# Unit 5

## I] The Internet Transport Protocols:

The Internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one. The protocols complement each other. The connectionless protocol is UDP. It does almost nothing beyond sending packets between applications, letting applications build their own protocols on top as needed. The connection-oriented protocol is TCP. It does almost everything. It makes connections and adds reliability with retransmissions, along with flow control and congestion control, all on behalf of the applications that use it.

### 1. Introduction to UDP

The Internet protocol suite supports a connectionless transport protocol called UDP (User Datagram Protocol). UDP provides a way for applications to send encapsulated IP datagrams without having to establish a connection. UDP transmits segments consisting of an 8-byte header followed by the payload. The header is shown in Fig. 6-27. The two ports serve to identify the endpoints within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port. Without the port fields, the transport layer would not know what to do with each incoming packet. With them, it delivers the embedded segment to the correct application.

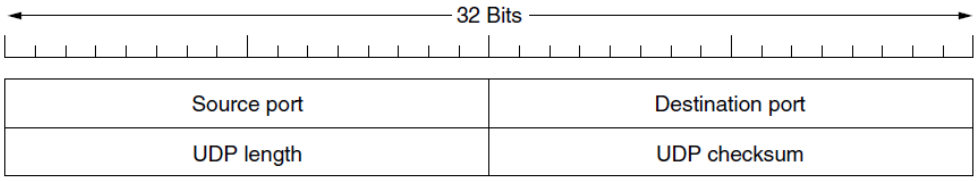


**Figure 6-27.** The UDP header.

The source port is primarily needed when a reply must be sent back to the source. By copying the Source port field from the incoming segment into the Destination port field of the outgoing segment, the process sending the reply can specify which process on the sending machine is to get it. The UDP length field includes the 8-byte header and the data. The minimum length is 8 bytes, to cover the header. The maximum length is 65,515 bytes, which is lower than the largest number that will fit in 16 bits because of the size limit on IP packets.

An optional Checksum is also provided for extra reliability. It checksums the header, the data, and a conceptual IP pseudoheader. When performing this computation, the Checksum field is set to zero and the data field is padded out with an additional zero byte if its length is an odd number. The checksum algorithm is simply to add up all the 16-bit words in one's complement and to take the

one's complement of the sum. As a consequence, when the receiver performs the calculation on the entire segment, including the Checksum field, the result should be 0. If the checksum is not computed, it is stored as a 0, since by a happy coincidence of one's complement arithmetic a true computed 0 is stored as all 1s. The pseudoheader for the case of IPv4 is shown in Fig. 6-28. It contains the 32-bit IPv4 addresses of the source and destination machines, the protocol number for UDP (17), and the byte count for the UDP segment (including the header). It is different but analogous for IPv6. Including the pseudoheader in the UDP checksum computation helps detect misdelivered packets, but including it also violates the protocol hierarchy since the IP addresses in it belong to the IP layer, not to the UDP layer. TCP uses the same pseudoheader for its checksum.

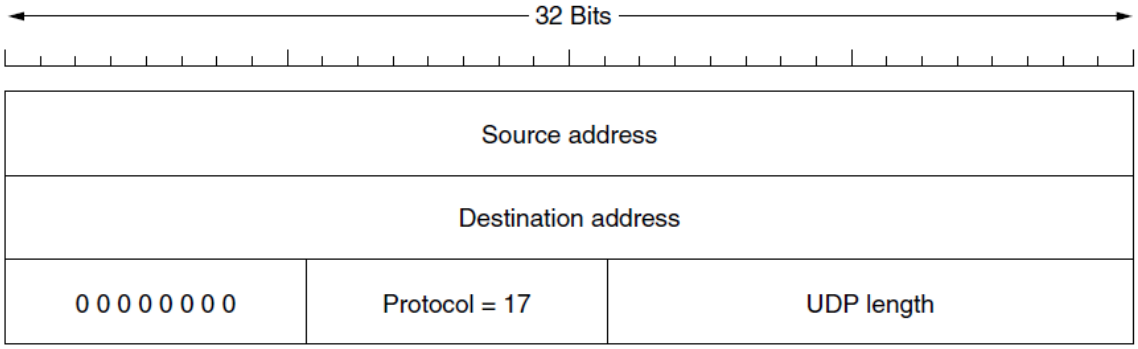


**Figure 6-28.** The IPv4 pseudoheader included in the UDP checksum.

It is probably worth mentioning explicitly some of the things that UDP does not do. It does not do flow control, congestion control, or retransmission upon receipt of a bad segment. All of that is up to the user processes. What it does do is provide an interface to the IP protocol with the added feature of demultiplexing multiple processes using the ports and optional end-to-end error detection. That is all it does.

## 2. Introduction to TCP

UDP is a simple protocol and it has some very important uses, such as clientserver interactions and multimedia, but for most Internet applications, reliable, sequenced delivery is needed. UDP cannot provide this, so another protocol is required. It is called TCP and is the main workhorse of the Internet.

TCP (Transmission Control Protocol) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters. TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.

Each machine supporting TCP has a TCP transport entity, either a library procedure, a user process, or most commonly part of the kernel. In all cases, it manages TCP streams and interfaces to the IP layer. A TCP entity accepts user data streams from local processes, breaks them up into pieces not

exceeding 64 KB (in practice, often 1460 data bytes in order to fit in a single Ethernet frame with the IP and TCP headers), and sends each piece as a separate IP datagram. When datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which reconstructs the original byte streams. For simplicity, we will sometimes use just ‘‘TCP’’ to mean the TCP transport entity (a piece of software) or the TCP protocol (a set of rules). From the context it will be clear which is meant. For example, in ‘‘The user gives TCP the data,’’ the TCP transport entity is clearly intended. The IP layer gives no guarantee that datagrams will be delivered properly, nor any indication of how fast datagrams may be sent. It is up to TCP to send datagrams fast enough to make use of the capacity but not cause congestion, and to time out and retransmit any datagrams that are not delivered. Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence. In short, TCP must furnish good performance with the reliability that most applications want and that IP does not provide.

## 3. The TCP Service Model

TCP service is obtained by both the sender and the receiver creating end points, called sockets, Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a port. A port is the TCP name for a TSAP. [TSAP stands for "Transport Service Access Point." It's a concept used in networking to identify a specific service or application on a device within a network. TSAPs are often associated with OSI (Open Systems Interconnection) model]. For TCP service to be obtained, a connection must be explicitly established between a socket on one machine and a socket on another machine. The socket calls are listed in Fig. 6-5. A socket may be used for multiple connections at the same time. In other words, two or more connections may terminate at the same socket. Connections are identified by the socket identifiers at both ends, that is, (socket1, socket2). No virtual circuit numbers or other identifiers are used. Port numbers below 1024 are reserved for standard services that can usually only be started by privileged users (e.g., root in UNIX systems). They are called well-known ports. For example, any process wishing to remotely retrieve mail from a host can connect to the destination host’s port 143 to contact its IMAP daemon. The list of well-known ports is given at www.iana.org. Over 700 have been assigned. A few of the better-known ones are listed in Fig. 6-34.

| Port | Protocol | Use |
|---|---|---|
| 20, 21 | FTP | File transfer |
| 22 | SSH | Remote login, replacement for Telnet |
| 25 | SMTP | Email |
| 80 | HTTP | World Wide Web |
| 110 | POP-3 | Remote email access |
| 143 | IMAP | Remote email access |
| 443 | HTTPS | Secure Web (HTTP over SSL/TLS) |
| 543 | RTSP | Media player control |
| 631 | IPP | Printer sharing |

**Figure 6-34.** Some assigned ports.

Other ports from 1024 through 49151 can be registered with IANA for use by unprivileged users, but applications can and do choose their own ports. For example, the BitTorrent peer-to-peer file-sharing application (unofficially) uses ports 6881–6887, but may run on other ports as well. inetd (Internet daemon) in UNIX, attach itself to multiple ports and wait for the first incoming connection. When that occurs, inetd forks off a new process and executes the appropriate daemon in it, letting that daemon handle the request. In this way, the daemons other than inetd are only active when there is work for them to do. Inetd learns which ports it is to use from a configuration file. Consequently, the system administrator can set up the system to have permanent daemons on the busiest ports (e.g., port 80) and inetd on the rest. All TCP connections are full duplex and point-to-point. All TCP connections are full duplex and point-to-point. Full duplex means that traffic can go in both directions at the same time. Point-to-point means that each connection has exactly two end points. TCP does not support multicasting or broadcasting.

A TCP connection is a byte stream, not a message stream. Message boundaries are not preserved end to end. For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as four 512-byte chunks, two 1024-byte chunks, one 2048-byte chunk (see Fig. 6-35), or some other way. There is no way for the receiver to detect the unit(s) in which the data were written, no matter how hard it tries.

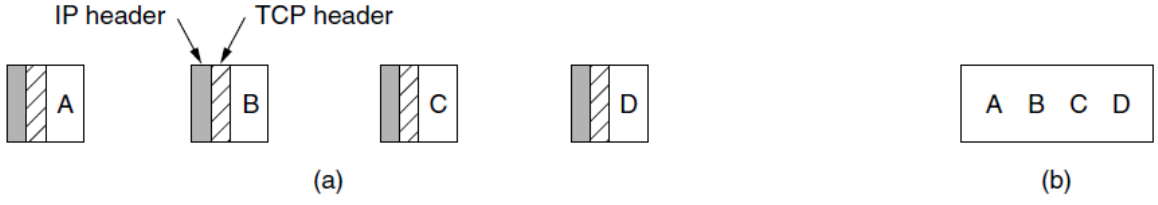


**Figure 6-35.** (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

Files in UNIX have this property too. The reader of a file cannot tell whether the file was written a block at a time, a byte at a time, or all in one blow. As with a UNIX file, the TCP software has no idea of what the bytes mean and no interest in finding out. A byte is just a byte. When an application passes data to TCP, TCP may send it immediately or buffer it (in order to collect a larger amount to send at once), at its discretion. However, sometimes the application really wants the data to be sent immediately. For example, suppose a user of an interactive game wants to send a stream of updates. It is essential that the updates be sent immediately, not buffered until there is a collection of them. To force data out, TCP has the notion of a PUSH flag that is carried on packets. The original intent was to let applications tell TCP implementations via the PUSH flag not to delay the transmission. However, applications cannot literally set the PUSH flag when they send data. Instead, different operating systems have evolved different options to expedite transmission (e.g., TCP NODELAY in Windows and Linux). For Internet archaeologists, we will also mention one interesting feature of TCP service that remains in the protocol but is rarely used: urgent data. When an application has high priority data that should be processed immediately, for example, if an interactive user hits the CTRL-C key to break off a remote computation that has already begun, the sending application can put some control information in the data stream and give it to TCP along with the URGENT flag. This event causes TCP to stop accumulating data and transmit everything it has for that connection immediately. When the urgent data are received at the destination, the receiving application is interrupted (e.g., given a signal in UNIX terms) so it can stop whatever it was doing and read the data stream to find the urgent data. The end of the urgent data is marked so the application knows when it is over. The start of the urgent data is not marked. It is up to the application to figure that out. This scheme provides a crude signaling mechanism and leaves everything else up to the application. However, while urgent data is potentially useful, it found no compelling application early on and fell into disuse. Its use is now discouraged because of implementation differences, leaving applications to handle their own signaling. Perhaps future transport protocols will provide better signaling.

## II] The TCP Protocol:

### 1. TCP protocol

A key feature of TCP, and one that dominates the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number. The sending and receiving TCP entities exchange data in the form of segments. A TCP segment consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes. The TCP software decides how big segments should be. It can accumulate data from several writes into one segment or can split data from one write over multiple segments. Two limits restrict the segment size.

- First, each segment, including the TCP header, must fit in the 65,515- byte IP payload.
- Second, each link has an MTU (Maximum Transfer Unit).

Each segment must fit in the MTU at the sender and receiver so that it can be sent and received in a single, unfragmented packet. In practice, the MTU is generally 1500 bytes (the Ethernet payload size) and thus defines the upper bound on segment size. It is still possible for IP packets carrying TCP segments to be fragmented when passing over a network path for which some link has a small MTU. If this happens, it degrades performance and causes other problems. Instead, modern TCP implementations perform path MTU discovery by using the technique which uses ICMP error messages to find the smallest MTU for any link on the path. TCP then adjusts the segment size downwards to avoid fragmentation. The basic protocol used by TCP entities is the sliding window protocol with a dynamic window size. When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exist, and otherwise without) bearing an acknowledgement number equal to the next sequence number it expects to receive and the remaining window size. If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again. Although this protocol sounds simple, there are many sometimes subtle ins and outs. Segments can arrive out of order, so bytes 3072–4095 can arrive but cannot be acknowledged because bytes 2048–3071 have not turned up yet. Segments can also be delayed so long in transit that the sender times out and retransmits them. The retransmissions may include different byte ranges than the original transmission, requiring careful administration to keep track of which bytes have been correctly received so far. However, since each byte in the stream has its own unique offset, it can be done. TCP must be prepared to deal with these problems and solve them in an efficient way. A considerable amount of effort has gone into optimizing the performance of TCP streams, even in the face of network problems.

## 2. TCP Segment Header

Figure 6-36 shows the layout of a TCP segment. Every segment begins with a fixed-format, 20-byte header. The fixed header may be followed by header options. After the options, if any, up to 65,535 − 20 − 20 = 65,495 data bytes may follow, where the first 20 refer to the IP header and the second to the TCP header. Segments without any data are legal and are commonly used for acknowledgements and control messages.

The Source port and Destination port fields identify the local end points of the connection. A TCP port plus its host's IP address forms a 48-bit unique end point. The source and destination end points together identify the connection. This connection identifier is called a 5 tuple because it consists of five pieces of information: the protocol (TCP), source IP and source port, and destination IP and destination port.

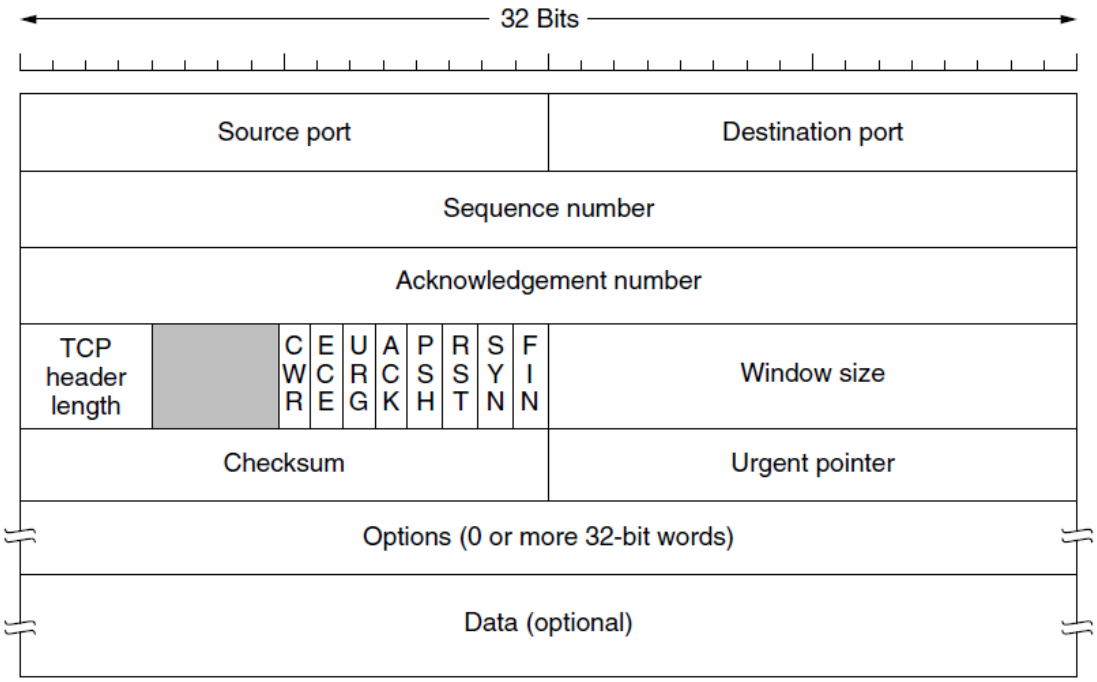


**Figure 6-36.** The TCP header.

The Sequence number and Acknowledgement number fields perform their usual functions. Note that the latter specifies the next in-order byte expected, not the last byte correctly received. It is a cumulative acknowledgement because it summarizes the received data with a single number. It does not go beyond lost data. Both are 32 bits because every byte of data is numbered in a TCP stream.

The TCP header length tells how many 32-bit words are contained in the TCP header. This information is needed because the Options field is of variable length, so the header is, too. Technically, this field really indicates the start of the data within the segment, measured in 32-bit words, but that number is just the header length in words, so the effect is the same.

Next comes a 4-bit field that is not used. The fact that these bits have remained unused for 30 years (as only 2 of the original reserved 6 bits have been reclaimed) is testimony to how well thought out TCP is. Lesser protocols would have needed these bits to fix bugs in the original design. Now come eight 1-bit flags. CWR and ECE are used to signal congestion when ECN (Explicit Congestion Notification) is used.

ECE is set to signal an ECN-Echo to a TCP sender to tell it to slow down when the TCP receiver gets a congestion indication from the network. CWR is set to signal Congestion Window Reduced from the TCP sender to the TCP receiver so that it knows the sender has slowed down and can stop sending the ECN-Echo.

URG is set to 1 if the Urgent pointer is in use. The Urgent pointer is used to indicate a byte offset from the current sequence number at which urgent data are to be found. This facility is in lieu of interrupt messages. This facility is a bare-bones way of allowing the sender to signal the receiver without getting TCP itself involved in the reason for the interrupt, but it is seldom used. The ACK bit is set to 1 to indicate that the Acknowledgement number is valid. This is the case for nearly all packets. If ACK is 0, the segment does not contain an acknowledgement, so the Acknowledgement

number field is ignored. The PSH bit indicates PUSHed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency).

The RST bit is used to abruptly reset a connection that has become confused due to a host crash or some other reason. It is also used to reject an invalid segment or refuse an attempt to open a connection. In general, if you get a segment with the RST bit on, you have a problem on your hands.

The SYN bit is used to establish connections. The connection request has SYN = 1 and ACK = 0 to indicate that the piggyback acknowledgement field is not in use. The connection reply does bear an acknowledgement, however, so it has SYN = 1 and ACK = 1. In essence, the SYN bit is used to denote both CONNECTION REQUEST and CONNECTION ACCEPTED, with the ACK bit used to distinguish between those two possibilities.

The FIN bit is used to release a connection. It specifies that the sender has no more data to transmit. However, after closing a connection, the closing process may continue to receive data indefinitely. Both SYN and FIN segments have sequence numbers and are thus guaranteed to be processed in the correct order. Flow control in TCP is handled using a variable-sized sliding window. The Window size field tells how many bytes may be sent starting at the byte acknowledged. A Window size field of 0 is legal and says that the bytes up to and including Acknowledgement number − 1 have been received, but that the receiver has not had a chance to consume the data and would like no more data for the moment, thank you. The receiver can later grant permission to send by transmitting a segment with the same Acknowledgement number and a nonzero Window size field. In TCP, acknowledgements and permission to send additional data are completely decoupled. In effect, a receiver can say: ‘‘I have received bytes up through k but I do not want any more just now, thank you.’’ This decoupling (in fact, a variable-sized window) gives additional flexibility.

A Checksum is also provided for extra reliability. It checksums the header, the data, and a conceptual pseudoheader in exactly the same way as UDP, except that the pseudoheader has the protocol number for TCP (6) and the checksum is mandatory.

The Options field provides a way to add extra facilities not covered by the regular header. Many options have been defined and several are commonly used. The options are of variable length, fill a multiple of 32 bits by using padding with zeros, and may extend to 40 bytes to accommodate the longest TCP header that can be specified. Some options are carried when a connection is established to negotiate or inform the other side of capabilities. Other options are carried on packets during the lifetime of the connection. Each option has a Type-Length-Value encoding.

A widely used option is the one that allows each host to specify the MSS (Maximum Segment Size) it is willing to accept. Using large segments is more efficient than using small ones because the 20-byte header can be amortized over more data, but small hosts may not be able to handle big segments. During connection setup, each side can announce its maximum and see its partner’s. If a host does not use this option, it defaults to a 536-byte payload. All Internet hosts are required to

accept TCP segments of 536 + 20 = 556 bytes. The maximum segment size in the two directions need not be the same. For lines with high bandwidth, high delay, or both, the 64-KB window corresponding to a 16-bit field is a problem. For example, on an OC-12 line (of roughly 600 Mbps), it takes less than 1 msec to output a full 64-KB window. If the round-trip propagation delay is 50 msec (which is typical for a transcontinental fiber), the sender will be idle more than 98% of the time waiting for acknowledgements. A larger window size would allow the sender to keep pumping data out.

The window scale option allows the sender and receiver to negotiate a window scale factor at the start of a connection. Both sides use the scale factor to shift the Window size field up to 14 bits to the left, thus allowing windows of up to 230 bytes. Most TCP implementations support this option.

The timestamp option carries a timestamp sent by the sender and echoed by the receiver. It is included in every packet, once its use is established during connection setup, and used to compute round-trip time samples that are used to estimate when a packet has been lost. It is also used as a logical extension of the 32- bit sequence number. On a fast connection, the sequence number may wraparound quickly, leading to possible confusion between old and new data. The PAWS (Protection Against Wrapped Sequence numbers) scheme discards arriving segments with old timestamps to prevent this problem. Finally, the SACK (Selective ACKnowledgement) option lets a receiver tell a sender the ranges of sequence numbers that it has received. It supplements the Acknowledgement number and is used after a packet has been lost but subsequent (or duplicate) data has arrived. The new data is not reflected by the Acknowledgement number field in the header because that field gives only the next in-order byte that is expected. With SACK, the sender is explicitly aware of what data the receiver has and hence can determine what data should be retransmitted.

## 3. TCP Connection Establishment

Connections are established in TCP by means of the three-way handshake. To establish a connection, one side, say, the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives in that order, either specifying a specific source or nobody in particular.

The other side, say, the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password). The CONNECT primitive sends a TCP segment with the SYN bit on and ACK bit off and waits for a response.

When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the Destination port field. If not, it sends a reply with the RST bit on to reject the connection. If some process is listening to the port, that process is given the incoming TCP segment. It can either accept or reject the connection. If it accepts, an

acknowledgement segment is sent back. The sequence of TCP segments sent in the normal case is shown in Fig. 6-37(a). Note that a SYN segment consumes 1 byte of sequence space so that it can be acknowledged unambiguously.

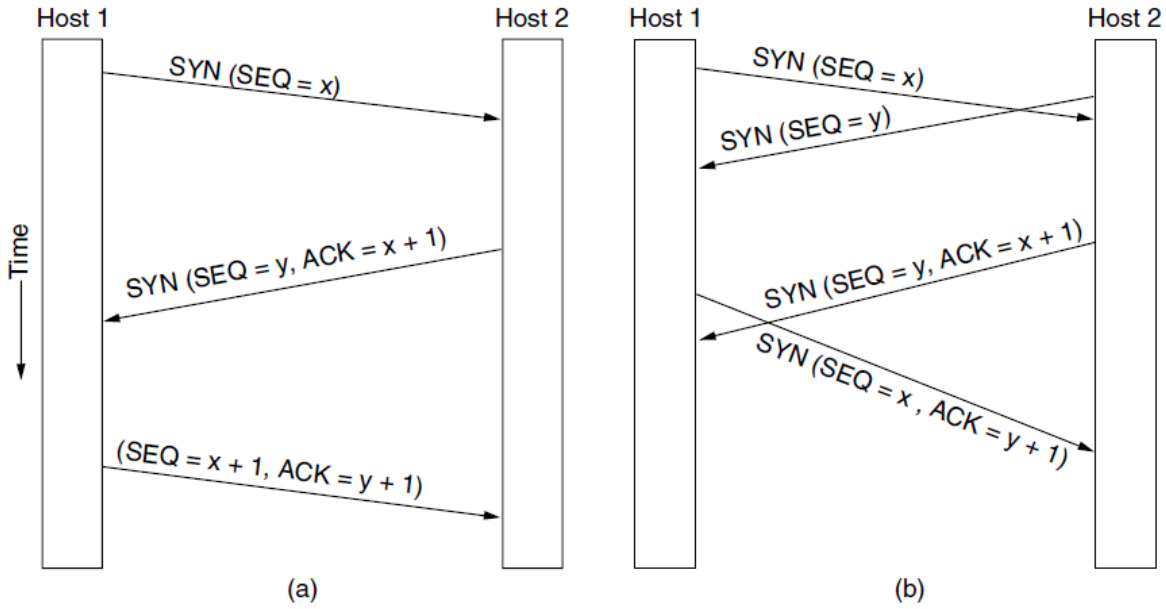


**Figure 6-37.** (a) TCP connection establishment in the normal case. (b) Simultaneous connection establishment on both sides.

In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the sequence of events is as illustrated in Fig. 6- 37(b). The result of these events is that just one connection is established, not two, because connections are identified by their end points. If the first setup results in a connection identified by (x, y) and the second one does too, only one table entry is made, namely, for (x, y). Recall that the initial sequence number chosen by each host should cycle slowly, rather than be a constant such as 0. This rule is to protect against delayed duplicate packets, as we discussed in Sec 6.2.2. Originally this was accomplished with a clock-based scheme in which the clock ticked every 4 μsec. However, a vulnerability with implementing the three-way handshake is that the listening process must remember its sequence number as soon it responds with its own SYN segment. This means that a malicious sender can tie up resources on a host by sending a stream of SYN segments and never following through to complete the connection. This attack is called a SYN flood, and it crippled many Web servers in the 1990s.

One way to defend against this attack is to use SYN cookies. Instead of remembering the sequence number, a host chooses a cryptographically generated sequence number, puts it on the outgoing segment, and forgets it. If the three-way handshake completes, this sequence number (plus 1) will be returned to the host. It can then regenerate the correct sequence number by running the same cryptographic function, as long as the inputs to that function are known, for example, the other host's IP address and port, and a local secret. This procedure allows the host to check that an acknowledged sequence number is correct without having to remember the sequence number separately. There are some caveats, such as the inability to handle TCP options, so SYN cookies

may be used only when the host is subject to a SYN flood. However, they are an interesting twist on connection establishment.

Although TCP connections are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections. Each simplex connection is released independently of its sibling. To release a connection, either party can send a TCP segment with the FIN bit set, which means that it has no more data to transmit. When the FIN is acknowledged, that direction is shut down for new data. Data may continue to flow indefinitely in the other direction, however.

When both directions have been shut down, the connection is released. Normally, four TCP segments are needed to release a connection: one FIN and one ACK for each direction. However, it is possible for the first ACK and the second FIN to be contained in the same segment, reducing the total count to three. Just as with telephone calls in which both people say goodbye and hang up the phone simultaneously, both ends of a TCP connection may send FIN segments at the same time. These are each acknowledged in the usual way, and the connection is shut down. There is, in fact, no essential difference between the two hosts releasing sequentially or simultaneously.

To avoid the two-army problem, timers are used. If a response to a FIN is not forthcoming within two maximum packet lifetimes, the sender of the FIN releases the connection. The other side will eventually notice that nobody seems to be listening to it anymore and will time out as well. While this solution is not perfect, given the fact that a perfect solution is theoretically impossible, it will have to do. In practice, problems rarely arise.

## 4. TCP Connection Release

The steps required to establish and release connections can be represented in a finite state machine with the 11 states listed in Fig. 6-38. In each state, certain events are legal. When a legal event happens, some action may be taken. If some other event happens, an error is reported. Each connection starts in the *CLOSED* state. It leaves that state when it does either a passive open (LISTEN) or an active open (CONNECT). If the other side does the opposite one, a connection is established and the state becomes *ESTABLISHED*. Connection release can be initiated by either side. When it is complete, the state returns to *CLOSED*. The finite state machine itself is shown in Fig. 6-39. The common case of a client actively connecting to a passive server is shown with heavy lines—solid for the client, dotted for the server. The lightface lines are unusual event sequences.

| State | Description |
|---|---|
| CLOSED | No connection is active or pending |
| LISTEN | The server is waiting for an incoming call |
| SYN RCVD | A connection request has arrived; wait for ACK |
| SYN SENT | The application has started to open a connection |
| ESTABLISHED | The normal data transfer state |
| FIN WAIT 1 | The application has said it is finished |
| FIN WAIT 2 | The other side has agreed to release |
| TIME WAIT | Wait for all packets to die off |
| CLOSING | Both sides have tried to close simultaneously |
| CLOSE WAIT | The other side has initiated a release |
| LAST ACK | Wait for all packets to die off |

**Figure 6-38.** The states used in the TCP connection management finite state machine.

Each line in Fig. 6-39 is marked by an *event/action* pair. The event can either be a user-initiated system call (CONNECT, LISTEN, SEND, or CLOSE), a segment arrival (*SYN*, *FIN*, *ACK*, or *RST*), or, in one case, a timeout of twice the maximum packet lifetime. The action is the sending of a control segment (*SYN*, *FIN*, or *RST*) or nothing, indicated by —. Comments are shown in parentheses. One can best understand the diagram by first following the path of a client (the heavy solid line), then later following the path of a server (the heavy dashed line). When an application program on the client machine issues a CONNECT request, the local TCP entity creates a connection record, marks it as being in the *SYN SENT* state, and shoots off a *SYN* segment. Note that many connections may be open (or being opened) at the same time on behalf of multiple applications, so the state is per connection and recorded in the connection record. When the *SYN+ACK* arrives, TCP sends the final *ACK* of the three-way handshake and switches into the *ESTABLISHED* state. Data can now be sent and received. When an application is finished, it executes a CLOSE primitive, which causes the local TCP entity to send a *FIN* segment and wait for the corresponding *ACK* (dashed box marked ‘‘active close’’). When the *ACK* arrives, a transition is made to the state *FIN WAIT 2* and one direction of the connection is closed. When the other side closes, too, a *FIN* comes in, which is acknowledged. Now both sides are closed, but TCP waits a time equal to twice the maximum packet lifetime to guarantee that all packets from the connection have died off, just in case the acknowledgement was lost. When the timer goes off, TCP deletes the connection record.

**Connection management from the server’s viewpoint:**

The server does a LISTEN and settles down to see who turns up. When a *SYN* comes in, it is acknowledged and the server goes to the *SYN RCVD* state. When the server’s *SYN* is itself

acknowledged, the three-way handshake is complete and the server goes to the *ESTABLISHED* state. Data transfer can now occur. When the client is done transmitting its data, it does a CLOSE, which causes a *FIN* to arrive at the server (dashed box marked ‘‘passive close’’). The server is then signaled. When it, too, does a CLOSE, a *FIN* is sent to the client. When the client’s acknowledgement shows up, the server releases the connection and deletes the connection record.

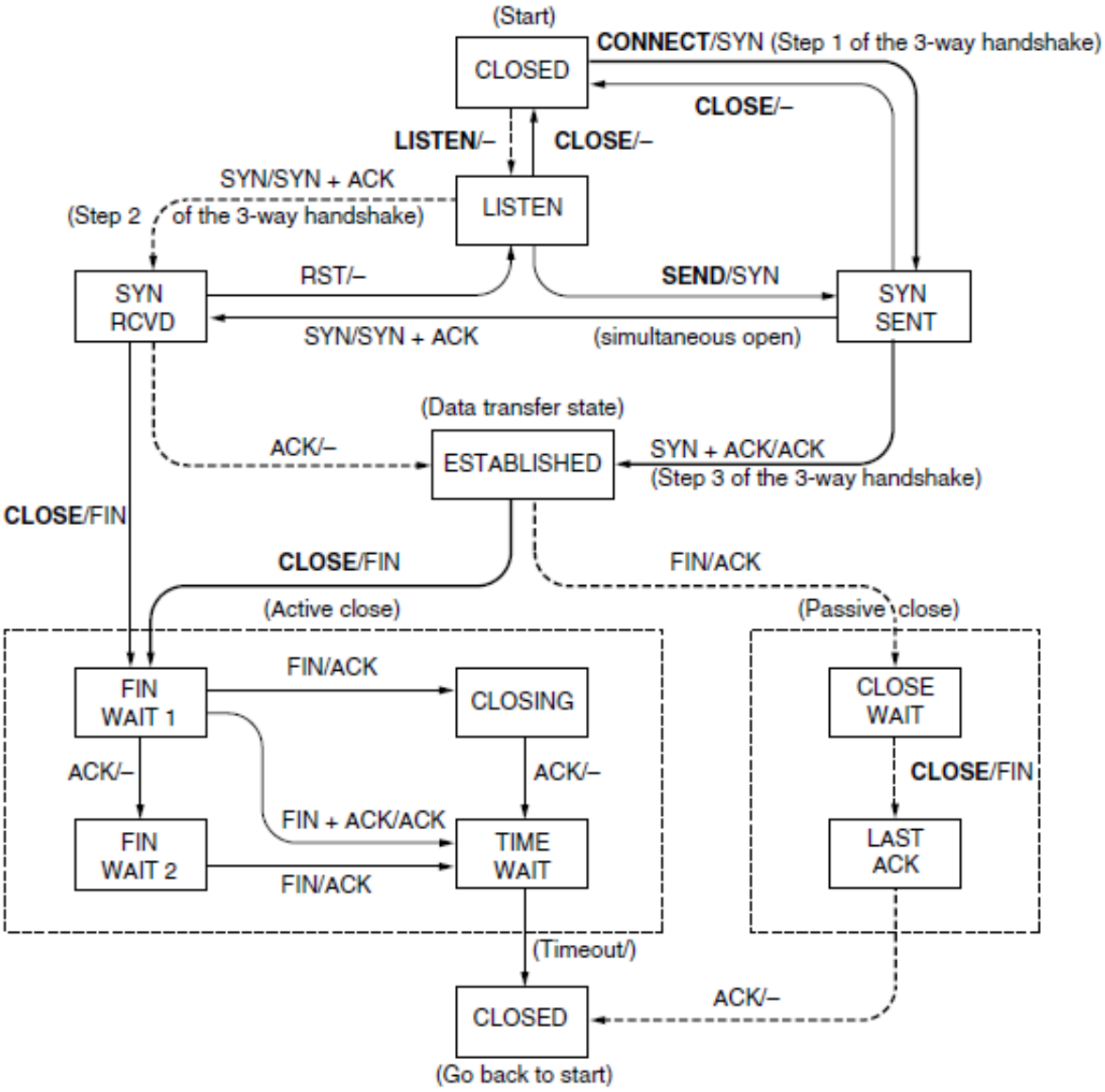


**Figure 6-39.** TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled with the event causing it and the action resulting from it, separated by a slash.

## 5. TCP Transmission Policy

## 6. TCP Congestion Control

## 7. TCP Timer Management

TCP uses multiple timers (at least conceptually) to do its work. The most important of these is the **RTO** (**Retransmission TimeOut**). When a segment is sent, a retransmission timer is started. If the segment is acknowledged before the timer expires, the timer is stopped. If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted (and the timer os started again). The question that arises is: how long should the timeout be? This problem is much more difficult in the transport layer than in data link protocols such as 802.11. In the latter case, the expected delay is measured in microseconds and is highly predictable (i.e., has a low variance), so the timer can be set to go off just slightly after the acknowledgement is expected, as shown in Fig. 6-42(a). Since acknowledgements are rarely delayed in the data link layer (due to lack of congestion), the absence of an acknowledgement at the expected time generally means either the frame or the acknowledgement has been lost.

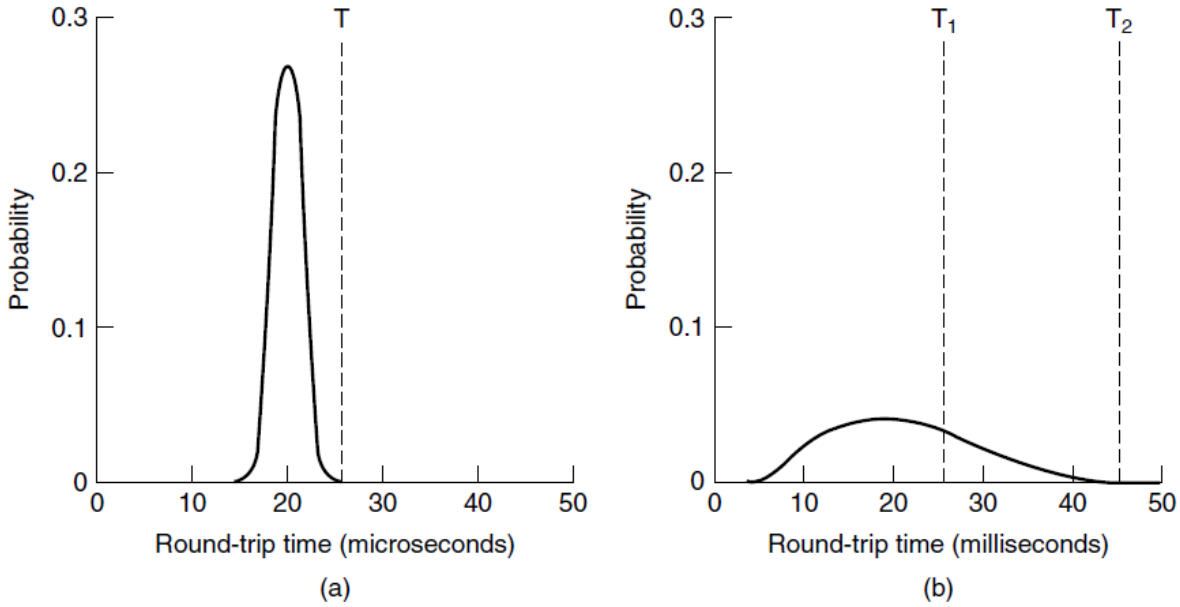


**Figure 6-42.** (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

TCP is faced with a radically different environment. The probability density function for the time it takes for a TCP acknowledgement to come back looks more like Fig. 6-42(b) than Fig. 6-42(a). It is larger and more variable. Determining the round-trip time to the destination is tricky. Even when it is known, deciding on the timeout interval is also difficult. If the timeout is set too short, say, *T*1 in Fig. 6-42(b), unnecessary retransmissions will occur, clogging the Internet with useless packets. If it is set too long (e.g., *T*2), performance will suffer due to the long retransmission delay whenever a packet is lost. Furthermore, the mean and variance of the acknowledgement arrival distribution can change rapidly within a few seconds as congestion builds up or is resolved. The solution is to use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance. The algorithm generally used by TCP is due to

Jacobson (1988) and works as follows. For each connection, TCP maintains a variable, *SRTT* (Smoothed Round-Trip Time), that is the best current estimate of the round-trip time to the destination in question. When a segment is sent, a timer is started, both to see how long the acknowledgement takes and also to trigger a retransmission if it takes too long. If the acknowledgement gets back before the timer expires, TCP measures how long the acknowledgement took, say, *R*. It then updates *SRTT* according to the formula

$$SRTT = \alpha\, SRTT + (1 - \alpha)\, R$$

where α is a smoothing factor that determines how quickly the old values are forgotten. Typically, α = 7/8. This kind of formula is an **EWMA (Exponentially Weighted Moving Average)** or low-pass filter that discards noise in the samples. Even given a good value of *SRTT*, choosing a suitable retransmission timeout is a nontrivial matter. Initial implementations of TCP used 2*xRTT*, but experience showed that a constant value was too inflexible because it failed to respond when the variance went up. In particular, queueing models of random (i.e., Poisson) traffic predict that when the load approaches capacity, the delay becomes large and highly variable. This can lead to the retransmission timer firing and a copy of the packet being retransmitted although the original packet is still transiting the network. It is all the more likely to happen under conditions of high load, which is the worst time at which to send additional packets into the network.

To fix this problem, Jacobson proposed making the timeout value sensitive to the variance in round-trip times as well as the smoothed round-trip time. This change requires keeping track of another smoothed variable, *RTTVAR* (Round- Trip Time VARiation) that is updated using the formula

$$RTTVAR = \beta\, RTTVAR + (1 - \beta)\, |SRTT - R|$$

This is an EWMA as before, and typically β = 3/4. The retransmission timeout, *RTO*, is set to be

*RTO = SRTT + 4 × RTTVAR (570)* The choice of the factor 4 is somewhat arbitrary, but multiplication by 4 can be done with a single shift, and less than 1% of all packets come in more than four standard deviations late. Note that *RTTVAR* is not exactly the same as the standard deviation (it is really the mean deviation), but it is close enough in practice. Jacobson's paper is full of clever tricks to compute timeouts using only integer adds, subtracts, and shifts. This economy is not needed for modern hosts, but it has become part of the culture that allows TCP to run on all manner of devices, from supercomputers down to tiny devices. So far nobody has put it on an RFID chip, but someday? Who knows.

The retransmission timer is also held to a minimum of 1 second, regardless of the estimates. This is a conservative value chosen to prevent spurious retransmissions based on measurements. One problem that occurs with gathering the samples, *R*, of the round-trip time is what to do when a segment times out and is sent again. When the acknowledgement comes in, it is unclear whether the acknowledgement refers to the first transmission or a later one. Guessing wrong can seriously contaminate the retransmission timeout. Phil Karn discovered this problem the hard way. Karn is

an amateur radio enthusiast interested in transmitting TCP/IP packets by ham radio, a notoriously unreliable medium. He made a simple proposal: do not update estimates on any segments that have been retransmitted. Additionally, the timeout is doubled on each successive retransmission until the segments get through the first time. This fix is called **Karn's algorithm.** Most TCP implementations use it. The retransmission timer is not the only timer TCP uses. A second timer is the **persistence timer**. It is designed to prevent the following deadlock. The receiver sends an acknowledgement with a window size of 0, telling the sender to wait. Later, the receiver updates the window, but the packet with the update is lost. Now the sender and the receiver are each waiting for the other to do something. When the persistence timer goes off, the sender transmits a probe to the receiver. The response to the probe gives the window size. If it is still 0, the persistence timer is set again and the cycle repeats. If it is nonzero, data can now be sent. A third timer that some implementations use is the **keepalive timer**. When a connection has been idle for a long time, the keepalive timer may go off to cause one side to check whether the other side is still there. If it fails to respond, the connection is terminated. This feature is controversial because it adds overhead and may terminate an otherwise healthy connection due to a transient network partition. The last timer used on each TCP connection is the one used in the *TIME WAIT* state while closing. It runs for twice the maximum packet lifetime to make sure that when a connection is closed, all packets created by it have died off.

## III] Application Layer:

1. **World Wide web and HTTP**
2. **Telnet**