CHAPTER 20

TRANSPORT PROTOCOLS

The foregoing observations should make us reconsider the widely held view that birds live only in the present. In fact, birds are aware of more than immediately present stimuli; they remember the past and anticipate the future.

—*The Minds of Birds*, Alexander Skutch

KEY POINTS

- The transport protocol provides an end-to-end data transfer service that shields upper-layer protocols from the details of the intervening network or networks. A transport protocol can be either connection oriented, such as TCP, or connectionless, such as UDP.
- If the underlying network or internetwork service is unreliable, such as with the use of IP, then a reliable connection-oriented transport protocol becomes quite complex. The basic cause of this complexity is the need to deal with the relatively large and variable delays experienced between end systems. These large, variable delays complicate the flow control and error control techniques.
- TCP uses a credit-based flow control technique that is somewhat different from the sliding-window flow control found in X.25 and HDLC. In essence, TCP separates acknowledgments from the management of the size of the sliding window.
- Although the TCP credit-based mechanism was designed for end-to-end flow control, it is also used to assist in internetwork congestion control. When a TCP entity detects the presence of congestion in the Internet, it reduces the flow of data onto the Internet until it detects an easing in congestion.

In a protocol architecture, the transport protocol sits above a network or internetwork layer, which provides network-related services, and just below application and other upper-layer protocols. The transport protocol provides services to transport service (TS) users, such as FTP, SMTP, and TELNET. The local transport entity communicates with some remote transport entity, using the services of some lower layer, such as the Internet Protocol. The general service provided by a transport protocol is the end-to-end transport of data in a way that shields the TS user from the details of the underlying communications systems.

We begin this chapter by examining the protocol mechanisms required to provide these services. We find that most of the complexity relates to reliable connection-oriented services. As might be expected, the less the network service provides, the more the transport protocol must do. The remainder of the chapter looks at two widely used transport protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

Refer to Figure 2.5 to see the position within the TCP/IP suite of the protocols discussed in this chapter.

20.1 CONNECTION-ORIENTED TRANSPORT PROTOCOL MECHANISMS

Two basic types of transport service are possible: connection oriented and connectionless or datagram service. A connection-oriented service provides for the establishment, maintenance, and termination of a logical connection between TS users. This has, so far, been the most common type of protocol service available and has a wide variety of applications. The connection-oriented service generally implies that the service is reliable. This section looks at the transport protocol mechanisms needed to support the connection-oriented service.

A full-feature connection-oriented transport protocol, such as TCP, is very complex. For purposes of clarity we present the transport protocol mechanisms in an evolutionary fashion. We begin with a network service that makes life easy for the transport protocol, by guaranteeing the delivery of all transport data units in order and defining the required mechanisms. Then we will look at the transport protocol mechanisms required to cope with an unreliable network service. All of this discussion applies in general to transport-level protocols. In Section 20.2, we apply the concepts developed in this section to describe TCP.

Reliable Sequencing Network Service

Let us assume that the network service accepts messages of arbitrary length and, with virtually 100% reliability, delivers them in sequence to the destination. Examples of such networks are as follows:

- A highly reliable packet-switching network with an X.25 interface
- A frame relay network using the LAPF control protocol
- An IEEE 802.3 LAN using the connection-oriented LLC service

In all of these cases, the transport protocol is used as an end-to-end protocol between two systems attached to the same network, rather than across an internet.

The assumption of a reliable sequencing networking service allows the use of a quite simple transport protocol. Four issues need to be addressed:

- Addressing
- Multiplexing
- Flow control
- Connection establishment/termination

Addressing The issue concerned with addressing is simply this: A user of a given transport entity wishes either to establish a connection with or make a data transfer to a user of some other transport entity using the same transport protocol. The target user needs to be specified by all of the following:

- User identification
- Transport entity identification

- Host address
- Network number

The transport protocol must be able to derive the information listed above from the TS user address. Typically, the user address is specified as (Host, Port). The **Port** variable represents a particular TS user at the specified host. Generally, there will be a single transport entity at each host, so a transport entity identification is not needed. If more than one transport entity is present, there is usually only one of each type. In this latter case, the address should include a designation of the type of transport protocol (e.g., TCP, UDP). In the case of a single network, **Host** identifies an attached network device. In the case of an internet, *Host* is a global internet address. In TCP, the combination of port and host is referred to as a **socket**.

Because routing is not a concern of the transport layer, it simply passes the *Host* portion of the address down to the network service. *Port* is included in a transport header, to be used at the destination by the destination transport protocol entity.

One question remains to be addressed: How does the initiating TS user know the address of the destination TS user? Two static and two dynamic strategies suggest themselves:

1. The TS user knows the address it wishes to use ahead of time. This is basically a system configuration function. For example, a process may be running that is only of concern to a limited number of TS users, such as a process that collects statistics on performance. From time to time, a central network management routine connects to the process to obtain the statistics. These processes generally are not, and should not be, well known and accessible to all.
2. Some commonly used services are assigned "well-known addresses." Examples include the server side of FTP, SMTP, and some other standard protocols.
3. A name server is provided. The TS user requests a service by some generic or global name. The request is sent to the name server, which does a directory lookup and returns an address. The transport entity then proceeds with the connection. This service is useful for commonly used applications that change location from time to time. For example, a data entry process may be moved from one host to another on a local network to balance load.
4. In some cases, the target user is to be a process that is spawned at request time. The initiating user can send a process request to a well-known address. The user at that address is a privileged system process that will spawn the new process and return an address. For example, a programmer has developed a private application (e.g., a simulation program) that will execute on a remote server but be invoked from a local workstation. A request can be issued to a remote job-management process that spawns the simulation process.

Multiplexing Multiplexing was discussed in general terms in Section 18.1. With respect to the interface between the transport protocol and higher-level protocols, the transport protocol performs a multiplexing/demultiplexing function. That is, multiple users employ the same transport protocol and are distinguished by port numbers or service access points.

The transport entity may also perform a multiplexing function with respect to the network services that it uses. Recall that we defined upward multiplexing as the multiplexing of multiple connections on a single lower-level connection, and downward multiplexing as the splitting of a single connection among multiple lower-level connections (Section 18.1).

Consider, for example, a transport entity making use of an X.25 service. Why should the transport entity employ upward multiplexing? There are, after all, 4095 virtual circuits available. In the typical case, this is more than enough to handle all active TS users. However, most X.25 networks base part of their charge on virtual circuit connect time, because each virtual circuit consumes some node buffer resources. Thus, if a single virtual circuit provides sufficient throughput for multiple TS users, upward multiplexing is indicated.

On the other hand, downward multiplexing or splitting might be used to improve throughput. For example, each X.25 virtual circuit is restricted to a 3-bit or 7-bit sequence number. A larger sequence space might be needed for high-speed, high-delay networks. Of course, throughput can only be increased so far. If there is a single host-node link over which all virtual circuits are multiplexed, the throughput of a transport connection cannot exceed the data rate of that link.

Flow Control Whereas flow control is a relatively simple mechanism at the link layer, it is a rather complex mechanism at the transport layer, for two main reasons:

- The transmission delay between transport entities is generally long compared to actual transmission time. This means that there is a considerable delay in the communication of flow control information.
- Because the transport layer operates over a network or internet, the amount of the transmission delay may be highly variable. This makes it difficult to effectively use a timeout mechanism for retransmission of lost data.

In general, there are two reasons why one transport entity would want to restrain the rate of segment[1] transmission over a connection from another transport entity:

- The user of the receiving transport entity cannot keep up with the flow of data.
- The receiving transport entity itself cannot keep up with the flow of segments.

How do such problems manifest themselves? Presumably a transport entity has a certain amount of buffer space. Incoming segments are added to the buffer. Each buffered segment is processed (i.e., the transport header is examined) and the data are sent to the TS user. Either of the two problems just mentioned will cause the buffer to fill up. Thus, the transport entity needs to take steps to stop or slow the flow of segments to prevent buffer overflow. This requirement is difficult to fulfill because of the annoying time gap between sender and receiver. We return to this point subsequently. First, we present four ways of coping with the flow control requirement. The receiving transport entity can

[1]Recall from Chapter 2 that the blocks of data (protocol data units) exchanged by TCP entities are referred to as TCP segments.

1. Do nothing.
2. Refuse to accept further segments from the network service.
3. Use a fixed sliding-window protocol.
4. Use a credit scheme.

Alternative 1 means that the segments that overflow the buffer are discarded. The sending transport entity, failing to get an acknowledgment, will retransmit. This is a shame, because the advantage of a reliable network is that one never has to retransmit. Furthermore, the effect of this maneuver is to exacerbate the problem. The sender has increased its output to include new segments plus retransmitted old segments.

The second alternative is a backpressure mechanism that relies on the network service to do the work. When a buffer of a transport entity is full, it refuses additional data from the network service. This triggers flow control procedures within the network that throttle the network service at the sending end. This service, in turn, refuses additional segments from its transport entity. It should be clear that this mechanism is clumsy and coarse grained. For example, if multiple transport connections are multiplexed on a single network connection (virtual circuit), flow control is exercised only on the aggregate of all transport connections.

The third alternative is already familiar to you from our discussions of link layer protocols in Chapter 7. The key ingredients, recall, are

- The use of sequence numbers on data units
- The use of a window of fixed size
- The use of acknowledgments to advance the window

With a reliable network service, the sliding-window technique would work quite well. For example, consider a protocol with a window size of 7. When the sender receives an acknowledgment to a particular segment, it is automatically authorized to send the succeeding seven segments (of course, some may already have been sent). When the receiver's buffer capacity gets down to seven segments, it can withhold acknowledgment of incoming segments to avoid overflow. The sending transport entity can send at most seven additional segments and then must stop. Because the underlying network service is reliable, the sender will not time out and retransmit. Thus, at some point, a sending transport entity may have a number of segments outstanding for which no acknowledgment has been received. Because we are dealing with a reliable network, the sending transport entity can assume that the segments will get through and that the lack of acknowledgment is a flow control tactic. This tactic would not work well in an unreliable network, because the sending transport entity would not know whether the lack of acknowledgment is due to flow control or a lost segment.

The fourth alternative, a credit scheme, provides the receiver with a greater degree of control over data flow. Although it is not strictly necessary with a reliable network service, a credit scheme should result in a smoother traffic flow. Further, it is a more effective scheme with an unreliable network service, as we shall see.

The credit scheme decouples acknowledgment from flow control. In fixed sliding-window protocols, such as X.25 and HDLC, the two are synonymous. In a credit scheme, a segment may be acknowledged without granting new credit, and vice versa. For the credit scheme, each individual octet of data that is transmitted is considered to have a unique sequence number. In addition to data, each transmitted segment includes in its header three fields related to flow control: **sequence number** (*SN*), **acknowledgment number** (*AN*), and **window** (*W*). When a transport entity sends a segment, it includes the sequence number of the first octet in the segment data field. Implicitly, the remaining data octets are numbered sequentially following the first data octet. A transport entity acknowledges an incoming segment with a return segment that includes $(AN = i, W = j)$, with the following interpretation:

- All octets through sequence number $SN = i - 1$ are acknowledged; the next expected octet has sequence number i.
- Permission is granted to send an additional window of $W = j$ octets of data; that is, the j octets corresponding to sequence numbers i through $i + j - 1$.

Figure 20.1 illustrates the mechanism (compare Figure 7.4). For simplicity, we show data flow in one direction only and assume that 200 octets of data are sent in each segment. Initially, through the connection establishment process, the sending and receiving sequence numbers are synchronized and A is granted an initial credit allocation of 1400 octets, beginning with octet number 1001. The first segment transmitted by A contains data octets numbered 1001 through 1200. After sending 600 octets in three segments, A has shrunk its window to a size of 800 octets (numbers 1601 through 2400). After B receives these three segments, 600 octets out of its original 1400 octets of credit are accounted for, and 800 octets of credit are outstanding. Now suppose that, at this point, B is capable of absorbing 1000 octets of incoming data on this connection. Accordingly, B acknowledges receipt of all octets through 1600 and issues a credit of 1000 octets. This means that A can send octets 1601

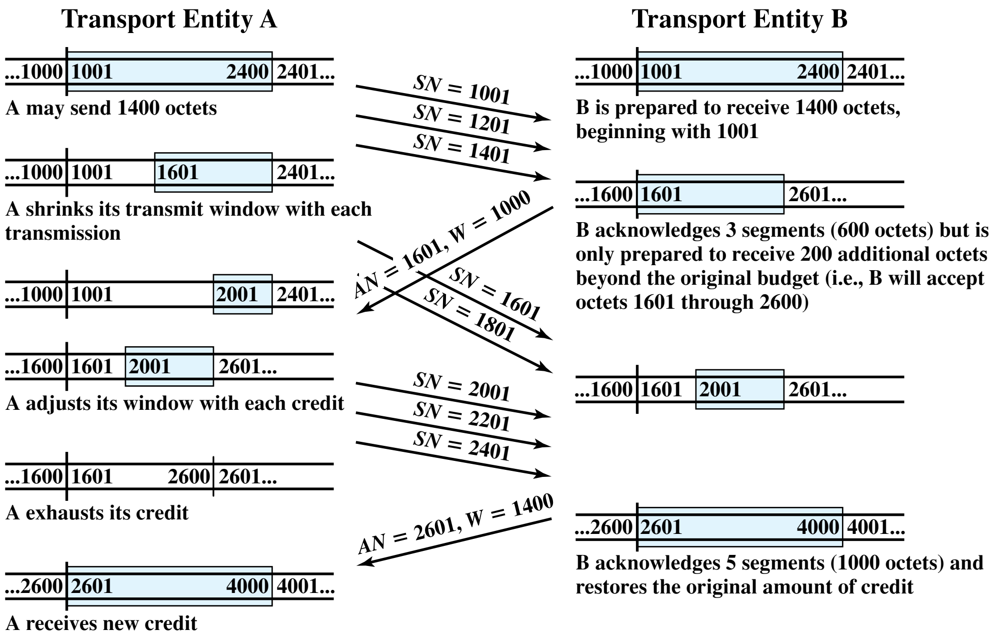


Figure 20.1 Example of TCP Credit Allocation Mechanism

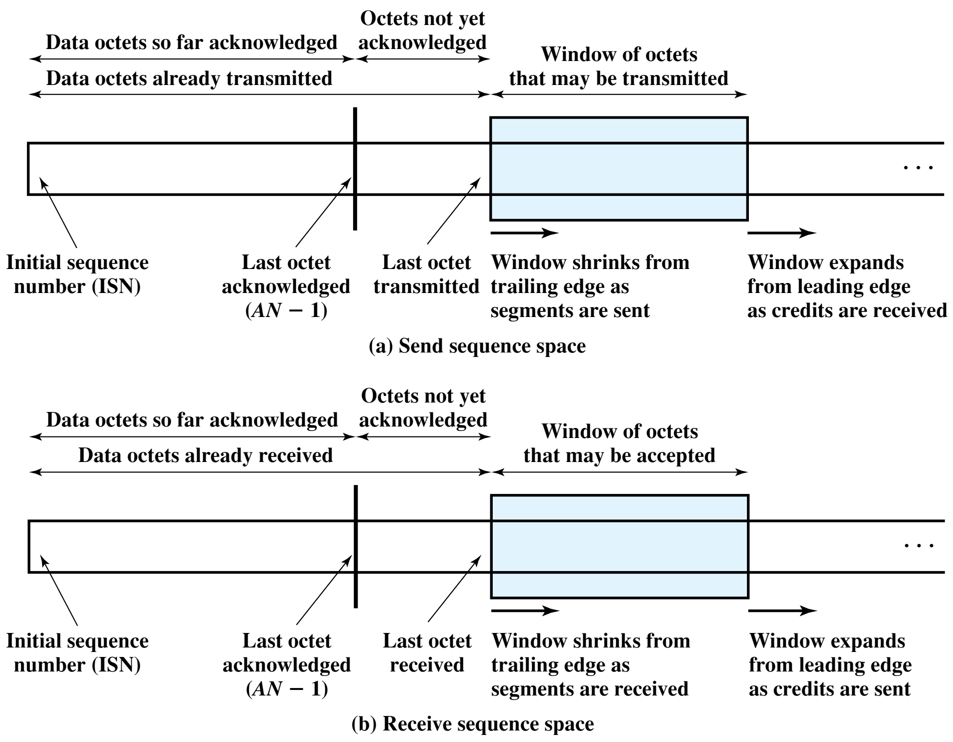

Data octets so far acknowledged
Octets not yet acknowledged
Data octets already transmitted
Window of octets that may be transmitted
Initial sequence number (ISN)
Last octet acknowledged ($AN - 1$)
Last octet transmitted
Window shrinks from trailing edge as segments are sent
Window expands from leading edge as credits are received
(a) Send sequence space
Data octets so far acknowledged
Octets not yet acknowledged
Data octets already received
Window of octets that may be accepted
Initial sequence number (ISN)
Last octet acknowledged ($AN - 1$)
Last octet received
Window shrinks from trailing edge as segments are received
Window expands from leading edge as credits are sent
(b) Receive sequence space

Figure 20.2 Sending and Receiving Flow Control Perspectives

through 2600 (5 segments). However, by the time that B's message has arrived at A, A has already sent two segments, containing octets 1601 through 2000 (which was permissible under the initial allocation). Thus, A's remaining credit upon receipt of B's credit allocation is only 600 octets (3 segments). As the exchange proceeds, A advances the trailing edge of its window each time that it transmits and advances the leading edge only when it is granted credit.

Figure 20.2 shows the view of this mechanism from the sending and receiving sides (compare Figure 7.3). Typically, both sides take both views because data may be exchanged in both directions. Note that the receiver is not required to immediately acknowledge incoming segments but may wait and issue a cumulative acknowledgment for a number of segments.

The receiver needs to adopt some policy concerning the amount of data it permits the sender to transmit. The conservative approach is to only allow new segments up to the limit of available buffer space. If this policy were in effect in Figure 20.1, the first credit message implies that B has 1000 available octets in its buffer, and the second message that B has 1400 available octets.

A conservative flow control scheme may limit the throughput of the transport connection in long-delay situations. The receiver could potentially increase throughput by optimistically granting credit for space it does not have. For example, if a receiver's buffer is full but it anticipates that it can release space for 1000 octets within a round-trip propagation time, it could immediately send a credit of 1000. If the receiver can keep up with the sender, this scheme may increase throughput and can do no harm. If the sender is faster than the receiver, however, some segments may be discarded, necessitating a retransmission. Because retransmissions are not otherwise necessary with a reliable network service (in the absence of internet congestion), an optimistic flow control scheme will complicate the protocol.

Connection Establishment and Termination Even with a reliable network service, there is a need for connection establishment and termination procedures to support connection-oriented service. Connection establishment serves three main purposes:

- It allows each end to assure that the other exists.
- It allows exchange or negotiation of optional parameters (e.g., maximum segment size, maximum window size, quality of service).
- It triggers allocation of transport entity resources (e.g., buffer space, entry in connection table).

Connection establishment is by mutual agreement and can be accomplished by a simple set of user commands and control segments, as shown in the state diagram of Figure 20.3. To begin, a TS user is in an CLOSED state (i.e., it has no open transport connection). The TS user can signal to the local TCP entity that it will passively wait for

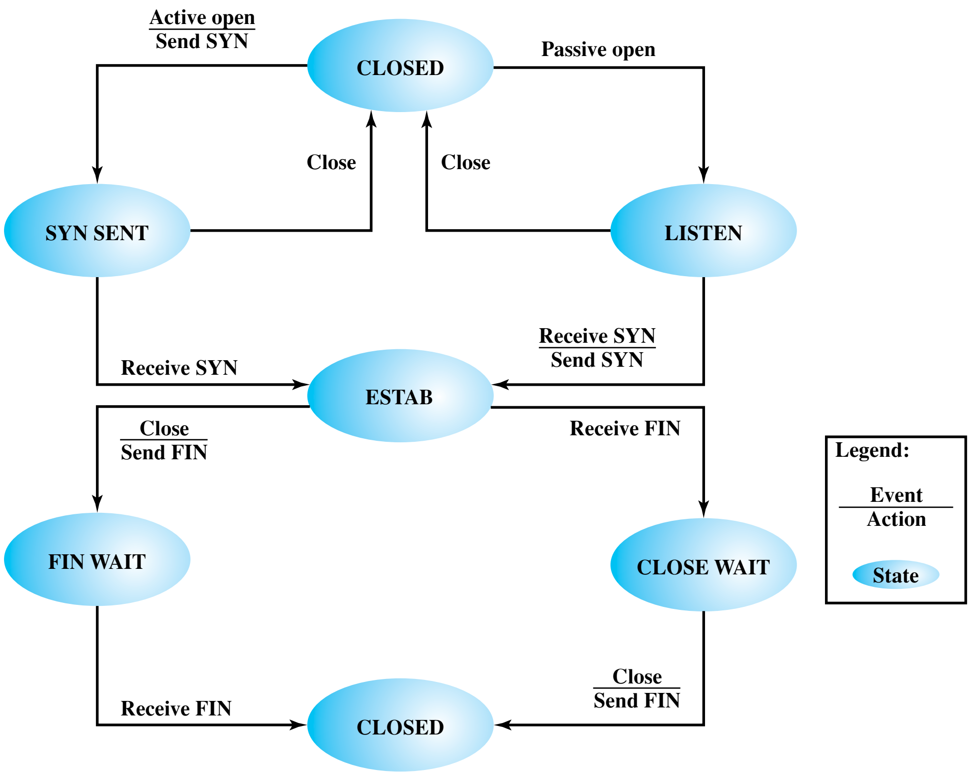


Figure 20.3 Simple Connection State Diagram

a request with a Passive Open command. A server program, such as time-sharing or a file transfer application, might do this. The TS user may change its mind by sending a Close command. After the Passive Open command is issued, the transport entity creates a connection object of some sort (i.e., a table entry) that is in the LISTEN state.

From the CLOSED state, a TS user may open a connection by issuing an Active Open command, which instructs the transport entity to attempt connection establishment with a designated remote TS user, which triggers the transport entity to send a SYN (for synchronize) segment. This segment is carried to the receiving transport entity and interpreted as a request for connection to a particular port. If the destination transport entity is in the LISTEN state for that port, then a connection is established by the following actions by the receiving transport entity:

- Signal the local TS user that a connection is open.
- Send a SYN as confirmation to the remote transport entity.
- Put the connection object in an ESTAB (established) state.

When the responding SYN is received by the initiating transport entity, it too can move the connection to an ESTAB state. The connection is prematurely aborted if either TS user issues a Close command.

Figure 20.4 shows the robustness of this protocol. Either side can initiate a connection. Further, if both sides initiate the connection at about the same time, it is established without confusion. This is because the SYN segment functions both as a connection request and a connection acknowledgment.

The reader may ask what happens if a SYN comes in while the requested TS user is idle (not listening). Three courses may be followed:

- The transport entity can reject the request by sending a RST (reset) segment back to the other transport entity.
- The request can be queued until the local TS user issues a matching Open.
- The transport entity can interrupt or otherwise signal the local TS user to notify it of a pending request.

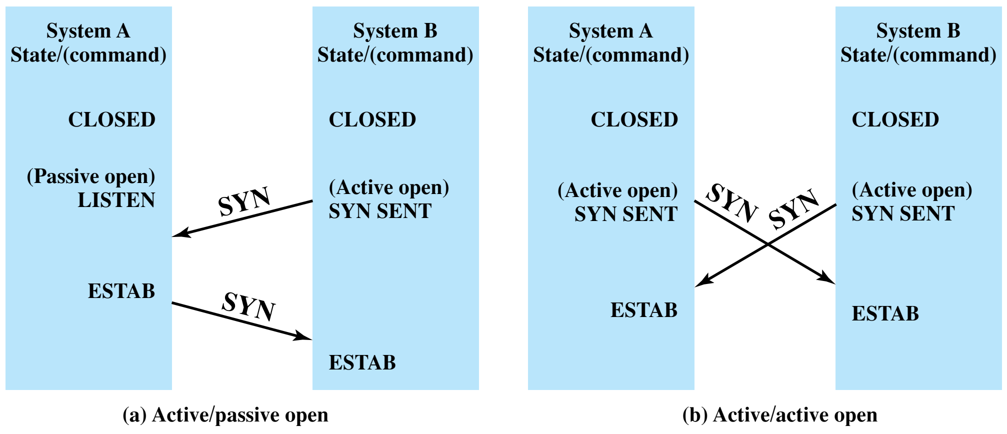


Figure 20.4 Connection Establishment Scenarios

Note that if the third mechanism is used, a Passive Open command is not strictly necessary but may be replaced by an Accept command, which is a signal from the user to the transport entity that it accepts the request for connection.

Connection termination is handled similarly. Either side, or both sides, may initiate a close. The connection is closed by mutual agreement. This strategy allows for either abrupt or graceful termination. With abrupt termination, data in transit may be lost; a graceful termination prevents either side from closing the connection until all data have been delivered. To achieve the latter, a connection in the FIN WAIT state must continue to accept data segments until a FIN (finish) segment is received.

Figure 20.3 defines the procedure for graceful termination. First, consider the side that initiates the termination procedure:

1. In response to a TS user's Close primitive, a transport entity sends a FIN segment to the other side of the connection, requesting termination.
2. Having sent the FIN, the transport entity places the connection in the FIN WAIT state. In this state, the transport entity must continue to accept data from the other side and deliver that data to its user.
3. When a FIN is received in response, the transport entity informs its user and closes the connection.

From the point of view of the side that does not initiate a termination,

1. When a FIN segment is received, the transport entity informs its user of the termination request and places the connection in the CLOSE WAIT state. In this state, the transport entity must continue to accept data from its user and transmit it in data segments to the other side.
2. When the user issues a Close primitive, the transport entity sends a responding FIN segment to the other side and closes the connection.

This procedure ensures that both sides have received all outstanding data and that both sides agree to connection termination before actual termination.

Unreliable Network Service

A more difficult case for a transport protocol is that of an unreliable network service. Examples of such networks are as follows:

- An internetwork using IP
- A frame relay network using only the LAPF core protocol
- An IEEE 802.3 LAN using the unacknowledged connectionless LLC service

The problem is not just that segments are occasionally lost, but that segments may arrive out of sequence due to variable transit delays. As we shall see, elaborate machinery is required to cope with these two interrelated network deficiencies. We shall also see that a discouraging pattern emerges. The combination of unreliability and nonsequencing creates problems with every mechanism we have discussed so far. Generally, the solution to each problem raises new problems. Although there are problems to be overcome for protocols at all levels, it seems that there are more difficulties with a reliable connection-oriented transport protocol than any other sort of protocol.

In the remainder of this section, unless otherwise noted, the mechanisms discussed are those used by TCP. Seven issues need to be addressed:

- Ordered delivery
- Retransmission strategy
- Duplicate detection
- Flow control
- Connection establishment
- Connection termination
- Failure recovery

Ordered Delivery With an unreliable network service, it is possible that segments, even if they are all delivered, may arrive out of order. The required solution to this problem is to number segments sequentially. We have seen that for data link control protocols, such as HDLC, and for X.25, each data unit (frame, packet) is numbered sequentially with each successive sequence number being one more than the previous sequence number. This scheme is used in some transport protocols, such as the ISO transport protocols. However, TCP uses a somewhat different scheme in which each data octet that is transmitted is implicitly numbered. Thus, the first segment may have a sequence number of 1. If that segment has 200 octets of data, then the second segment would have the sequence number 201, and so on. For simplicity in the discussions of this section, we will continue to assume that each successive segment's sequence number is 200 more than that of the previous segment; that is, each segment contains exactly 200 octets of data.

Retransmission Strategy Two events necessitate the retransmission of a segment. First, a segment may be damaged in transit but nevertheless arrive at its destination. If a checksum is included with the segment, the receiving transport entity can detect the error and discard the segment. The second contingency is that a segment fails to arrive. In either case, the sending transport entity does not know that the segment transmission was unsuccessful. To cover this contingency, a positive acknowledgment scheme is used: The receiver must acknowledge each successfully received segment by returning a segment containing an acknowledgment number. For efficiency, we do not require one acknowledgment per segment. Rather, a cumulative acknowledgment can be used, as we have seen many times in this book. Thus, the receiver may receive segments numbered 1, 201, and 401, but only send $AN = 601$ back. The sender must interpret $AN = 601$ to mean that the segment with $SN = 401$ and all previous segments have been successfully received.

If a segment does not arrive successfully, no acknowledgment will be issued and a retransmission is in order. To cope with this situation, there must be a timer associated with each segment as it is sent. If the timer expires before the segment is acknowledged, the sender must retransmit.

So the addition of a timer solves that problem. Next problem: At what value should the timer be set? Two strategies suggest themselves. A fixed timer value could be used, based on an understanding of the network's typical behavior. This suffers from an inability to respond to changing network conditions. If the value is too small, there will be many unnecessary retransmissions, wasting network capacity. If the value is too large, the protocol will be sluggish in responding to a lost segment. The timer should be set at a value a bit longer than the round trip time (send segment, receive ACK). Of course, this delay is variable even under constant network load. Worse, the statistics of the delay will vary with changing network conditions.

An adaptive scheme has its own problems. Suppose that the transport entity keeps track of the time taken to acknowledge data segments and sets its **retransmission timer** based on the average of the observed delays. This value cannot be trusted for three reasons:

- The peer transport entity may not acknowledge a segment immediately. Recall that we gave it the privilege of cumulative acknowledgments.
- If a segment has been retransmitted, the sender cannot know whether the received acknowledgment is a response to the initial transmission or the retransmission.
- Network conditions may change suddenly.

Each of these problems is a cause for some further tweaking of the transport algorithm, but the problem admits of no complete solution. There will always be some uncertainty concerning the best value for the retransmission timer. We return to this issue in Section 20.3.

Incidentally, the retransmission timer is only one of a number of timers needed for proper functioning of a transport protocol. These are listed in Table 20.1, together with a brief explanation.

Duplicate Detection If a segment is lost and then retransmitted, no confusion will result. If, however, one or more segments in sequence are successfully delivered, but the corresponding ACK is lost, then the sending transport entity will time out and one or more segments will be retransmitted. If these retransmitted segments arrive successfully, they will be duplicates of previously received segments. Thus, the receiver must be able to recognize duplicates. The fact that each segment carries a sequence number helps, but, nevertheless, duplicate detection and handling is not simple. There are two cases:

- A duplicate is received prior to the close of the connection.
- A duplicate is received after the close of the connection.

Table 20.1 Transport Protocol Timers

Retransmission timer	Retransmit an unacknowledged segment
2MSL (maximum segment lifetime) timer	Minimum time between closing one connection and opening another with the same destination address
Persist timer	Maximum time between ACK/CREDIT segments
Retransmit-SYN timer	Time between attempts to open a connection
Keepalive timer	Abort connection when no segments are received

The second case is discussed in the subsection on connection establishment. We deal with the first case here.

Notice that we say "a" duplicate rather than "the" duplicate. From the sender's point of view, the retransmitted segment is the duplicate. However, the retransmitted segment may arrive before the original segment, in which case the receiver views the original segment as the duplicate. In any case, two tactics are needed to cope with a duplicate received prior to the close of a connection:

- The receiver must assume that its acknowledgment was lost and therefore must acknowledge the duplicate. Consequently, the sender must not get confused if it receives multiple acknowledgments to the same segment.
- The sequence number space must be long enough so as not to "cycle" in less than the maximum possible segment lifetime (time it takes segment to transit network).

Figure 20.5 illustrates the reason for the latter requirement. In this example, the sequence space is of length 1600; that is, after $SN = 1600$, the sequence numbers cycle back and begin with $SN = 1$. For simplicity, we assume the receiving transport entity maintains a credit window size of 600. Suppose that A has transmitted data segments with $SN = 1$, 201, and 401. B has received the two segments with $SN = 201$ and $SN = 401$, but the segment with $SN = 1$ is delayed in transit. Thus, B does not send any acknowledgments. Eventually, A times out and retransmits segment $SN = 1$. When the duplicate segment $SN = 1$ arrives, B acknowledges 1, 201, and 401 with $AN = 601$. Meanwhile, A has timed out again and retransmits $SN = 201$, which B acknowledges with another $AN = 601$. Things now seem to have sorted themselves out and data transfer continues. When the sequence space is exhausted, A cycles back to $SN = 1$ and continues. Alas, the old segment $SN = 1$ makes a belated appearance and is accepted by B before the new segment $SN = 1$ arrives. When the new segment $SN = 1$ does arrive, it is treated as a duplicate and discarded.

It should be clear that the untimely emergence of the old segment would have caused no difficulty if the sequence numbers had not yet wrapped around. The larger the sequence number space (number of bits used to represent the sequence number), the longer the wraparound is avoided. How big must the sequence space be? This depends on, among other things, whether the network enforces a maximum packet lifetime, and the rate at which segments are being transmitted. Fortunately, each addition of a single bit to the sequence number field doubles the sequence space, so it is rather easy to select a safe size.

Flow Control The credit allocation flow control mechanism described earlier is quite robust in the face of an unreliable network service and requires little enhancement. As was mentioned, a segment containing $(AN = i, W = j)$ acknowledges all octets through number $i - 1$ and grants credit for an additional j octets beginning with octet i. The credit allocation mechanism is quite flexible. For example, suppose that the last octet of data received by B was octet number $i - 1$ and that the last segment issued by B was $(AN = i, W = j)$. Then

- To increase credit to an amount k $(k > j)$ when no additional data have arrived, B issues $(AN = i, W = k)$.
- To acknowledge an incoming segment containing m octets of data $(m < j)$ without granting additional credit, B issues $(AN = i + m, W = j - m)$.

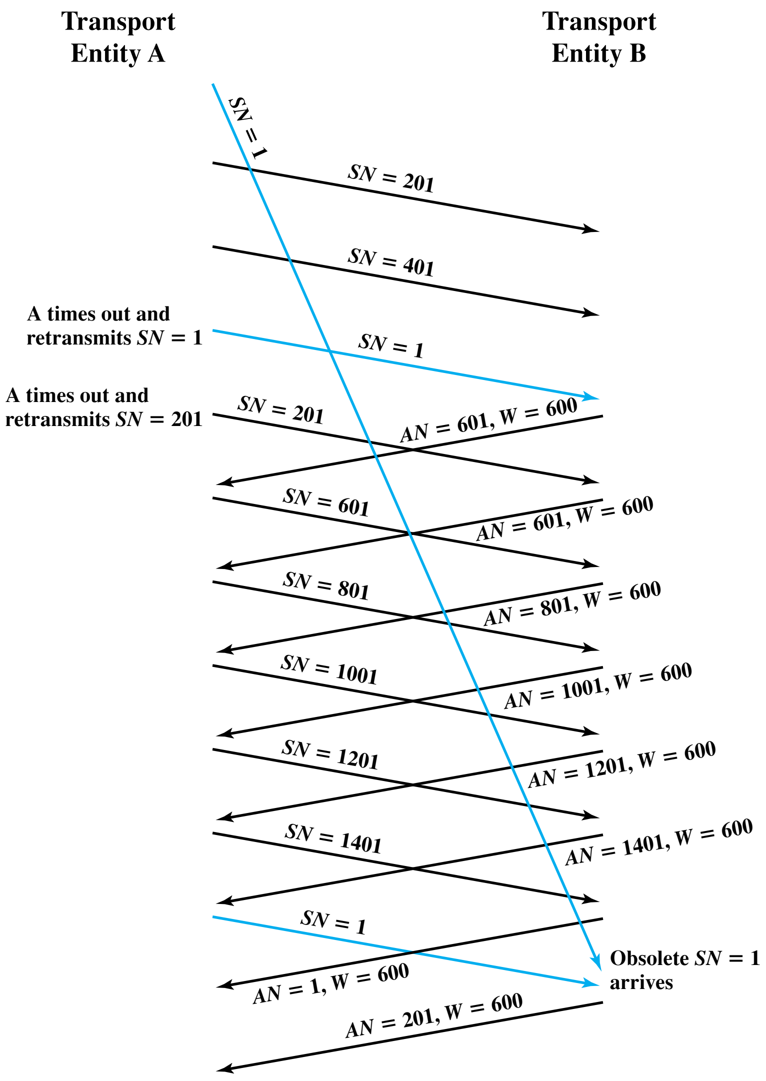


Figure 20.5 Example of Incorrect Duplicate Detection

If an ACK/CREDIT segment is lost, little harm is done. Future acknowledgments will resynchronize the protocol. Further, if no new acknowledgments are forthcoming, the sender times out and retransmits a data segment, which triggers a new acknowledgment. However, it is still possible for deadlock to occur. Consider a situation in which B sends $(AN = i, W = 0)$, temporarily closing the window. Subsequently, B sends $(AN = i, W = j)$, but this segment is lost. A is awaiting the opportunity to send data and B thinks that it has granted that opportunity. To overcome this problem, a **persist timer** can be used. This timer is reset with each outgoing segment (all segments contain the AN and W fields). If the timer ever expires, the protocol entity is required to send a segment, even if it duplicates a previous one. This breaks the deadlock and assures the other end that the protocol entity is still alive.

Connection Establishment As with other protocol mechanisms, connection establishment must take into account the unreliability of a network service. Recall that a connection establishment calls for the exchange of SYNs, a procedure sometimes referred to as a two-way handshake. Suppose that A issues a SYN to B. It expects to get a SYN back, confirming the connection. Two things can go wrong: A's SYN can be lost or B's answering SYN can be lost. Both cases can be handled by use of a **retransmit-SYN timer** (Table 20.1). After A issues a SYN, it will reissue the SYN when the timer expires.

This gives rise, potentially, to duplicate SYNs. If A's initial SYN was lost, there are no duplicates. If B's response was lost, then B may receive two SYNs from A. Further, if B's response was not lost, but simply delayed, A may get two responding SYNs. All of this means that A and B must simply ignore duplicate SYNs once a connection is established.

There are other problems to contend with. Just as a delayed SYN or lost response can give rise to a duplicate SYN, a delayed data segment or lost acknowledgment can give rise to duplicate data segments, as we have seen in Figure 20.5. Such a delayed or duplicated data segment can interfere with data transfer, as illustrated in Figure 20.6.

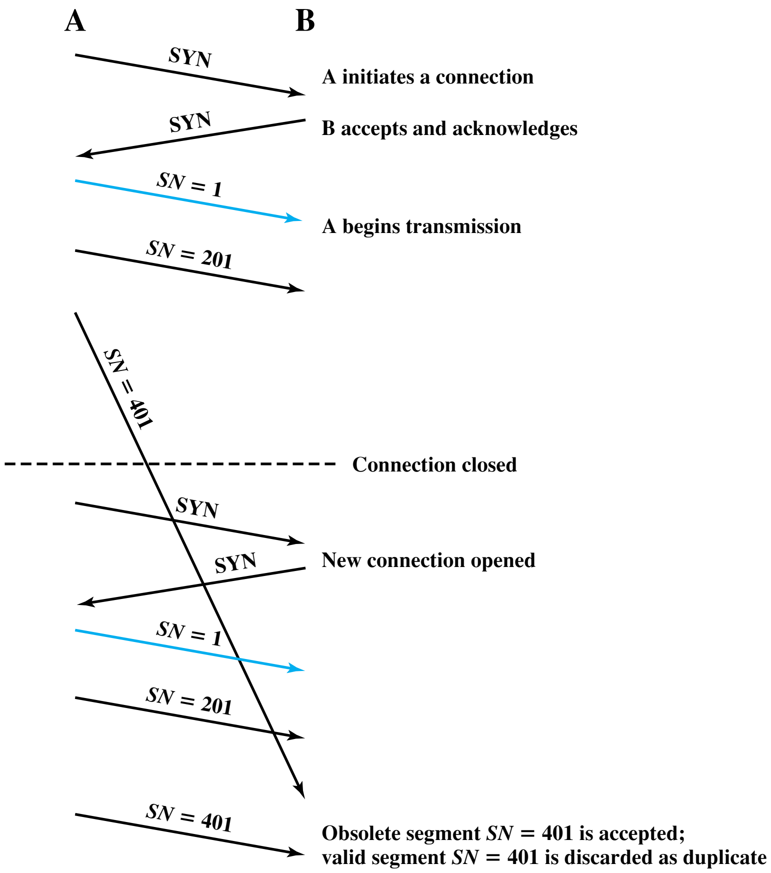


Figure 20.6 The Two-Way Handshake: Problem with Obsolete Data Segment

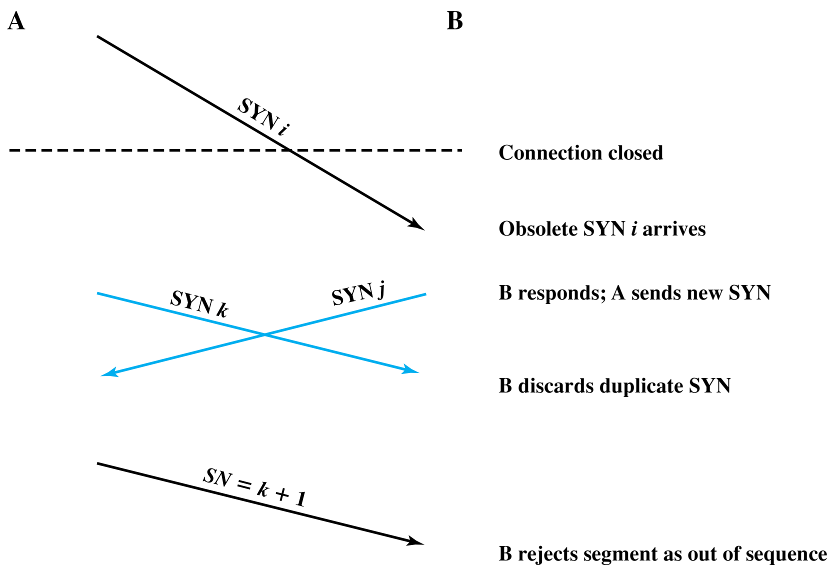


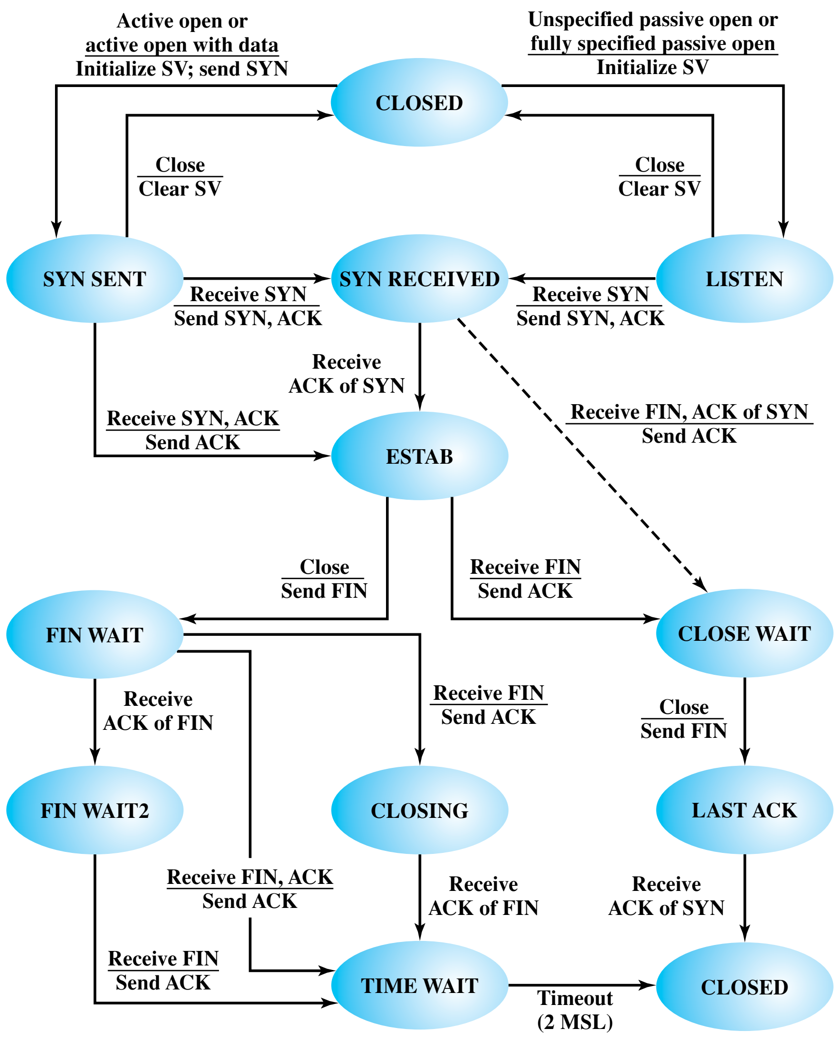
Figure 20.7 Two-Way Handshake: Problem with Obsolete SYN Segments

Assume that with each new connection, each transport protocol entity begins numbering its data segments with sequence number 1. In the figure, a duplicate copy of segment $SN = 401$ from an old connection arrives during the lifetime of a new connection and is delivered to B before delivery of the legitimate data segment $SN = 401$. One way of attacking this problem is to start each new connection with a different sequence number that is far removed from the last sequence number of the most recent connection. For this purpose, the connection request is of the form SYN $i + 1$, where i is the sequence number of the first data segment that will be sent on this connection.

Now consider that a duplicate SYN i may survive past the termination of the connection. Figure 20.7 depicts the problem that may arise. An old SYN i arrives at B after the connection is terminated. B assumes that this is a fresh request and responds with SYN j, meaning that B accepts the connection request and will begin transmitting with $SN = j + 1$. Meanwhile, A has decided to open a new connection with B and sends SYN k. B discards this as a duplicate. Now both sides have transmitted and subsequently received a SYN segment, and therefore think that a valid connection exists. However, when A initiates data transfer with a segment numbered $k + 1$. B rejects the segment as being out of sequence.

The way out of this problem is for each side to acknowledge explicitly the other's SYN and sequence number. The procedure is known as a **three-way handshake**. The revised connection state diagram, which is the one employed by TCP, is shown in the upper part of Figure 20.8. A new state (SYN RECEIVED) is added. In this state, the transport entity hesitates during connection opening to assure that the SYN segments sent by the two sides have both been acknowledged before the connection is declared established. In addition to the new state, there is a control segment (RST) to reset the other side when a duplicate SYN is detected.

Figure 20.9 illustrates typical three-way handshake operations. In Figure 20.9a, transport entity A initiates the connection, with a SYN including the sending



SV = state vector
MSL = maximum segment lifetime

Figure 20.8 TCP Entity State Diagram

sequence number, *i*. The value *i* is referred to as the initial sequence number (ISN) and is associated with the SYN; the first data octet to be transmitted will have sequence number $i + 1$. The responding SYN acknowledges the ISN with $(AN = i + 1)$ and includes its ISN. A acknowledges B's SYN/ACK in its first data segment, which begins with sequence number $i + 1$. Figure 20.9b shows a situation in which an old SYN *i* arrives at B after the close of the relevant connection. B assumes that this is a fresh request and responds with SYN *j*, $AN = i + 1$. When A receives this message, it realizes that it has not requested a connection and therefore sends an RST, $AN = j$. Note that the AN $= j$ portion of the RST message is essential so that an old duplicate RST does not abort a legitimate connection establishment. Figure 20.9c shows a case in which an old SYN/ACK arrives in the middle of

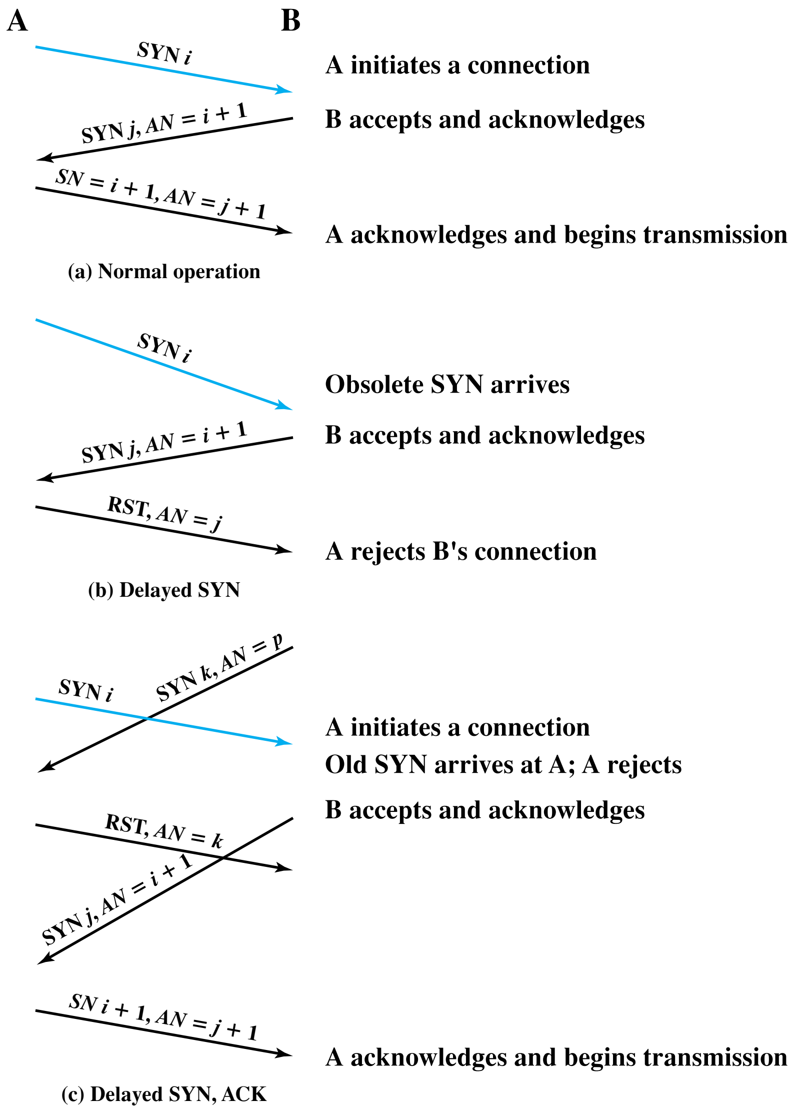


Figure 20.9 Examples of Three-Way Handshake

a new connection establishment. Because of the use of sequence numbers in the acknowledgments, this event causes no mischief.

For simplicity, the upper part of Figure 20.8 does not include transitions in which RST is sent. The basic rule is as follows: Send an RST if the connection state is not yet OPEN and an invalid ACK (one that does not reference something that was sent) is received. The reader should try various combinations of events to see that this connection establishment procedure works in spite of any combination of old and lost segments.

Connection Termination The state diagram of Figure 20.3 defines the use of a simple two-way handshake for connection establishment, which was found to be unsatisfactory in the face of an unreliable network service. Similarly, the two-way handshake defined in that diagram for connection termination is inadequate for an unreliable network service. Misordering of segments could cause the following scenario. A transport entity in the CLOSE WAIT state sends its last data segment, followed by a FIN segment, but the FIN segment arrives at the other side before the last data segment. The receiving transport entity will accept that FIN, close the connection, and lose the last segment of data. To avoid this problem, a sequence number can be associated with the FIN, which can be assigned the next sequence number after the last octet of transmitted data. With this refinement, the receiving transport entity, upon receiving a FIN, will wait if necessary for the late-arriving data before closing the connection.

A more serious problem is the potential loss of segments and the potential presence of obsolete segments. Figure 20.8 shows that the termination procedure adopts a similar solution to that used for connection establishment. Each side must explicitly acknowledge the FIN of the other, using an ACK with the sequence number of the FIN to be acknowledged. For a graceful close, a transport entity requires the following:

- It must send a FIN i and receive $AN = i + 1$.
- It must receive a FIN j and send $AN = j + 1$.
- It must wait an interval equal to twice the maximum expected segment lifetime.

Failure Recovery When the system upon which a transport entity is running fails and subsequently restarts, the state information of all active connections is lost. The affected connections become *half open* because the side that did not fail does not yet realize the problem.

The still active side of a half-open connection can close the connection using a **keepalive timer**. This timer measures the time the transport machine will continue to await an acknowledgment (or other appropriate reply) of a transmitted segment after the segment has been retransmitted the maximum number of times. When the timer expires, the transport entity assumes that the other transport entity or the intervening network has failed, closes the connection, and signals an abnormal close to the TS user.

In the event that a transport entity fails and quickly restarts, half-open connections can be terminated more quickly by the use of the RST segment. The failed side returns an RST i to every segment i that it receives. When the RST i reaches the other side, it must be checked for validity based on the sequence number i, because the RST could be in response to an old segment. If the reset is valid, the transport entity performs an abnormal termination.

These measures clean up the situation at the transport level. The decision as to whether to reopen the connection is up to the TS users. The problem is one of synchronization. At the time of failure, there may have been one or more outstanding segments in either direction. The TS user on the side that did not fail knows how much data it has received, but the other user may not, if state information were lost. Thus, there is the danger that some user data will be lost or duplicated.

20.2 TCP

In this section we look at TCP (RFC 793), first at the service it provides to the TS user and then at the internal protocol details.

TCP Services

TCP is designed to provide reliable communication between pairs of processes (TCP users) across a variety of reliable and unreliable networks and internets. TCP provides two useful facilities for labeling data: push and urgent:

- **Data stream push:** Ordinarily, TCP decides when sufficient data have accumulated to form a segment for transmission. The TCP user can require TCP to transmit all outstanding data up to and including that labeled with a push flag. On the receiving end, TCP will deliver these data to the user in the same manner. A user might request this if it has come to a logical break in the data.
- **Urgent data signaling:** This provides a means of informing the destination TCP user that significant or "urgent" data is in the upcoming data stream. It is up to the destination user to determine appropriate action.

As with IP, the services provided by TCP are defined in terms of primitives and parameters. The services provided by TCP are considerably richer than those provided by IP, and hence the set of primitives and parameters is more complex. Table 20.2 lists TCP service request primitives, which are issued by a TCP user to TCP, and Table 20.3 lists TCP service response primitives, which are issued by TCP to a local TCP user.

Table 20.2 TCP Service Request Primitives

Primitive	Parameters	Description
Unspecified Passive Open	source-port, [timeout], [timeout-action], [precedence], [security-range]	Listen for connection attempt at specified security and precedence from any remote destination.
Fully Specified Passive Open	source-port, destination-port, destination-address, [timeout], [timeout-action], [precedence], [security-range]	Listen for connection attempt at specified security and precedence from specified destination.
Active Open	source-port, destination-port, destination-address, [timeout], [timeout-action], [precedence], [security]	Request connection at a particular security and precedence to a specified destination.
Active Open with Data	source-port, destination-port, destination-address, [timeout], [timeout-action], [precedence], [security], data, data-length, PUSH-flag, URGENT-flag	Request connection at a particular security and precedence to a specified destination and transmit data with the request.
Send	local-connection-name, data, data-length, PUSH-flag, URGENT-flag, [timeout], [timeout-action]	Transfer data across named connection.
Allocate	local-connection-name, data-length	Issue incremental allocation for receive data to TCP.
Close	local-connection-name	Close connection gracefully.
Abort	local-connection-name	Close connection abruptly.
Status	local-connection-name	Query connection status.

Note: Square brackets indicate optional parameters.

Table 20.3 TCP Service Response Primitives

Primitive	Parameters	Description
Open ID	local-connection-name, source-port, destination-port,* destination-address*	Informs TCP user of connection name assigned to pending connection requested in an Open primitive
Open Failure	local-connection-name	Reports failure of an Active Open request
Open Success	local-connection-name	Reports completion of pending Open request
Deliver	local-connection-name, data, data-length, URGENT-flag	Reports arrival of data
Closing	local-connection-name	Reports that remote TCP user has issued a Close and that all data sent by remote user has been delivered
Terminate	local-connection-name, description	Reports that the connection has been terminated; a description of the reason for termination is provided
Status Response	local-connection-name, source-port, source-address, destination-port, destination-address, connection-state, receive-window, send-window, amount-awaiting-ACK, amount-awaiting-receipt, urgent-state, precedence, security, timeout	Reports current status of connection
Error	local-connection-name, description	Reports service-request or internal error

* =Not used for Unspecified Passive Open.

Table 20.4 provides a brief definition of the parameters involved. The two passive open commands signal the TCP user's willingness to accept a connection request. The active open with data allows the user to begin transmitting data with the opening of the connection.

TCP Header Format

TCP uses only a single type of protocol data unit, called a TCP segment. The header is shown in Figure 20.10. Because one header must serve to perform all protocol mechanisms, it is rather large, with a minimum length of 20 octets. The fields are as follows:

- **Source Port (16 bits):** Source TCP user. Example values are Telnet = 23; TFTP = 69; HTTP = 80. A complete list is maintained at **http://www.iana.org/assignments/port-numbers**.
- **Destination Port (16 bits):** Destination TCP user.
- **Sequence Number (32 bits):** Sequence number of the first data octet in this segment except when the SYN flag is set. If SYN is set, this field contains the initial sequence number (ISN) and the first data octet in this segment has sequence number ISN + 1.

Table 20.4 TCP Service Parameters

Source Port	Local TCP user
Timeout	Longest delay allowed for data delivery before automatic connection termination or error report; user specified
Timeout-action	Indicates whether the connection is terminated or an error is reported to the TCP user in the event of a timeout
Precedence	Precedence level for a connection. Takes on values zero (lowest) through seven (highest); same parameter as defined for IP
Security-range	Allowed ranges in compartment, handling restrictions, transmission control codes, and security levels
Destination Port	Remote TCP user
Destination Address	Internet address of remote host
Security	Security information for a connection, including security level, compartment, handling restrictions, and transmission control code; same parameter as defined for IP
Data	Block of data sent by TCP user or delivered to a TCP user
Data Length	Length of block of data sent or delivered
PUSH flag	If set, indicates that the associated data are to be provided with the data stream push service
URGENT flag	If set, indicates that the associated data are to be provided with the urgent data signaling service
Local Connection Name	Identifier of a connection defined by a (local socket, remote socket) pair; provided by TCP
Description	Supplementary information in a Terminate or Error primitive
Source Address	Internet address of the local host
Connection State	State of referenced connection (CLOSED, ACTIVE OPEN, PASSIVE OPEN, ESTABLISHED, CLOSING)
Receive Window	Amount of data in octets the local TCP entity is willing to receive
Send Window	Amount of data in octets permitted to be sent to remote TCP entity
Amount Awaiting ACK	Amount of previously transmitted data awaiting acknowledgment
Amount Awaiting Receipt	Amount of data in octets buffered at local TCP entity pending receipt by local TCP user
Urgent State	Indicates to the receiving TCP user whether there are urgent data available or whether all urgent data, if any, have been delivered to the user

- **Acknowledgment Number (32 bits):** Contains the sequence number of the next data octet that the TCP entity expects to receive.
- **Data Offset (4 bits):** Number of 32-bit words in the header.
- **Reserved (4 bits):** Reserved for future use.
- **Flags (6 bits):** For each flag, if set to 1, the meaning is

  CWR: congestion window reduced.

  ECE: ECN-Echo; the CWR and ECE bits, defined in RFC 3168, are used for the explicit congestion notification function; a discussion of this function is beyond our scope.

  URG: urgent pointer field significant.

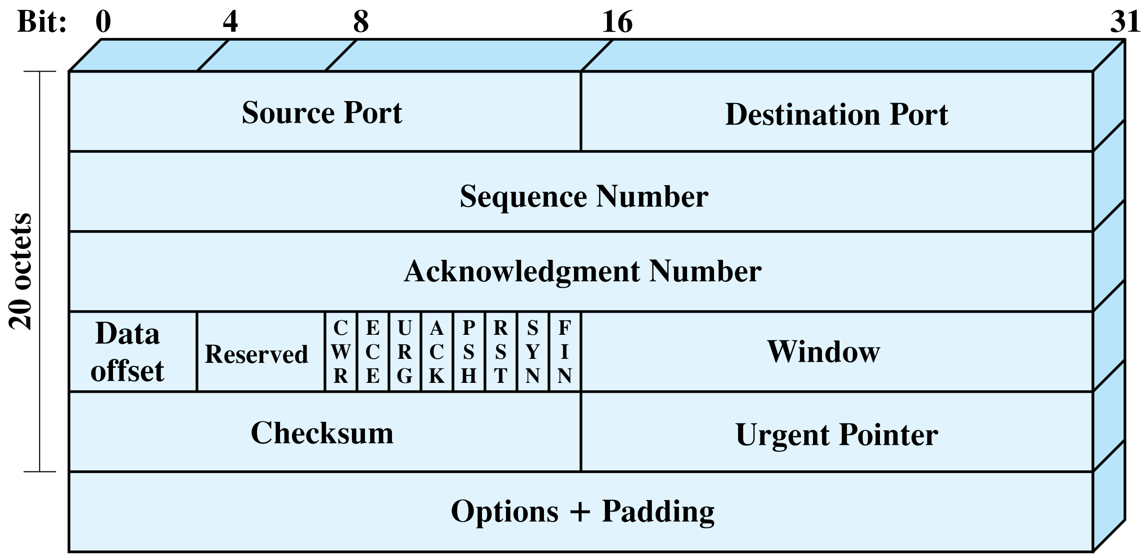


Figure 20.10 TCP Header

ACK: acknowledgment field significant.

PSH: push function.

RST: reset the connection.

SYN: synchronize the sequence numbers.

FIN: no more data from sender.

- **Window (16 bits):** Flow control credit allocation, in octets. Contains the number of data octets, beginning with the sequence number indicated in the acknowledgment field that the sender is willing to accept.
- **Checksum (16 bits):** The ones complement of the ones complement sum of all the 16-bit words in the segment plus a pseudoheader, described subsequently.[2]
- **Urgent Pointer (16 bits):** This value, when added to the segment sequence number, contains the sequence number of the last octet in a sequence of urgent data. This allows the receiver to know how much urgent data is coming.
- **Options (Variable):** An example is the option that specifies the maximum segment size that will be accepted.

The **Sequence Number** and **Acknowledgment Number** are bound to octets rather than to entire segments. For example, if a segment contains sequence number 1001 and includes 600 octets of data, the sequence number refers to the first octet in the data field; the next segment in logical order will have sequence number 1601. Thus, TCP is logically stream oriented: It accepts a stream of octets from the user, groups them into segments as it sees fit, and numbers each octet in the stream.

The **Checksum** field applies to the entire segment plus a pseudoheader prefixed to the header at the time of calculation (at both transmission and reception). The pseudoheader includes the following fields from the IP header: source and destination internet address and protocol, plus a segment length field. By including the pseudoheader, TCP protects itself from misdelivery by IP. That is, if IP delivers a packet to the wrong host, even if the packet contains no bit errors, the receiving TCP entity will detect the delivery error.

By comparing the TCP header to the TCP user interface defined in Tables 20.2 and 20.3, the reader may feel that some items are missing from the TCP header; that is indeed the case. TCP is intended to work with IP. Hence, some TCP user parameters are passed down by TCP to IP for inclusion in the IP header. The precedence parameter can be mapped into the DS (Differentiated Services) field, and the security parameter into the optional security field in the IP header.

It is worth observing that this TCP/IP linkage means that the required minimum overhead for every data unit is actually 40 octets.

TCP Mechanisms

We can group TCP mechanisms into the categories of connection establishment, data transfer, and connection termination.

[2]A discussion of this checksum is contained in Appendix K.

Connection Establishment Connection establishment in TCP always uses a three-way handshake. When the SYN flag is set, the segment is essentially a request for connection and functions as explained in Section 20.1. To initiate a connection, an entity sends a SYN, $SN = X$, where X is the initial sequence number. The receiver responds with SYN, $SN = Y$, $AN = X + 1$ by setting both the SYN and ACK flags. Note that the acknowledgment indicates that the receiver is now expecting to receive a segment beginning with data octet $X + 1$, acknowledging the SYN, which occupies $SN = X$. Finally, the initiator responds with $AN = Y + 1$. If the two sides issue crossing SYNs, no problem results: Both sides respond with SYN/ACKs (Figure 20.4).

A connection is uniquely determined by the source and destination sockets (host, port). Thus, at any one time, there can only be a single TCP connection between a unique pair of ports. However, a given port can support multiple connections, each with a different partner port.

Data Transfer Although data are transferred in segments over a transport connection, data transfer is viewed logically as consisting of a stream of octets. Hence every octet is numbered, modulo 2^{32}. Each segment contains the sequence number of the first octet in the data field. Flow control is exercised using a credit allocation scheme in which the credit is a number of octets rather than a number of segments, as explained in Section 20.1.

Data are buffered by the transport entity on both transmission and reception. TCP normally exercises its own discretion as to when to construct a segment for transmission and when to release received data to the user. The PUSH flag is used to force the data so far accumulated to be sent by the transmitter and passed on by the receiver. This serves an end-of-block function.

The user may specify a block of data as urgent. TCP will designate the end of that block with an urgent pointer and send it out in the ordinary data stream. The receiving user is alerted that urgent data are being received.

If, during data exchange, a segment arrives that is apparently not meant for the current connection, the RST flag is set on an outgoing segment. Examples of this situation are delayed duplicate SYNs and an acknowledgment of data not yet sent.

Connection Termination The normal means of terminating a connection is a graceful close. Each TCP user must issue a CLOSE primitive. The transport entity sets the FIN bit on the last segment that it sends out, which also contains the last of the data to be sent on this connection.

An abrupt termination occurs if the user issues an ABORT primitive. In this case, the entity abandons all attempts to send or receive data and discards data in its transmission and reception buffers. An RST segment is sent to the other side.

TCP Implementation Policy Options

The TCP standard provides a precise specification of the protocol to be used between TCP entities. However, certain aspects of the protocol admit several possible implementation options. Although two implementations that choose alternative options will be interoperable, there may be performance implications. The design areas for which options are specified are the following:

- Send policy
- Deliver policy
- Accept policy
- Retransmit policy
- Acknowledge policy

Send Policy In the absence of both pushed data and a closed transmission window (see Figure 20.2a), a sending TCP entity is free to transmit data at its own convenience, within its current credit allocation. As data are issued by the user, they are buffered in the transmit buffer. TCP may construct a segment for each batch of data provided by its user or it may wait until a certain amount of data accumulates before constructing and sending a segment. The actual policy will depend on performance considerations. If transmissions are infrequent and large, there is low overhead in terms of segment generation and processing. On the other hand, if transmissions are frequent and small, the system is providing quick response.

Deliver Policy In the absence of a Push, a receiving TCP entity is free to deliver data to the user at its own convenience. It may deliver data as each in-order segment is received, or it may buffer data from a number of segments in the receive buffer before delivery. The actual policy will depend on performance considerations. If deliveries are infrequent and large, the user is not receiving data as promptly as may be desirable. On the other hand, if deliveries are frequent and small, there may be unnecessary processing both in TCP and in the user software, as well as an unnecessary number of operating system interrupts.

Accept Policy When all data segments arrive in order over a TCP connection, TCP places the data in a receive buffer for delivery to the user. It is possible, however, for segments to arrive out of order. In this case, the receiving TCP entity has two options:

- **In-order:** Accept only segments that arrive in order; any segment that arrives out of order is discarded.
- **In-window:** Accept all segments that are within the receive window (see Figure 20.2b).

The in-order policy makes for a simple implementation but places a burden on the networking facility, as the sending TCP must time out and retransmit segments that were successfully received but discarded because of misordering. Furthermore, if a single segment is lost in transit, then all subsequent segments must be retransmitted once the sending TCP times out on the lost segment.

The in-window policy may reduce transmissions but requires a more complex acceptance test and a more sophisticated data storage scheme to buffer and keep track of data accepted out of order.

Retransmit Policy TCP maintains a queue of segments that have been sent but not yet acknowledged. The TCP specification states that TCP will retransmit a segment if it fails to receive an acknowledgment within a given time. A TCP implementation may employ one of three retransmission strategies:

- **First-only:** Maintain one retransmission timer for the entire queue. If an acknowledgment is received, remove the appropriate segment or segments from the queue and reset the timer. If the timer expires, retransmit the segment at the front of the queue and reset the timer.
- **Batch:** Maintain one retransmission timer for the entire queue. if an acknowledgment is received, remove the appropriate segment or segments from the queue and reset the timer. If the timer expires, retransmit all segments in the queue and reset the timer.
- **Individual:** Maintain one timer for each segment in the queue. If an acknowledgment is received, remove the appropriate segment or segments from the queue and destroy the corresponding timer or timers. If any timer expires, retransmit the corresponding segment individually and reset its timer.

The first-only policy is efficient in terms of traffic generated, because only lost segments (or segments whose ACK was lost) are retransmitted. Because the timer for the second segment in the queue is not set until the first segment is acknowledged, however, there can be considerable delays. The individual policy solves this problem at the expense of a more complex implementation. The batch policy also reduces the likelihood of long delays but may result in unnecessary retransmissions.

The actual effectiveness of the retransmit policy depends in part on the accept policy of the receiver. If the receiver is using an in-order accept policy, then it will discard segments received after a lost segment. This fits best with batch retransmission. If the receiver is using an in-window accept policy, then a first-only or individual retransmission policy is best. Of course, in a mixed network of computers, both accept policies may be in use.

Acknowledge Policy When a data segment arrives that is in sequence, the receiving TCP entity has two options concerning the timing of acknowledgment:

- **Immediate:** When data are accepted, immediately transmit an empty (no data) segment containing the appropriate acknowledgment number.
- **Cumulative:** When data are accepted, record the need for acknowledgment, but wait for an outbound segment with data on which to piggyback the acknowledgment. To avoid long delay, set a persist timer (Table 20.1); if the timer expires before an acknowledgment is sent, transmit an empty segment containing the appropriate acknowledgment number.

The immediate policy is simple and keeps the remote TCP entity fully informed, which limits unnecessary retransmissions. However, this policy results in extra segment transmissions, namely, empty segments used only to ACK. Furthermore, the policy can cause a further load on the network. Consider that a TCP entity receives a segment and immediately sends an ACK. Then the data in the segment are released to the application, which expands the receive window, triggering another empty TCP segment to provide additional credit to the sending TCP entity.

Because of the potential overhead of the immediate policy, the cumulative policy is typically used. Recognize, however, that the use of this policy requires more processing at the receiving end and complicates the task of estimating round-trip time by the sending TCP entity.

20.3 TCP CONGESTION CONTROL

The credit-based flow control mechanism of TCP was designed to enable a destination to restrict the flow of segments from a source to avoid buffer overflow at the destination. This same flow control mechanism is now used in ingenious ways to provide congestion control over the Internet between the source and destination. Congestion, as we have seen a number of times in this book, has two main effects. First, as congestion begins to occur, the transit time across a network or internetwork increases. Second, as congestion becomes severe, network or internet nodes drop packets. The TCP flow control mechanism can be used to recognize the onset of congestion (by recognizing increased delay times and dropped segments) and to react by reducing the flow of data. If many of the TCP entities operating across a network exercise this sort of restraint, internet congestion is relieved.

Since the publication of RFC 793, a number of techniques have been implemented that are intended to improve TCP congestion control characteristics. Table 20.5 lists some of the most popular of these techniques. None of these techniques extends or violates the original TCP standard; rather the techniques represent implementation policies that are within the scope of the TCP specification. Many of these techniques are mandated for use with TCP in RFC 1122 (*Requirements for Internet Hosts*) while some of them are specified in RFC 2581. The labels Tahoe, Reno, and NewReno refer to implementation packages available on many operating systems that support TCP. The techniques fall roughly into two categories: retransmission timer management and window management. In this section, we look at some of the most important and most widely implemented of these techniques.

Retransmission Timer Management

As network or internet conditions change, a static retransmission timer is likely to be either too long or too short. Accordingly, virtually all TCP implementations attempt to estimate the current round-trip time by observing the pattern of delay

Table 20.5 Implementation of TCP Congestion Control Measures

Measure	RFC 1122	TCP Tahoe	TCP Reno	NewReno
RTT Variance Estimation	√	√	√	√
Exponential RTO Backoff	√	√	√	√
Karn's Algorithm	√	√	√	√
Slow Start	√	√	√	√
Dynamic Window Sizing on Congestion	√	√	√	√
Fast Retransmit		√	√	√
Fast Recovery			√	√
Modified Fast Recovery				√

for recent segments, and then set the timer to a value somewhat greater than the estimated round-trip time.

Simple Average A simple approach is to take the average of observed round-trip times over a number of segments. If the average accurately predicts future round-trip times, then the resulting retransmission timer will yield good performance. The simple averaging method can be expressed as

$$\mathrm{ARTT}(K+1) = \frac{1}{K+1}\sum_{i=1}^{K+1}\mathrm{RTT}(i) \tag{20.1}$$

where RTT(i) is the round-trip time observed for the ith transmitted segment, and ARTT(K) is the average round-trip time for the first K segments.

This expression can be rewritten as

$$\mathrm{ARTT}(K+1) = \frac{K}{K+1}\mathrm{ARTT}(K) + \frac{1}{K+1}\mathrm{RTT}(K+1) \tag{20.2}$$

With this formulation, it is not necessary to recalculate the entire summation each time.

Exponential Average Note that each term in the summation is given equal weight; that is, each term is multiplied by the same constant $1/(K+1)$. Typically, we would like to give greater weight to more recent instances because they are more likely to reflect future behavior. A common technique for predicting the next value on the basis of a time series of past values, and the one specified in RFC 793, is exponential averaging:

$$\mathrm{SRTT}(K+1) = \alpha \times \mathrm{SRTT}(K) + (1-\alpha) \times \mathrm{RTT}(K+1) \tag{20.3}$$

where SRTT(K) is called the smoothed round-trip time estimate, and we define $\mathrm{SRTT}(0) = 0$. Compare this with Equation (20.2). By using a constant value of α $(0 < \alpha < 1)$, independent of the number of past observations, we have a circumstance in which all past values are considered, but the more distant ones have less weight. To see this more clearly, consider the following expansion of Equation (20.3):

$$\mathrm{SRTT}(K+1) = (1-\alpha)\mathrm{RTT}(K+1) + \alpha(1-\alpha)\mathrm{RTT}(K) + \alpha^2(1-\alpha)\mathrm{RTT}(K-1) + \cdots + \alpha^K(1-\alpha)\mathrm{RTT}(1)$$

Because both α and $(1-\alpha)$ are less than one, each successive term in the preceding equation is smaller. For example, for $\alpha = 0.8$, the expansion is

$$\mathrm{SRTT}(K+1) = (0.2)\mathrm{RTT}(K+1) + (0.16)\mathrm{RTT}(K) + (0.128)\mathrm{RTT}(K-1) + \cdots$$

The older the observation, the less it is counted in the average.

The smaller the value of α, the greater the weight given to the more recent observations. For $\alpha = 0.5$, virtually all of the weight is given to the four or five most recent observations, whereas for $\alpha = 0.875$, the averaging is effectively spread out over the ten or so most recent observations. The advantage of using a small value of α is that the average will quickly reflect a rapid change in the observed quantity. The disadvantage is that if there is a brief surge in the value of the observed quantity and it then settles back to some relatively constant value, the use of a small value of α will result in jerky changes in the average.

Figure 20.11 compares simple averaging with exponential averaging (for two different values of α). In part (a) of the figure, the observed value begins at 1, grows gradually to a value of 10, and then stays there. In part (b) of the figure, the

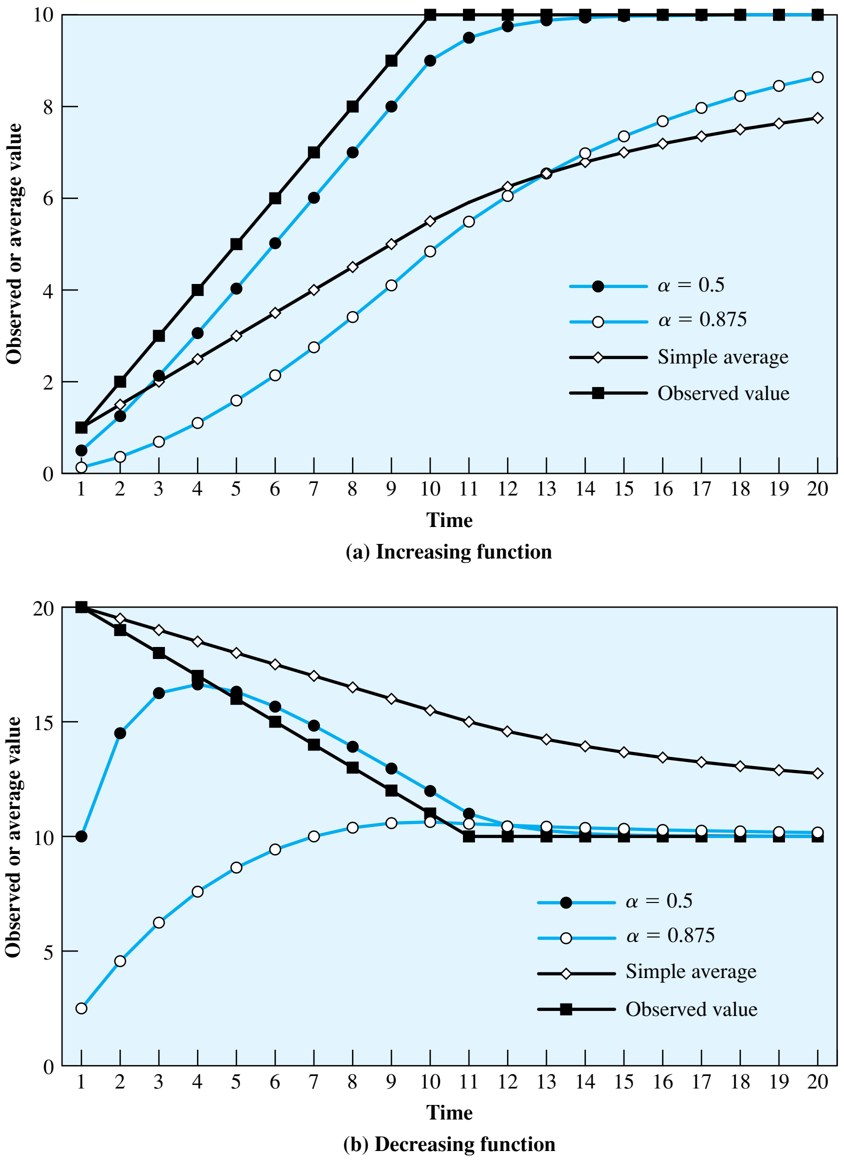


Figure 20.11 Use of Exponential Averaging

observed value begins at 20, declines gradually to 10, and then stays there. Note that exponential averaging tracks changes in RTT faster than does simple averaging and that the smaller value of α results in a more rapid reaction to the change in the observed value.

Equation (20.3) is used in RFC 793 to estimate the current round-trip time. As was mentioned, the retransmission timer should be set at a value somewhat greater than the estimated round-trip time. One possibility is to use a constant value:

$$\text{RTO}(K+1) = \text{SRTT}(K+1) + \Delta$$

where RTO is the retransmission timer (also called the retransmission timeout) and Δ is a constant. The disadvantage of this is that Δ is not proportional to SRTT. For large values of SRTT, Δ is relatively small and fluctuations in the actual RTT will result in unnecessary retransmissions. For small values of SRTT, Δ is relatively large and causes unnecessary delays in retransmitting lost segments. Accordingly, RFC 793 specifies the use of a timer whose value is proportional to SRTT, within limits:

$$\text{RTO}(K+1) = \text{MIN}(\text{UBOUND}, \text{MAX}(\text{LBOUND}, \beta \times \text{SRTT}(K+1))) \quad \textbf{(20.4)}$$

where UBOUND and LBOUND are prechosen fixed upper and lower bounds on the timer value and β is a constant. RFC 793 does not recommend specific values but does list as "example values" the following: α between 0.8 and 0.9 and β between 1.3 and 2.0.

RTT Variance Estimation (Jacobson's Algorithm) The technique specified in the TCP standard, and described in Equations (20.3) and (20.4), enables a TCP entity to adapt to changes in round-trip time. However, it does not cope well with a situation in which the round-trip time exhibits a relatively high variance. [ZHAN86] points out three sources of high variance:

1. If the data rate on the TCP connection is relatively low, then the transmission delay will be relatively large compared to propagation time and the variance in delay due to variance in IP datagram size will be significant. Thus, the SRTT estimator is heavily influenced by characteristics that are a property of the data and not of the network.
2. Internet traffic load and conditions may change abruptly due to traffic from other sources, causing abrupt changes in RTT.
3. The peer TCP entity may not acknowledge each segment immediately because of its own processing delays and because it exercises its privilege to use cumulative acknowledgments.

The original TCP specification tries to account for this variability by multiplying the RTT estimator by a constant factor, as shown in Equation (20.4). In a stable environment, with low variance of RTT, this formulation results in an unnecessarily high value of RTO, and in an unstable environment a value of $\beta = 2$ may be inadequate to protect against unnecessary retransmissions.

A more effective approach is to estimate the variability in RTT values and to use that as input into the calculation of an RTO. A variability measure that is easy to estimate is the mean deviation, defined as

$$\text{MDEV}(X) = \text{E}[|X - \text{E}[X]|]$$

where E[X] is the expected value of X.

As with the estimate of RTT, a simple average could be used to estimate MDEV:

$$\begin{aligned} \text{AERR}(K+1) &= \text{RTT}(K+1) - \text{ARTT}(K) \\ \text{ADEV}(K+1) &= \frac{1}{K+1}\sum_{i=1}^{K+1} |\text{AERR}(i)| \\ &= \frac{K}{K+1}\text{ADEV}(K) + \frac{1}{K+1}|\text{AERR}(K+1)| \end{aligned}$$

where ARTT(K) is the simple average defined in Equation (20.1) and AERR(K) is the sample mean deviation measured at time K.

As with the definition of ARRT, each term in the summation of ADEV is given equal weight; that is, each term is multiplied by the same constant $1/(K+1)$. Again, we would like to give greater weight to more recent instances because they are more likely to reflect future behavior. Jacobson, who proposed the use of a dynamic estimate of variability in estimating RTT [JACO88], suggests using the same exponential smoothing technique as is used for the calculation of SRTT. The complete algorithm proposed by Jacobson can be expressed as follows:

$$\begin{aligned} \text{SRTT}(K+1) &= (1-g) \times \text{SRTT}(K) + g \times \text{RTT}(K+1) \\ \text{SERR}(K+1) &= \text{RTT}(K+1) - \text{SRTT}(K) \\ \text{SDEV}(K+1) &= (1-h) \times \text{SDEV}(K) + h \times |\text{SERR}(K+1)| \\ \text{RTO}(K+1) &= \text{SRTT}(K+1) + f \times \text{SDEV}(K+1) \end{aligned} \tag{20.5}$$

As in the RFC 793 definition [Equation (20.3)], SRTT is an exponentially smoothed estimate of RTT, with $(1-g)$ equivalent to α. Now, however, instead of multiplying the estimate SRTT by a constant [Equation (20.4)], a multiple of the estimated mean deviation is added to SRTT to form the retransmission timer. Based on his timing experiments, Jacobson proposed the following values for the constants in his original paper [JACO88]:

$$\begin{aligned} g &= 1/8 = 0.125 \\ h &= 1/4 = 0.25 \\ f &= 2 \end{aligned}$$

After further research [JACO90a], Jacobson recommended using $f = 4$, and this is the value used in current implementations.

Figure 20.12 illustrates the use of Equation 20.5 on the same data set used in Figure 20.11. Once the arrival times stabilize, the variation estimate SDEV declines. The values of RTO for both $f = 2$ and $f = 4$ are quite conservative as long as RTT is changing but then begin to converge to RTT when it stabilizes.

Experience has shown that Jacobson's algorithm can significantly improve TCP performance. However, it does not stand by itself. Two other factors must be considered:

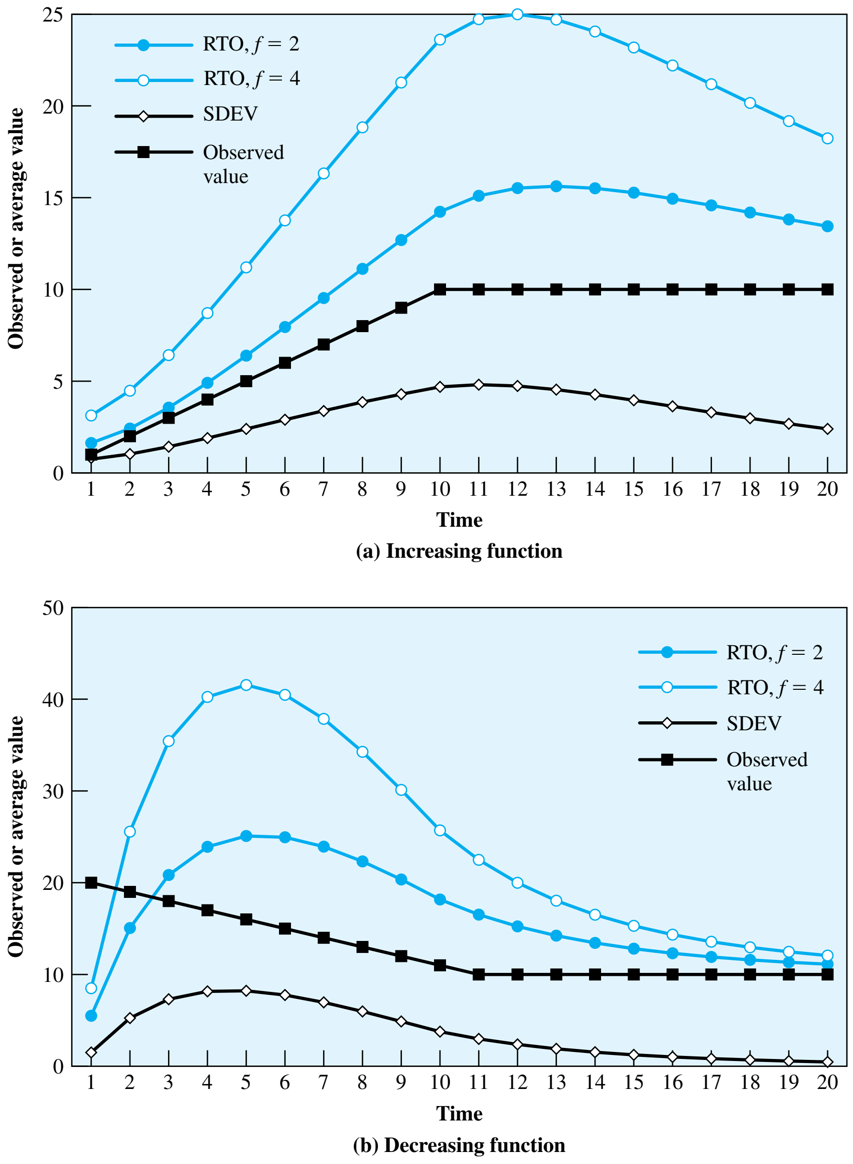


Figure 20.12 Jacobson's RTO Calculation

1. What RTO value should be used on a retransmitted segment? The exponential RTO backoff algorithm is used for this purpose.
2. Which round-trip samples should be used as input to Jacobson's algorithm? Karn's algorithm determines which samples to use.

Exponential RTO Backoff When a TCP sender times out on a segment, it must retransmit that segment. RFC 793 assumes that the same RTO value will be used for this retransmitted segment. However, because the timeout is probably due to network congestion, manifested as a dropped packet or a long delay in round-trip time, maintaining the same RTO value is ill advised.

Consider the following scenario. There are a number of active TCP connections from various sources sending traffic into an internet. A region of congestion develops such that segments on many of these connections are lost or delayed past the RTO time of the connections. Therefore, at roughly the same time, many segments will be retransmitted into the internet, maintaining or even increasing the congestion. All of the sources then wait a local (to each connection) RTO time and retransmit yet again. This pattern of behavior could cause a sustained condition of congestion.

A more sensible policy dictates that a sending TCP entity increase its RTO each time a segment is retransmitted; this is referred to as a *backoff* process. In the scenario of the preceding paragraph, after the first retransmission of a segment on each affected connection, the sending TCP entities will all wait a longer time before performing a second retransmission. This may give the internet time to clear the current congestion. If a second retransmission is required, each sending TCP entity will wait an even longer time before timing out for a third retransmission, giving the internet an even longer period to recover.

A simple technique for implementing RTO backoff is to multiply the RTO for a segment by a constant value for each retransmission:

$$\text{RTO} = q \times \text{RTO} \qquad \textbf{(20.6)}$$

Equation (20.6) causes RTO to grow exponentially with each retransmission. The most commonly used value of q is 2. With this value, the technique is referred to as *binary exponential backoff.* This is the same technique used in the Ethernet CSMA/CD protocol (Chapter 16).

Karn's Algorithm If no segments are retransmitted, the sampling process for Jacobson's algorithm is straightforward. The RTT for each segment can be included in the calculation. Suppose, however, that a segment times out and must be retransmitted. If an acknowledgment is subsequently received, there are two possibilities:

1. This is the ACK to the first transmission of the segment. In this case, the RTT is simply longer than expected but is an accurate reflection of network conditions.
2. This is the ACK to the second transmission.

The sending TCP entity cannot distinguish between these two cases. If the second case is true and the TCP entity simply measures the RTT from the first transmission until receipt of the ACK, the measured time will be much too long. The measured RTT will be on the order of the actual RTT plus the RTO. Feeding this false RTT into Jacobson's algorithm will produce an unnecessarily high value of SRTT and therefore RTO. Furthermore, this effect propagates forward a number of iterations, since the SRTT value of one iteration is an input value in the next iteration.

An even worse approach would be to measure the RTT from the *second* transmission to the receipt of the ACK. If this is in fact the ACK to the first transmission, then the measured RTT will be much too small, producing a too low value of SRTT and RTO. This is likely to have a positive feedback effect, causing additional retransmissions and additional false measurements.

Karn's algorithm [KARN91] solves this problem with the following rules:

1. Do not use the measured RTT for a retransmitted segment to update SRTT and SDEV [Equation (20.5)].
2. Calculate the backoff RTO using Equation (20.6) when a retransmission occurs.
3. Use the backoff RTO value for succeeding segments until an acknowledgment arrives for a segment that has not been retransmitted.

When an acknowledgment is received to an unretransmitted segment, Jacobson's algorithm is again activated to compute future RTO values.

Window Management

In addition to techniques for improving the effectiveness of the retransmission timer, a number of approaches to managing the send window have been examined. The size of TCP's send window can have a critical effect on whether TCP can be used efficiently without causing congestion. We discuss two techniques found in virtually all modern implementations of TCP: slow start and dynamic window sizing on congestion.[3]

Slow Start The larger the send window used in TCP, the more segments that a sending TCP entity can send before it must wait for an acknowledgment. This can create a problem when a TCP connection is first established, because the TCP entity is free to dump the entire window of data onto the internet.

One strategy that could be followed is for the TCP sender to begin sending from some relatively large but not maximum window, hoping to approximate the window size that would ultimately be provided by the connection. This is risky because the sender might flood the internet with many segments before it realized from timeouts that the flow was excessive. Instead, some means is needed of gradually expanding the window until acknowledgments are received. This is the purpose of the slow start mechanism.

With slow start, TCP transmission is constrained by the following relationship:

$$awnd = \text{MIN}[credit, cwnd] \tag{20.7}$$

where

awnd = allowed window, in segments. This is the number of segments that TCP is currently allowed to send without receiving further acknowledgments.

cwnd = congestion window, in segments. A window used by TCP during startup and to reduce flow during periods of congestion.

credit = the amount of unused credit granted in the most recent acknowledgment, in segments. When an acknowledgment is received, this value is calculated as *window/segment_size*, where *window* is a field in the incoming TCP segment (the amount of data the peer TCP entity is willing to accept).

[3]These algorithms were developed by Jacobson [JACO88] and are also described in RFC 2581. Jacobson describes things in units of TCP segments, whereas RFC 2581 relies primarily on units of TCP data octets, with some reference to calculations in units of segments. We follow the development in [JACO88].

When a new connection is opened, the TCP entity initializes $cwnd = 1$. That is, TCP is only allowed to send 1 segment and then must wait for an acknowledgment before transmitting a second segment. Each time an acknowledgment to new data is received, the value of *cwnd* is increased by 1, up to some maximum value.

In effect, the slow-start mechanism probes the internet to make sure that the TCP entity is not sending too many segments into an already congested environment. As acknowledgments arrive, TCP is able to open up its window until the flow is controlled by the incoming ACKs rather than by *cwnd*.

The term *slow start* is a bit of a misnomer, because *cwnd* actually grows exponentially. When the first ACK arrives, TCP opens *cwnd* to 2 and can send two segments. When these two segments are acknowledged, TCP can slide the window 1 segment for each incoming ACK and can increase *cwnd* by 1 for each incoming ACK. Therefore, at this point TCP can send four segments. When these four are acknowledged, TCP will be able to send eight segments.

Dynamic Window Sizing on Congestion The slow-start algorithm has been found to work effectively for initializing a connection. It enables the TCP sender to determine quickly a reasonable window size for the connection. Might not the same technique be useful when there is a surge in congestion? In particular, suppose a TCP entity initiates a connection and goes through the slow-start procedure. At some point, either before or after *cwnd* reaches the size of the credit allocated by the other side, a segment is lost (timeout). This is a signal that congestion is occurring. It is not clear how serious the congestion is. Therefore, a prudent procedure would be to reset $cwnd = 1$ and begin the slow-start process all over.

This seems like a reasonable, conservative procedure, but in fact it is not conservative enough. Jacobson [JACO88] points out that "it is easy to drive a network into saturation but hard for the net to recover." In other words, once congestion occurs, it may take a long time for the congestion to clear.[4] Thus, the exponential growth of *cwnd* under slow start may be too aggressive and may worsen the congestion. Instead, Jacobson proposed the use of slow start to begin with, followed by a linear growth in *cwnd*. The rules are as follows. When a timeout occurs,

1. Set a slow-start threshold equal to half the current congestion window; that is, set $ssthresh = cwnd/2$.
2. Set $cwnd = 1$ and perform the slow-start process until $cwnd = ssthresh$. In this phase, *cwnd* is increased by 1 for every ACK received.
3. For $cwnd \geq ssthresh$, increase *cwnd* by one for each round-trip time.

Figure 20.13 illustrates this behavior. Note that it takes 11 round-trip times to recover to the *cwnd* level that initially took 4 round-trip times to achieve.

Fast Retransmit The retransmission timer (RTO) that is used by a sending TCP entity to determine when to retransmit a segment will generally be noticeably longer than the actual round-trip time (RTT) that the ACK for that segment will take to reach the sender. Both the original RFC 793 algorithm and the Jacobson

[4]Kleinrock refers to this phenomenon as the long-tail effect during a rush-hour period. See Sections 2.7 and 2.10 of [KLEI76] for a detailed discussion.

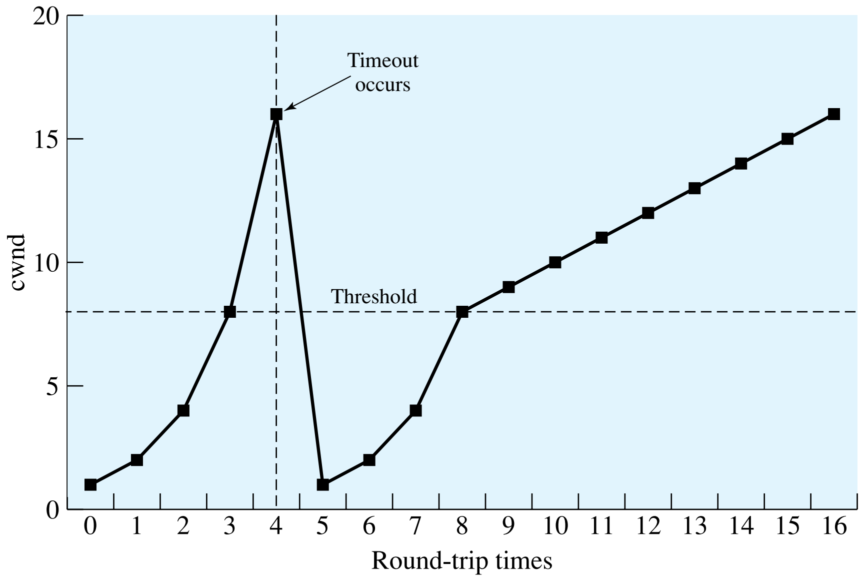


Figure 20.13 Illustration of Slow Start and Congestion Avoidance

algorithm set the value of RTO at somewhat greater than the estimated round-trip time SRTT. Several factors make this margin desirable:

1. RTO is calculated on the basis of a prediction of the next RTT, estimated from past values of RTT. If delays in the network fluctuate, then the estimated RTT may be smaller than the actual RTT.
2. Similarly, if delays at the destination fluctuate, the estimated RTT becomes unreliable.
3. The destination system may not ACK each segment but cumulatively ACK multiple segments, while at the same time sending ACKs when it has any data to send. This behavior contributes to fluctuations in RTT.

A consequence of these factors is that if a segment is lost, TCP may be slow to retransmit. If the destination TCP is using an in-order accept policy (see Section 6.3), then many segments may be lost. Even in the more likely case that the destination TCP is using an in-window accept policy, a slow retransmission can cause problems. To see this, suppose that A transmits a sequence of segments, the first of which is lost. So long as its send window is not empty and RTO does not expire, A can continue to transmit without receiving an acknowledgment. B receives all of these segments except the first. But B must buffer all of these incoming segments until the missing one is retransmitted; it cannot clear its buffer by sending the data to an application until the missing segment arrives. If retransmission of the missing segment is delayed too long, B will have to begin discarding incoming segments.

Jacobson [JACO90b] proposed two procedures, called fast retransmit and fast recovery, that under some circumstances improve on the performance provided by RTO. Fast retransmit takes advantage of the following rule in TCP. If a TCP entity receives a segment out of order, it must immediately issue an ACK for the last in-order segment that was received. TCP will continue to repeat this ACK with each incoming segment until the missing segment arrives to "plug the hole" in its buffer. When the hole is plugged, TCP sends a cumulative ACK for all of the in-order segments received so far.

When a source TCP receives a duplicate ACK, it means that either (1) the segment following the ACKed segment was delayed so that it ultimately arrived out of order, or (2) that segment was lost. In case (1), the segment does ultimately arrive and therefore TCP should not retransmit. But in case (2) the arrival of a duplicate ACK can function as an early warning system to tell the source TCP that a segment has been lost and must be retransmitted. To make sure that we have case (2) rather than case (1), Jacobson recommends that a TCP sender wait until it receives three duplicate ACKs to the same segment (that is, a total of four ACKs to the same segment). Under these circumstances, it is highly likely that the following segment has been lost and should be retransmitted immediately, rather than waiting for a timeout.

Fast Recovery When a TCP entity retransmits a segment using fast retransmit, it knows (or rather assumes) that a segment was lost, even though it has not yet timed out on that segment. Accordingly, the TCP entity should take congestion avoidance measures. One obvious strategy is the slow-start/congestion avoidance procedure used when a timeout occurs. That is, the entity could set *ssthresh* to *cwnd*/2, set *cwnd* = 1 and begin the exponential slow-start process until *cwnd* = *ssthresh*, and then increase *cwnd* linearly. Jacobson [JACO90b] argues that this approach is unnecessarily conservative. As was just pointed out, the very fact that multiple ACKs have returned indicates that data segments are getting through fairly regularly to the other side. So Jacobson proposes a fast recovery technique: retransmit the lost segment, cut *cwnd* in half, and then proceed with the linear increase of *cwnd*. This technique avoids the initial exponential slow-start process.

RFC 2582 (The NewReno Modification to TCP's Fast Recovery Mechanism) modifies the fast recovery algorithm to improve the response when two segments are lost within a single window. Using fast retransmit, a sender retransmits a segment before timeout because it infers that the segment was lost. If the sender subsequently receives an acknowledgement that does not cover all of the segments transmitted before fast retransmit was initiated, the sender may infer that two segments were lost from the current window and retransmit an additional segment. The details of both fast recovery and modified fast recovery are complex; the reader is referred to RFCs 2581 and 2582.

20.4 UDP

In addition to TCP, there is one other transport-level protocol that is in common use as part of the TCP/IP protocol suite: the user datagram protocol (UDP), specified in RFC 768. UDP provides a connectionless service for application-level procedures. Thus, UDP is basically an unreliable service; delivery and duplicate protection are not guaranteed. However, this does reduce the overhead of the protocol and may be adequate in many cases. An example of the use of UDP is in the context of network management, as described in Chapter 22.

The strengths of the connection-oriented approach are clear. It allows connection-related features such as flow control, error control, and sequenced delivery. Connectionless service, however, is more appropriate in some contexts. At lower layers (internet, network), a connectionless service is more robust (e.g., see discussion in Section 10.5). In addition, it represents a "least common denominator" of service to be expected at higher layers. Further, even at transport and above there is justification for a connectionless service. There are instances in which the overhead of connection establishment and termination is unjustified or even counterproductive. Examples include the following:

- **Inward data collection:** Involves the periodic active or passive sampling of data sources, such as sensors, and automatic self-test reports from security equipment or network components. In a real-time monitoring situation, the loss of an occasional data unit would not cause distress, because the next report should arrive shortly.
- **Outward data dissemination:** Includes broadcast messages to network users, the announcement of a new node or the change of address of a service, and the distribution of real-time clock values.
- **Request-response:** Applications in which a transaction service is provided by a common server to a number of distributed TS users, and for which a single request-response sequence is typical. Use of the service is regulated at the application level, and lower-level connections are often unnecessary and cumbersome.
- **Real-time applications:** Such as voice and telemetry, involving a degree of redundancy and/or a real-time transmission requirement. These must not have connection-oriented functions such as retransmission.

Thus, there is a place at the transport level for both a connection-oriented and a connectionless type of service.

UDP sits on top of IP. Because it is connectionless, UDP has very little to do. Essentially, it adds a port addressing capability to IP. This is best seen by examining the UDP header, shown in Figure 20.14. The header includes a source port and destination port. The Length field contains the length of the entire UDP segment, including header and data. The checksum is the same algorithm used for TCP and IP. For UDP, the checksum applies to the entire UDP segment plus a pseudoheader prefixed to the UDP header at the time of calculation and which is the same pseudoheader used for TCP. If an error is detected, the segment is discarded and no further action is taken.

The Checksum field in UDP is optional. If it is not used, it is set to zero. However, it should be pointed out that the IP checksum applies only to the IP header and not to the data field, which in this case consists of the UDP header and the user data. Thus, if no checksum calculation is performed by UDP, then no check is made on the user data at either the transport or internet protocol layers.

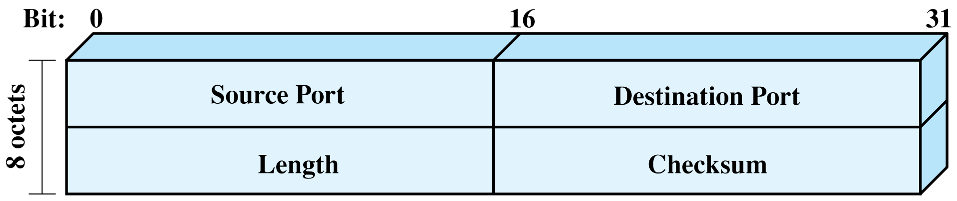


Figure 20.14 UDP Header

20.5 RECOMMENDED READING AND WEB SITES

[IREN99] is a comprehensive survey of transport protocol services and protocol mechanisms, with a brief discussion of a number of different transport protocols. Perhaps the best coverage of the various TCP strategies for flow and congestion control is to be found in [STEV94]. An essential paper for understanding the issues involved is the classic [JACO88].

IREN99 Iren, S.; Amer, P.; and Conrad, P. “The Transport Layer: Tutorial and Survey.” *ACM Computing Surveys*, December 1999.

JACO88 Jacobson, V. “Congestion Avoidance and Control.” *Proceedings, SIGCOMM ’88, Computer Communication Review*, August 1988; reprinted in *Computer Communication Review*, January 1995; a slightly revised version is available at ftp.ee.lbl.gov/papers/congavoid.ps.Z

STEV94 Stevens, W. *TCP/IP Illustrated, Volume 1: The Protocols.* Reading, MA: Addison-Wesley, 1994.

Recommended Web sites:

- **Center for Internet Research:** One of the most active groups in the areas covered in this chapter. The site contains many papers and useful pointers.
- **TCP Maintenance Working Group:** Chartered by IETF to make minor revisions to TCP and to update congestion strategies and protocols. The Web site includes all relevant RFCs and Internet drafts.
- **TCP-Friendly Web site:** Summarizes some of the recent work on adaptive congestion control algorithms for non-TCP-based applications, with a specific focus on schemes that share bandwidth fairly with TCP connections.

20.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

checksum	port	Transmission Control Protocol (TCP)
credit	retransmission strategy	transport protocol
data stream push	sequence number	urgent data signaling
duplicate detection	slow start	User Datagram Protocol (UDP)
exponential average	socket	
flow control	TCP congestion control	
Karn’s algorithm	TCP implementation policy options	
multiplexing		

Review Questions

20.1. What addressing elements are needed to specify a target transport service (TS) user?

20.2. Describe four strategies by which a sending TS user can learn the address of a receiving TS user.

20.3. Explain the use of multiplexing in the context of a transport protocol.

20.4. Briefly describe the credit scheme used by TCP for flow control.

20.5. What is the key difference between the TCP credit scheme and the sliding-window flow control scheme used by many other protocols, such as HDLC?

20.6. Explain the two-way and three-way handshake mechanisms.

20.7. What is the benefit of the three-way handshake mechanism?

20.8. Define the urgent and push features of TCP.

20.9. What is a TCP implementation policy option?

20.10. How can TCP be used to deal with network or internet congestion?

20.11. What does UDP provide that is not provided by IP?

Problems

20.1 It is common practice in most transport protocols (indeed, most protocols at all levels) for control and data to be multiplexed over the same logical channel on a per-user-connection basis. An alternative is to establish a single control transport connection between each pair of communicating transport entities. This connection would be used to carry control signals relating to all user transport connections between the two entities. Discuss the implications of this strategy.

20.2 The discussion of flow control with a reliable network service referred to a backpressure mechanism utilizing a lower-level flow control protocol. Discuss the disadvantages of this strategy.

20.3 Two transport entities communicate across a reliable network. Let the normalized time to transmit a segment equal 1. Assume that the end-to-end propagation delay is 3, and that it takes a time 2 to deliver data from a received segment to the transport user. The sender is initially granted a credit of seven segments. The receiver uses a conservative flow control policy, and updates its credit allocation at every opportunity. What is the maximum achievable throughput?

20.4 Someone posting to comp.protocols.tcp-ip complained about a throughput of 120 kbps on a 256-kbps link with a 128-ms round-trip delay between the United States and Japan, and a throughput of 33 kbps when the link was routed over a satellite.

a. What is the utilization over the two links? Assume a 500-ms round-trip delay for the satellite link.

b. What does the window size appear to be for the two cases?

c. How big should the window size be for the satellite link?

20.5 Draw diagrams similar to Figure 20.4 for the following (assume a reliable sequenced network service):

a. Connection termination: active/passive

b. Connection termination: active/active

c. Connection rejection

d. Connection abortion: User issues an OPEN to a listening user, and then issues a CLOSE before any data are exchanged.

20.6 With a reliable sequencing network service, are segment sequence numbers strictly necessary? What, if any, capability is lost without them?

20.7 Consider a connection-oriented network service that suffers a reset. How could this be dealt with by a transport protocol that assumes that the network service is reliable except for resets?

20.8 The discussion of retransmission strategy made reference to three problems associated with dynamic timer calculation. What modifications to the strategy would help to alleviate those problems?

20.9 Consider a transport protocol that uses a connection-oriented network service. Suppose that the transport protocol uses a credit allocation flow control scheme, and the network protocol uses a sliding-window scheme. What relationship, if any, should there be between the dynamic window of the transport protocol and the fixed window of the network protocol?

20.10 In a network that has a maximum packet size of 128 bytes, a maximum packet lifetime of 30 s, and an 8-bit packet sequence number, what is the maximum data rate per connection?

20.11 Is a deadlock possible using only a two-way handshake instead of a three-way handshake? Give an example or prove otherwise.

20.12 Listed are four strategies that can be used to provide a transport user with the address of the destination transport user. For each one, describe an analogy with the Postal Service user.

a. Know the address ahead of time.
b. Make use of a "well-known address."
c. Use a name server.
d. Addressee is spawned at request time.

20.13 In a credit flow control scheme such as that of TCP, what provision could be made for credit allocations that are lost or misordered in transit?

20.14 What happens in Figure 20.3 if a SYN comes in while the requested user is in CLOSED? Is there any way to get the attention of the user when it is not listening?

20.15 In discussing connection termination with reference to Figure 20.8, it was stated that in addition to receiving an acknowledgement of its FIN and sending an acknowledgement of the incoming FIN, a TCP entity must wait an interval equal to twice the maximum expected segment lifetime (the TIME WAIT state). Receiving an ACK to its FIN assures that all of the segments it sent have been received by the other side. Sending an ACK to the other side's FIN assures the other side that all its segments have been received. Give a reason why it is still necessary to wait before closing the connection.

20.16 Ordinarily, the Window field in the TCP header gives a credit allocation in octets. When the Window Scale option is in use, the value in the Window field is multiplied by a 2^F, where F is the value of the window scale option. The maximum value of F that TCP accepts is 14. Why is the option limited to 14?

20.17 Suppose the round-trip time (RTT) between two hosts is 100 ms, and that both hosts use a TCP window of 32 Kbytes. What is the maximum throughput that can be achieved by means of TCP in this scenario?

20.18 Suppose two hosts are connected with each other by a means of a 100 mbps link, and assume the round-trip time (RTT) between them is 1 ms. What is the minimum TCP window size that would let TCP achieve the maximum possible throughput between these two hosts? (*Note:* Assume no overhead.)

20.19 A host is receiving data from a remote peer by means of TCP segments with a payload of 1460 bytes. If TCP acknowledges every other segment, what is the minimum uplink bandwidth needed to achieve a data throughput of 1 MBytes per second, assuming there is no overhead below the network layer? (*Note:* Assume no options are used by TCP and IP.)

20.20 Analyze the advantages and disadvantages of performing congestion control at the transport layer, rather than at the network layer.

20.21 Jacobson's congestion control algorithm assumes most packet losses are caused by routers dropping packets due to network congestion. However, packets may be also dropped if they are corrupted in their path to destination. Analyze the performance of TCP in a such lossy environment, due to Jacobson's congestion control algorithm.

20.22 One difficulty with the original TCP SRTT estimator is the choice of an initial value. In the absence of any special knowledge of network conditions, the typical approach is to pick an arbitrary value, such as 3 seconds, and hope that this will converge quickly to an accurate value. If this estimate is too small, TCP will perform unnecessary retransmissions. If it is too large, TCP will wait a long time before retransmitting if the first segment is lost. Also, the convergence may be slow, as this problem indicates.

- **a.** Choose $\alpha = 0.85$ and $\text{SRTT}(0) = 3$ seconds, and assume all measured RTT values $= 1$ second and no packet loss. What is SRTT(19)? *Hint*: Equation (20.3) can be rewritten to simplify the calculation, using the expression $(1 - \alpha^n)/(1 - \alpha)$.
- **b.** Now let $\text{SRTT}(0) = 1$ second and assume measured RTT values $= 3$ seconds and no packet loss. What is SRTT(19)?

20.23 A poor implementation of TCP's sliding-window scheme can lead to extremely poor performance. There is a phenomenon known as the Silly Window Syndrome (SWS), which can easily cause degradation in performance by several factors of 10. As an example of SWS, consider an application that is engaged in a lengthy file transfer, and that TCP is transferring this file in 200-octet segments. The receiver initially provides a credit of 1000. The sender uses up this window with 5 segments of 200 octets. Now suppose that the receiver returns an acknowledgment to each segment and provides an additional credit of 200 octets for every received segment. From the receiver's point of view, this opens the window back up to 1000 octets. However, from the sender's point of view, if the first acknowledgment arrives after five segments have been sent, a window of only 200 octets becomes available. Assume that at some point, the sender calculates a window of 200 octets but has only 50 octets to send until it reaches a "push" point. It therefore sends 50 octets in one segment, followed by 150 octets in the next segment, and then resumes transmission of 200-octet segments. What might now happen to cause a performance problem? State the SWS in more general terms.

20.24 TCP mandates that both the receiver and the sender should incorporate mechanisms to cope with SWS.

- **a.** Suggest a strategy for the receiver. *Hint*: Let the receiver "lie" about how much buffer space is available under certain circumstances. State a reasonable rule of thumb for this.
- **b.** Suggest a strategy for the sender. *Hint*: Consider the relationship between the maximum possible send window and what is currently available to send.

20.25 Derive Equation (20.2) from Equation (20.1).

20.26 In Equation (20.5), rewrite the definition of $\text{SRTT}(K + 1)$ so that it is a function of $\text{SERR}(K + 1)$. Interpret the result.

20.27 A TCP entity opens a connection and uses slow start. Approximately how many round-trip times are required before TCP can send N segments.

20.28 Although slow start with congestion avoidance is an effective technique for coping with congestion, it can result in long recovery times in high-speed networks, as this problem demonstrates.

- **a.** Assume a round-trip time of 60 ms (about what might occur across a continent) and a link with an available bandwidth of 1 Gbps and a segment size of 576 octets. Determine the window size needed to keep the pipe full and the time it will take to reach that window size after a timeout using Jacobson's approach.
- **b.** Repeat (a) for a segment size of 16 Kbytes.