its operation.

Interfaces to Transport Services

The formal interaction that takes place between a service user and the service provider for connection establishment, data transfer, expedited data transfer, and connection release is the same for the transport layer as it is for the upper layers which have already been discussed. It is perhaps more interesting to consider how an interface to the transport service

might be implemented in a representative UNIX implementation, such as Wisconsin ARGO[1] 1.0 (*Wisconsin ARGO 1.0 Kernel Programmer's Guide for Operating Systems 4.3*), and 4.3 RENO UNIX (*RENO 4.3 UNIX Operating System*).

Socket Interface for OSI Transport Service

In ARGO 1.0 UNIX, the OSI upper layers are implemented in user space—that portion of the UNIX operating system available for application programming—making the application service elements accessible to application developers through library/procedure calls rather than system calls. OSI's transport layer is implemented in the UNIX kernel along with most of OSI's network layer; some network-layer functions, and the data link and physical-layer components appropriate for the transmission technologies supported (e.g., a LAN or serial interface), reside in network interfaces (see Figure 12.1).

The transport service is implemented as an extension to the UNIX interprocess communication that supports TCP and UDP. Like TCP and UDP, OSI transports are accessed via *sockets* (type *sock-seqpacket*). The

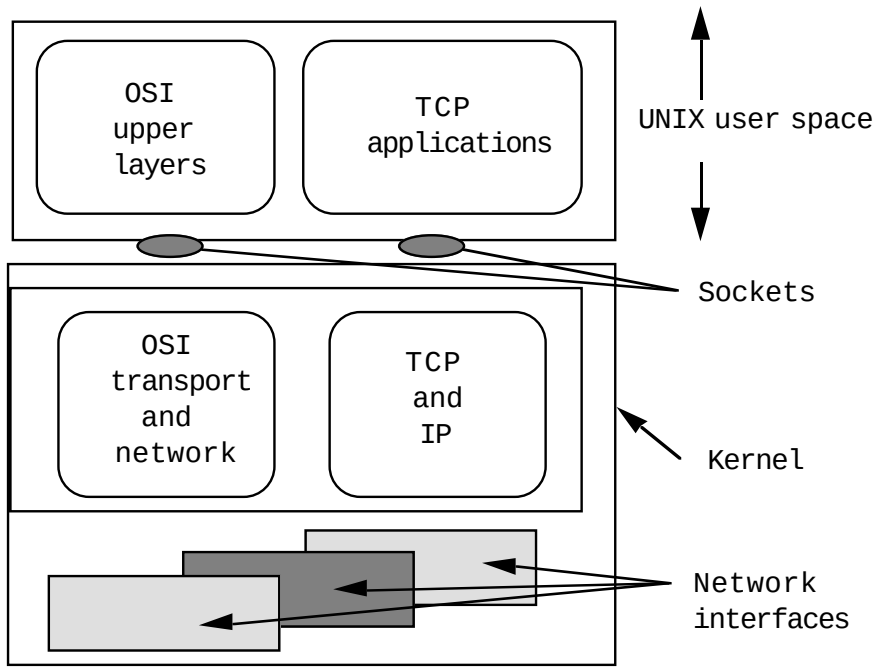

FIGURE 12.1 ARGO 1.0

1. *ARGO* stands for "A Really Good OSI" implementation.

TABLE 12.2 TCP service primitives (from ULP to TCP)

Request Primitives	*Parameters*
Unspecified Passive Open	source port, ULP_timeout, ULP_timeout_action, precedence, security_range—actually, if IP, any IP option supported
Fully Specified Passive Open	source port, destination port, destination (IP) address, ULP_timeout, ULP_timeout_action, precedence, security_range
Active Open	source port, destination port, destination (IP) address, ULP_timeout, ULP_timeout_action, precedence, security_range
Active Open with data	source port, destination port, destination (IP) address, ULP_timeout, ULP_timeout_action precedence, security_range, data, data length, PUSH flag, URGENT flag
Send	local connection name, data, data length , PUSH flag, URGENT flag, ULP_timeout, ULP_timeout_action
Allocate	local connection name, data length
Close	local connection name—graceful
Abort	local connection name—disruptive
Status	local connection name

transport service primitives and parameters are mapped onto a set of UNIX system calls (see Table 12.4).

Greatly abbreviated, the process of establishing a transport connection in ARGO 1.0 is as follows. The UNIX system call *socket()* creates a communication endpoint. The parameters of the *socket()* system call used to establish an OSI transport connection include the address format (AF-ISO), type (sock-seqpacket), and protocol identifier (ISO TP). The UNIX system call *bind()* is used to assign an address to a socket. This now addressable endpoint can be used to initiate or listen for an incoming transport connection.

A transport user—a session entity or an application that runs directly over the transport service, in user space—initiates a transport connection by issuing the *connect()* system call to another transport user. If the called transport user is also located on a UNIX machine running ARGO 1.0, it must have already issued a *socket()* system call (to create its own communication endpoint) and *bind()* to assign an address to the

TABLE 12.3 TCP Service Primitives (from TCP to ULP)

Request Primitives	*Parameters*
Open Id	local connection name, source port, destination port, destination (IP) address
Open failure	local connection name
Open success	local connection name—completion of one of the open requests
Deliver	local connection name, data, data length, URGENT flag—data have arrived across the named connection
Closing	local connection name—remote ULP issued close, TCP has delivered all outstanding data
Terminate	local connection name—indication of remote reset, service failure, or connection closing by local ULP
Status response	local connection name, source port, source (IP) address, destination port, destination (IP) address, connection state, amount of data local TCP willing to accept, amount of data allowed to send, amount of data waiting acknowledgment, amount of data pending receipt by local ULP, urgent state, precedence, security, ULP_timeout
Error	local connection name, error description—indication that

TABLE 12.4 Mapping of Transport Service Primitives to UNIX System Calls

TS Primitives		*UNIX System Calls*
T-CONNECT	request	socket(), bind(), connect(), setsockopt()
	indication	Return from accept(), getsockopt(), following socket(), bind(), listen()
	response	(No applicable system calls)
	confirmation	Return from connect()
T-DATA	request	recv(), sendv(), (new calls)
	indication	Return from recv(), sendv(), select()
T-EXPEDITED-DATA	request	sendv() with MSG_OOB flag set
	indication	SIGURG, getsockopt() with TPFLAG-XPD, return from select()
T-DISCONNECT	request	close(), setsockopt()
	indication	SIGURG, error return, getsockopt()

socket. The transport service user on the called machine must have also initiated a *listen()* system call to passively await an incoming transport connection request. When a request arrives, the called transport user uses the *accept()* system call to accept or reject the request. The time-sequence diagram in Figure 12.2 illustrates the flow of UNIX calls and the transport connection-establishment primitives implied by the calls. (It is safe to assume that the relevant parameters of the transport service primitives are conveyed in the UNIX system calls. Both ARGO 1.0 and RENO 4.3 UNIX use new system calls—*sendv()* and *recvv()*—for data transfer (these correspond to the T-DATA.request and T-DATA.indication, re-spectively). These are scatter-gather or "vectored data" system calls: an array of pointers and lengths describe areas from (or to) which data will be gathered (or scattered). An indication of the end of a transport service data unit is provided using the flags parameter, which allows processing of a

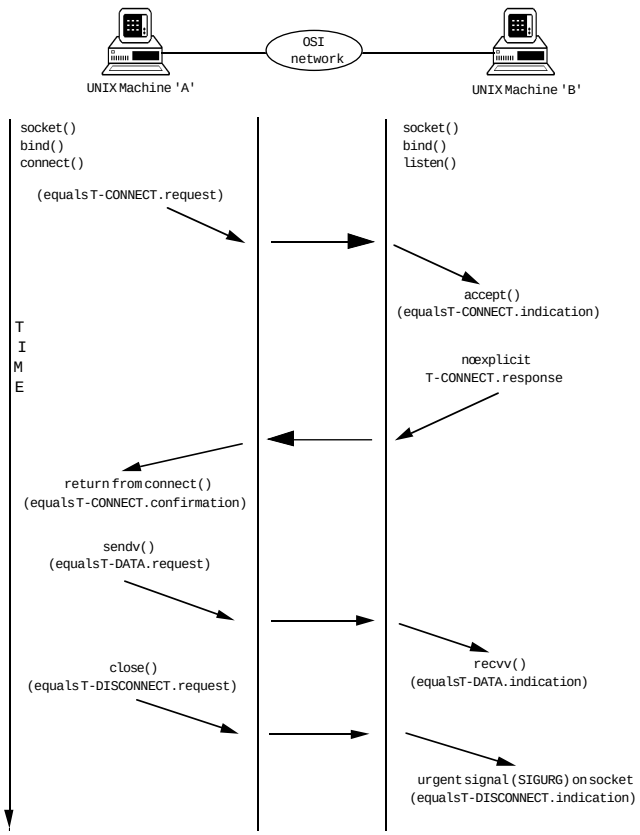

FIGURE 12.2 Transport Connection Establishment in ARGO 1.0

transport service data unit to span multiple system calls. A sending user process distinguishes transport expedited data from normal data by setting the MSG_OOB flag in the *sendv()* call; the receiver is notified of the arrival of expedited data via a UNIX signal mechanism (SIGURG, interpreted as an "urgent condition on the socket"). User processes initiate transport connection release using the *close()* system call. An application process is notified of a peer transport user (or transport provider) initiated release via an urgent condition on the socket (or via an error return on other primitives), whereupon the socket is closed.

The X/OPEN Transport Interface

Like sockets, the X/OPEN Transport Interface (XTI) offers a programmatic interface to OSI transport protocols. Specifically, the X/OPEN Transport Interface allows transport users (user processes) to request transport classes 0, 2, and 4 over network connections and transport class 4 over network datagrams. The X/OPEN Transport Interface supports transport protocol class selection, expedited data, quality of service, orderly release, and variable-length transport addresses and supports both *synchronous* and *asynchronous* communication. With *synchronous* communication, the calling transport user attempts to connect and waits for an accept; similarly, a called transport user will wait and listen for an incoming transport connection (passive). With *asynchronous* communication, the calling transport user attempts to connect and goes off to do other things until notified that the transport connection has been accepted (active); a transport user awaiting an incoming transport connection may listen as a background activity and attend to other operations.

Synchronous communication is the default mode of communication for the *t_connect()* system call. The transport connection-establishment process is similar to the UNIX socket interface: a transport user creates a transport endpoint using the *t_open()* system call, then binds a transport address to the endpoint using the *t_bind()* system call. An active open is invoked via the *t_connect()* system call, followed by a call to *poll()* with time-out value to await confirmation. A passive open is invoked via the *t_listen()* system call. All active processes must create a listening endpoint; a separate responding endpoint is created for each transport connection. A listening transport user accepts an incoming transport connection by invoking the *t_accept()* system call; to reject an incoming transport connection, a listening transport user returns a *t_snddis()* system call rather than the *t_accept()*.

If the transport connection is accepted, the transport users exchange data using the *t_snd()* and *t_rcv()* system calls. In the synchronous mode, a sending transport user waits if flow is constrained; in the

asynchronous mode, explicit flow restrictions are signaled via [TFLOW] errors, a flow-control constraint error signaled in the *t_snd()* call return. If expedited data are to be sent, a flag (T-EXPEDITED) is set in the *t_snd()* call request. The *t_rcv()* system calls are used to receive partial or complete TSDUs; if a partial TSDU is received, a flag (T-MORE) is set, reflecting the arrival of an *end of TSDU* indication in the transport protocol. If expedited data are to be received, a flag (T-EXPEDITED) is set among the *t_rcv()* system call parameters.

The transport connection is released upon completion of a *t_snddis()* and *t_rcvdis()* system call sequence. Figure 12.3 illustrates the entire sequence of events associated with a transport connection in which a single transfer of data is performed.

For asynchronous communication, transport users create endpoints but set the O_NONBLOCK flag in the *t_open()* system call. Again, both the calling and called transport users bind a transport address to the endpoint using the *t_bind()* system call. The calling user process (transport user) invokes the *t_connect()* system call, which returns immediately. The calling process may perform other operations while the transport connection-establishment process proceeds.

The listening or passive user process calls:

- *poll()* with a time-out value to await an incoming event

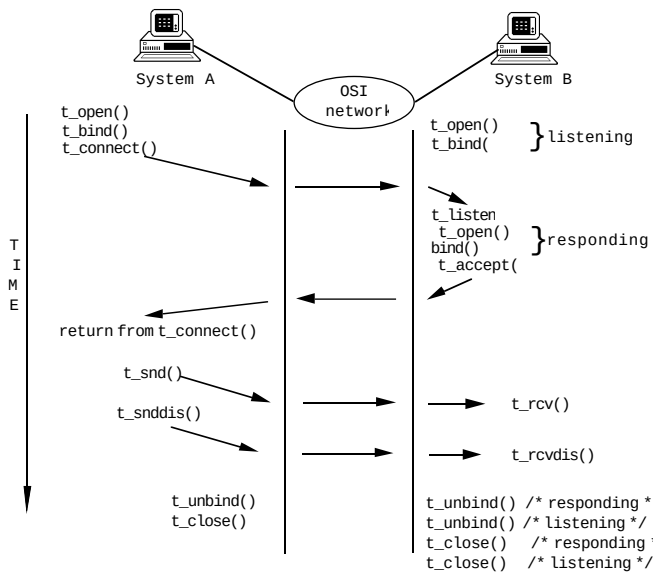

FIGURE 12.3 XTI: Synchronous Communication

- *t_look()* to obtain details about incoming events (in this case, an incoming transport connect request indication)
- *t_listen()* to obtain the information that describes the incoming transport connection request
- *t_accept()* to notify the calling transport user that the connection will be accepted

The calling transport user, periodically calling *poll()*, learns of an incoming event, then calls *t_look()* to check whether the event is a confirmation of the transport connection request it has made. The calling transport user determines the results of transport connection negotiation (described later in this chapter) from the information returned in the *t_rcvconnect()* system call.

The only "twist" on sending and receiving data in asynchronous mode is that user processes must *poll()* for incoming events, then *t_look()* to determine that a particular event is incoming data prior to performing the *t_rcv()* system call. The same twist applies for connection release: a "release event" is returned through *t_look()*. Figure 12.4 illustrates the entire sequence of events associated with asynchronous communication.

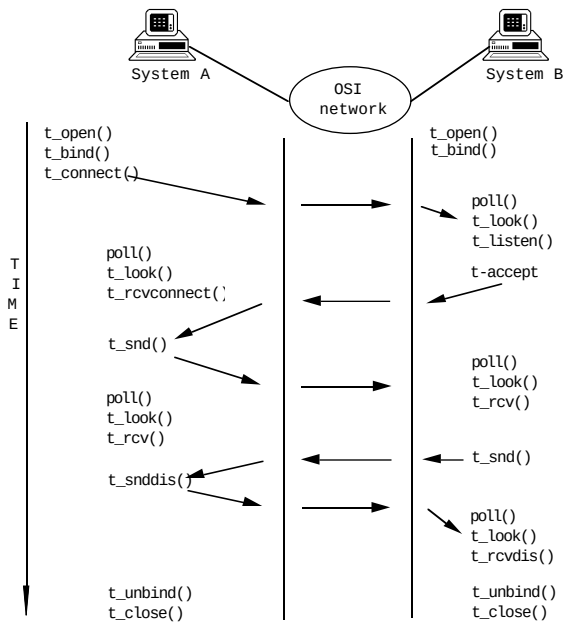

FIGURE 12.4 XTI: Asynchronous Communication

Transport Addressing

In the set of UNIX system calls used to establish interprocess communication between TCP/IP applicatons, a set of *Internet port numbers* is used to differentiate users of the stream and datagram services provided by the transport layers. There are, for example, well-known or "assigned" numbers for the Internet standard applications servers (e.g., FTP, SMTP, SNMP, TELNET). For OSI-based interprocess communication in UNIX, ARGO 1.0 uses a separate address space, *transport service access point identifiers*,[2] to name the interface between the transport service users and the transport entities in the kernel. Like ports, transport service access point identifiers are a particularly important set of names, since they are often visible to program interfaces such as the UNIX socket.

The parameters of the *socket()* system call used to establish an OSI transport connection include the address family/format (AF-ISO), type (sock-seqpacket), and protocol identifier (ISO TP). The transport service access point identifier assigned to a socket using the *bind()* system call usually takes the following form:

```
struct sockaddr_iso {
  short siso_family;          /* address family is iso */
  u_short  siso_tsuffix;      /* the TSAP identifier  */
  struct iso_addr  siso_addr; /* the NSAP address */
  char siso_zero[2];          /* unused */
}
```

The transport service access point address is represented as the concatenation of the network service access point address and a transport selector. The C programming-language data structure and the data structure used to assign an Internet port/address when establishing a socket (type stream) are very similar; e.g.:

```
struct sockaddr_in {
  short  sin_family;        /* address family is internet */
  u_short  sin_port;        /* the internet port */
  struct in_addr  sin_addr; /* the IP address, a.b.c.d */
  char  sin_zero[8];        /* unused */
}
```

In TCP/IP, ports identify upper-level protocol entities or processes,

2. The full address of a transport service access point consists of a network service access point address and a transport service access point identifier. The term "transport service access point address" refers to this full address; where only the identifier part (sometimes called a "transport selector") is meant, the term "transport service access point identifier" or simply "transport identifier" will be used.

and pairs of endpoints identify TCP connections. Specifically, the pair {IP address, port number} roughly corresponds to the OSI transport service access point address, which is the pair {network service access point address, transport selector}, and two of these {IP address, port number} pairs uniquely identify a TCP connection. Internet port number assignment follows a *client/server* paradigm. A server process binds to a well-known port number and listens at {server-IP-address, well known-tcp-port} for connection requests from any source; i.e., from {any-IP-address, any-port}. A client process binds to {client-tcp-port, client-IP-address}, where client-tcp-port is dynamically allocated from unused ports on the client's system. When a client process initially connects to a server process, the endpoints of the transport connection are {client-IP-address, client-tcp-port} and {server-IP-address, well known-tcp-port}, but when the server accepts an incoming request, it creates a second socket and binds the incoming call to this new socket; the resulting endpoints of the transport connection are {client-IP-address, client-tcp-port} and {server-IP-address, server-tcp-port}. Once the server process has mapped the transport connection onto this new pair of endpoints, it continues to listen at {server-IP-address, server-tcp-port} for requests from any source; i.e., from {any-IP-address, any-port}.

A Comparison of Internet Ports and OSI TSAPAs

Internet port numbers are always 16 bits. OSI allows for variable-length transport service access point identifiers, up to a maximum length of 64 octets. ARGO 1.0 accommodates the assignment of extended transport identifiers through the use of the *getsockopt()* or *setsockopt()* system calls following the call to *bind()*. Another difference between OSI transport service access point identifiers and Internet port numbers is that, generally speaking, a set of Internet port numbers are globally and uniquely assigned to represent a well-known application; transport service access point identifiers are frequently derived from the presentation address attribute of an application entity title (see Chapters 5, 7, and 11).

Five Classes of OSI Transport Protocol

The *Connection Oriented Transport Protocol Specification* (ISO/IEC 8073: 1986) defines five classes of procedures for the connection-oriented OSI transport protocol. The transmission control protocol (TCP), on the other hand, has no concept of "class"; it simply defines one standard way in which to provide a connection-oriented transport service.

The answer to the question "why five classes of transport protocol

in OSI?" lies not in the transport layer but in the network layer. If one's model of the world has end-user equipment (hosts) attached to specific individual networks, then one is likely to design a separate transport protocol for each individual network, optimized for the particular characteristics of that network. If one's model of the world has end-user equipment attached to a global internet, consisting of an arbitrary number of interconnected individual networks, then one is likely to design a single transport protocol that provides an end-to-end service with respect to the internet as a whole and makes few (if any) assumptions about the characteristics of individual networks (which, as part of the internet, are not individually visible to the transport protocol). Each of the OSI transport protocol classes 0 through 3 is intended to be used with a particular type of connection-oriented network, according to the probability of signaled and unsignaled errors on that network. Class 4, like TCP, is intended to be used in internets and is not defined with respect to any particular network type.

Class 0: The Simple Class

Class 0 is designed to have "minimum functionality." It provides only the functions necessary to establish a transport connection, transfer data (well, it can segment transport service data units), and report protocol errors. Class 0 relies on the underlying network connection to provide all of the end-to-end capabilities of the transport layer, including sequencing, flow control, and error detection. It was designed with the expectation that the underlying network connection would provide a virtually error-free data-transfer service; for example, in those subnetworks in which a protocol such as CCITT X.25-1984 is used. Class 0 does not provide multiplexing capabilities, and it is so resolutely simple that it does not even have its own disconnection procedures: when the underlying network connection goes away, transport class 0 goes with it.

Class 0 adds no value whatsoever to the underlying network service, which is just the way some network service providers like it. ("Our network service is absolutely splendid. How could you end users possibly imagine that any further work is necessary on your part to obtain a reliable end-to-end connection?") Class 0 is equivalent to the transport protocol defined for the CCITT Teletex service; it persists because it has been written into the CCITT X.400-series recommendations for message handling systems as the required protocol class for connection to public network messaging systems.[3]

3. The CCITT X-series recommendations differ from ISO standards in that each service definition identifies *precisely* the protocol that must be used to provide the service; e.g., the

Class 0 does have one redeeming feature: it is used to create an OSI transport service on TCP/IP networks, enabling OSI applications to run over TP/IP. RFC 1006, *ISO Transport Services on Top of the TCP*, specifies a widely used convention for operating OSI applications over TCP/IP networks. TCP, complemented by a simple 4-octet packetization protocol, provides the essence of the connection-oriented OSI network service across TCP/IP networks (in fact, it does so better than the OSI connection-oriented network protocol, X.25), and OSI transport protocol class 0 provides two service features—the transfer of transport addresses and transport service data unit delimiting—to complete the OSI transport service.

Class 1: The Basic Error Recovery Class

Transport class 1 is a small improvement over class 0, providing error recovery following a failure signaled by the network service, expedited data, and an explicit transport connection release (distinct from network connection release). Class 1 could almost be called "the apologist's transport protocol." It recognizes that no connection-oriented network in the real world is truly perfect. Where class 0 says, "Trust me, the network will never do anything bad," class 1 says, "The network will never do anything bad without telling me about it, and if it does, I'll take care of it."

Several error recovery procedures are included in class 1:

- *Retention until acknowledgment:* Following a signaled failure, copies of outstanding transport protocol data units are retained until receipt of an indication that the remote transport entity is alive, at which time resynchronization procedures are invoked.
- *Reassignment after failure:* If the underlying network connection signals a Disconnect, class 1 can map the existing transport connection onto a new network connection.
- *Resynchronization:* Following a recoverable failure signaled by the network connection (e.g., a reset; see Chapter 13) or a reassignment after failure, both transport entities retransmit unacknowledged transport protocol data units and resynchronize the data stream.

Class 1 can recover only from errors explicitly signaled by the network service provider. Errors not detected and reported by the network service provider will also go undetected by the transport protocol.

CCITT Recommendation T.70 for teletex terminals *must* be used to provide the OSI transport service when connecting to a public MHS. ISO service-definition standards describe only the features of a service, implying that a number of protocols might be used to provide that service.

Class 2: The Multiplexing Class

The experts who introduced class 0 were quite pleased about having identified a minimally functional, low-overhead transport protocol until they realized that they had failed to consider one very important aspect of subscriber access to an X.25 public data network: for economy, subscribers often multiplex data streams from more than one piece of data terminal equipment over a single network connection provided by an X.25 network to maximize use of the throughput available over that network connection. The most obvious way to correct this oversight would have been to extend the functionality of class 0. Class 0 "protectionists," while acknowledging that multiplexing was useful, argued for stability and consistency between the ISO transport protocol standard and CCITT Recommendation T.70, the transport protocol for teletex terminals. (It should also be noted that the folks who work for carrier networks were not as keenly interested in multiplexing as were end users.) Thus was born Class 2.

The following functions are present in class 2:

- *Reference numbers* enable two communicating transport entities to distinguish the transport protocol data units associated with one transport connection from those associated with a different transport connection.
- *Explicit flow control,* when selected, allows the transport entities to regulate the amount and pacing of data transferred between them.
- *Expedited data transfer* permits the transmission of up to 16 octets of user data that are not subject to normal flow-control procedures.
- *Extended transport protocol data unit numbering*, when selected, allows transport entities to use a larger sequence number space (31-bit, rather than the normal 7-bit) for transport protocol data unit numbering and acknowledgments. This increases the number of transport protocol data units that a sender can transmit (the "send window") before it must wait for explicit acknowledgment of reception by the receiver. Normally, the use of a larger window increases throughput. Increasing the size of the sequence number space is also necessary to eliminate the possibility of sequence numbers "wrapping"; for example, if the sequence space were {1 . . . 10}, and a window of 15 were allowed, a sender could transmit packets in sequence {1 . . . 10, 1, 2, 3, 4, 5}. Receiving this sequence of packets, the receiver would be unable to determine whether the second transport protocol data unit received containing the sequence number 1 was a duplicate of a transport protocol data unit received earlier with sequence number 1 or a new transport protocol data unit with the same "wrapped" sequence number.

Curiously, class 2 adds only those functions necessary to support multiplexing; in particular, the error-detection capabilities of class 1 are missing from class 2.[4]

Class 3: The Error Recovery and Multiplexing Class

When the work on OSI transport-layer standards began in the late 1970s, most of the people involved approached the job with a particular type of network in mind—an X.25 packet-switching network, for example, or an X.21 circuit-switching network—each of which suggested different design criteria for a corresponding transport protocol. Lacking the internetworking perspective of the people who designed TCP/IP, the ISO experts proceeded to deal with each of the available network services on a case-by-case basis, accommodating differences in reliability and features not by designing a single, highly resilient, and competent transport protocol but by *introducing one after another.*

So it came to pass that yet another group of experts studied the existing set of transport protocols and—feeling either remorse, concern, or both—decided that at least one transport protocol should combine all of the features of the existing transport protocols. Class 3 represents the union of the functions and capabilities of classes 0 through 2 and then some; the transport protocol standard describes class 3 as having ". . . the characteristics of Class 2 plus the ability to recover from network disconnect or reset." Specifically, transport protocol class 3 provides the multiplexing functions of class 2 plus the error recovery functions of class 1.

Class 4: The Error-Detection and Recovery Class

Implicit in the design of the first four OSI transport protocol classes is the assumption that any errors that might occur in the transfer of data across a network connection will be detected by the network service provider and signaled to the transport entities. Prior to the introduction of class 4, transport protocol design was based on a "network-centric" view of the world: like the dial tone in the telephone system, the OSI network service would in all configurations be provided by one or more common carriers, and service uniformity and homogeneity would be the rule rather than the exception. Further, compelling political and economic arguments existed that made perpetuating the notion that the bulk of end-to-end functionality could be provided at the network layer a standards imperative.

4. Many implementers who are initially appalled by the fact that there are five distinct classes of transport protocol in OSI comfort themselves with the assumption that at least they will be able to implement them by incrementally adding functions to lower classes to create successively higher ones. They are truly aghast when they discover that this is not the case.

Technically speaking, the "network is everything" school of thought views the service provided from network entry to network exit as having *end-to-end* significance. In implementation and practice, however, the networking protocols and interworking among those protocols in fact provide only *edge-to-edge* significance. As will be seen in Chapter 13, the provision of network services, like all human endeavor, is fallible, and variability exists. This is particularly true when multiple providers and diverse technologies are involved in the process. Humans tolerate variability in the voice network because they intuitively apply error-detection and recovery mechanisms; computers lack intuition, so the protocols they use to transfer data must be designed to recognize a variety of failures and recover from them. Until the introduction of transport protocol class 4 (generally referred to as "TP4"), many of the necessary *reliability* functions were absent.

Several years prior to the OSI transport standardization effort, research in the United States questioned the premise that reliability could be assured by establishing uniformity across all networking services. Practical experience gained in the implementation and use of research networks such as the U.S. Department of Defense Advanced Research Projects Agency (DARPA) Internet Research Project (RFC 791), the Livermore Interactive Network Communication System (LINCS; Watson 1982), and the Ethernet-oriented architecture developed at the Xerox Palo Alto Research Center (Xerox Corporation 1981) demonstrated the benefits of incorporating *all* end-to-end reliability functions into a single protocol that operated in host computers (end systems) and relying on the network to perform only the functions essential to the forwarding and delivery of information from source to destination. Since the transport layer takes nothing for granted in this model, the variability of service quality among interconnected networks becomes a non-issue. An entirely different axiom was espoused: *"Sadly, it is a fact of networking life that bits get smashed, octets and packets arrive out of order, some arrive twice, and some do not arrive at all."* Host protocols must therefore be prepared to deal with these problems. From this "host-centric" school of thought, transport protocols such as TCP and OSI transport protocol class 4 emerged.

The OSI transport protocol standard describes class 4 as having ". . . the functionality of class 3 plus the ability to detect and recover from lost, duplicated, or out of sequence transport protocol data units." This is somewhat misleading, since the mechanisms to provide reliability in class 4 differ substantially from those in class 3. It is more accurate to say that transport protocol class 4 provides the multiplexing functions of class 2 and explicit flow control, plus error detection and error recovery

based on the "positive acknowledgment and retransmission" paradigm of TCP. The functions of TP4 include those of class 2 plus:

- A *checksum* computed on the transport header and user data. A 16-bit arithmetic checksum based on Fletcher (1982) is computed to detect bit-level errors in the data stream.
- *Resequencing,* which enables a receiver to determine when transport protocol data units have arrived out of order and provides a means of correctly ordering the octet stream before passing it up to the transport service user.
- *Inactivity control,* which enables a transport connection to survive the (temporary) unsignaled loss of network layer connectivity.
- *Splitting* and *recombining,* which enable a transport connection to transfer data simultaneously over multiple network connections to increase throughput or provide resiliency from single network connection failure.
- *Detection* and *recovery* of *lost* and *duplicate transport protocol data units.*

TP4 is genetically closer to TCP, Xerox's RTP, and their proprietary networking relatives (e.g., the reliable transports in Digital Equipment Corporation's DECnet and Burroughs Network Architecture) than the "Tinkertoy" classes that do not provide an actual end-to-end transport service. Later in this chapter we will demonstrate just how similar TCP and TP4 are.

How Do You Choose the Right One?

Having five transport protocols to choose from is clearly a problem for both implementers and users. The way in which the OSI standard describes how to choose which of the transport protocols to use in a given configuration only adds to the confusion. So how do you choose one? The transport protocol standard recommends that a transport protocol class be chosen to support a given transport connection based on the *type of network connection* available at the time of connection establishment and the *quality of service* requested by the transport service user. Since no meaningful guidelines for the specification or interpretation of quality of service parameters have ever been produced for OSI, class selection relies almost entirely on the underlying network type. The standard identifies three types of network connection:

1. *Type A:* a network connection with an acceptable residual error rate and an acceptable rate of signaled errors.
2. *Type B:* a network connection with an acceptable residual error rate but an unacceptable rate of signaled errors.

3. *Type C:* a network connection with an unacceptable residual error rate.[5]

This, of course, begs the question of how the service quality of network connections can be known in advance. Good question, and also one left unanswered by the standard. Like Pilate, the transport protocol standard washes its hands of the responsibility, saying only that "It is assumed that each transport entity is aware of the quality of service provided by particular network connections . . . ." An immediate reaction by any right-thinking individual is that this is a joke. It is impossible to imagine how any individual host could know *a priori* the type and characteristics of all the network connections that might be available to connect to all the host machines in a small, private network, much less a global OSI network! The transport standard's assumption of this nature belies the notion of typing altogether, since it is meaningful only if service uniformity is assumed across every possible set of interconnected networks. (Such uniformity cannot be guaranteed even in the global voice network, which is far more homogenous than any conceivable global data network.)

In fact, the typing of network connections was merely a convenient premise for the existence of multiple transport protocol classes, which served a different purpose altogether. The political realities of the time demanded that each of several existing public network services be characterized within OSI as having at least the potential to be the cornerstone of the *pervasive worldwide network, connecting all hosts everywhere. Each of these must, perforce, have its own specially optimized transport protocol. But since compromise and consensus are supposed to be the essence of standardization, it would hardly do to have several separate incompatible standards. The solution was to talk not about specific existing networks but about "types" of networks, with which different "classes" of a single protocol standard could be associated. Achieving this "compromise" made some of the standards developers very happy, while creating a legacy of confusion and incompatibility for OSI implementers and users. Thankfully, at least one of the five classes was based on the practical realization that all of these subnetworks—along with LANs, MANs, spaghetti-nets, and future-nets—would someday be interconnected.*

5. In the early stages of OSI transport protocol development, only the connection-oriented network service existed. The connectionless network service and a corresponding addendum to the transport protocol describing how to operate transport protocol class 4 were introduced later. For reasons unknown, networks that offered a connectionless service were considered to offer service functionally equivalent to type-C NCs.

The whole notion of typing network connections was myopic, ill-advised, and ultimately destructive. Ironically, by insisting on a menu of "tailor-made" transport protocols, the "network-centric" individuals who had compelling political and economic motivations to promote a uniform network service accomplished just the opposite, exposing how diverse even publicly provided network services were!

Conformance

What happens when you have more than one choice of protocol? In the case of OSI transport, every delegation, liaison body, even individuals within delegations championed a different protocol. It was popular, for example, but not mandatory, to champion the protocol that one's delegation introduced. The phrase "spirit of compromise" was at first eagerly embraced and subsequently worn extremely thin during the joint ISO/CCITT meetings at which the issue of determining the conformance clause for the OSI transport protocol was discussed.

Combinations were popular, especially if the combination included the protocol one championed. In fact, once combinations were recognized as instrumental in proceeding, the joint committees quickly agreed to eliminate two of the possible combinations (support none and support all). Eventually, it became clear that under no circumstances would the network-centric community agree to abandon support for TP0, and hell would freeze over before the host-centric community would consider anything other than TP4 sufficient for its purposes. The resultant conformance clause is a travesty, a *status quo ante openum*: To claim conformance to the standard, you must implement class 0 or class 2 or *both*. Further, if you implement class 1, you must implement class 0. If you implement class 3 or class 4, you must implement class 2. Confused? There's more. You can operate only class 4 over the connectionless network service.

The solution? If you are implementing OSI transport for use with an internetworking protocol (such as CLNP), it's easy—class 4 is the only class that will work at all over a connectionless network service. To cover all possible cases, including those in which your system will operate directly over a connection-oriented network (such as an X.25 network) with no internetwork protocol, implement classes 0, 2, and 4. It's no big deal adding classes 0 and 2, since both are no-brainers. Notwithstanding what must have

seemed like good arguments in their favor at the time, classes 1 and 3 have no modern constituency and can safely be ignored. If you are an end user, and you care about reliability, you should insist on class 4; tell your vendors "No discussion—just do it."

Comparing TP4 to TCP

A 1985 study performed jointly by the U.S. Defense Communications Agency and the National Academy of Sciences (National Academy of Sciences Report 1985) concludes that TCP and TP4 are functionally equivalent and provide essentially similar services. Table 12.5 compares the functions provided by the two protocols.

Table 12.5 Comparison of TP4 and TCP Functions

Function	*TCP*	*TP4*
Data transfer	Streams	Blocks
Flow control	Octets	Segments
Error detection[6]	Checksum	Checksum
Error correction	Retransmission	Retransmission
Addressing	16-bit ports	Variable TSAPA
Interrupt service	Urgent data	Expedited data
Security	Not available	Variable in TP
Precedence	Not available	16 bits in TP
Connection termination	Graceful	Nongraceful

6. The TCP and TP4 checksum functions are both intended to detect errors that may be introduced into the data stream between two transport users, but they do not operate in the same way. The TP4 checksum computation is carried out on the transport packet (header and user data) only, and it may be disabled (by explicitly selecting the "nonuse of checksum" option during connection establishment; the default is to use checksums). The TCP checksum is carried out on the combination of the transport packet (header and user data) and a prepended "pseudoheader" consisting of the source and destination IP addresses, the IP PROTO field, and the TCP segment length, and it may not be disabled. The TP4 checksum is also slightly more complicated (both to generate at the sender and to verify at the receiver) than the TCP checksum, although the additional complexity does not make TP4 significantly more resistant to undetected errors than TCP. An excellent discussion of how to efficiently implement the TP4 checksum is contained in Sklower (1989); a similar analysis for TCP may be found in RFC 1071, Clark (1989), and RFC 1141.

TP4 and TCP are not only functionally equivalent but operationally similar as well. This is best understood by examining the process of establishing a transport connection, providing reliable data transfer through the use of retransmission on time-out mechanisms, and connection termination of each protocol.

OSI Transport Connection Establishment

In the OSI reference model and the transport service definition, establishing a transport connection is described as a process of matching the transport service user's requested quality of service with available network services. One dubious aspect of this process is network service selection—connection-oriented or connectionless. In many end-system configurations, this decision process may not exist; for example, many LAN-based OSI systems will support only TP4 operating over a connectionless network service, and teletex-based systems that are only configured to access a public message-handling service via an X.25 public data network will support only TP0 over X.25. In configurations in which both a connection-oriented and a connectionless network service are available, the transport layer supposedly determines which type of network service to select on the basis of quality of service information submitted along with the T-CONNECT.request, information either stored in a user-defined configuration file or retrieved from a directory, or some set of operating system parameters.

During the exchange of transport protocol data units used for connection establishment, parameters that characterize the nature of the transport connection are negotiated by the two transport entities,[7] including:

- *TP class:* Since we have "choices," both transport entities must agree on the class of protocol to be used. (Successful negotiation of a transport connection implies that the two transport entities share at least one protocol that is sufficient to provide the transport service requested by the initiating application.) The calling transport entity selects a *preferred TP class* and may indicate *alternative TP classes* it is willing to use if the called transport entity does not support the preferred class.

7. The current standards for OSI transport define only two-party connections. New work in ISO on multiparty transport connections had been under way for a year when this book went to press and is expected to result in amendments to the OSI transport protocol standard to support connections in which more than two transport entities participate as peers.

- *TSAP IDs:* Both the calling and called transport service access point identifiers are encoded in the *connect request* transport protocol data unit.

The transport protocol standard states that encoding of the transport service access point identifiers in the connect request transport protocol service access point is optional and may be omitted when "either network address unambiguously defines the transport address" (that is, the transport service access point address can be specified by the network service access point alone, without the additional information provided by the transport service access point identifier). This is an unfortunate artifact of the incorporation of the T.70 teletex transport protocol into OSI; despite what the standard says, the identifiers are indeed necessary!

- *Options:* Options are defined for each transport protocol class. If class 1 is selected, both parties must agree to either use or not use receipt confirmation (see Chapter 13), and expedited data transfer provided by the network layer. If class 2 is selected, both parties must agree whether to use or not use explicit or implicit flow control. If classes 2, 3, or 4 are selected, both parties must agree to use normal or extended transport protocol data unit numbering and to use or not use transport expedited data. If class 4 is selected, both parties must agree on whether or not to perform a checksum on transport data packets and which checksum algorithm they will use.
- *Transport protocol data unit size:* The OSI transport protocol uses a fixed maximum packet size ranging (in powers of 2) from 128 octets to 8,192 octets,[8] including the header.
- *QOS parameters:* The calling transport entity may include values of quality of service parameters indicating the throughput, transit delay, and residual error rate expectations of the end-user application initiating the communication. In theory, these values assist the called transport entity in deciding whether a transport connection can be established that satisfies the end-user criteria for communication.

8. An amendment to the OSI transport protocol that allows for the negotiation of much larger maximum packet sizes in much smaller increments was recently adopted by both ISO and CCITT. The new "preferred maximum TPDU size" parameter is encoded as a variable-length field of up to 32 bits that gives the maximum data packet size in units of 128 octets, allowing for the negotiation of any maximum data packet size from 128 octets to 2^{36} octets in increments of 128 octets.

Other parameters exchanged during transport connection establishment include the called and calling transport service access point identifiers, the value of the *initial credit* (how large a window the called transport entity will offer), and timer values germane to the operation of specific protocol classes. For example, if TP4 is selected as the preferred class, an *acknowledgment time* is exchanged by the called and calling transport entities. This timer value is used during data transfer as an approximation of the amount of time a transport entity will delay following reception of a data packet before sending an acknowledgment packet. For classes 1, 3, and 4 operating over a network connection, the value of *reassignment time* (how long before the called transport entity will attempt to reassign this transport connection to another network connection following a network connection failure) is exchanged.

To establish a transport connection, a transport entity composes a *connect request* packet (CR TPDU) and submits it to the network layer for delivery to the destination transport entity. The destination transport entity is identified by the network address indicated in the network service primitive that conveys the connect request packet (an N-CONNECT.request or N-UNITDATA.request; see Chapter 13). Figure 12.5 shows the composition of the CR TPDU.

Transport protocol classes 2, 3, and 4 provide the ability to multiplex many transport connections onto a single network connection. To distinguish one transport connection established between a given pair of transport entities from another, two 16-bit *references* fields are used. In

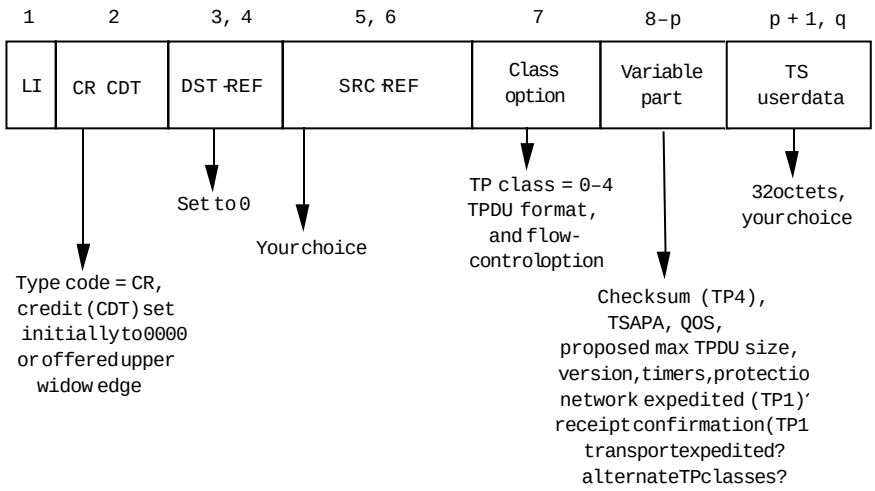

FIGURE 12.5 Connect Request Packet

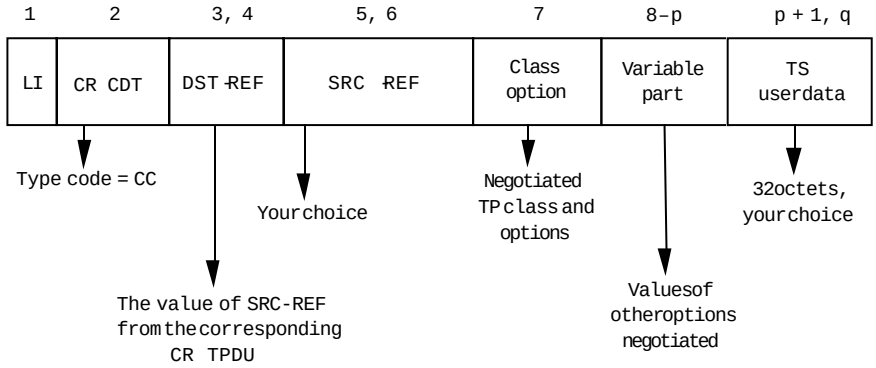

FIGURE 12.6 Connect Confirm Packet

the connection request package, the calling transport entity encodes a value for the *source reference* (SRC-REF in Figures 12.5 and 12.6) field; *a destination reference* (DST-REF in Figures 12.5 and 12.6) field is set to 0, to be determined by the called transport entity. The called transport entity encodes its own source reference in the source reference field of the *connect confirm packet* (CC TPDU) and places the value from the source reference field of the connect request packet into the destination reference field of the connect confirm packet. The pair of references uniquely identifies this transport connection between this pair of transport entities. The selection of the values for references is a *local matter.*[9] Typically (e.g., for simplicity), references are assigned sequentially.

Upon reception of a connect request packet, the called transport entity parses the packet and determines whether it can support the *preferred TP class* indicated; if it cannot, it determines whether it can support any of the *alternative TP classes* identified. If it cannot support any of the TP classes indicated, it must reject the connection (see "Connection Release (Connection Refusal) in the OSI transport protocol"). Table 12.6 illustrates the permissible combinations of preferred and alternative TP class parameter encodings.

The called transport entity next determines which options it can support among those selected by the calling transport entity. The rules governing the response to options selected are straightforward. If the calling transport entity proposes the use of an option—i.e., by setting the flag

9. ISO/IEC 8073 defines a "local matter" as "a decision made by a system concerning its behaviour in the Transport Layer that is not subject to the requirements of this protocol." In plain-speak, use any value you please, so long as you can guarantee its uniqueness for the duration of time during which references are to be *frozen*.

Table 12.6 Combinations of Preferred/Alternative TP classes

Preferred class	Alternative class					
	0	*1*	*2*	*3*	*4*	*None*
0	Not valid	Not valid	Not valid	Not valid	Not valid	TP0
1	TP1 or TP0	TP1 or TP0	Not valid	Not valid	Not valid	TP1 or TP0
2	TP2 or TP0	Not valid	TP2	Not valid	Not valid	TP2
3	TP3, TP2, TP0	TP3, TP2, TP1, TP0	TP3 or TP2	TP3 or TP2	Not valid	TP3 or TP2
4	TP4, TP2, TP0	TP4, TP2, TP1, TP0	TP4 or TP2	TP4, TP3, TP2	TP4 or TP2	TP4 or TP2

representing the option to 1—the called transport entity may agree to use the option by leaving the flag set to 1, or it may refuse to use the option by setting the flag to 0. If the calling transport entity does not propose the use of an option, the called transport entity may not propose its use.

The called transport entity must also determine whether it can support the maximum packet size indicated or whether it must indicate that a *smaller* maximum packet size should be used, and whether it can maintain the QOS indicated.

Once these decisions are made, and assuming that a transport connection can be successfully negotiated, the called transport entity returns a connect confirm packet, indicating what choices it has selected from the negotiable parameters (see Figure 12.6).

The called transport entity also records values of parameters relevant to the operation of the transport protocol that may have been sent in the connect request transport protocol data unit; e.g., the values of either the *acknowledgment time* or the *reassignment time*, and the *initial credit*. These values are used during the data-transfer phase and for error recovery purposes.

Three-Way Handshake

The process of establishing a transport connection is only partially completed when the calling transport entity receives a connect confirm packet unit from a called transport entity; the calling transport entity has parsed the connect confirm packet, and it knows that the called transport entity is indeed willing to establish a transport connection. The calling transport entity also knows the characteristics that the called transport entity has *negotiated* for the transport connection (the called transport entity has indicated these in the connect confirm packet it has composed). Like the called transport entity, the calling transport entity will have recorded values of parameters relevant to the operation of the transport protocol that may have been sent in the connect confirm

packet—the values of either the acknowledgment time or the reassignment time, and the initial credit.

At this point, however, the called transport entity hasn't a clue whether the negotiated characteristics for the transport connection are acceptable or even whether the connect confirm packet has been delivered to the calling transport entity. The calling transport entity thus has the additional responsibility in the transport connection-establishment process of providing the called transport entity with an indication of whether transport connection establishment succeeded or failed. Assuming that the transport connection has been successfully negotiated, the calling transport entity has two choices:

1. Return an *acknowledgment* packet (AK TPDU).
2. Start sending data.

The first mechanism is always available to the calling transport entity. Figure 12.7 shows the format of the acknowledgment packet.

Reception of an acknowledgment packet by the called transport entity following transmission of a connect confirm packet provides the called transport entity with an explicit acknowledgment that the calling transport entity accepted the transport connection with the negotiated characteristics. Confirmation that the transport connection is successfully negotiated in this manner is called a *three-way handshake.* Figure 12.8 provides a simplified illustration of this process.

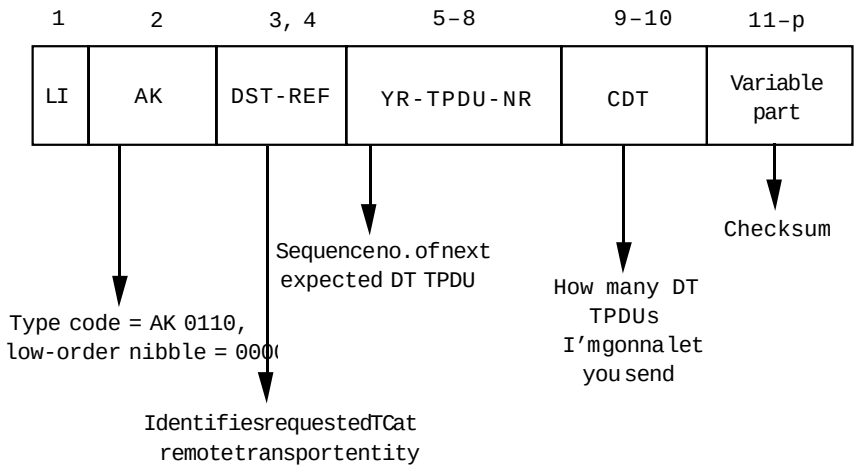

FIGURE 12.7 Acknowledgment Packet

Relevant information encoded in the acknowledgment packet includes:

- *Destination reference,* which contains the value of the calling transport entity's reference number.
- *Your transport protocol data unit number* (YR-TPDU-NR), which contains the value of the next expected sequence number (*n*), implicitly acknowledging the receipt of all transport protocol data packets up to and including sequence number *n*–1 (modulo 2^7 arithmetic if normal formats, modulo 2^{31} if extended formats). Sequence numbers are used in the data-transfer phase of a transport connection to distinguish packets containing normal data from one another and to assist in determining whether data packets have arrived in order or have been lost or duplicated. The OSI transport protocol begins every transport connection with an initial sequence number of 0; since no data have yet been transferred, YR-TPDU-NR here contains a value of 1.
- *CDT* contains the *initial credit* allocated by the calling transport entity to the called transport entity (the number of transport protocol data packets I'll allow you to send before you must wait for me to acknowledge that I've received them).
- Other parameters, including a checksum computed on the acknowledgment packet and an *acknowledgment sequence number* (see "Normal Data Transfer in OSI transport protocols," later in this chapter).

A second mechanism is available for situations in which the called transport entity has indicated a non-0 initial credit value in the connection packet. The calling transport entity then has the option of immediately sending any normal or expedited transport service user data waiting for transfer; the first *data* packet (DT TPDU) or *expedited data* packet (ED TPDU) received by the called transport entity in this case is interpreted as a completion of the three-way handshake (again, refer to Figure 12.8).

Setting It All to Unix

If one were to trace the OSI transport connection-establishment process through the UNIX system calls described earlier in this chapter and the associated transport protocol state machine, then trace the packets that traversed the network while the process took place, and assuming that

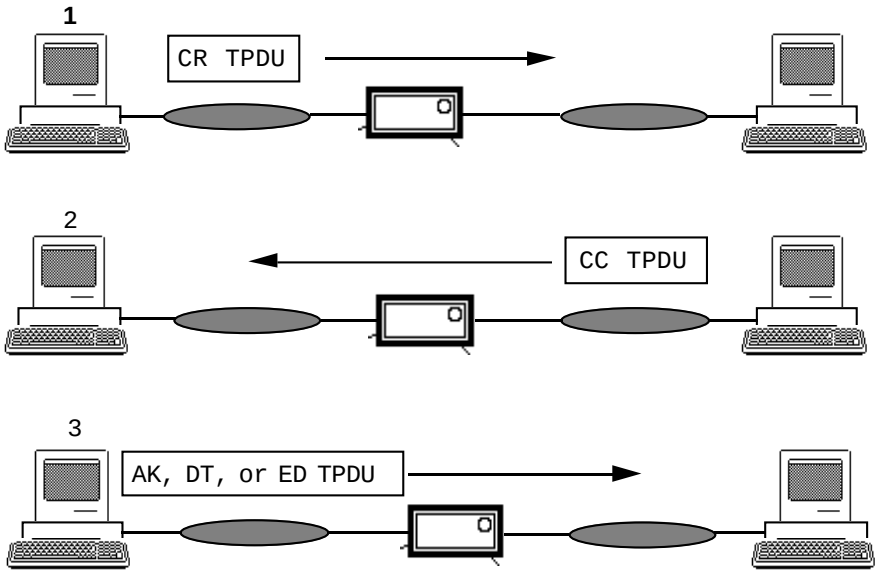

FIGURE 12.8 Three-Way Handshake

both transport users are on ARGO 1.0 UNIX, the time sequence of events might look like the one shown in Figure 12.9.

In the figure, transport user A invokes the UNIX system call *socket(AF-ISO, sock-seqpacket, ISO TP)* to create a local communication endpoint. Transport user B does the same. Both parties invoke the UNIX system call *bind(),* which is used to assign a transport service access point address to their respective sockets. Transport user A requests a transport connection by issuing the *connect()* system call. The transport entity composes a CR packet, requesting TP class 4 and indicating that the reference number it will use for this transport connection is 5 (SR in the first step of Figure 12.9). The network layer is called upon to transfer the CR packet.

The called transport entity at B receives the CR packet (transport user B has invoked system calls to passively await an incoming transport connection—i.e., it has issued a *listen()*. It agrees that TP class 4 is a wonderful choice and agrees with all the rest of the choices the calling transport entity has indicated. It composes a CC packet reflecting consent to all these selections. The called transport entity copies the source reference from the CR packet into the destination reference field (DR in the second step of Figure 12.9) and indicates that the reference number it will use for this transport connection is 2 (SR in the second step of Figure 12.9); the pair of reference numbers {5, 2} now distinguishes this trans-

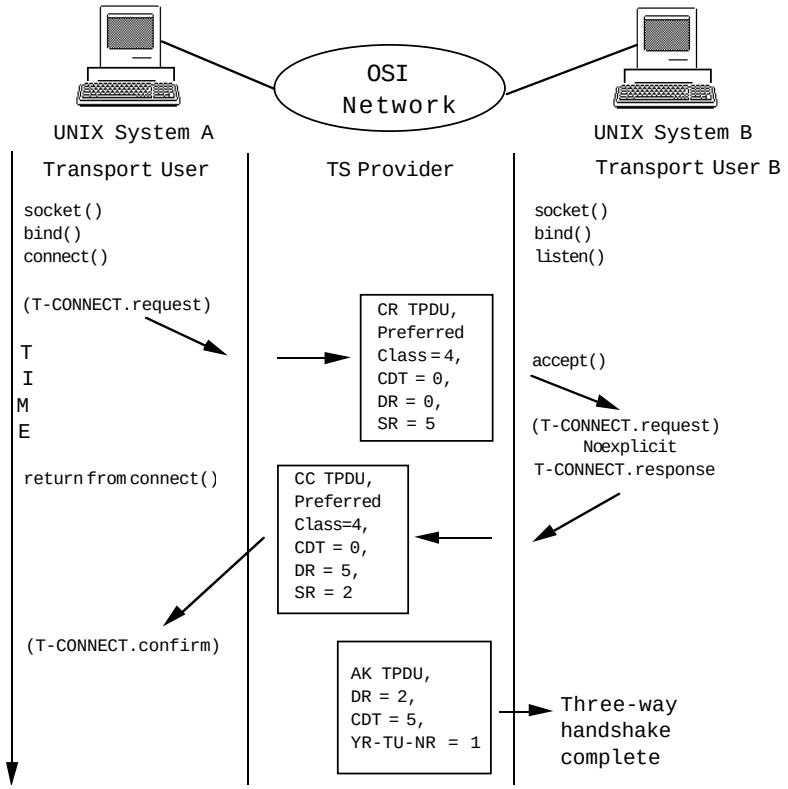

FIGURE 12.9 Setting Connection Establishment to UNIX

port connection from others established between transport service users A and B. In the example, the called transport entity also indicates an initial credit (CDT) of 0. The called transport entity then calls upon the network layer to transfer the CC packet.

The calling transport entity receives the CC packet, noting with great enthusiasm that the called transport entity has accepted all proposed transport connection selections. It composes an acknowledgment packet, setting the destination reference field to 2, the value identified by the called transport entity for this transport connection; offers a credit of 5; and indicates that the next expected sequence number (YR-TPDU-NR) is 1. The network layer is again called upon to transfer the AK packet. A return from *connect()* completes transport connection establishment for transport service user A. The called transport entity receives the AK packet, which completes the three-way handshake. A return from the *accept()* call completes transport connection establishment for transport service user B.

Frozen References

An important consideration when using references is providing a mechanism to avoid reuse of a reference to identify a new transport connection while a packet associated with a previous use of the same reference may still be "trollicking"[10] along a network connection or bounding its way along as a connectionless NPDU. A reference would be reused when the value of the 16-bit reference counter "wraps" in modulo 2 arithmetic; although the circumstances under which this might occur seem extraordinary, betting the ranch that it won't isn't a very good idea. A procedure called *frozen references* offers some guidance to the implementer on how to deal with this phenomenon. In TP4, one way to bound the time *(L)* to freeze a reference is to wait a minimum of the computed or estimated *round-trip time*—the sum of the time required to transfer an NSDU from the local transport entity to the remote transport entity (M_{LR}) and back again (M_{RL})—plus the value of *acknowledgment time* the remote transport entity indicated in the CR transport protocol data unit (A_R), plus the value of what is known as the *persistence time (R)*, which is how long the local transport entity will try to resend an unacknowledged transport protocol data unit before "giving up." The formula is:

$$L = M_{LR} + M_{RL} + R + A_R$$

Like all simple things, however, this formula has its weaknesses (see "Timers and Open Transport Protocols," later in this chapter). In practice, this value of *L* may be too large; better too large, however, than too small! One must remember the purpose of the timer and choose a value that can be lived with.

TCP Connection Establishment

The *synchronize stream (SYN)* process of TCP is functionally equivalent to connection establishment in the OSI transport protocol. TCP operates as a pair of independent streams of (octets of) data between upper-layer protocols. The synchronization process establishes the beginning of the byte stream in both directions of information flow. During SYN process-

10. "Trollicking" is derived from the word *trundle*, meaning "to roll on little wheels; to bowl along," and *frollicking*, meaning "dancing, playing tricks, or frisking about." Any transport packets remaining in a network connection during a "frozen references" time period clearly could only be there to cause mischief . . .

ing, information is exchanged between TCP processes that is similar to the information negotiated (or implied) during OSI transport connection establishment:

- *Addressing:* The *named sockets* (source and destination port addresses) for the upper-layer protocol pairs that will use the TCP connection are exchanged during the SYN phase.
- *Initial sequence number (ISN):* All OSI transport protocol classes begin a transport connection using an initial sequence number of 0. TCP offers greater latitude, allowing TCP entities to identify the initial octet sequence number in the SYN segment.
- *Data offset:* contains the number of 32-bit words in the TCP segment header; and therefore points to the first octet of user data.
- *Window:* Similar to OSI TP's credit, the value in the window field indicates the number of bytes of information the originator of the SYN is willing to accept.
- *Checksum:* A 16-bit arithmetic checksum is computed on the header and data of all TCP segments.
- *Maximum segment size (MSS):* Upper-layer protocols negotiate the maximum transport segment size (in octets). The default value on the Internet for TCP maximum segment size is 536 bytes (RFC 879).[11]

Unlike OSI's transport protocol, all TCP segments have the same format.[12] Figure 12.10 illustrates the encodings of fields significant for the SYN segment.

Many of the option facilities negotiated in OSI TP connection establishment are nonoptions in TCP. For example, *urgent data*—the closest thing to OSI's expedited data transfer—is a service that is always available in TCP connections. TCP always uses explicit flow control. TCP is only operated over a datagram service. And TCP wouldn't know receipt confirmation if it were bitten by the warty little beast.

To confirm the establishment of a TCP segment, the responding

11. Extensions to TCP by Van Jacobson, R. Braden, and D. Borman (RFC 1323) describe how large segment sizes and windows are negotiated by TCPs; like the amendment to the OSI transport protocol, this mechanism allows TCP entities to negotiate and use very large TCP packets, necessary for networks that exhibit high bandwidth but have long round-trip delays. Van Jacobson and company have colloquially termed such transmission facilities "long fat networks (LFNs)," pronounced "elephan(t)s."

12. The Internet convention for illustrating protocol headers in its standards is to depict the fields in 32-bit sequences. In this chapter, the authors have elected to draw both OSI transport protocol and TCP headers according to ISO conventions (with apologies to the Internet community).

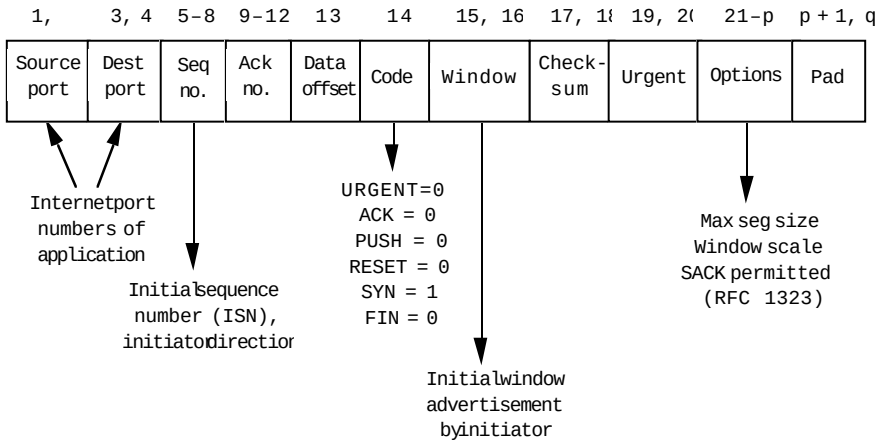

FIGURE 12.10 TCP SYN Segment

TCP entity *acknowledges* receipt of the SYN segment by generating a TCP segment with the code field bits SYN and ACK set to 1. The responding TCP entity acknowledges receipt of the initiator's SYN and attempts to synchronize the byte stream in the responder-initiator direction in a single TCP segment. This process, called *piggybacking,* improves protocol efficiency in several ways. Since only one segment must be created and transmitted to accomplish two tasks, less processing is devoted to protocol composition/decomposition. Fewer bytes of protocol header information are required, so less bandwidth is required. And since two tasks are accomplished in one transmission, overall delay is improved.

Setting the ACK bit in the code field indicates that the *acknowledgment number* field is significant. In the case of a SYN/ACK segment, the responding TCP entity should encode this field with the value of the ISN derived from the originator's SYN packet, incremented by 1. The values of the source and destination ports are encoded in reverse order (i.e., the called TCP entity is the source of the SYN/ACK segment, and the calling TCP entity is the destination). The responding TCP entity synchronizes sequence numbers by encoding the ISN for the responder-to-initiator direction of information flow. Figure 12.11 illustrates the encoding of the fields significant to the SYN/ACK segment.

A three-way handshake is common to OSI's transport protocol and TCP. In TCP, the three-way handshake is completed when the initiator returns a TCP segment with the ACK bit of the code field set to 1 (see Figure 12.12). A TCP segment containing an acknowledgment can also contain user data (another use of piggybacking); thus, if the responder

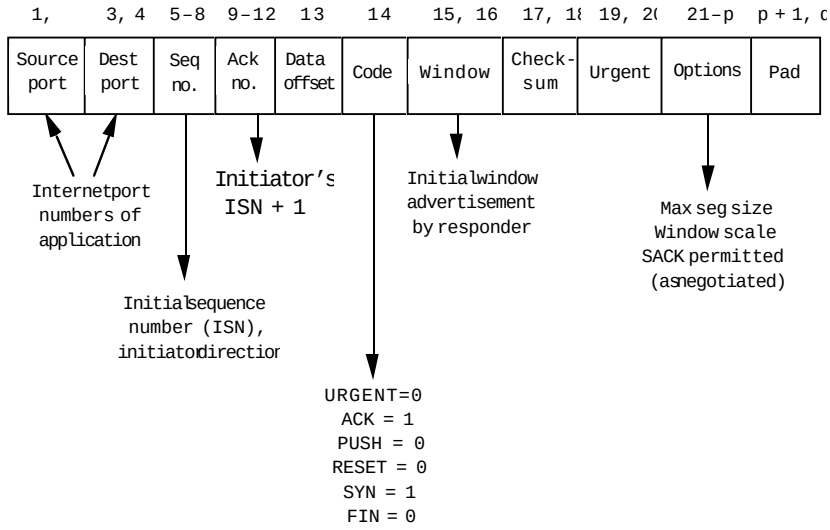

FIGURE 12.11 SYN/ACK Segment

indicated a non-0 initial window in the SYN/ACK segment, the initiator can send up to "responder's initial window" number of bytes of data in the SYN/ACK segment.

Figure 12.13 sets TCP connection establishment to UNIX in much the same fashion as Figure 12.9 does for OSI TP4.

In the figure, a client process (ULP A) on host A creates a socket, binds a port number to that socket, and attempts to connect to ULP B on host B. A's TCP entity composes a TCP segment with the SYN flag set and sets the ISN to 200. A server process (ULP B on host B has also created a socket and has bound a port number to that socket, and is awaiting

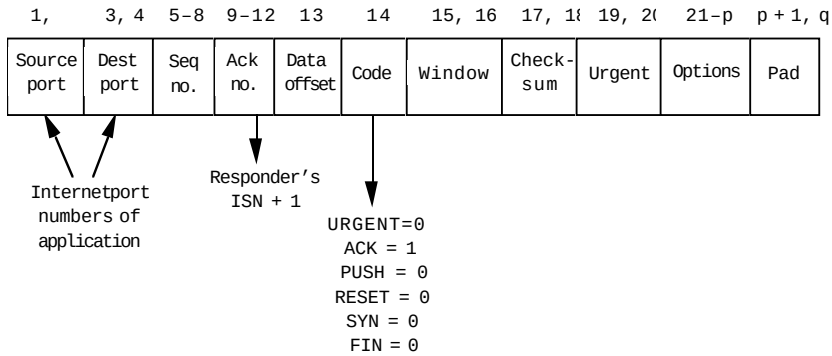

FIGURE 12.12 ACK Segment Encoding

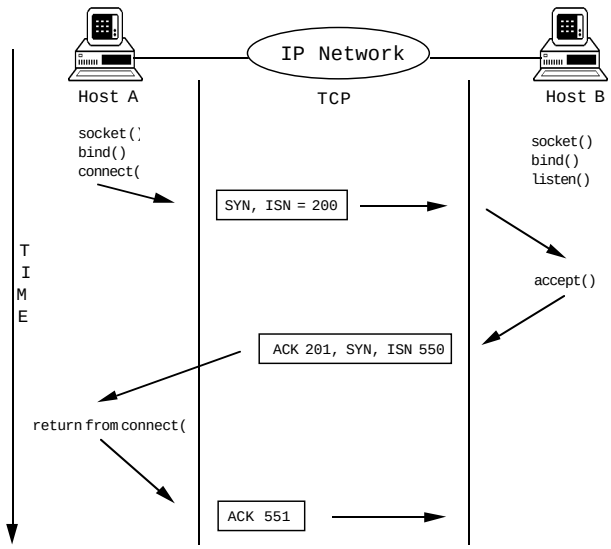

FIGURE 12.13 Setting TCP Connection Establishment to UNIX

incoming connect requests at a well-known port number). The SYN packet arrives, is processed by host B's TCP entity, and the server process at B is notified of the incoming connect request. ULP B accepts the connection (managing TCP endpoints as described earlier in this chapter). Host B's TCP entity composes a TCP segment with both the SYN and ACK flags set and with an ISN of 550. The ACK flag indicates that the acknowledgment field is significant, and it is set to the value of the ISN received in A's SYN packet plus 1 (i.e., 201). When the SYN/ACK segment arrives at host A, ULP A is informed, and A's TCP entity attempts to complete the three-way handshake by returning a segment to host B with the ACK flag set, indicating that the ackowledgment field is significant in this segment (and set to the value of the ISN received in the SYN/ACK packet plus 1; in this case, 551). If host B had offered an initial window, host A could have piggybacked data along with the acknowledgment.

"Keep Quiet"—TCP's "Frozen References"

TCP also worries about reusing a sequence number before its time (well, chronologically speaking, TCP worried first!). To prevent sequence numbers from a previous incarnation of a TCP connection from being mistaken for segments of a new connection, RFC 793 suggests that TCP send no segments for a time equal to the maximum segment lifetime (MSL). The recommended value for MSL in RFC 793 is 2 minutes, although this is an *engineering choice* (a cousin to OSI's "local matter").

Normal Data Transfer in OSI Transport Protocol

Once a transport connection has been established, transport service user data may be transferred bidirectionally between communicating session entities (transport service users). The OSI transport service allows corresponding transport service users to submit arbitrarily large (theoretically unbounded) transport service data units. In the context of providing reliable transfer, the sending transport layer performs segmentation of transport service data units into transport protocol data units up to the *maximum transport protocol data unit size* negotiated during transport connection establishment and submits the transport protocol data units to the network layer for forwarding and delivery to the destination transport entity.

The means by which a sending transport entity determines whether transport protocol data units arrived without mishap at their destination varies depending on the transport protocol class negotiated. Mechanisms are provided in class 4 to assure that individual transport protocol data units are explicitly acknowledged and that the transport service data units are reassembled correctly into the original transport service data unit prior to delivery to the destination transport service user. The lesser or *invertebrate* transport protocol classes rely mainly on gimmicks, chicanery, hand waving, and the network layer to provide all or nearly all aspects of reliable data transfer.

In transport classes 0 and 2, for example, transport service data units are segmented and submitted to the network layer for in-sequence transfer, and the network service is relied upon to detect (but not correct for) loss. Users of a transport connection supported by TP0 or TP2 act in blind faith; unless the underlying network connection signals an error, they continue to submit transport service data units and assume that they will be delivered correctly. If, however, an error is signaled by the network service provider, the transport connection is abruptly terminated, and both transport service users are left to fend for themselves. TP2 users are left multi- and per-plexed, and the upper layers are left to clean up the mess.

Transport classes 1 and 3 provide error recovery following a failure signaled by the network service using *reassignment after failure, retention until acknowledgment,* and *resynchronization* functions, as follows:

- If the underlying network connection signals a disconnect, TP1 can map the existing transport connection onto a new network connection. Copies of unacknowledged data packets are retained by a

sending transport entity until a new network connection is opened. The retained packets are then retransmitted by the sending transport entity (remember, information flow across a transport connection is bidirectional, so both parties may act as senders) and explicitly acknowledged by the receiver, thus resynchronizing the information flow of the transport connection.[13]

- If the underlying network connection signals an error (a reset), transport entities retransmit unacknowledged data packets to resynchronize the data streams in both directions (over the same network connection).

Remember, these Mickey Mouse mechanisms are considered to be sufficient because *acceptable* levels of service quality are purportedly provided by the network service; in the real world, they are probably adequate when both transport service users subscribe to the same public network provider or are attached to the same physical subnetwork. Since you can't be too careful these days, it's better to practice "safe networking" than to deny the possibility of network connection failures and say, "Well, it won't happen to me."

Flow Control

A major concern in maintaining service quality in a network is seeing that information flows into the network at a manageable rate. Just as a highway can handle only so many automobiles before traffic initially slows and then comes to a stop, networks can only switch a finite number of packets before experiencing similar *congestion.* To prevent traffic jams on major California highways surrounding Los Angeles, for example, traffic signals are positioned on the entrance ramps. These allow automobiles to enter the roadway at intervals that vary according to the amount of traffic already present on the roadway. By regulating the flow of automobiles onto the roadway, congestion is temporarily avoided.[14] Of course, roads are rarely built to handle peak loads (rush hours), and under these conditions, highway congestion inevitably occurs (sometimes the traffic lights at entranceways are turned off, a visible sign of surrender or *congestion collapse*).

Similar techniques are used in networks to avoid congestion. The

13. Copies of connect request and connect confirm TPDUs—and for reasons unknown, TPDUs used for connection release (DR and DC TPDUs)—are retained in the same fashionto permit completion of transport connection establishment or release if the network connection disconnects during that phase of operation.

14. Los Angelenos may deny that this works at all, but the fault lies in the fact that congestion-avoidance mechanisms have been applied to highways that have existed in a congested state for decades.

rate of information flow into a network (incoming packets) is regulated according to the network's ability to switch packets. Flow control is a classic networking problem, and a variety of *flow-control* techniques have been employed to avoid congestion. Of these, OSI employs both implicit and explicit flow-control techniques in its transport protocol classes.

In a public network service, users subscribe to and expect a certain rate of throughput (often, a negotiated quality of service characteristic); thus, it is no surprise to find that transport classes 0 and 1 are designed to rely entirely on the network service provider to regulate the flow of transport protocol data units. This *implicit flow control*[15] is straightforward (and autocratic): the sending transport entity submits transport protocol data units to the network service (in the form of NSDUs); the network layer accepts the NSDUs and forwards them *at a rate it chooses*, typically one consistent with maintaining some degree of uniform service quality for all subscribers. Transport service users continue to submit transport service data units, and TP0 and TP1 faithfully continue to create transport protocol data units and submit them to the network layer until the local buffer pool allocated to the transport layer is exhausted. TP0 and TP1 then halt and accept no transport service data units from the session layer; the effect of halting percolates through the upper layers and has the same *back-pressure* effect on information flow that (gradually) closing a faucet has on the flow of water. Normal user data cease to flow out of the end system until the back-pressure condition is eased; i.e., until the network layer accepts a sufficient number from the queue of previously submitted NSDUs to allow the flow of information to continue.

A shortcoming of this form of flow control is that although the network layer is somewhat insulated from congestion, an end system receiving packets has no explicit means of indicating that it cannot accept packets at the incoming rate. TP0 and TP1 again rely upon the network service to deal with this situation; protocols used to provide the OSI connection-oriented network service (e.g., X.25) have facilities that enable an end system to signal that it is temporarily unable to receive incoming packets (there are no explicit network service primitives to signal "congestion"; how the transport layer indicates an "unable to receive" state is a local matter).

Flow control is no less important in private internets; the mechanisms, however, are more democratic. All transport entities are expected

15. Implicit flow control is optionally available in TP2 and must be negotiated during connection establishment.

to participate in an explicit flow-control process. TP4, for example, uses a *sliding window* mechanism; very simply stated, this form of flow control proceeds as follows:

1. Each transport entity indicates a number of data packets that it is able to receive; this is called the *credit* (CDT). The initial credit value is exchanged during connection establishment. During data transfer, CDT is added to the value of the highest sequence number acknowledged (called the *lower window edge* [LWE], initially 0) to create the *send window*. This sum is called the *upper window edge* (UWE).
2. A sending transport entity sends a number of transport protocol data units equal to the credit, then waits for an explicit acknowledgment packet before continuing to send.
3. Upon reception of an acknowledgment packet, a sending transport entity extracts the value of the sequence number (YR-TPDU-NR) from the AK packet and uses this value as the new LWE; it also extracts the value of CDT from the AK packet and adds this to the new LWE to determine the new UWE.

As steps 2 and 3 are repeated, the sequence numbers are incremented using modulo 2 arithmetic, and the send window "slides." By increasing or decreasing the value of CDT, the size of the send window increases or decreases. This is also called opening or closing the send window.

Different policies are applied to determine the appropriate value of CDT. One simple policy for determining the initial value of CDT is to divide the number of bytes of buffer space available for the receiver side of the transport protocol by the maximum transport protocol data unit size to yield an integer value for CDT; for example, if there are 4,096 octets of receive buffer available, and the maximum transport protocol data unit size negotiated for this transport connection is 1,024, then CDT could be set initially to 4.

Reliability Mechanisms to Deal with the Real World

Of the OSI transport protocol classes, only TP4 was developed to deal with real-world situations in which, despite the best efforts exerted by the network layer, bad things happen to transport protocol data units on their way from source to destination: specifically, transport protocol data units may get lost, multiple copies of the same transport protocol data unit may be delivered, transport protocol data units may be delivered to

the wrong end system, bits of the transport protocol data unit header and (worse) user data may get corrupted, and the transport protocol data units may arrive out of order. TP4 deals with these errors as follows:

- Receipt of each data packet is explicitly acknowledged using an acknowledgment packet; absent an explicit acknowledgment for a data packet sent, the sender assumes that it has been lost (or misdelivered).
- Destination transport connection reference information encoded in each data and acknowledgment packet is used to distinguish the data and acknowledgments of one transport connection established between a pair of transport service access points from those of another and to determine whether a data packet has been misdelivered.
- Transport protocol data unit numbers encoded in the data and acknowledgment packet headers are used to detect out-of-order arrival and duplicate arrival.
- Adaptive timer mechanisms are used to avoid injection of duplicate packets into the network.
- When selected, an arithmetic checksum computed on the user data of each data packet is used to detect bit-level corruption.

An important function performed by sending TP4 entity is determining that loss has occurred and correcting for the error; this mechanism is called *retransmission on time-out* in OSI (see Figure 12.14). Each

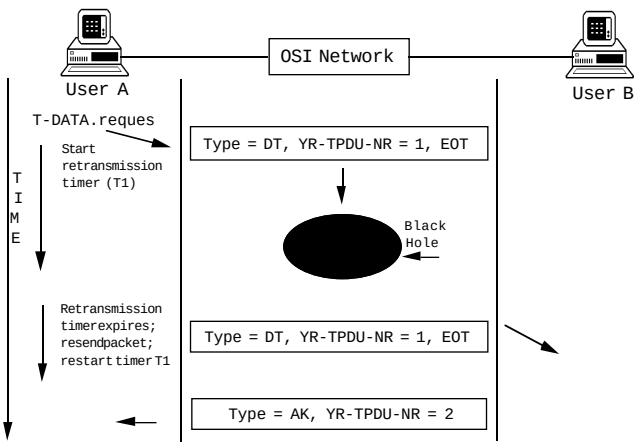

FIGURE 12.14 Retransmission on Time-out

time a TP4 entity sends a DT transport protocol data unit, it runs a retransmission timer *(T1)*; if the explicit acknowledgment of receipt of the data packet is not received by the sender prior to the expiration of the T1 timer, the sender assumes that the data packet is lost and retransmits it.

The detailed manner in which the sender and receiver cooperate to achieve reliability through retransmission is best understood by examining each separately.

Sender TP4 Responsibilities

Like the lesser transport protocol classes, the sending TP4 transport entity segments transport service data units into data packets (if necessary); the data packets are often a fixed size (except the last segment of a packet), up to the maximum packet size negotiated during connection establishment.[16] A sequence number is assigned to each data packet (YR-TPDU-NR in Figure 12.14); the initial value of the YR-TPDU-NR field is always 0, and YR-TPDU-NR is incremented by 1 for all subsequent data packets transferred in the transport connection. The combination of DST-REF and YR-TPDU-NR is used to differentiate transport protocol data units of different transport connections multiplexed between the same pair of transport entities. If multiple transport protocol data units are required to transfer a single transport service data unit, the *end of transport service data unit* (EOT) bit in the data packet header is set to 0 in all but the packet containing the final segment of the transport service data unit (indicating that there are more user data to come); the EOT bit is set to 1 in the data packet containing the final segment of the transport service data unit (see Figure 12.15).

Initially, the sending TP4 entity may send one or more data packets, up to the value indicated in the CDT field of

- The connect request packet if the sender was the responder during connection establishment.
- The connect confirm packet if the sender was the initiator during connection establishment.

The sender retains a copy of each data packet sent. It also maintains information about the number of transport protocol data units it has sent, as well as the *sequence numbers of* those packets, and the sequence number of the next data packet the receiver expects[17] (the lower window edge).

16. In practice, the maximum packet size should be less than or equal to the maximum NSDU size offered by the network layer between the source and destination end systems.
17. Sequence numbers in acknowledgment packets are interpreted as meaning "I acknowledge receipt of all data packets, in sequence, up to but not including the sequence number indicated in the YR-TPDU-NR field."

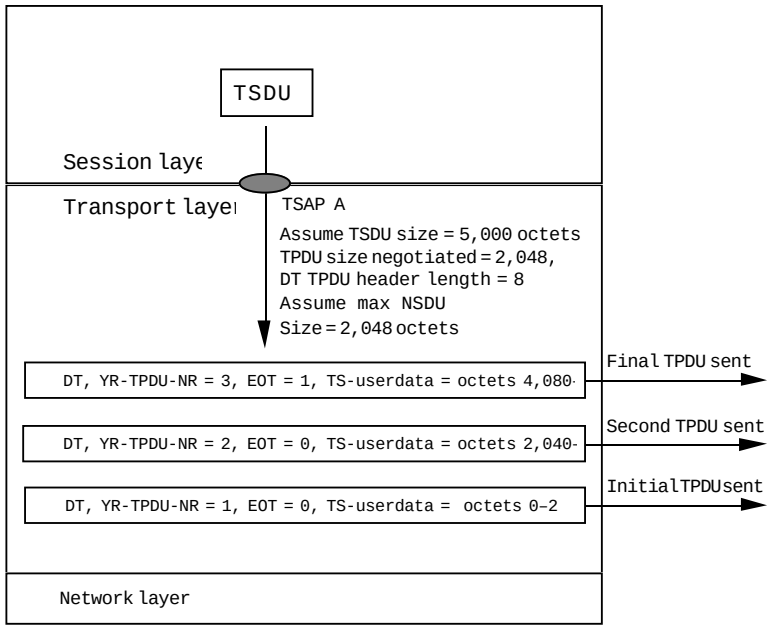

FIGURE 12.15 Example of Transport Segmentation/Reassembly[18]

The sender adds the value of the credit indicated by its peer transport entity to the lower window edge, creating the new upper window edge. These two values determine the new send window; acknowledgments containing sequence numbers outside this window are as-sumed to be duplicates.

Retransmission Timer

A retransmission timer is run for each data packet, or for a set of data packets forming a contiguous sequence (send window). If an acknowledgment is received containing a sequence number within the send window, the sender safely assumes that all data packets having sequence numbers up to (but not including) this value have been received. The value of YR-TPDU-NR from the acknowledgment packet becomes the new lower window edge, the value of CDT in the acknowledgment packet is added to the lower window edge to arrive at a new upper window edge, and the new send window is recomputed.

If the retransmission timer expires, several implementation alterna-

18. To avoid situations in which dividing the transport user data into maximum length packets would result in an exceedingly small "final" TPDU, algorithms that determine the optimal segmentation based on available network maximum data unit size are applied.

tives are available to the sender; for example, it can update the lower window edge and retransmit only the data packet whose YR-TPDU-NR is equal to the value of the lower window edge, or it can retransmit all or a part of the reevaluated send window (from new lower window edge to old upper window edge). (The value of the retransmission timer is discussed separately; see "Timers and Open Transport Protocols," later in this chapter.)

Receiver TP4 Responsibilities

The primary responsibilities of the receiver are to explicitly acknowledge correct receipt of individual data packets using an acknowledgment packet, correctly reassemble data packets into transport service data units (when necessary), and deliver transport service data units to the called transport service user. Acknowledgment packets are generated

- Upon successful receipt and processing of a data packet.
- When advertised credit is reached (the window is full).
- To *allocate credit;* i.e., to increase or reduce the upper window edge (to open or close the window) and thus identify the number of data packets the receiver is willing to handle. (Often, the initial value is a simple calculation of available receive buffer size divided by the negotiated maximum packet size.)

A receiver TP4 entity also returns an acknowledgment packet upon

- Receipt of a duplicate data packet.
- Expiration of the window timer (see "Timers and Open Transport Protocols").

When the receiver determines that a transport service data unit is arriving in segments (multiple data packets), it must also worry about correct reassembly of the transport service data unit. The retransmission procedures assure that the sender will resend data packets the receiver does not acknowledge (they were not received or were received and failed the checksum computation), and the receiver weeds out duplicate data packets using the TPDU numbering and reference fields of the data packet, but transport protocol data packets may still arrive out of order. One possible strategy that a receiver may use to correctly order data packets that have arrived out of order is to maintain

- An in-order list, containing data packets units of a partially reassembled transport service data unit that arrive in sequence.
- An out-of-order list, containing data packets of a partially reassembled transport service data unit that arrive out of sequence.
- A next expected transport protocol data unit number (NEXT-EX-

PECTED-TPDU-NR) variable, containing the value of the next expected TPDU sequence number (i.e., the sequence number of the data packet that would sequentially follow the last data packet in the in-order list).

When a data packet arrives, if the value of YR-TPDU-NR transport protocol data unit NR is not the same as the value of next expected transport protocol data unit number, the receiver adds this data packet to the out-of-order list, *in order* with respect to the rest of that list. If a data packet arrives and the value of YR-TPDU-NR is the same as the value of *next expected transport protocol data unit number*, the receiver adds the data packet to the tail of the in-order list, checks the out-of-order list for one or more data packets that follow the newly arrived data packet in sequence, and moves these to the in-order list. If at any time in this process a data packet is encountered containing the final segment of the transport service data unit (EOT = 1), the process is stopped; otherwise, the receiver adjusts *next expected transport protocol data unit number* to contain the value of the sequence transport protocol data unit number of the next expected data packet. Alternatively, the receiver can maintain only an in-order list and discard without explicitly acknowledging any data packets received out of order; the sender will dutifully retransmit these data packets according to the retransmission on time-out procedures.

The receiver has an explicit mechanism—setting the value of *CDT*—for controlling the number of transport protocol data units it is expected to be able to receive at a given time. The receiver can increase or decrease credit (open or close the send window) as necessary to exercise some control over the way its local resources are used. Suppose, for example, that a transport implementation has 64K of buffer space. If a maximum packet size of 1K is negotiated for a transport connection, a theoretical credit of 64 is available; anticipating that multiple transport connections to multiple destinations may be established, an implementation may allocate an initial credit of 8 for up to eight transport connections, then reduce the credit for all transport connections as additional transport connections share the buffer space. (It is expected that some rational value for the maximum number of concurrent transport connections is applied in all implementations; certainly, here, the value must be less than 65!)

Sequencing Acknowledgment Packets in OSI TP4

Like data packets, acknowledgment packets can also be lost, duplicated, corrupted, or delivered out of order. Failing to correct these errors for data packets will corrupt user data; failing to correct these errors for

acknowledgment packets will cause the transport protocol to misbehave.

Lost or corrupted acknowledgment packets are recovered as part of the retransmission or window resynchronization processing. Acknowledgment packets are retransmitted when a data packet is received that contains a sequence number outside the send window (lower than the lower window edge or greater than the upper window edge) or when the window timer expires (see "Timers and Open Transport Protocols," later in this chapter). It is also important to be able to determine whether an acknowledgment packet is a duplicate or an indication of a new credit value (a window update). Acknowledgment packets thus often have a *subsequence number* encoded in the variable part of the transport protocol header. This number is used to order acknowledgment packets to ensure that the same credit value is used by both the sending and receiving parties (i.e., that both have the same understanding of what the window looks like).

Transport Protocol Data Unit Concatenation—Piggybacking Acknowledgments, ISO-style

To improve protocol performance, especially in multiplexing scenarios, OSI TPs may group multiple transport protocol data units into a single NSDU. The rules are straightforward: any number of acknowledgment, expedited acknowledgment, reject, error, or disconnect confirmation packets from any number of transport connections may be prepended to a single connect request, disconnect request, connect confirm, data, or expedited data packet; i.e., only one packet from the latter set may be present, and it must be the last transport protocol data unit in the NSDU. In the case of TP4, the most common occurrence of concatenation is likely to be that of a single acknowledgment packet with a data packet for the same transport connection (see Figure 12.16). For TP2 and TP4 over a net-

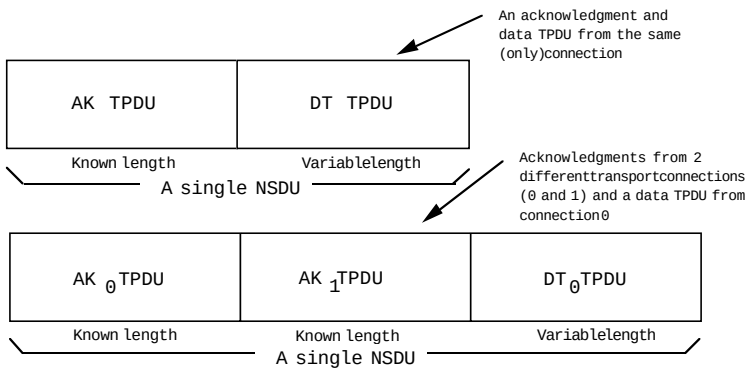

FIGURE 12.16 Examples of TPDU Concatenation

work connection, concatenation has additional benefits: one can, for example, concatenate acknowledgments from different transport connections with a single data packet, send a single large NSDU, and save "turnaround time." Compared to the TCP piggyback mechanisms, in which one need only set the TCP acknowledgment AK flag to true and populate the TCP acknowledgment sequence number, this admittedly is overhead and overkill—another demonstration that flexibility costs.

Data Transfer in TCP—More of the Same

TCP's name for retransmission on time-out is *positive acknowledgment and retransmission*. Mechanisms exist in TCP for detecting and correcting the same set of errors as in TP4. What distinguishes TCP from TP4 in data transfer is *encoding* rather than *functionality*; for example:

- TCP transfers octet streams, not fixed blocks of user data. The 32-bit sequence number in a TCP segment represents the number of the octet in the stream, not the number of the TCP packet.
- The TCP acknowledgment number indicates the next expected *octet*, as opposed to the next expected TCP segment.
- The acknowledgment flag may be set to true to indicate that the acknowledgment sequence number (and window) is significant in data segments in the return stream (piggybacking).
- The 16-bit TCP window indicates in octets the amount of data the receiver is willing to accept in the next TCP segment(s); this value is added to the acknowledgment sequence number to determine the send window. Thus, window is TCP's octet equivalent of TP4's credit.
- A *push* bit in the code field of the TCP segment may be used to decrease delay; i.e., its use overrides TCP's attempt to fill a maximum segment sized packet before sending. (Although it can be misused, push is not intended to be a delimiter of segments as is OSI's EOT bit.)

Figure 12.17 depicts two TCP scenarios—successful transfer with positive acknowledgment and loss followed by retransmission. Many of the sender and receiver responsibilities and strategies described for TP4 were

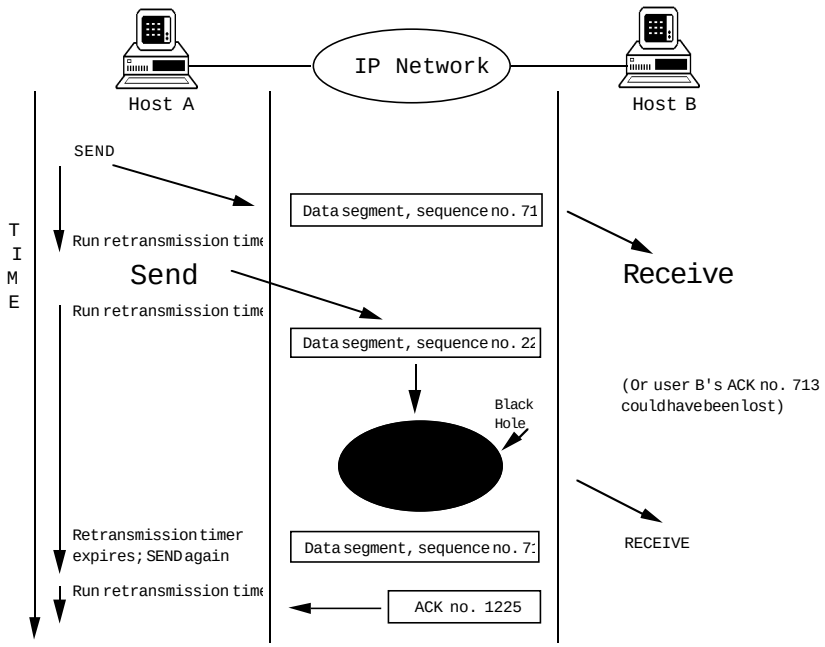

FIGURE 12.17 Data Transfer, TCP-style

derived from the operation of TCP, so these will be familiar to readers.

The sending TCP entity collects data from the upper-layer protocol process and sends those data "at its convenience" (seriously, that's what RFC 793 says . . .). Typically, the sender attempts to fill a maximum segment size (MSS) packet before sending (unless a PUSH is invoked). The default maximum segment size is 536 octets, which allows for a standard TCP header and 512 octets of user data and fits neatly into the default IP packet of 576 octets, assuming an IP header of 40 octets (RFC 879).

The sending TCP entity runs a retransmission timer for each TCP data segment. If the retransmission timer expires and no acknowledgment packet has arrived indicating successful delivery of the segment to the receiver, TCP assumes the data segment is lost, arrived corrupted, or was misdelivered; resends the data segment; and restarts the retransmission timer for this segment. RFC 793 suggests two resend strategies: if the retransmission timer expires, TCP may resend the next unacknowledged segment ("first-only" retransmission), or it may resend all the data segments on the retransmission queue ("batch" retransmission).

The receiving TCP entity may apply one of two acceptance strategies. If an "in-order" data-acceptance strategy is used, the receiving TCP

entity accepts only data that arrive in octet-sequence order and discards all other data. The receiving TCP entity returns an acknowledgment to the sender and makes the octet stream available to the upper-layer protocol process as it arrives. If an "in-window" data-acceptance strategy is employed, the receiving TCP entity maintains segments containing octets that arrive out of order separately from those that have arrived in order and examines newly arrived data segments to determine whether the next expected octet in the ordered stream of octets has arrived. If so, the receiving TCP entity adds this segment's worth of octets to the end of the octet stream that had previously arrived in order and looks at the out-of-order stream to see whether additional octets may now be appended to the end of the in-order stream. The TCP entity returns an acknowledgment and makes the accumulated stream of in-order octets available to the upper-layer protocol process.

An explicit acknowledgment is returned in a TCP segment (potentially "piggybacked" with data flowing in the opposite direction). The *acknowledgment sequence number X* indicates that all octets up to but not including *X* have been received, and the next octet expected is at sequence number *X*. The *segment window* indicates the number of octets the receiver is willing to accept (beginning with sequence number *X*). Acknowledgment packets reflect only what has been received in sequence; they do not acknowledge data packets that arrived successfully but out of sequence.

When an acknowledgment packet arrives, the sending TCP entity may choose to resend all unacknowledged data from sequence number *X* up to the maximum permitted by the segment window. In theory, applying the "batch" retransmission strategy results in more traffic but possibly less delay. The sending TCP entity may resend only the data segment containing the first unacknowledged octet. This negates a large window and may increase delay, but it is preferred because it introduces less traffic into the network. Batch retransimssion strategies are generally regarded as bad ideas, since their excessive retransmission of segments is likely to contribute to network congestion.

Window Considerations for TP4 and TCP

Managing the send and receive windows is critical to the performance of OSI and TCP networks. Every network has a finite forwarding capacity, and absent constant monitoring of network "busy-ness," transport entities can easily submit packets faster than the network can forward and

deliver them, even if they are all dutifully abiding by the windows advertised for their respective transport connections (this is simply a case in which the sum of the advertised windows exceeds the capacity of the network). Networks that become too busy or congested do unkind things such as discard packets. Since congestion has the undesirable effect of causing retransmission (either because delays increase and transport entities presume loss and retransmit or because the network is in fact discarding packets due to congestion) and retransmission results in delay, it is important that transport implementations try their best not to retransmit unless they are very sure they must. On the other hand, too much caution will also cause delay; an overly conservative retransmission timer will wait too long before causing genuinely lost packets to be retransmitted. The trick, evidently, is to wait long enough, but not too long.

A number of different mechanisms are available to deal with this conundrum, most of them applicable to both TCP and TP4. One of the most successful is called "slow-start" (Jacobson 1988). Slow-start is a simple mechanism and follows a simple philosophy: as new transport connections are established, they shouldn't upset the equilibrium that may exist in a network by transmitting large amounts of data right away. In other words, transport connections should not be opened with large windows (or credits); rather, the window (or credit) should initially be small and should grow as evidence of the network's ability to handle more packets is returned in the form of acknowledgments for each packet sent. Slow-start recognizes that a receiver advertises a window or credit of a certain size based on the receiver's ability to handle incoming packets, which is closely related to the availability of buffers and processing cycles at the receiver but has nothing whatsoever to do with congestion in the network. It does not follow, therefore, that the most appropriate strategy for the sender is to immediately fill the window offered by the receiver; the sender must also take into account the effect of its behavior on the network.[19] Slow-start couples *flow control* (ensuring that a sender

19. A variation of the familiar "tragedy of the commons" applies to the behavior of hosts sending traffic into an internetwork, since most of the algorithms that have been devised for congestion avoidance and control in internetworks depend on a "good network citizen" collaboration among host transport protocol implementations to globally maximize the traffic that can be handled by the network without congestion collapse. An unscrupulous host can attempt to take advantage of its well-behaved neighbors by deliberately sending traffic into the network at a rate that would produce serious congestion but for the willingness of other hosts to "back off" as the overall network load rises. This problem has been addressed by a combination of legislation (the Internet standards require, for example, that all Internet TCP implementations use Jacobson's slow-start algorithm) and negative reinforcement (operating the internetwork in such a way that "selfish host

does not send faster than its receiver can receive) with *congestion control* (ensuring that the traffic generated by all senders does not overwhelm the capacity of the network).[20]

A transport implementation using slow-start maintains two windows that govern the rate at which it sends packets: the normal "usable" window (the difference between the window or credit offered by the receiver and the amount of outstanding [unacknowledged] data that are already in the window) and a separate "congestion" window, which is a running estimate of how much data can be sent without congesting the network. The transport protocol then uses the congestion window, rather than the usable window, to control the rate at which it sends new data. (Correct operation of the transport protocol requires, of course, that the size of the congestion window never exceed the size of the usable window.)

Slow-start divides the lifetime of a transport connection into "phases." The first phase begins when the connection is established. The congestion window at the beginning of a phase is always set to 1 packet (for TCP, this is the maximum segment size; for OSI transport, 1 transport protocol data unit); thereafter, as long as no packets are lost, the congestion window is increased by 1 packet every time an acknowledgment packet is received, subject to an upper bound of either the current usable window (which the congestion window must never exceed) or the current "slow-start threshold" (which is half the value of the congestion window at the end of the previous phase). This has the effect of opening the congestion window rapidly[21] until a threshold (or absolute upper bound) is reached or a packet loss occurs (which suggests that the window may have been opened too far).

The detection of a packet loss, which triggers retransmission of the lost data, ends a phase. The next phase begins with the congestion window back at 1 and a new slow-start threshold of half the congestion-window value that was in effect when packet loss terminated the previous

behavior is punished—for example, by using "fair queuing" in routers, so that individual hosts see the effects of congestion [dropped packets and increased transit delay] caused by their own traffic as well as by the total traffic load on the network).

20. It is important to understand that flow control is strictly an element of the host-to-host transport protocol (in which the network does not participate), whereas congestion control has both host-based and network-based elements.

21. The congestion window value grows exponentially during this part of the slow-start procedure, since every time a window of N data packets is sent, N ackowledgments are received in return, increasing the window by N (1 for each acnowledgment); starting at $N = 1$, the progression (assuming there is no packet loss) is 1, 2, 4, 8 . . . .

phase.[22] This "be more conservative next time" strategy, which halves the maximum congestion window every time a packet is lost, would by itself eventually shrink the window to 1 packet—solving the congestion problem, to be sure, but also reducing the transport protocol to an inefficient send-and-wait mode of operation. To avoid this, slow-start is paired with a strategy that Jacobson (1988) calls "congestion avoidance," which allows the congestion window to grow past the threshold—but much more slowly. After the congestion window has reached the slow-start threshold, it is incremented by its reciprocal (rather than by 1) each time an acknowledgment packet arrives.

The slow-start/congestion-avoidance algorithm has been widely implemented in TCP but has only recently found its way into TCP's counterpart in OSI, TP4. In OSI networks based on TP4 and the connectionless network protocol CLNP, it is also possible to detect and signal network congestion by using the *congestion experienced* flag in the QOS Maintenance field of the CLNP header (see Chapter 13) and the base credit-management and retransmission strategies on the work of Raj Jain (1985, 1986a, 1986b, 1990); in fact, Jacobson's congestion-avoidance strategy is nearly identical to Jain's, which differs primarily in its use of a smaller window-shrinking factor when congestion is signaled.

OSI's Expedited Data

OSI transport expedited data is an entirely separate data flow packet. It is not subject to normal data flow control and has its own packet type, acknowledgment, and sequence space (see Figure 12.18). In theory, transport expedited data is used when user data of great urgency must be transferred. Expedited data reminds one of the childhood practice of cutting ahead in line: an expedited data packet is placed at the head of the outbound queue, and although it is not expected to overtake any previously submitted data packets, it must be delivered before any data packet is submitted after it. Expedited data can cut in line, but it may not really be processed with the urgency it expects and may well end up being transferred no more quickly than if it had been submitted as normal data.

Expedited data is highly constrained. Only one expedited data packet

22. The actual formula for calculating the new threshold value is not really as simple as "half the old threshold," but it is close enough for the purposes of this discussion. Not all the details (such as doubling the value of the retransmission timers for unacknowledged packets waiting in the window when a phase-ending packet loss occurs) are covered here; implementers should see Jacobson (1988) and Zhang (1991).

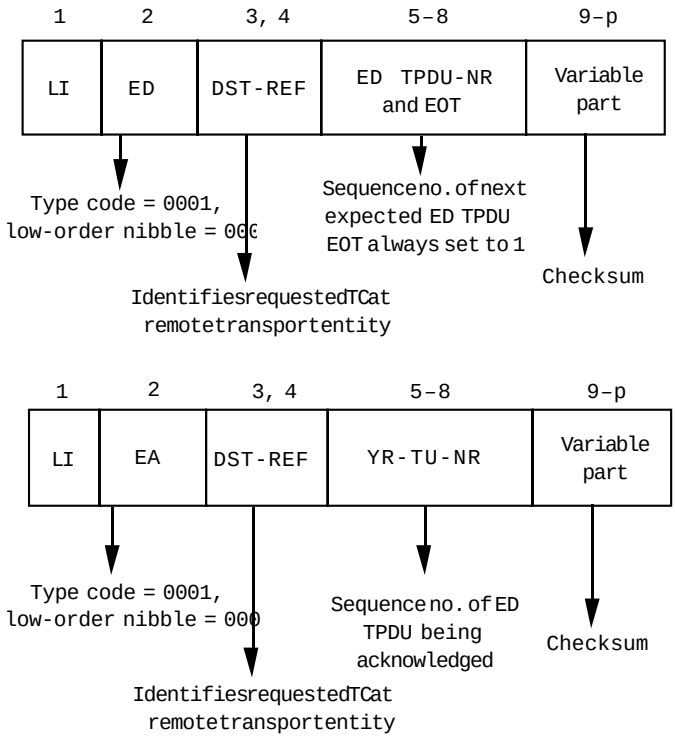

FIGURE 12.18 ED and EA Transport Protocol Data Units

may be outstanding (unacknowledged) at a time; each expedited data TSDU maps onto a single *expedited data* packet (EDT PDU) and it can be used only to transfer 16 weenie little octets.[23] Another curious bit of protocol encoding is the presence of an EOT bit that is always set to 1.

TCP's Urgent Data

TCP's notion of urgent data is somewhat more flexible. TCP allows an upper layer protocol to mark data in the stream as *urgent* for the receiver. The sender TCP does so by setting the URG bit in the code field of the

23. The 16-octet limit to user data in expedited data packets is a consequence of attempting to map expedited data packets at the transport layers onto a single X.25 *interrupt* packet, which offers only 32 octets of user data at the network layer. Substract the maximum protocol overhead of an expedited data packet (16 octets), and only 16 octets remain for user data.

TCP segment to 1, indicating that the URGENT field is significant. The value of the URGENT field represents the number of priority delivery octets in this TCP segment (and perhaps in subsequent segments, if the number of octets in this segment is less than the value of the current field). This value is added to the segment sequence number to assist the receiver TCP in identifying the last octet of urgent data. When the TCP segment containing the URG bit arrives, the receiver TCP notifies the upper-layer protocol that urgent data are coming. Although not explicitly stated in RFC 793, it is assumed that the upper-layer protocol will begin processing the urgent data; when the last octet of urgent data arrives, the receiving TCP delivers the urgent data in the TCP segment and notifies the upper-layer protocol that normal data transfer has resumed. (It has been suggested that the urgent data capability is roughly equivalent to the session layer's *activity interrupt* or *capability data* services. Some say this is a stretch.)

Timers and Open Transport Protocols

The most fundamental thing you can say about TP4 and TCP is that they are timer-based: to operate correctly, both protocols rely on the certainty that either an expected event will occur or a timer will expire. This characteristic is responsible for the robustness and flexibility of both of these protocols. The dependence on timers, however, means that the performance of TP4 or TCP is highly sensitive to the choice of timer values and to the way in which the values of different timers are related. It's not terribly difficult to choose reasonably good (initial) timer values or to build implementations that can dynamically adjust them, but the consequences of choosing bad values, or building implementations that either cannot adapt or adapt inappropriately, are much more serious with TP4 and TCP than they are with protocols that do not depend as heavily on timers.

The OSI transport protocol specification provides rudimentary guidelines for establishing initial values for some of the many timers that transport class 4 relies on for correct operation. If network-service behavior were uniform and stable, these guidelines might be sufficient. Unfortunately, the behavior of real-world networks is anything but uniform and stable. The practice of adjusting timer values to react correctly to change in the behavior of networks is fundamental to correct and efficient transport implementations, be they OSI TP4 or TCP. What must be taken into consideration for many of these timers is described in a gener-

al way in the following subsections and is, for the most part, applicable to all retransmission and timer-based transport protocols.

Retransmission Timer

Almost every TP4 packet (or TCP segment) must be either explicitly or implicitly acknowledged. When a transport packet is first assembled and transmitted, it is associated with a retransmission timer. If the expected acknowledgment of the packet is not received before the timer expires, the packet is retransmitted and the timer is restarted (using either the same or a different value for the time-out period); when the acknowledgment is received, the timer is canceled. Until the acknowledgment is received (or the transmission attempt is abandoned after "too many retries"), the sending system must retain enough information to be able to retransmit the original packet if necessary. Dealing with timers and timer-generated events and holding information about packets for some period after they have been sent represent a significant load on the resources of a TP4 or TCP-based system. One of the most important goals of an efficient transport protocol implementation is therefore to minimize this overhead.

The basic problem with retransmission timers is choosing the right time-out interval. An overly sensitive timer will cause the unnecessary retransmission—duplication—of packets that were in fact received and acknowledged correctly; a sluggish timer will respond too slowly to the actual loss of a packet or its acknowledgment, increasing the delay associated with error detection and recovery. Ideally, a retransmission timer should expire only when it is actually the case that a packet or its acknowledgment has been lost or discarded. In practice, however, there is no way to be certain whether or not this has happened—that's why the timers are there.

The realistic goal is to pick (or dynamically approach) a time-out interval that allows the protocol to recover quickly when a packet has been lost (the timer value must not be too large) but reduces below some acceptable threshold the number of occasions on which a packet that was in fact correctly received is retransmitted because the acknowledgment did not arrive before the expiration of the time-out interval (the timer value must not be too small). In any environment in which the loss or corruption of packets is not a rare occurrence, the performance of TP4 and TCP is extremely sensitive to these timer values.

The internet over which TP4 and TCP operates may contain paths with very different delay and throughput characteristics, which may change dramatically during the lifetime of a transport connection, and the timer-based behavior of the transport implementation at the other

end of a transport connection cannot be completely predetermined. Because of this variability, one cannot simply pick an "average" value for any retransmission timer (except in very limited, static configurations); there will always be configurations in which the protocol will not operate at all with such a static value, much less operate efficiently. These timers must adapt dynamically to the actual, observed (or inferred) delay characteristics of each individual transport connection.

The timer-value problem has two parts. To maximize performance, the interval between the initial transmission of a packet and its first retransmission should be tuned as finely as possible, with an adaptive granularity small enough to keep the value close to its theoretical ideal. To ensure that the protocol will nevertheless operate correctly when the attempt to maximize performance leads to the choice of a much-too-small initial timer value for the connection-establishment phase and to allow it to cope with sudden, relatively large transient or persistent changes in end-to-end delay during the data-transfer phase, the interval must be increased for the second and subsequent retransmissions (when necessary) in such a way that a sufficiently large interval is allowed to expire before the transmission attempt is deliberately abandoned ("too many retries").

Choosing and Adjusting Retransmission Timer Values

If it were possible to periodically measure the actual end-to-end delay between two transport connection endpoints, the corresponding retransmission timer values could be adjusted up or down accordingly. Unfortunately, TP4's normal data packets and their acknowledgments cannot always be used for delay measurement, since a single acknowledgment packet can acknowledge more than one data packet, and the use of a selective acknowledgment strategy by the receiver can artificially skew round-trip delay measurements. TP4 expedited data packets, which must be acknowledged immediately and individually, could be used for this purpose, but the expedited data option is not always selected, and even when it is, there is no guarantee that expedited data will flow regularly enough (or in some cases, even over the same path) to provide the necessary dynamic delay information.

Although it is not feasible to obtain a direct measurement of end-to-end delay, the first part of the timer-value problem can be solved successfully by using a trial-and-error technique that adjusts the timer value based on observed retransmission behavior: crank the retransmission time-out interval down until the number of retransmissions per measurement interval starts to climb and then gently bump the time-out value back up until the number of retransmissions drops just below some

acceptable threshold. When the number of retransmissions rises above the threshold, the retransmission interval is increased; when it drops below the threshold, the interval is decreased. The goal is to maintain an equilibrium just below the threshold.[24]

This "adaptive retransmission" scheme assumes that "false" retransmissions caused by a too-short retransmission time-out interval can be distinguished from "real" retransmissions caused by the actual loss of data packets—that when the retransmission interval is reduced below a certain threshold, the resulting increase in the number of "false" retransmissions will be detectable against the fluctuating background of "real" retransmissions. Since there is no objective way to determine whether any individual retransmission is "real" or "false," this scheme depends on recognizing patterns in the observed retransmission behavior that can be related to deliberate adjustments of the retransmission time-out interval. The basic technique, described earlier, is simply to raise the time-out value when the number of retransmissions increases and lower it when the number of retransmissions decreases, in an attempt to keep the number of retransmissions at some "optimal" level (the threshold).

For most real-world configurations, this basic technique is much too simplistic. It works only if changes in round-trip delay are the only significant cause of changes in the number of retransmissions per measurement interval; the "threshold" number of "real" retransmissions must be known in advance and must not change significantly. Even when these conditions are met, the retransmission timer value will oscillate whether or not the retransmission behavior changes, unless a longitudinal damping function is used to stabilize it. There are a number of ways to improve the basic adaptive retransmission scheme. A smoothing function that accounts for recent history (one or two measurement intervals back) can be used to damp oscillation of the time-out value around the threshold. A simple first-order smoothing function might operate to ensure that the time-out value is adjusted only when a change in retransmission behavior has persisted for two or more consecutive measurement intervals. A second-order function (which accounts for changes in the rate of change of the number of retransmissions) can be used to damp oscillations even further, depending on how widely the number of "real" retransmissions is expected to fluctuate and on how firmly these oscillations must be damped to provide acceptable perfor-

24. Although they are discussed separately in this chapter, the dynamic adjustment of retransmission timers and the sliding window flow-control strategy (introduced earlier) will be closely coupled in any actual transport protocol implementation.

mance. No damping function, however, can prevent performance-killing inflation of the re-transmission time-out interval or of the "threshold" number of retransmissions that are interpreted as "real" (and therefore acceptable).

Consider, for example, the following scenario. A change in the characteristics of the end-to-end path over which packets are flowing causes an increase in the number of packets that are lost and/or corrupted; this causes a corresponding increase in the number of packets that must be retransmitted. These are "real" retransmissions, but neither of the transport protocol machines involved has any way of knowing this; as far as they can tell, the observed increase in the number of retransmissions might just as well be caused by premature expiration of the re-transmission timer due to an increase in the end-to-end transit delay. If this condition persists, the retransmission time-out will be adjusted upward until the number of retransmissions per measurement interval stops increasing (if the condition disappears fast enough, and the number of retransmissions drops back to its former level within the granularity of the damping function, the retransmission time-out will not be changed). The increase in the number of retransmissions had nothing to do with the value of the retransmission timer, but the adaptive retransmission algorithm thinks that its action in raising the time-out value is responsible for halting the increase in the number of retransmissions (because when the time-out interval was raised, the increase in the number of retransmissions stopped—Piaget would love this algorithm). If the algorithm's analysis goes no further than this, the time-out interval will stabilize at a new (higher) value and will be driven back down only if the number of retransmissions starts to decline. Just when efficient retransmission behavior is most important (to minimize the adverse effects of the increase in the number of lost and/or corrupted packets), the retransmission time-out interval is inflated, increasing the time it takes the protocol to recover from errors. This is not good.

In principle, the retransmission time-out interval should be adjusted only to account for changes in transit delay; changing the time-out interval will not affect the number of retransmissions that are due to other causes (such as a change in the number of lost or corrupted packets). But because it is not possible to distinguish "real" retransmissions (due to loss and/or corruption of packets) from "false" ones (due to premature retransmission timer expiration, caused by a mismatch between the time-out value and the actual end-to-end transit delay), the inflation just described cannot be prevented. It can, however, be corrected after it has occurred by making the basic adaptive retransmission algorithm more

sophisticated. When the number of retransmissions per measurement interval changes spontaneously, the algorithm has no choice but to change the retransmission timer value accordingly (damping small oscillations). When the number of retransmissions is stable, however, the algorithm can deliberately alter the retransmission time-out: increase it to see whether the number drops or decrease it to see whether the number rises. By periodically challenging a stable timer value, the algorithm can correct inflation of the time-out interval and can also correct a too-short time-out interval that is producing an unnecessarily high (but stable, and therefore unprovocative) number of retransmissions.

Adapting retransmission timer values to cope with variable delay is certainly not unique to OSI; it has been observed and managed in TCP networks for many years. In RFC 793, it is recommended that the retransmission time-out be based on *round-trip time* (RTT), which is computed by recording the time elapsed between sending a data segment and receiving the corresponding acknowledgment and by sampling frequently. The algorithm used to compute the round-trip time is:

```
smoothedRoundTripTime = (å* oldRoundTripTime) + ((1 – å)*newRoundTripTime)
```

where å, a weighting factor, is selected such that $0 < å < 1$.

A small å responds to delay quickly; a large å, slowly. The time-out value should be greater than the round-trip time but within reason; for example:

```
time-out = minimum(upperBound,ß*smoothedRoundTripTime)
```

where ß, a delay variance factor, is selected such that $1.3 < ß < 2$.

Some deficiences have been identified and corrected in this initial algorithm. Karn and Partridge (1987) observed that retransmitted segments cause ambiguities in the round-trip time computation; specifically, if the sender cannot determine whether the acknowledgment corresponds to an original data packet or a retransmission, it cannot determine the correct round-trip time for that packet. The Karn/Partridge algorithm computes the round-trip time only for packets that are not retransmitted and increases the retransmission timer by a multiplicative factor (2 is suggested) each time a segment is retransmitted.

Further study showed that limiting ß in the manner described in RFC 793 will fail if delays vary widely, and Jacobson (1988) proposes that estimates for both the average round-trip time and the variance should be provided and that the estimated variance be used in place of ß. These algorithms work as well in TP4 implementations as in TCP implementations.

Connection-Establishment Timers

During OSI transport connection establishment, two timers govern the re-transmission of the connect request and connect confirm packets. These packets are sent out before there has been any opportunity to observe or infer the end-to-end round-trip delay; and in general, no reliable "pregenerated" information about the probable delay to a given destination is available (although when it is, it can be used to guide the selection of initial CR and CC timer values). The end-to-end delay over a single subnetwork (link) might be anywhere from 1 or 2 milliseconds (for a LAN) to 250 milliseconds (for a satellite link), and there could be almost any number of these links, in various combinations, in the actual end-to-end path. The round-trip delay also includes processing time in the two end systems and in an unpredictable number of intermediate (gateway) systems. Under these circumstances, the probability of correctly guessing the optimal timer value (or even something acceptably close to it) is very small.

There is an alternative to simply picking a timer value at random. As long as the retransmission interval is increased substantially for second and subsequent retransmissions (when necessary), a very small initial time-out value (on the order of 250–500 milliseconds) can be used. This will give good performance when the actual delay is small (and configurations with small end-to-end delay are precisely the ones in which high performance is likely to be most important). The possible unnecessary retransmission of one or more connect request or connect confirm packets when the actual delay is larger than the small initial value chosen for the timer is usually acceptable, occurring as it does only during the connection-establishment phase. Incrementally backing off the retransmission timer each time it expires (using, for example, a 500-millisecond increment) and setting the "maximum number of retries" threshold fairly high (at 15, for example) can ensure that very few (if any) connection-establishment attempts are abandoned (timed out) prematurely (see "Backing Off for Subsequent Retransmissions," later in this section).

Data Retransmission Timer Value

A simple way to pick a starting value for an adaptive data packet retransmission scheme is to measure, during the connection-establishment phase, the delay between sending a connect request and receiving the corresponding connect confirm (or at the other end, between sending a connect confirm and receiving the corresponding acknowledgment or first data packet). Because the connect request and/or connect confirm packets may be retransmitted, and because processing delays associated with connection establishment are usually greater than those associated with normal data flow, this value cannot be used as a constant for the

data packet retransmission time-out value; it is likely to be accurate enough, however, to ensure that an adaptive retransmission algorithm quickly converges on a satisfactory value. An adaptive retransmission scheme is most useful either when no prior information about the round-trip delay variance is available or when the available information suggests that the delay variance could be large. When it is possible to expect that the delay variance will be relatively small, better performance can be obtained from a well-chosen constant value for the data retransmission time-out value, based on a slight overestimate of the expected maximum round-trip delay (this is especially true for operation of TP4 over a network connection). The retransmission timer associated with expedited data packet can be managed in the same way as the timer associated with normal data packets.

Backing Off for Subsequent Retransmissions

No matter how cleverly the initial value for a retransmission timer is chosen, there will be circumstances in which the timer expires, the associated packet is retransmitted, and the timer must be reset. A simple approach to choosing a new timer value is to reuse the initial value. This approach will produce a series of retransmissions at evenly spaced intervals, which will terminate when the retransmission timer is canceled by the arrival of an appropriate acknowledgment or the maximum number of retries is reached. If the initial time-out value is not too far off the mark, or the maximum number of retries parameter is very large, this approach will work. If the initial time-out value is much too short, however, either the maximum number of retries will be exhausted before any acknowledgment has had time to arrive, or a large number of unnecessarily retransmitted packets will be pumped out of the sending system before the acknowledgment arrives. In the former case, the transport connection or connection-establishment attempt will be aborted when the sender's give-up timer expires (the value of the give-up timer depends on the retransmission timer value and the maximum number of retries, as discussed in the following subsection). In the latter case, adaptation of the initial time-out interval (as described earlier) will eventually correct the problem for data packets, but the situation will persist for connect request and connect confirm packets, for which no adaptive adjustment of the retransmission timer value is possible.

Using an equal-interval approach to retransmission constrains the choice of an initial retransmission timer value: if the two pathological situations just described are to be avoided, the time-out interval cannot be reduced below a certain "safety threshold." In configurations in which the mean transit delay is low but the delay variance is relatively high,

this constraint limits the effective performance of the protocol.[25]

An algorithm that backs off geometrically for each retransmission rather than linearly eliminates this constraint. When a packet is first transmitted, the corresponding retransmission timer is set to the appropriate initial value for that packet (which is either a constant, for connect request and connect confirm packets, or a dynamic value determined by an adaptive retransmission scheme, for data packets). If this timer expires, the packet is retransmitted, and the retransmission timer is set to a value that is the sum of the initial value and a fixed increment (the "back-off" increment). If this timer expires, the packet is again retransmitted, and the retransmission timer is set to a value that is the sum of the initial value and twice the back-off increment. This continues until the timer is canceled by the arrival of a suitable acknowledgment (or the transmission attempt is abandoned after "too many retries"). Each retransmission interval is therefore longer than the one before it. When the retransmission interval is increased in this way after each retransmission, the partial sums that represent the accumulated time since the first transmission of a packet grow geometrically rather than linearly. This allows a transport protocol to recover quickly from the choice of a too-small initial timer value, without falling into either of the two traps described earlier. The initial retransmission interval can be made as small as necessary to achieve good performance, relying on the geometric algorithm to back the value off safely if something goes wrong.

For the algorithm just described, the aggregate retransmission time is a function of the constant parameters for the maximum number of retries and the back-off increment. Without making the function too complicated, we can also allow the first-retransmission interval to be different from the back-off increment. Letting x be the maximum number of retries, y the interval between the initial transmission of a data packet and its first retransmission, and z the fixed increment by which the retransmission time is increased for each retransmission after the first, we obtain the following formula:

```
aggregate = y(x + 1) + [zx(x + 1)/2]
```

Note: The value of y for data packet retransmission will change dynamically if an adaptive retransmission scheme is used.

As an example, assume an implementation that has chosen $x = 5$,

25. If the delay variance is low, the probability of either of the two pathological conditions occurring is also low, and the "safety threshold" can be set as low as necessary to avoid this performance limit.

y = 500 milliseconds, and z = 1 second. At most, 18 seconds will elapse between the first transmission of a packet and the expiration of the last retransmission timer ("too many retries"). This value sets a lower bound on the value of the give-up timer.

Give-Up Timer

Associated with each OSI transport connection is a "give-up" timer, which is started (or restarted) whenever the first incarnation of a data packet is sent out (that is, it is not reset when a data packet is retransmitted, as is the data packet retransmission timer). Whenever an acknowledgment covering all outstanding data packets is received, the give-up timer is canceled.[26]

The give-up timer establishes an upper bound on the amount of time that can elapse between the first transmission of a packet and the receipt of an acknowledgment that covers that transport protocol data unit. The expectation of an acknowledgment can remain unfulfilled for no more than the give-up time-out interval before TP4 decides that its peer is either dead, disabled, or malfunctioning; if the give-up timer expires, the corresponding transport connection is torn down.

The give-up timer value should be large enough so that it includes any reasonable combination of end-to-end processing and transmission delays (including the maximum number of retransmissions). It must, however, be less than the value of the reference timer in all other transport protocol machines with which a given implementation will communicate, to ensure that no retransmission can occur after a remote peer has decided that it is safe to reuse a transport connection reference (see "Reference Timer," later in this section). When, for whatever reason, a transport protocol machine stops receiving the packets that are being sent by its peer, the operation of the give-up timer in the sending system ensures that the sender will not inject a packet into the pipe that might (if the pipe eventually clears) arrive at the receiver after the expiration of the receiver's reference timer.

The value of the give-up timer must be greater than the maximum aggregate data retransmission time and less than the value of the reference timer in all other transport systems. In the example described in the previous subsection, the aggregate data retransmission time is 18 seconds. A typical reference timer value (again, see "Reference Timer") is 100 seconds. Within these bounds, a reasonable give-up timer value would be 40 seconds.

26. Whenever the expedited data option is implemented, there are actually two separate give-up timers: one for normal data TPDUs and one for expedited data TPDUs. They operate independently, but in the same way.

Inactivity Timer

On every transport connection, each of the two transport protocol machines involved must regularly demonstrate both existence and sanity to its peer, by sending a correctly formed packet. This is true whether or not the peers have any user data to exchange; in the absence of data flow, the peers exchange acknowledgment packets in response to the expiration of their window timers (discussed later in this section). The protocol depends on this "I'm OK, you're OK" form of phatic communion to maintain connectivity and to detect its loss. Silence, therefore, is an abnormal (and eventually fatal) condition.

The inactivity timer detects silence. It is started when a connection is first established and is reset whenever any valid packet is received for that connection. It expires, therefore, only when a period of silence has persisted for long enough that the local transport protocol machine must assume that its peer has either died or become disabled. The expiration of an inactivity timer results in termination of the corresponding transport connection.

The value of the inactivity timer is chosen to reflect the most appropriate compromise between the desire to keep a sick transport connection open as long as there is a reasonable hope that it can be revived (within the "quality of service" constraints, if any, specified by the transport user) and the desire to recognize a genuinely dead connection as quickly as possible. Since the value of the window timer is directly related to the value of the inactivity timer in all other transport implementations, the inactivity timer value must be chosen carefully.

Window Timer

Whenever there is two-way data traffic on a transport connection, the corresponding flow of acknowledgments in both directions ensures that both transport machines have up-to-date window (flow-control) information. If data packets flow in only one direction, or if there are no data packets flowing in either direction (there are no user data to send, or one or both of the transport protocol machines has closed its receive window), this information must be exchanged by some other mechanism.

The window timer generates a flow of acknowledgment packets that depends only on the existence and health of the sending transport protocol machine; even when there is no need to acknowledge a data packet, an acknowledgment packet will be sent at regular intervals as the window timer expires. This serves two essential purposes: it prevents the remote peer's inactivity timer from expiring in response to a long stretch of silence (that is, it convinces the remote peer that its partner is still alive and well), and it conveys up-to-date window information (credit), which may change whether or not there is current data traffic. To understand

the importance of the latter function, consider that when a receive window closes, a potential sender—which can't send into a closed window, of course, and therefore will not be getting new acknowledgments as a result of having sent new data—will never learn that the receive window has reopened unless the receiver sends an "unprovoked" acknowledgment containing a new credit.

Because one of the roles of the window timer is to prevent the expiration of the inactivity timer in a remote peer, its value must be chosen with respect to the value of the inactivity timer in other transport protocol machines. Since acknowledgments, like other packets, can be lost, the value of the inactivity timer is usually chosen to allow one or two "window" acknowledgments to be generated and lost without risking expiration of the inactivity timer. A typical value for the window timer allows for three window time-outs within an interval that is slightly smaller than the inactivity time-out interval.

Reference Timer

When an OSI transport connection is closed (normally or abnormally), a *reference timer* is started; until it expires, the local connection reference number that was used for the old connection cannot be reused for a new one. The reference timer ensures that a new connection based on a previously used connection reference number cannot be opened until it is certain that all packets generated during the lifetime of the old connection have disappeared.

The reference timer is designed to cope with situations such as the one described in the following scenario. Two transport protocol machines establish a transport connection and start exchanging data packets. At some point, congestion in the path between the two transport peers causes a data packet (and its subsequent retransmissions) to be delayed long enough for the give-up timer in the sending transport protocol machine to expire, terminating the connection. Eventually, however, the congestion clears, and the packet (and/or one or more of its retransmissions) arrives at its destination. If the reference number that was used for the original transport connection has been reused in the interim to establish a new connection, the late-arriving packet could be (mis)interpreted as belonging to the new connection.

It is important to note that no value of the reference timer is large enough to guarantee, by itself, that the confusion just described cannot arise. It is also necessary for the underlying internetwork protocol to operate in such a way that internetwork protocol data units, which carry packets, are discarded after a specific "packet lifetime" is exceeded. The OSI connectionless network protocol includes this function (see Chapter

13). A local system-management function must ensure that the reference time-out value associated with each transport connection is greater than the packet lifetime specified in the network protocol packets that carry data for that transport connection.

Approximately 2^{16} reference numbers are available for use with transport connection endpoints associated with a single NSAP (network) address. Most transport implementations—assuming that even under the most extreme circumstances, it will take at least a couple of milliseconds to set up and tear down a transport connection—will use a value of about 2^7 or 2^8 seconds for the reference timer. If it is very important to recover the local system resources dedicated to a transport connection quickly after the connection is closed, a smaller value will be used. A value of 100 seconds is about the lowest that can be used safely in a general-purpose implementation.

Connection Release (Connection Refusal) in the OSI Transport Protocol

There are actually two circumstances that dictate the release of an OSI transport connection. The first, *connection refusal*, occurs when the called transport entity cannot satisfy one or more of the conditions of transport connection establishment conveyed in the connection request packet, or when the connect request packet received is in error. To refuse a transport connection, the called transport entity composes and returns a *disconnect request* packet (DR TPDU) (Figure 12.19) containing the reason for refusing the connection. The expected reasons for refusing a connection are identified (negotiation failed, reference overflow, transport service access point address unknown), and several more creative reasons for refusing a connection are provided (congestion at transport service access point, session entity not attached to transport service access point).

If the connection is refused due to an inability to parse the connect request packet, the called transport entity returns an error packet (ER TPDU), indicating the reason for refusal. This form of connection refusal typically reflects an error in implementation (e.g., an invalid parameter or parameter value was encountered) or detection of a bit-level error as a result of computing the checksum on the connect request packet. All of the octets of the connect request packet that caused the rejection are returned in the variable part of the error packet.

Connection release is an extremely mundane phase of operation. To

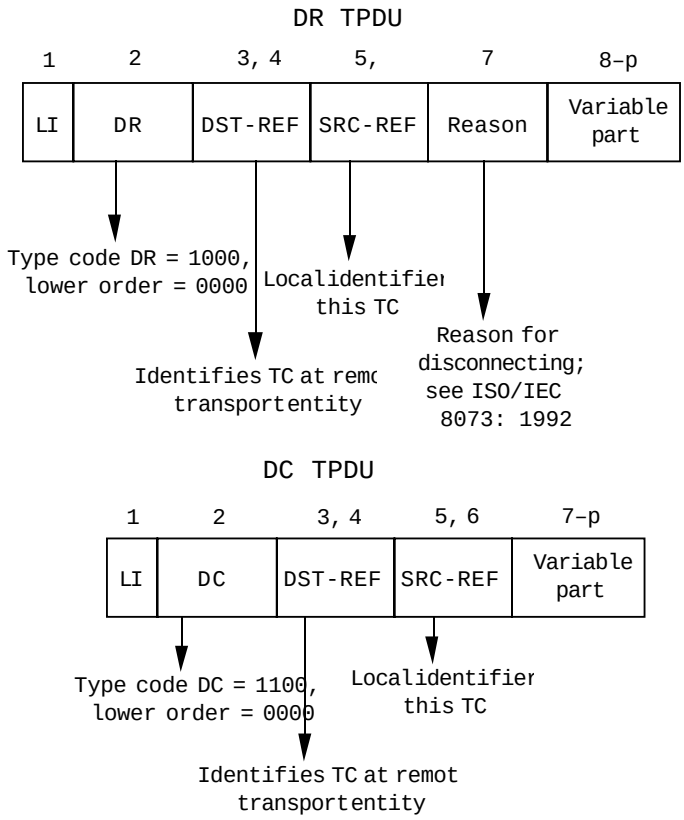

FIGURE 12.19 Transport Protocol Data Units for Connection Release

the degree that one finds transport connections interesting, all of the interesting things have already occurred: both parties have negotiated the characteristics of a connection, meaningful information transfer has taken place, and for those readers old enough to remember, "Now it's time to say goodbye . . . to all our com-pah-nee."[27] The process is especially mundane in the OSI transport protocol since no effort is made to ensure that all data transmitted in both directions have been acknowledged before the transport connection is released (according to the letter of the law as prescribed by the OSI reference model, if a graceful or orderly release is desired, it will be performed by the session layer, when the orderly release functional unit is selected (see Chapter 11); whether

27. For those too young to remember, these are the first words of the closing song of the "Mickey Mouse Club" television show of the fifties.

the functionality implemented at the session layer is equivalent to TCP's graceful close is a subject of ongoing debate).

Essentially, OSI transport connection release is a process of abruptly announcing one's departure from the conversation: one party—the calling or called transport entity, either as the result of an explicit request by a transport service user (a T-DISCONNECT.request) or as a local matter—issues a *disconnect request* packet (DR TPDU) (see Figure 12.20). Following transmission (or reception) of the disconnect request packet, there's lots of tidying up to do:

- All timers related to this transport connection are stopped. For TP4, this may include the retransmission, inactivity, and window timers.
- The receiver composes and returns a *disconnect confirm* packet (DC TPDU) (see Figure 12.19) to the initiating transport entity, and notifies the transport user via a T-DISCONNECT.indication primitive.
- Both parties freeze reference numbers (see "Timers and Open transport Protocols," earlier in this chapter).

Upon expiration of the timer bounding the use of references, the transport entities consider the transport connection closed.

Connection Release (Refusal) in TCP

The circumstances that dictate the release of an OSI transport connection exist for TCP as well. *Connection refusal* occurs in TCP when the responding TCP entity cannot establish a TCP connection or when the SYN packet received is in error. To refuse a TCP connection, the called TCP entity sets the RST and ACK bits in the code field of the TCP packet to 1, and

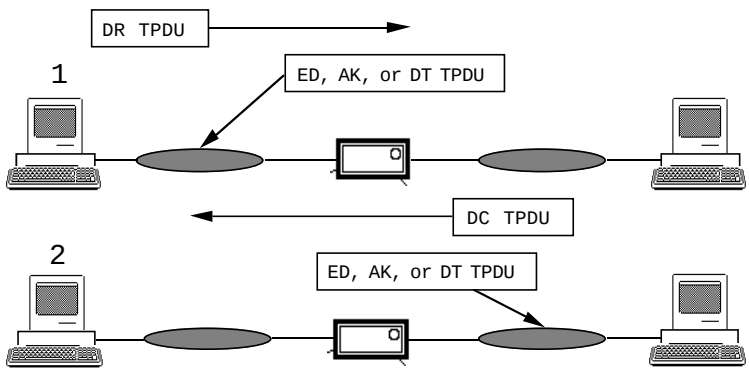

FIGURE 12.20 Connection Release, TP4

sets the acknowledgment sequence number to the initiator's ISN, incremented by 1; unlike the OSI transport protocol, no reason for refusing the connection is indicated in the RST/ACK segment. (A quick examination of the OSI reason codes suggests that this is no great loss.)

TCP offers two forms of *connection release*: abrupt and graceful. Abrupt release indicates that something seriously wrong has occurred. Again, the RST bit of the code field is set to 1; depending on the current state of the TCP entity that receives the RST segment, the sequence-number and acknowledgment sequence number fields may be significant. Graceful close in TCP is an orderly shutdown process. All information transmitted in both directions must be acknowledged before the TCP connection is considered "finished" and may be closed. When an upper-layer protocol has finished sending data and wishes to close the TCP connection, the TCP entity indicates this state to its peer by sending a TCP segment with the FIN bit of the code field set to 1. The sequence-number field is set to the value of the last byte transmitted. The receiver of the FIN segment must acknowledge receipt of the last octet but is not required to close its half of the connection; it may continue to transfer data, and the initiator of the FIN segment must dutifully acknowledge all data received until it receives a TCP segment with the FIN and ACK bits of the code field set to 1 and an acknowledgment number set to the sequence number of the last octet received from the FIN segment initiator. Upon receiving the FIN/ACK segment, the FIN initiator returns an ACK segment, completing a three-way "good-bye," as illustrated in Figure 12.21.

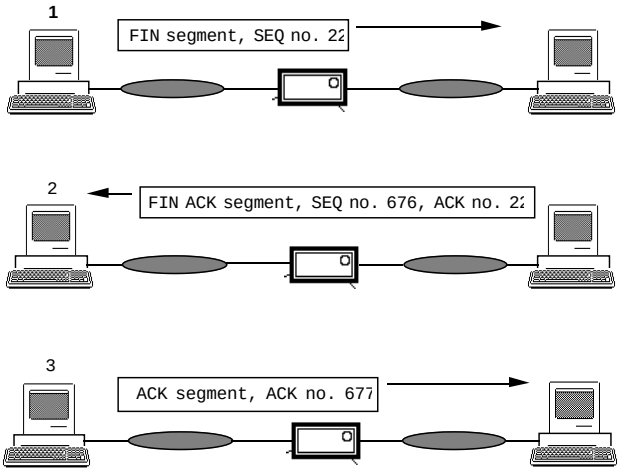

FIGURE 12.21 Graceful Close in TCP

Datagram Transport Protocols—CLTP and UDP

Both OSI and TCP/IP support connectionless (datagram) operation at the transport layer as an alternative to connections for upper-layer protocols that do not need the reliability and other characteristics of a transport connection. The service model is simple unconfirmed best-effort delivery (see Figure 12.22). Formally, the OSI connectionless transport service (ISO/IEC 8072: 1993) is supported by the OSI connectionless transport protocol (ISO/IEC 8602: 1987). Like TCP/IP's user datagram protocol (UDP; RFC 768), the primary purpose of which is to differentiate user-level processes identified by the port number, the primary purpose of the OSI connectionless transport protocol is to differentiate transport service users identified by the transport service access point identifier of the transport service access point address.

The differences between the two are unremarkable (see Figure 12.23). OSI connectionless transport supports variable-length transport service access point addresses, whereas the user datagram protocol supports 16-bit port numbers. Both provide user data-integrity checks by means of a 16-bit checksum; in both protocols, if the checksum verification fails, the packet is dropped with no indication to the upper layer process (transport service user).

The user datagram protocol is used to support a number of widely used applications in TCP/IP networks: Sun's Network File System/Remote Procedure Call, the Domain Name System, the Simple Network Management Protocol, even a routing protocol (the Routing Information

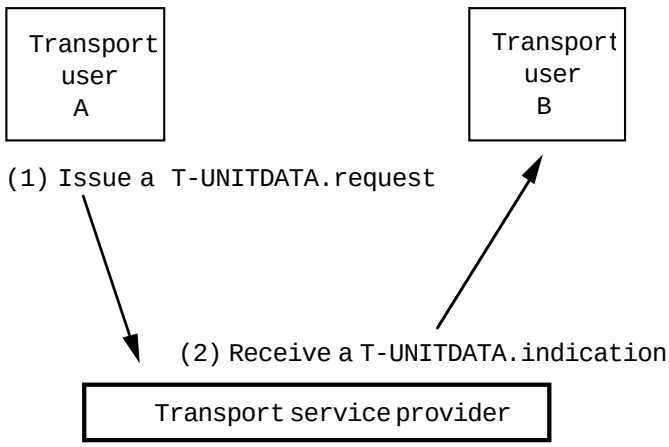

FIGURE 12.22 Connectionless Transport Service Primitives

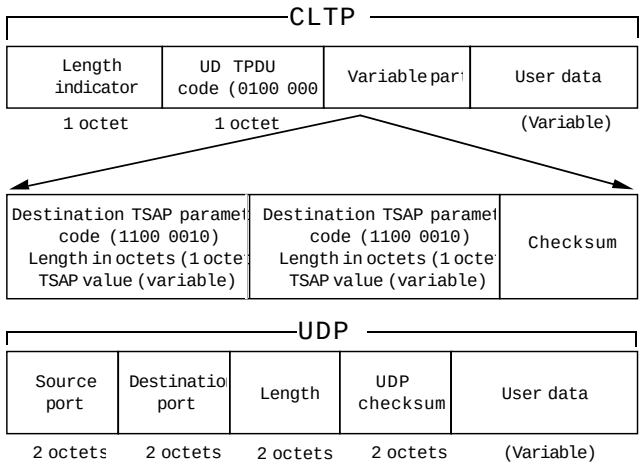

FIGURE 12.23 CLTP and UDP Formats

Protocol; see Chapter 14). The OSI connectionless transport protocol is ostensibly used by OSI connectionless upper layers, but to date, no application service elements have been developed to use these. Sun's Network File System is run over the OSI connectionless transport protocol today, and in dual-stack environments the Simple Network Management Protocol may be operated over it to manage CLNP-based networks in those configurations in which the managed agent does not support the User Datagram Protocol—i.e., those configurations in which an OSI-only host or router is present in an otherwise dual-stack topology (RFC 1418). (Network operators may also find it convenient to use the Simple Network Management Protocol over the OSI connectionless transport protocol in certain network diagnostic/debugging modes, in which it is useful to have management information traverse the same logical topology as data packets.)

Socket Interfaces to Datagram Transport Services

In ARGO 1.0 and RENO UNIX, the user datagram protocol and the OSI connectionless transport protocol are accessed via sockets (type *sock-dgram*). The Internet address family is used for UDP, and the ISO address family for CLTP. Both UDP and CLTP sockets support a best-effort datagram service via the *sendto* and *recvfrom* system calls. (In certain applications, the *connect()* call can be used to "fix" the destination address, and subsequent packets can be sent/received using the *recv()*, *read()*, *send()*, or *write()* system calls.)

Conclusion

The transport layer plays the same critical role in both the OSI and TCP/IP architectures: it defines the concept of "host" or "end system," in which applications live, and distinguishes these end-user hosts from systems that are concerned only with the "intermediate" functions (routing, relaying, switching, and transmission) of networking. As it is typically deployed at a real boundary between facilities belonging to end users and facilities belonging to the network, transport is the host's opportunity to ensure that its applications get the data pipe that *they* want, regardless of what the *network* is prepared to provide. The OSI transport protocol fudges this a bit by defining classes 0 through 3 in such a way that the end-to-end reliability seen by applications in fact depends very much on what the network is able to provide; but class 4, and TCP, support a genuinely network-independent transport service, which can be provided as reliably over connectionless internets as over connection-oriented networks. For applications that do not require the reliability of a transport connection, OSI and TCP/IP provide connectionless (transport datagram) alternatives as separate protocols.

Much has been written about the differences between TCP and the OSI class-4 transport protocol in an effort to prove that one or the other is "better." There is ample fuel for this debate: TCP is octet-sequenced, TP4 is packet-sequenced; TCP has graceful close, TP4 does not; TCP's port numbers are fixed at 16 bits, TP4's are variable-length. The list is long, but it contains no "killer argument" in favor of either protocol. Readers interested in pursuing this question will find a convincing argument for the essential irrelevance of this "which is better?" debate in Chapin (1990).