18

TCP Connection Establishment and Termination

18.1 Introduction

TCP is a *connection-oriented* protocol. Before either end can send data to the other, a *connection* must be established between them. In this chapter we take a detailed look at how a TCP connection is established and later terminated.

This establishment of a connection between the two ends differs from a *connectionless* protocol such as UDP. We saw in Chapter 11 that with UDP one end just sends a datagram to the other end, without any preliminary handshaking.

18.2 Connection Establishment and Termination

To see what happens when a TCP connection is established and then terminated, we type the following command on the system `svr4`:

```
svr4 % telnet bsdi discard
Trying 140.252.13.35 ...
Connected to bsdi.
Escape character is '^]'.
^]                                  type Control, right bracket to talk to the Telnet client
telnet> quit                        terminate the connection
Connection closed.
```

The `telnet` command establishes a TCP connection with the host `bsdi` on the port corresponding to the discard service (Section 1.12). This is exactly the type of service we need to see what happens when a connection is established and terminated, without having the server initiate any data exchange.

tcpdump Output

Figure 18.1 shows the tcpdump output for the segments generated by this command.

```
1  0.0                  svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                  win 4096 <mss 1024>
2  0.002402 (0.0024)    bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                  ack 1415531522 win 4096
                                                  <mss 1024>
3  0.007224 (0.0048)    svr4.1037 > bsdi.discard: . ack 1823083522 win 4096
4  4.155441 (4.1482)    svr4.1037 > bsdi.discard: F 1415531522:1415531522(0)
                                                  ack 1823083522 win 4096
5  4.156747 (0.0013)    bsdi.discard > svr4.1037: . ack 1415531523 win 4096
6  4.158144 (0.0014)    bsdi.discard > svr4.1037: F 1823083522:1823083522(0)
                                                  ack 1415531523 win 4096
7  4.180662 (0.0225)    svr4.1037 > bsdi.discard: . ack 1823083523 win 4096
```

Figure 18.1 tcpdump output for TCP connection establishment and termination.

These seven TCP segments contain TCP headers only. No data is exchanged.

For TCP segments, each output line begins with

source > *destination: flags*

where *flags* represents four of the six flag bits in the TCP header (Figure 17.2). Figure 18.2 shows the five different characters that can appear in the *flags* output.

flag	3-character abbreviation	Description
S	SYN	synchronize sequence numbers
F	FIN	sender is finished sending data
R	RST	reset connection
P	PSH	push data to receiving process as soon as possible
.		none of above four flags is on

Figure 18.2 *flag* characters output by tcpdump for flag bits in TCP header.

In this example we see the S, F, and period. We'll see the other two *flags* (R and P) later. The other two TCP header flag bits—ACK and URG—are printed specially by tcpdump.

It's possible for more than one of the four flag bits in Figure 18.2 to be on in a single segment, but we normally see only one on at a time.

> RFC 1025 [Postel 1987], the *TCP and IP Bake Off*, calls a segment with the maximum combination of allowable flag bits turned on at once (SYN, URG, PSH, FIN, and 1 byte of data) a Kamikaze packet. It's also known as a nastygram, Christmas tree packet, and lamp test segment.

In line 1, the field `1415531521:1415531521(0)` means the sequence number of the packet was 1415531521 and the number of data bytes in the segment was 0. `tcpdump` displays this by printing the starting sequence number, a colon, the implied ending sequence number, and the number of data bytes in parentheses. The advantage of displaying both the sequence number and the implied ending sequence number is to see what the implied ending sequence number is, when the number of bytes is greater than 0. This field is output only if (1) the segment contains one or more bytes of data or (2) the SYN, FIN, or RST flag was on. Lines 1, 2, 4, and 6 in Figure 18.1 display this field because of the flag bits—we never exchange any data in this example.

In line 2 the field `ack 1415531522` shows the acknowledgment number. This is printed only if the ACK flag in the header is on.

The field `win 4096` in every line of output shows the window size being advertised by the sender. In these examples, where we are not exchanging any data, the window size never changes from its default of 4096. (We examine TCP's window size in Section 20.4.)

The final field that is output in Figure 18.1, `<mss 1024>` shows the *maximum segment size* (MSS) option specified by the sender. The sender does not want to receive TCP segments larger than this value. This is normally to avoid fragmentation (Section 11.5). We discuss the maximum segment size in Section 18.4, and show the format of the various TCP options in Section 18.10.

Time Line

Figure 18.3 shows the time line for this sequence of packets. (We described some general features of these time lines when we showed the first one in Figure 6.11, p. 80.) This figure shows which end is sending packets. We also expand some of the `tcpdump` output (e.g., printing SYN instead of `S`). In this time line we have also removed the window size values, since they add nothing to the discussion.

Connection Establishment Protocol

Now let's return to the details of the TCP protocol that are shown in Figure 18.3. To establish a TCP connection:

1. The requesting end (normally called the *client*) sends a SYN segment specifying the port number of the *server* that the client wants to connect to, and the client's *initial sequence number* (ISN, 1415531521 in this example). This is segment 1.
2. The server responds with its own SYN segment containing the server's initial sequence number (segment 2). The server also acknowledges the client's SYN by ACKing the client's ISN plus one. A SYN consumes one sequence number.
3. The client must acknowledge this SYN from the server by ACKing the server's ISN plus one (segment 3).

These three segments complete the connection establishment. This is often called the *three-way handshake*.

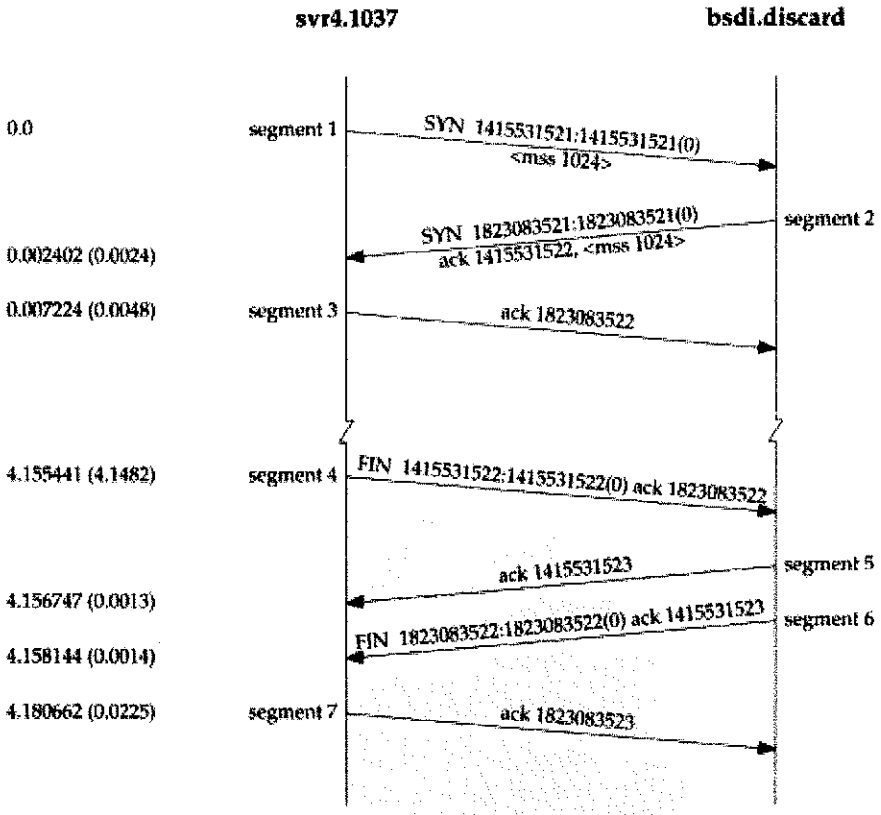

Figure 18.3 Time line of connection establishment and connection termination.

The side that sends the first SYN is said to perform an *active open*. The other side, which receives this SYN and sends the next SYN, performs a *passive open*. (In Section 18.8 we describe a simultaneous open where both sides can do an active open.)

When each end sends its SYN to establish the connection, it chooses an initial sequence number for that connection. The ISN should change over time, so that each connection has a different ISN. RFC 793 [Postel 1981c] specifies that the ISN should be viewed as a 32-bit counter that increments by one every 4 microseconds. The purpose in these sequence numbers is to prevent packets that get delayed in the network from being delivered later and then misinterpreted as part of an existing connection.

> How are the sequence numbers chosen? In 4.4BSD (and most Berkeley-derived implementations) when the system is initialized the initial send sequence number is initialized to 1. This practice violates the Host Requirements RFC. (A comment in the code acknowledges that this is wrong.) This variable is then incremented by 64,000 every half-second, and will cycle back to 0 about every 9.5 hours. (This corresponds to a counter that is incremented every 8

microseconds, not every 4 microseconds.) Additionally, each time a connection is established, this variable is incremented by 64,000.

The 4.1-second gap between segments 3 and 4 is the time between establishing the connection and typing the `quit` command to `telnet` to terminate the connection.

Connection Termination Protocol

While it takes three segments to establish a connection, it takes four to terminate a connection. This is caused by TCP's *half-close*. Since a TCP connection is full-duplex (that is, data can be flowing in each direction independently of the other direction), each direction must be shut down independently. The rule is that either end can send a FIN when it is done sending data. When a TCP receives a FIN, it must notify the application that the other end has terminated that direction of data flow. The sending of a FIN is normally the result of the application issuing a close.

The receipt of a FIN only means there will be no more data flowing in that direction. A TCP can still send data after receiving a FIN. While it's possible for an application to take advantage of this half-close, in practice few TCP applications use it. The normal scenario is what we show in Figure 18.3. We describe the half-close in more detail in Section 18.5.

We say that the end that first issues the close (e.g., sends the first FIN) performs the *active close* and the other end (that receives this FIN) performs the *passive close*. Normally one end does the active close and the other does the passive close, but we'll see in Section 18.9 how both ends can do an active close.

Segment 4 in Figure 18.3 initiates the termination of the connection and is sent when the Telnet client closes its connection. This happens when we type `quit`. This causes the client TCP to send a FIN, closing the flow of data from the client to the server.

When the server receives the FIN it sends back an ACK of the received sequence number plus one (segment 5). A FIN consumes a sequence number, just like a SYN. At this point the server's TCP also delivers an end-of-file to the application (the discard server). The server then closes its connection, causing its TCP to send a FIN (segment 6), which the client TCP must ACK by incrementing the received sequence number by one (segment 7).

Figure 18.4 shows the typical sequence of segments that we've described for the termination of a connection. We omit the sequence numbers. In this figure sending the FINs is caused by the applications closing their end of the connection, whereas the ACKs of these FINs are automatically generated by the TCP software.

Connections are normally initiated by the client, with the first SYN going from the client to the server. Either end can actively close the connection (i.e., send the first FIN). Often, however, it is the client that determines when the connection should be terminated, since client processes are often driven by an interactive user, who enters something like "quit" to terminate. In Figure 18.4 we can switch the labels at the top, calling the left side the server and the right side the client, and everything still works fine as shown. (The first example in Section 14.4, for example, shows the daytime server closing the connection.)

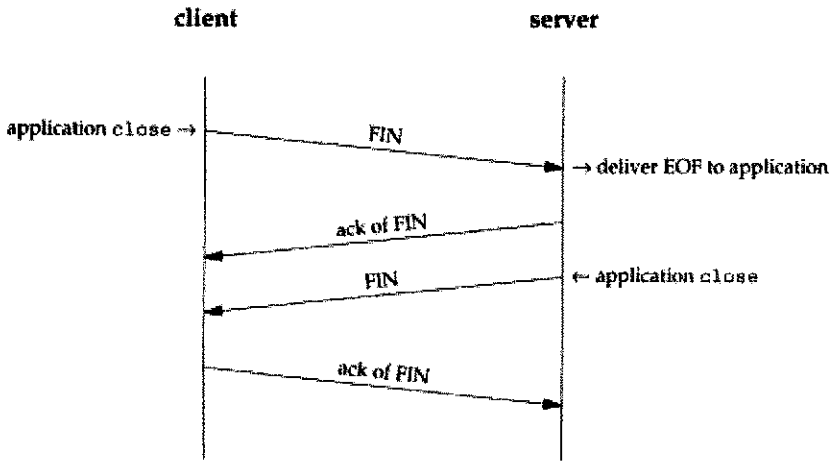

Figure 18.4 Normal exchange of segments during connection termination.

Normal tcpdump Output

Having to sort through all the huge sequence numbers is cumbersome, so the default `tcpdump` output shows the complete sequence numbers only on the SYN segments, and shows all following sequence numbers as relative offsets from the original sequence numbers. (To generate the output for Figure 18.1 we had to specify the `-S` option.) The normal `tcpdump` output corresponding to Figure 18.1 is shown in Figure 18.5.

```
1  0.0                  svr4.1037 > bsdi.discard: S 1415531521:1415531521(0)
                                                    win 4096 <mss 1024>
2  0.002402 (0.0024)    bsdi.discard > svr4.1037: S 1823083521:1823083521(0)
                                                    ack 1415531522
                                                    win 4096 <mss 1024>
3  0.007224 (0.0048)    svr4.1037 > bsdi.discard: . ack 1 win 4096
4  4.155441 (4.1482)    svr4.1037 > bsdi.discard: F 1:1(0) ack 1 win 4096
5  4.156747 (0.0013)    bsdi.discard > svr4.1037: . ack 2 win 4096
6  4.158144 (0.0014)    bsdi.discard > svr4.1037: F 1:1(0) ack 2 win 4096
7  4.180662 (0.0225)    svr4.1037 > bsdi.discard: . ack 2 win 4096
```

Figure 18.5 Normal `tcpdump` output for connection establishment and termination.

Unless we need to show the complete sequence numbers, we'll use this form of output in all following examples.

18.3 Timeout of Connection Establishment

There are several instances when the connection cannot be established. In one example the server host is down. To simulate this scenario we issue our `telnet` command after disconnecting the Ethernet cable from the server's host. Figure 18.6 shows the `tcpdump` output.

```
1   0.0                 bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                  win 4096 <mss 1024>
                                                  [tos 0x10]
2   5.814797 ( 5.8148)  bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                  win 4096 <mss 1024>
                                                  [tos 0x10]
3  29.815436 (24.0006)  bsdi.1024 > svr4.discard: S 291008001:291008001(0)
                                                  win 4096 <mss 1024>
                                                  [tos 0x10]
```

Figure 18.6 `tcpdump` output for connection establishment that times out.

The interesting point in this output is how frequently the client's TCP sends a SYN to try to establish the connection. The second segment is sent 5.8 seconds after the first, and the third is sent 24 seconds after the second.

As a side note, this example was run about 38 minutes after the client was rebooted. This corresponds with the initial sequence number of 291,008,001 (approximately $38 \times 60 \times 64000 \times 2$). Recall earlier in this chapter we said that typical Berkeley-derived systems initialize the initial sequence number to 1 and then increment it by 64,000 every half-second.

Also, this is the first TCP connection since the system was bootstrapped, which is why the client's port number is 1024.

What isn't shown in Figure 18.6 is how long the client's TCP keeps retransmitting before giving up. To see this we have to time the `telnet` command:

```
bsdi % date ; telnet svr4 discard ; date
Thu Sep 24 16:24:11 MST 1992
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection timed out
Thu Sep 24 16:25:27 MST 1992
```

The time difference is 76 seconds. Most Berkeley-derived systems set a time limit of 75 seconds on the establishment of a new connection. We'll see in Section 21.4 that the third packet sent by the client would have timed out around 16:25:29, 48 seconds after it was sent, had the client not given up after 75 seconds.

First Timeout Period

One puzzling item in Figure 18.6 is that the first timeout period, 5.8 seconds, is close to 6 seconds, but not exact, while the second period is almost exactly 24 seconds. Ten more

of these tests were run and the first timeout period took on various values between 5.59 seconds and 5.93 seconds. The second timeout period, however, was always 24.00 (to two decimal places).

What's happening here is that BSD implementations of TCP run a timer that goes off every 500 ms. This 500-ms timer is used for various TCP timeouts, all of which we cover in later chapters. When we type in the `telnet` command, an initial 6-second timer is established (12 clock ticks), but it may expire anywhere between 5.5 and 6 seconds in the future. Figure 18.7 shows what's happening.

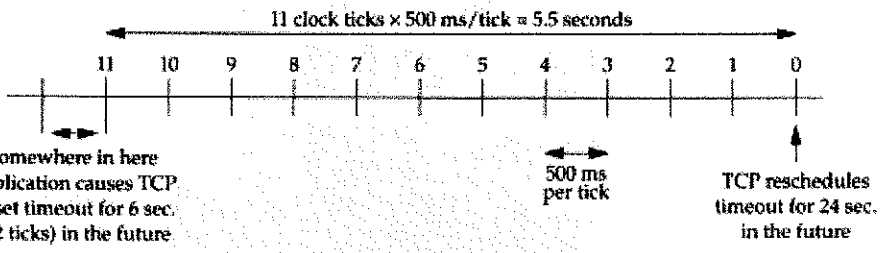

Figure 18.7 TCP 500-ms timer.

Although the timer is initialized to 12 ticks, the first decrement of the timer can occur between 0 and 500 ms after it is set. From that point on the timer is decremented about every 500 ms, but the first period can be variable. (We use the qualifier "about" because the time when TCP gets control every 500 ms can be preempted by other interrupts being handled by the kernel.)

When that 6-second timer expires at the tick labeled 0 in Figure 18.7, the timer is reset for 24 seconds (48 ticks) in the future. This next timer will be close to 24 seconds, since it was set at a time when the TCP's 500-ms timer handler was called by the kernel.

Type-of-Service Field

In Figure 18.6, the notation `[tos 0x10]` appears. This is the type-of-service (TOS) field in the IP datagram (Figure 3.2). The BSD/386 Telnet client sets the field for minimum delay.

18.4 Maximum Segment Size

The maximum segment size (MSS) is the largest "chunk" of data that TCP will send to the other end. When a connection is established, each end can announce its MSS. The values we've seen have all been 1024. The resulting IP datagram is normally 40 bytes larger: 20 bytes for the TCP header and 20 bytes for the IP header.

Some texts refer to this as a "negotiated" option. It is not negotiated in any way. When a connection is established, each end has the option of announcing the MSS it

expects to receive. (An MSS option can only appear in a SYN segment.) If one end does not receive an MSS option from the other end, a default of 536 bytes is assumed. (This default allows for a 20-byte IP header and a 20-byte TCP header to fit into a 576-byte IP datagram.)

In general, the larger the MSS the better, until fragmentation occurs. (This may not always be true. See Figures 24.3 and 24.4 for a counterexample.) A larger segment size allows more data to be sent in each segment, amortizing the cost of the IP and TCP headers. When TCP sends a SYN segment, either because a local application wants to initiate a connection, or when a connection request is received from another host, it can send an MSS value up to the outgoing interface's MTU, minus the size of the fixed TCP and IP headers. For an Ethernet this implies an MSS of up to 1460 bytes. Using IEEE 802.3 encapsulation (Section 2.2), the MSS could go up to 1452 bytes.

The values of 1024 that we've seen in this chapter, for connections involving BSD/386 and SVR4, are because many BSD implementations require the MSS to be a multiple of 512. Other systems, such as SunOS 4.1.3, Solaris 2.2, and AIX 3.2.2, all announce an MSS of 1460 when both ends are on a local Ethernet. Measurements in [Mogul 1993] show how an MSS of 1460 provides better performance on an Ethernet than an MSS of 1024.

If the destination IP address is "nonlocal," the MSS normally defaults to 536. While it's easy to say that a destination whose IP address has the same network ID and the same subnet ID as ours is local, and a destination whose IP address has a totally different network ID from ours is nonlocal, a destination with the same network ID but a different subnet ID could be either local or nonlocal. Most implementations provide a configuration option (Appendix E and Figure E.1) that lets the system administrator specify whether different subnets are local or nonlocal. The setting of this option determines whether the announced MSS is as large as possible (up to the outgoing interface's MTU) or the default of 536.

The MSS lets a host limit the size of datagrams that the other end sends it. When combined with the fact that a host can also limit the size of the datagrams that it sends, this lets a host avoid fragmentation when the host is connected to a network with a small MTU.

Consider our host `slip`, which has a SLIP link with an MTU of 296 to the router `bsdi`. Figure 18.8 shows these systems and the host `sun`.

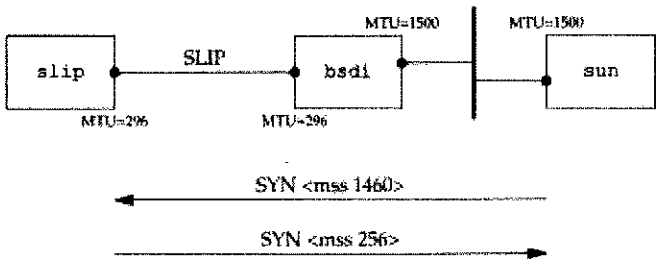

Figure 18.8 TCP connection from `sun` to `slip` showing MSS values.

We initiate a TCP connection from `sun` to `slip` and watch the segments using `tcpdump`. Figure 18.9 shows only the connection establishment (with the window size advertisements removed).

```
1  0.0                  sun.1093 > slip.discard: S 517312000:517312000(0)
                                                 <mss 1460>
2  0.10 (0.00)          slip.discard > sun.1093: S 509556225:509556225(0)
                                                 ack 517312001 <mss 256>
3  0.10 (0.00)          sun.1093 > slip.discard: . ack 1
```

Figure 18.9 `tcpdump` output for connection establishment from `sun` to `slip`.

The important fact here is that `sun` cannot send a segment with more than 256 bytes of data, since it received an MSS option of 256 (line 2). Furthermore, since `slip` knows that the outgoing interface's MTU is 296, even though `sun` announced an MSS of 1460, it will never send more than 256 bytes of data, to avoid fragmentation. It's OK for a system to send *less* than the MSS announced by the other end.

This avoidance of fragmentation works only if either host is directly connected to a network with an MTU of less than 576. If both hosts are connected to Ethernets, and both announce an MSS of 536, but an intermediate network has an MTU of 296, fragmentation will occur. The only way around this is to use the path MTU discovery mechanism (Section 24.2).

18.5 TCP Half-Close

TCP provides the ability for one end of a connection to terminate its output, while still receiving data from the other end. This is called a *half-close*. Few applications take advantage of this capability, as we mentioned earlier.

To use this feature the programming interface must provide a way for the application to say "I am done sending data, so send an end-of-file (FIN) to the other end, but I still want to receive data from the other end, until it sends me an end-of-file (FIN)."

> The sockets API supports the half-close, if the application calls `shutdown` with a second argument of 1, instead of calling `close`. Most applications, however, terminate both directions of the connection by calling `close`.

Figure 18.10 shows a typical scenario for a half-close. We show the client on the left side initiating the half-close, but either end can do this. The first two segments are the same: a FIN by the initiator, followed by an ACK of the FIN by the recipient. But it then changes from Figure 18.4, because the side that receives the half-close can still send data. We show only one data segment, followed by an ACK, but any number of data segments can be sent. (We talk more about the exchange of data segments and acknowledgments in Chapter 19.) When the end that received the half-close is done sending data, it closes its end of the connection, causing a FIN to be sent, and this delivers an end-of-file to the application that initiated the half-close. When this second FIN is acknowledged, the connection is completely closed.

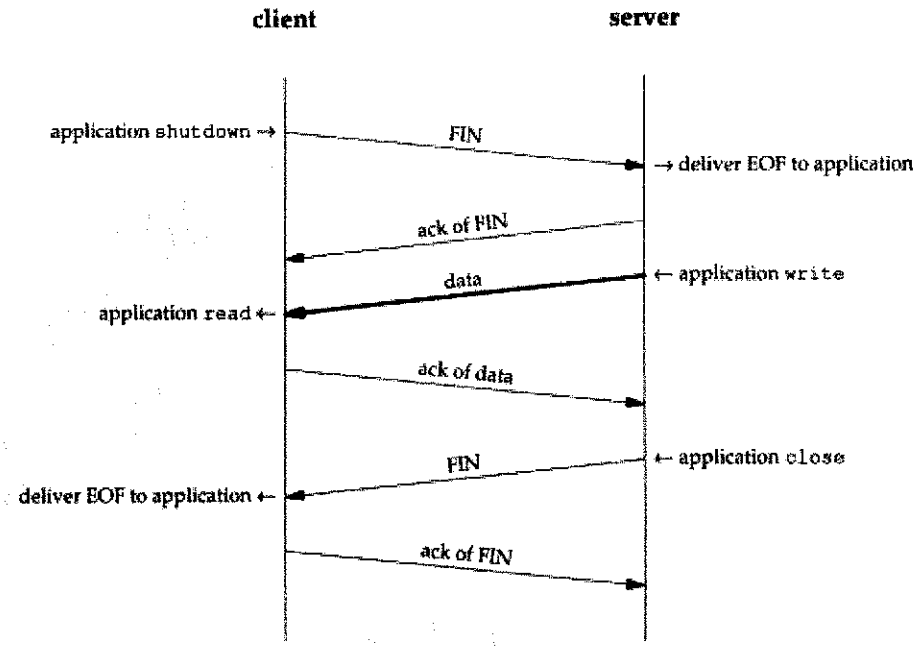

Figure 18.10 Example of TCP's half-close.

Why is there a half-close? One example is the Unix `rsh`(1) command, which executes a command on another system. The command

```
sun % rsh bsdi sort < datafile
```

executes the `sort` command on the host `bsdi` with standard input for the `rsh` command being read from the file named `datafile`. A TCP connection is created by `rsh` between itself and the program being executed on the other host. The operation of `rsh` is then simple: it copies standard input (`datafile`) to the connection, and copies from the connection to standard output (our terminal). Figure 18.11 shows the setup. (Remember that a TCP connection is full-duplex.)

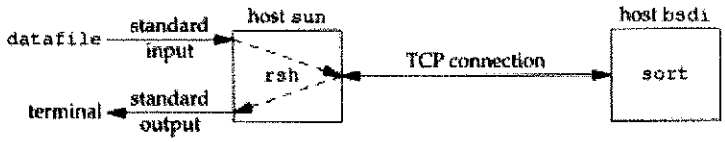

Figure 18.11 The command: `rsh bsdi sort < datafile`.

On the remote host `bsdi` the `rshd` server executes the `sort` program so that its standard input and standard output are both the TCP connection. Chapter 14 of [Stevens 1990] details the Unix process structure involved, but what concerns us here is the use of the TCP connection and the required use of TCP's half-close.

The `sort` program cannot generate any output until all of its input has been read. All the initial data across the connection is from the `rsh` client to the `sort` server, sending the file to be sorted. When the end-of-file is reached on the input (`datafile`), the `rsh` client performs a half-close on the TCP connection. The `sort` server then receives an end-of-file on its standard input (the TCP connection), sorts the file, and writes the result to its standard output (the TCP connection). The `rsh` client continues reading its end of the TCP connection, copying the sorted file to its standard output.

Without a half-close, some other technique is needed to let the client tell the server that the client is finished sending data, but still let the client receive data from the server. Two connections could be used as an alternative, but a single connection with a half-close is better.

18.6 TCP State Transition Diagram

We've described numerous rules regarding the initiation and termination of a TCP connection. These rules can be summarized in a state transition diagram, which we show in Figure 18.12.

The first thing to note in this diagram is that a subset of the state transitions is "typical." We've marked the normal client transitions with a darker solid arrow, and the normal server transitions with a darker dashed arrow.

Next, the two transitions leading to the ESTABLISHED state correspond to opening a connection, and the two transitions leading from the ESTABLISHED state are for the termination of a connection. The ESTABLISHED state is where data transfer can occur between the two ends in both directions. Later chapters describe what happens in this state.

We've collected the four boxes in the lower left of this diagram within a dashed box and labeled it "active close." Two other boxes (CLOSE_WAIT and LAST_ACK) are collected in a dashed box with the label "passive close."

The names of the 11 states (CLOSED, LISTEN, SYN_SENT, etc.) in this figure were purposely chosen to be identical to the states output by the `netstat` command. The `netstat` names, in turn, are almost identical to the names originally described in RFC 793. The state CLOSED is not really a state, but is the imaginary starting point and ending point for the diagram.

The state transition from LISTEN to SYN_SENT is legal but is not supported in Berkeley-derived implementations.

The transition from SYN_RCVD back to LISTEN is valid only if the SYN_RCVD state was entered from the LISTEN state (the normal scenario), not from the SYN_SENT state (a simultaneous open). This means if we perform a passive open (enter LISTEN), receive a SYN, send a SYN with an ACK (enter SYN_RCVD), and then receive a reset instead of an ACK, the end point returns to the LISTEN state and waits for another connection request to arrive.

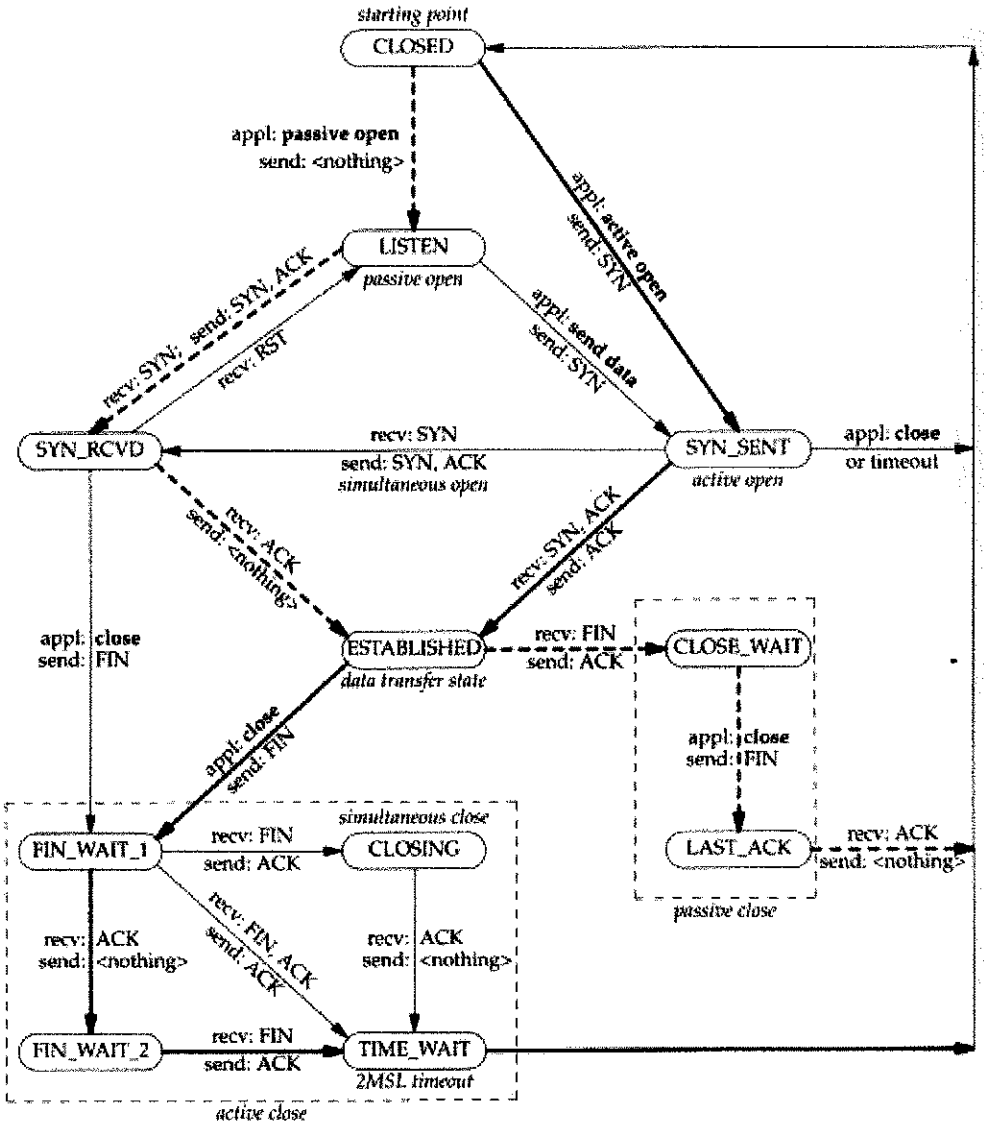

Figure 18.12 TCP state transition diagram.

Figure 18.13 shows the normal TCP connection establishment and termination, detailing the different states through which the client and server pass. It is a redo of Figure 18.3 showing only the states.

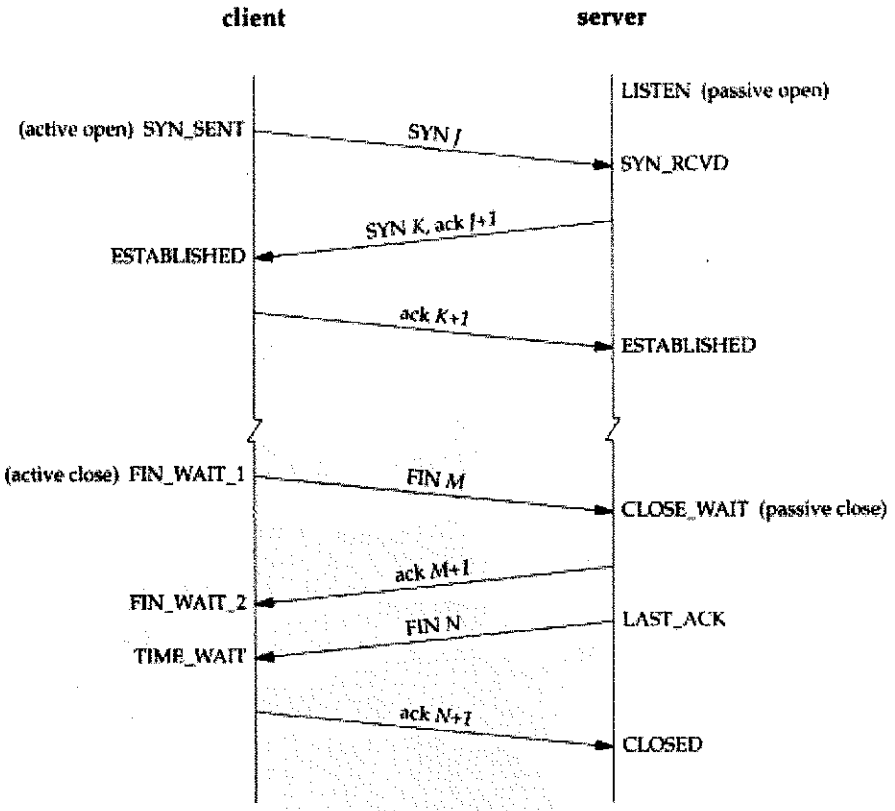

Figure 18.13 TCP states corresponding to normal connection establishment and termination.

We assume in Figure 18.13 that the client on the left side does an active open, and the server on the right side does a passive open. Although we show the client doing the active close, as we mentioned earlier, either side can do the active close.

You should follow through the state changes in Figure 18.13 using the state transition diagram in Figure 18.12, making certain you understand why each state change takes place.

2MSL Wait State

The TIME_WAIT state is also called the 2MSL wait state. Every implementation must choose a value for the *maximum segment lifetime* (MSL). It is the maximum amount of

time any segment can exist in the network before being discarded. We know this time limit is bounded, since TCP segments are transmitted as IP datagrams, and the IP datagram has the TTL field that limits its lifetime.

> RFC 793 [Postel 1981c] specifies the MSL as 2 minutes. Common implementation values, however, are 30 seconds, 1 minute, or 2 minutes.

Recall from Chapter 8 that the real-world limit on the lifetime of the IP datagram is based on the number of hops, not a timer.

Given the MSL value for an implementation, the rule is: when TCP performs an active close, and sends the final ACK, that connection must stay in the TIME_WAIT state for twice the MSL. This lets TCP resend the final ACK in case this ACK is lost (in which case the other end will time out and retransmit its final FIN).

Another effect of this 2MSL wait is that while the TCP connection is in the 2MSL wait, the socket pair defining that connection (client IP address, client port number, server IP address, and server port number) cannot be reused. That connection can only be reused when the 2MSL wait is over.

Unfortunately most implementations (i.e., the Berkeley-derived ones) impose a more stringent constraint. By default a local port number cannot be reused while that port number is the local port number of a socket pair that is in the 2MSL wait. We'll see examples of this common constraint below.

> Some implementations and APIs provide a way to bypass this restriction. With the sockets API, the `SO_REUSEADDR` socket option can be specified. It lets the caller assign itself a local port number that's in the 2MSL wait, but we'll see that the rules of TCP still prevent this port number from being part of a connection that is in the 2MSL wait.

Any delayed segments that arrive for a connection while it is in the 2MSL wait are discarded. Since the connection defined by the socket pair in the 2MSL wait cannot be reused during this time period, when we do establish a valid connection we know that delayed segments from an earlier incarnation of this connection cannot be misinterpreted as being part of the new connection. (A connection is defined by a socket pair. New instances of a connection are called *incarnations* of that connection.)

As we said with Figure 18.13, it is normally the client that does the active close and enters the TIME_WAIT state. The server usually does the passive close, and does not go through the TIME_WAIT state. The implication is that if we terminate a client, and restart the same client immediately, that new client cannot reuse the same local port number. This isn't a problem, since clients normally use ephemeral ports, and don't care what the local ephemeral port number is.

With servers, however, this changes, since servers use well-known ports. If we terminate a server that has a connection established, and immediately try to restart the server, the server cannot assign its well-known port number to its end point, since that port number is part of a connection that is in a 2MSL wait. It may take from 1 to 4 minutes before the server can be restarted.

We can see this scenario using our `sock` program. We start the server, connect to it from a client, and then terminate the server:

```
sun % sock -v -s 6666                    start as server, listening on port 6666
                                         (execute client on bsdi that connects to this port)
connection on 140.252.13.33.6666 from 140.252.13.35.1081
^?                                       then type interrupt key to terminate server
sun % sock -s 6666                       and immediately try to restart server on same port
can't bind local address: Address already in use
sun % netstat                            let's check the state of the connection
Active Internet connections
Proto Recv-Q Send-Q  Local Address      Foreign Address      (state)
tcp        0      0  sun.6666          bsdi.1081          TIME_WAIT
                                         many more lines that are deleted
```

When we try to restart the server, the program outputs an error message indicating it cannot bind its well-known port number, because it's already in use (i.e., it's in a 2MSL wait).

We then immediately execute `netstat` to see the state of the connection, and verify that it is indeed in the TIME_WAIT state.

> If we continually try to restart the server, and measure the time until it succeeds, we can measure the 2MSL value. On SunOS 4.1.3, SVR4, BSD/386, and AIX 3.2.2, it takes 1 minute to restart the server, meaning the MSL is 30 seconds. Under Solaris 2.2 it takes 4 minutes to restart the server, implying an MSL of 2 minutes.

We can see the same error from a client, if the client tries to allocate a port that is part of a connection in the 2MSL wait (something clients normally don't do):

```
sun % sock -v bsdi echo                  start as client, connect to echo server
connected on 140.252.13.33.1162 to 140.252.13.35.7
hello there                              type this line
hello there                              and it's echoed by the server
^D                                       type end-of-file character to terminate client
sun % sock -b1162 bsdi echo
can't bind local address: Address already in use
```

The first time we execute the client we specify the `-v` option to see what the local port number is (1162). The second time we execute the client we specify the `-b` option, telling the client to assign itself 1162 as its local port number. As we expect, the client can't do this, since that port number is part of a connection that is in a 2MSL wait.

We need to reemphasize one effect of the 2MSL wait because we'll encounter it in Chapter 27 with FTP, the File Transfer Protocol. As we said earlier, it is a socket pair (that is, the 4-tuple consisting of a local IP address, local port, remote IP address and remote port) that remains in the 2MSL wait. Although many implementations allow a process to reuse a port number that is part of a connection that is in the 2MSL wait (normally with an option named `SO_REUSEADDR`), TCP *cannot* allow a new connection to be created with the same socket pair. We can see this with the following experiment:

```
sun % sock -v -s 6666                    start as server, listening on port 6666
                                         (execute client on bsdi that connects to this port)
```

```
connection on 140.252.13.33.6666 from 140.252.13.35.1098
^?                                          then type interrupt key to terminate server

sun % sock -b6666 bsdi 1098                 try to start as client with local port 6666
can't bind local address: Address already in use

sun % sock -A -b6666 bsdi 1098              try again, this time with -A option
active open error: Address already in use
```

The first time we run our `sock` program, we run it as a server on port 6666 and connect to it from a client on the host `bsdi`. The client's ephemeral port number is 1098. We terminate the server so it does the active close. This causes the 4-tuple of 140.252.13.33 (local IP address), 6666 (local port number), 140.252.13.35 (foreign IP address), and 1098 (foreign port number) to enter the 2MSL wait on the server host.

The second time we run the program, we run it as a client and try to specify the local port number as 6666 and connect to host `bsdi` on port 1098. But the program gets an error when it tries to assign itself the local port number of 6666, because that port number is part of the 4-tuple that is in the 2MSL wait state.

To try and get around this error we run the program again, specifying the `-A` option, which enables the `SO_REUSEADDR` option that we mentioned. This lets the program assign itself the port number 6666, but we then get an error when it tries to issue the active open. Even though it can assign itself the port number 6666, it cannot create a connection to port 1098 on the host `bsdi`, because the socket pair defining that connection is in the 2MSL wait state.

What if we try to establish the connection from the other host? First we must restart the server on `sun` with the `-A` flag, since the local port it needs (6666) is part of a connection that is in the 2MSL wait:

```
sun % sock -A -s 6666                       start as server, listening on port 6666
```

Then, before the 2MSL wait is over on `sun`, we start the client on `bsdi`:

```
bsdi % sock -b1098 sun 6666
connected on 140.252.13.35.1098 to 140.252.13.33.6666
```

Unfortunately it works! This is a violation of the TCP specification, but is supported by most Berkeley-derived implementations. These implementations allow a new connection request to arrive for a connection that is in the TIME_WAIT state, if the new sequence number is greater than the final sequence number from the previous incarnation of this connection. In this case the ISN for the new incarnation is set to the final sequence number from the previous incarnation plus 128,000. The appendix of RFC 1185 [Jacobson, Braden, and Zhang 1990] shows the pitfalls still possible with this technique.

This implementation feature lets a client and server continually reuse the same port number at each end for successive incarnations of the same connection, but only if the server does the active close. We'll see another example of this 2MSL wait condition in Figure 27.8, with FTP. See Exercise 18.5 also.

Quiet Time Concept

The 2MSL wait provides protection against delayed segments from an earlier incarnation of a connection from being interpreted as part of a new connection that uses the same local and foreign IP addresses and port numbers. But this works only if a host with connections in the 2MSL wait does not crash.

What if a host with ports in the 2MSL wait crashes, reboots within MSL seconds, and immediately establishes new connections using the same local and foreign IP addresses and port numbers corresponding to the local ports that were in the 2MSL wait before the crash? In this scenario, delayed segments from the connections that existed before the crash can be misinterpreted as belonging to the new connections created after the reboot. This can happen regardless of how the initial sequence number is chosen after the reboot.

To protect against this scenario, RFC 793 states that TCP should not create any connections for MSL seconds after rebooting. This is called the *quiet time*.

> Few implementations abide by this since most hosts take longer than MSL seconds to reboot after a crash.

FIN_WAIT_2 State

In the FIN_WAIT_2 state we have sent our FIN and the other end has acknowledged it. Unless we have done a half-close, we are waiting for the application on the other end to recognize that it has received an end-of-file notification and close its end of the connection, which sends us a FIN. Only when the process at the other end does this close will our end move from the FIN_WAIT_2 to the TIME_WAIT state.

This means our end of the connection can remain in this state forever. The other end is still in the CLOSE_WAIT state, and can remain there forever, until the application decides to issue its close.

> Many Berkeley-derived implementations prevent this infinite wait in the FIN_WAIT_2 state as follows. If the application that does the active close does a complete close, not a half-close indicating that it expects to receive data, then a timer is set. If the connection is idle for 10 minutes plus 75 seconds, TCP moves the connection into the CLOSED state. A comment in the code acknowledges that this implementation feature violates the protocol specification.

18.7 Reset Segments

We've mentioned a bit in the TCP header named RST for "reset." In general, a reset is sent by TCP whenever a segment arrives that doesn't appear correct for the referenced connection. (We use the term "referenced connection" to mean the connection specified by the destination IP address and port number, and the source IP address and port number. This is what RFC 793 calls a socket.)

Connection Request to Nonexistent Port

A common case for generating a reset is when a connection request arrives and no process is listening on the destination port. In the case of UDP, we saw in Section 6.5 that an ICMP port unreachable was generated when a datagram arrived for a destination port that was not in use. TCP uses a reset instead.

This example is trivial to generate—we use the Telnet client and specify a port number that's not in use on the destination:

```
bsdi % telnet svr4 20000              port 20000 should not be in use
Trying 140.252.13.34...
telnet: Unable to connect to remote host: Connection refused
```

This error message is output by the Telnet client immediately. Figure 18.14 shows the packet exchange corresponding to this command.

```
1  0.0                    bsdi.1087 > svr4.20000: S 297416193:297416193(0)
                                                   win 4096 <mss 1024>
                                                   [tos 0x10]

2  0.003771 (0.0038)      svr4.20000 > bsdi.1087: R 0:0(0) ack 297416194 win 0
```

Figure 18.14 Reset generated by attempt to open connection to nonexistent port.

The values we need to examine in this figure are the sequence number field and acknowledgment number field in the reset. Because the ACK bit was not on in the arriving segment, the sequence number of the reset is set to 0 and the acknowledgment number is set to the incoming ISN plus the number of data bytes in the segment. Although there is no real data in the arriving segment, the SYN bit logically occupies 1 byte of sequence number space; therefore, in this example the acknowledgment number in the reset is set to the ISN, plus the data length (0), plus one for the SYN bit.

Aborting a Connection

We saw in Section 18.2 that the normal way to terminate a connection is for one side to send a FIN. This is sometimes called an *orderly release* since the FIN is sent after all previously queued data has been sent, and there is normally no loss of data. But it's also possible to abort a connection by sending a reset instead of a FIN. This is sometimes called an *abortive release*.

Aborting a connection provides two features to the application: (1) any queued data is thrown away and the reset is sent immediately, and (2) the receiver of the RST can tell that the other end did an abort instead of a normal close. The API being used by the application must provide a way to generate the abort instead of a normal close.

We can watch this abort sequence happen using our `sock` program. The sockets API provides this capability by using the "linger on close" socket option (`SO_LINGER`). We specify the `-L` option with a linger time of 0. This causes the abort to be sent when the connection is closed, instead of the normal FIN. We'll connect to a server version of our `sock` program on `svr4` and type one line of input:

```
bsdi % sock -L0 svr4 8888          this is the client; server shown later
hello, world                       type one line of input that's sent to other end
^D                                 type end-of-file character to terminate client
```

Figure 18.15 shows the `tcpdump` output for this example. (We have deleted all the window advertisements in this figure, since they add nothing to the discussion.)

```
1  0.0                  bsdi.1099 > svr4.8888: S 671112193:671112193(0)
                                                 <mss 1024>
2  0.004975 (0.0050)    svr4.8888 > bsdi.1099: S 3224959489:3224959489(0)
                                                 ack 671112194 <mss 1024>
3  0.006656 (0.0017)    bsdi.1099 > svr4.8888: . ack 1
4  4.833073 (4.8264)    bsdi.1099 > svr4.8888: P 1:14(13) ack 1
5  5.026224 (0.1932)    svr4.8888 > bsdi.1099: . ack 14
6  9.527634 (4.5014)    bsdi.1099 > svr4.8888: R 14:14(0) ack 1
```

Figure 18.15 Aborting a connection with a reset (RST) instead of a FIN.

Lines 1–3 show the normal connection establishment. Line 4 sends the data line that we typed (12 characters plus the Unix newline character), and line 5 is the acknowledgment of the received data.

Line 6 corresponds to our typing the end-of-file character (Control-D) to terminate the client. Since we specified an abort instead of a normal close (the `-L0` command-line option), the TCP on `bsdi` sends an RST instead of the normal FIN. The RST segment contains a sequence number and acknowledgment number. Also notice that the RST segment elicits no response from the other end—it is not acknowledged at all. The receiver of the reset aborts the connection and advises the application that the connection was reset.

We get the following error on the server for this exchange:

```
svr4 % sock -s 8888                run as server, listen on port 8888
hello, world                       this is what the client sent over
read error: Connection reset by peer
```

This server reads from the network and copies whatever it receives to standard output. It normally ends by receiving an end-of-file notification from its TCP, but here we see that it receives an error when the RST arrives. The error is what we expect: the connection was reset by the peer.

Detecting Half-Open Connections

A TCP connection is said to be *half-open* if one end has closed or aborted the connection without the knowledge of the other end. This can happen any time one of the two hosts crashes. As long as there is no attempt to transfer data across a half-open connection, the end that's still up won't detect that the other end has crashed.

Another common cause of a half-open connection is when a client host is powered off, instead of terminating the client application and then shutting down the client host. This happens when PCs are being used to run Telnet clients, for example, and the users

power off the PC at the end of the day. If there was no data transfer going on when the PC was powered off, the server will never know that the client disappeared. When the user comes in the next morning, powers on the PC, and starts a new Telnet client, a new occurrence of the server is started on the server host. This can lead to many half-open TCP connections on the server host. (In Chapter 23 we'll see a way for one end of a TCP connection to discover that the other end has disappeared using TCP's keepalive option.)

We can easily create a half-open connection. We'll execute the Telnet client on `bsdi`, connecting to the discard server on `svr4`. We type one line of input, and watch it go across with `tcpdump`, and then disconnect the Ethernet cable on the server's host, and reboot the server host. This simulates the server host crashing. (We disconnect the Ethernet cable before rebooting the server to prevent it from sending a FIN out the open connections, which some TCPs do when they are shut down.) After the server has rebooted, we reconnect the cable, and try to send another line from the client to the server. Since the server's TCP has rebooted, and lost all memory of the connections that existed before it was rebooted, it knows nothing about the connection that the data segment references. The rule of TCP is that the receiver responds with a reset.

```
bsdi % telnet svr4 discard              start the client
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
hi there                                this line is sent OK
                                        here is where we reboot the server host
another line                            and this one elicits a reset
Connection closed by foreign host.
```

Figure 18.16 shows the `tcpdump` output for this example. (We have removed from this output the window advertisements, the type-of-service information, and the MSS announcements, since they add nothing to the discussion.)

```
1    0.0                  bsdi.1102 > svr4.discard: S 1591752193:1591752193(0)
2    0.004811 (  0.0048)  svr4.discard > bsdi.1102: S 26368001:26368001(0)
                                                     ack 1591752194
3    0.006516 (  0.0017)  bsdi.1102 > svr4.discard: . ack 1

4    5.167679 (  5.1612)  bsdi.1102 > svr4.discard: P 1:11(10) ack 1
5    5.201662 (  0.0340)  svr4.discard > bsdi.1102: . ack 11

6  194.909929 (189.7083)  bsdi.1102 > svr4.discard: P 11:25(14) ack 1
7  194.914957 (  0.0050)  arp who-has bsdi tell svr4
8  194.915678 (  0.0007)  arp reply bsdi is-at 0:0:c0:6f:2d:40
9  194.918225 (  0.0025)  svr4.discard > bsdi.1102: R 26368002:26368002(0)
```

Figure 18.16 Reset in response to data segment on a half-open connection.

Lines 1–3 are the normal connection establishment. Line 4 sends the line "hi there" to the discard server, and line 5 is the acknowledgment.

At this point we disconnect the Ethernet cable from `svr4`, reboot it, and reconnect the cable. This takes almost 190 seconds. We then type the next line of input to the client ("another line") and when we type the return key the line is sent to the server

(line 6 in Figure 18.16). This elicits a response from the server, but note that since the server was rebooted, its ARP cache is empty, so an ARP request and reply are required (lines 7 and 8). Then the reset is sent in line 9. The client receives the reset and outputs that the connection was terminated by the foreign host. (The final message output by the Telnet client is not as informative as it could be.)

18.8 Simultaneous Open

It is possible, although improbable, for two applications to both perform an active open to each other at the same time. Each end must transmit a SYN, and the SYNs must pass each other on the network. It also requires each end to have a local port number that is well known to the other end. This is called a *simultaneous open*.

For example, one application on host A could have a local port of 7777 and perform an active open to port 8888 on host B. The application on host B would have a local port of 8888 and perform an active open to port 7777 on host A.

This is *not* the same as connecting a Telnet client on host A to the Telnet server on host B, at the same time that a Telnet client on host B is connecting to the Telnet server on host A. In this Telnet scenario, both Telnet servers perform passive opens, not active opens, and the Telnet clients assign themselves an ephemeral port number, not a port number that is well known to the other Telnet server.

TCP was purposely designed to handle simultaneous opens and the rule is that only one connection results from this, not two connections. (Other protocol suites, notably the OSI transport layer, create two connections in this scenario, not one.)

When a simultaneous open occurs the state transitions differ from those shown in Figure 18.13. Both ends send a SYN at about the same time, entering the SYN_SENT state. When each end receives the SYN, the state changes to SYN_RCVD (Figure 18.12), and each end resends the SYN and acknowledges the received SYN. When each end receives the SYN plus the ACK, the state changes to ESTABLISHED. These state changes are summarized in Figure 18.17.

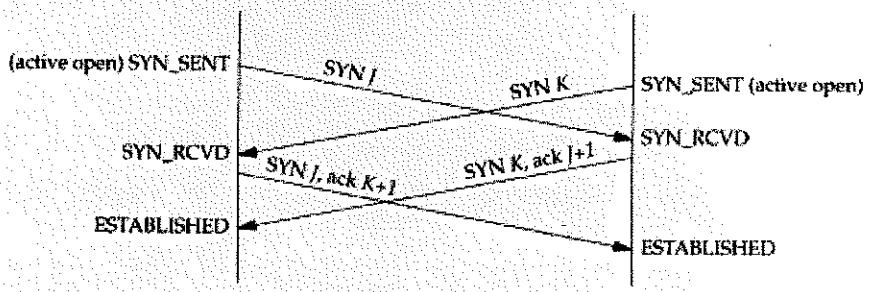

Figure 18.17 Segments exchanged during simultaneous open.

A simultaneous open requires the exchange of four segments, one more than the normal three-way handshake. Also notice that we don't call either end a client or a server, because both ends act as client and server.

An Example

It is possible, though hard, to generate a simultaneous open. The two ends must be started at about the same time, so that the SYNs cross each other. Having a long round-trip time between the two ends helps, to let the SYNs cross. To do this we'll execute one end on our host `bsdi`, and the other end on the host `vangogh.cs.berkeley.edu`. Since there is a dialup SLIP link between them, the round-trip time should be long enough (a few hundred milliseconds) to let the SYNs cross.

One end (`bsdi`) assigns itself a local port of 8888 (the `-b` command-line option) and performs an active open to port 7777 on the other host:

```
bsdi % sock -v -b8888 vangogh.cs.berkeley.edu 7777
connected on 140.252.13.35.8888 to 128.32.130.2.7777
TCP_MAXSEG = 512
hello, world                    we type this line
and hi there                    this line was typed on other end
connection closed by peer       this is output when FIN received
```

The other end is started at about the same time, assigns itself a local port of 7777, and performs an active open to port 8888:

```
vangogh % sock -v -b7777 bsdi.tuc.noao.edu 8888
connected on 128.32.130.2.7777 to 140.252.13.35.8888
TCP_MAXSEG = 512
hello, world                    this is typed on the other end
and hi there                    we type this line
^D                              and then type our EOF character
```

We specify the `-v` flag to our `sock` program to verify the IP address and port numbers on each end of the connection. This flag also prints the MSS used by each end of the connection. We also type in one line on each end, which is sent to the other end and printed, to verify that both ends are indeed talking to each other.

Figure 18.18 shows the exchange of segments across the connection. (We have deleted some new TCP options that appear in the original SYN from `vangogh`, a 4.4BSD system. We describe these newer options in Section 18.10.) Notice the two SYNs (lines 1 and 2) followed by the two SYNs with ACKs (lines 3 and 4). These perform the simultaneous open.

Line 5 shows the input line "hello, world" going from `bsdi` to `vangogh`, with the acknowledgment in line 6. Lines 7 and 8 correspond to the line "and hi there" going in the other direction. Lines 9–12 show the normal connection termination.

Many Berkeley-derived implementations do not support the simultaneous open correctly. On these systems, if you can get the SYNs to cross, you end up with an infinite exchange of segments, each with a SYN and an ACK, in each direction. The transition from the SYN_SENT state to the SYN_RCVD state in Figure 18.12 is not always tested in many implementations.

```
1   0.0                 bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                  win 4096 <mss 512>
2   0.213782 (0.2138)   vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                  win 8192 <mss 512>
3   0.215399 (0.0016)   bsdi.8888 > vangogh.7777: S 91904001:91904001(0)
                                                  ack 1058199042 win 4096
                                                  <mss 512>
4   0.340405 (0.1250)   vangogh.7777 > bsdi.8888: S 1058199041:1058199041(0)
                                                  ack 91904002 win 8192
                                                  <mss 512>
5   5.633142 (5.2927)   bsdi.8888 > vangogh.7777: P 1:14(13) ack 1 win 4096
6   6.100366 (0.4672)   vangogh.7777 > bsdi.8888: . ack 14 win 8192
7   9.640214 (3.5398)   vangogh.7777 > bsdi.8888: P 1:14(13) ack 14 win 8192
8   9.796417 (0.1562)   bsdi.8888 > vangogh.7777: . ack 14 win 4096
9   13.060395 (3.2640)  vangogh.7777 > bsdi.8888: F 14:14(0) ack 14 win 8192
10  13.061828 (0.0014)  bsdi.8888 > vangogh.7777: . ack 15 win 4096
11  13.079769 (0.0179)  bsdi.8888 > vangogh.7777: F 14:14(0) ack 15 win 4096
12  13.299940 (0.2202)  vangogh.7777 > bsdi.8888: . ack 15 win 8192
```

Figure 18.18 Exchange of segments during simultaneous open.

18.9 Simultaneous Close

We said earlier that one side (often, but not always, the client) performs the active close, causing the first FIN to be sent. It's also possible for both sides to perform an active close, and the TCP protocol allows for this *simultaneous close*.

In terms of Figure 18.12, both ends go from ESTABLISHED to FIN_WAIT_1 when the application issues the close. This causes both FINs to be sent, and they probably pass each other somewhere in the network. When the FIN is received, each end transitions from FIN_WAIT_1 to the CLOSING state, and each state sends its final ACK. When each end receives the final ACK, the state changes to TIME_WAIT. Figure 18.19 summarizes these state changes.

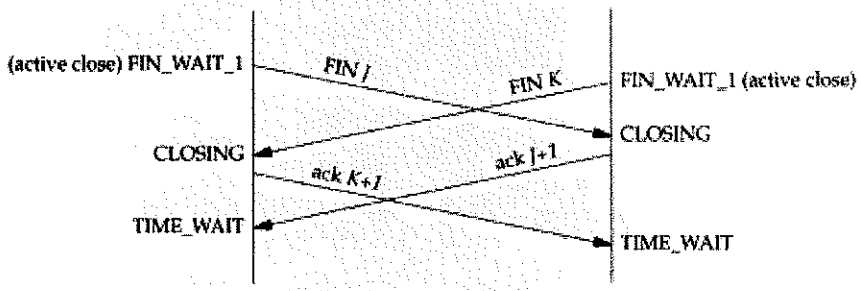

Figure 18.19 Segments exchanged during simultaneous close.

With a simultaneous close the same number of segments are exchanged as in the normal close.

18.10 TCP Options

The TCP header can contain options (Figure 17.2). The only options defined in the original TCP specification are the end of option list, no operation, and the maximum segment size option. We have seen the MSS option in almost every SYN segment in our examples.

Newer RFCs, specifically RFC 1323 [Jacobson, Braden, and Borman 1992], define additional TCP options, most of which are found only in the latest implementations. (We describe these new options in Chapter 24.) Figure 18.20 shows the format of the current TCP options—those from RFC 793 and RFC 1323.

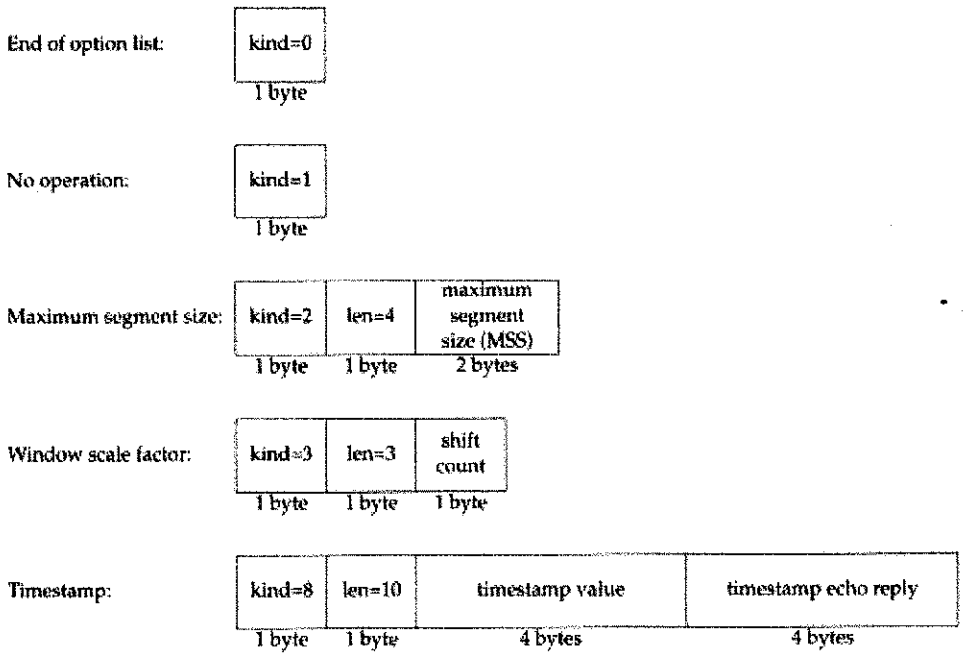

Figure 18.20 TCP options.

Every option begins with a 1-byte *kind* that specifies the type of option. The options with a *kind* of 0 and 1 occupy a single byte. The other options have a *len* byte that follows the *kind* byte. The length is the total length, including the *kind* and *len* bytes.

The reason for the no operation (NOP) option is to allow the sender to pad fields to a multiple of 4 bytes. If we initiate a TCP connection from a 4.4BSD system, the following TCP options are output by `tcpdump` on the initial SYN segment:

```
<mss 512,nop,wscale 0,nop,nop,timestamp 146647 0>
```

The MSS option is set to 512, followed by a NOP, followed by the window scale option. The reason for the first NOP is to pad the 3-byte window scale option to a 4-byte

boundary. Similarly, the 10-byte timestamp option is preceded by two NOPs, to occupy 12 bytes, placing the two 4-byte timestamps onto 4-byte boundaries.

Four other options have been proposed, with *kinds* of 4, 5, 6, and 7 called the selective-ACK and echo options. We don't show them in Figure 18.20 because the echo options have been replaced with the timestamp option, and selective ACKs, as currently defined, are still under discussion and were not included in RFC 1323. Also, the T/TCP proposal for TCP transactions (Section 24.7) specifies three options with *kinds* of 11, 12, and 13.

18.11 TCP Server Design

We said in Section 1.8 that most TCP servers are concurrent. When a new connection request arrives at a server, the server accepts the connection and invokes a new process to handle the new client. Depending on the operating system, various techniques are used to invoke the new server. Under Unix the common technique is to create a new process using the `fork` function. Lightweight processes (threads) can also be used, if supported.

What we're interested in is the interaction of TCP with concurrent servers. We need to answer the following questions: how are the port numbers handled when a server accepts a new connection request from a client, and what happens if multiple connection requests arrive at about the same time?

TCP Server Port Numbers

We can see how TCP handles the port numbers by watching any TCP server. We'll watch the Telnet server using the `netstat` command. The following output is on a system with no active Telnet connections. (We have deleted all the lines except the one showing the Telnet server.)

```
sun % netstat -a -n -f inet
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  *.23                   *.*                    LISTEN
```

The `-a` flag reports on all network end points, not just those that are ESTABLISHED. The `-n` flag prints IP addresses as dotted-decimal numbers, instead of trying to use the DNS to convert the address to a name, and prints numeric port numbers (e.g., 23) instead of service names (e.g., Telnet). The `-f inet` option reports only TCP and UDP end points.

The local address is output as `*.23`, where the asterisk is normally called the *wildcard* character. This means that an incoming connection request (i.e., a SYN) will be accepted on any local interface. If the host were multihomed, we could specify a single IP address for the local IP address (one of the host's IP addresses), and only connections received on that interface would be accepted. (We'll see an example of this later in this section.) The local port is 23, the well-known port number for Telnet.

The foreign address is output as `*.*`, which means the foreign IP address and foreign port number are not known yet, because the end point is in the LISTEN state, waiting for a connection to arrive.

We now start a Telnet client on the host `slip` (140.252.13.65) that connects to this server. Here are the relevant lines from the `netstat` output:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23       140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

The first line for port 23 is the ESTABLISHED connection. All four elements of the local and foreign address are filled in for this connection: the local IP address and port number, and the foreign IP address and port number. The local IP address corresponds to the interface on which the connection request arrived (the Ethernet interface, 140.252.13.33).

The end point in the LISTEN state is left alone. This is the end point that the concurrent server uses to accept future connection requests. It is the TCP module in the kernel that creates the new end point in the ESTABLISHED state, when the incoming connection request arrives and is accepted. Also notice that the port number for the ESTABLISHED connection doesn't change: it's 23, the same as the LISTEN end point.

We now initiate another Telnet client from the same client (`slip`) to this server. Here is the relevant `netstat` output:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.13.33.23       140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23       140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

We now have two ESTABLISHED connections from the same host to the same server. Both have a local port number of 23. This is not a problem for TCP since the foreign port numbers are different. They must be different because each of the Telnet clients uses an ephemeral port, and the definition of an ephemeral port is one that is not currently in use on that host (`slip`).

This example reiterates that TCP demultiplexes incoming segments using all four values that comprise the local and foreign addresses: destination IP address, destination port number, source IP address, and source port number. TCP cannot determine which process gets an incoming segment by looking at the destination port number only. Also, the only one of the three end points at port 23 that will receive incoming connection requests is the one in the LISTEN state. The end points in the ESTABLISHED state cannot receive SYN segments, and the end point in the LISTEN state cannot receive data segments.

Next we initiate a third Telnet client, from the host `solaris` that is across the SLIP link from `sun`, and not on its Ethernet.

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.23        140.252.1.32.34603     ESTABLISHED
tcp        0      0  140.252.13.33.23       140.252.13.65.1030     ESTABLISHED
tcp        0      0  140.252.13.33.23       140.252.13.65.1029     ESTABLISHED
tcp        0      0  *.23                   *.*                    LISTEN
```

The local IP address of the first ESTABLISHED connection now corresponds to the interface address of SLIP link on the multihomed host `sun` (140.252.1.29).

Restricting Local IP Address

We can see what happens when the server does not wildcard its local IP address, setting it to one particular local interface address instead. If we specify an IP address (or hostname) to our `sock` program when we invoke it as a server, that IP address becomes the local IP address of the listening end point. For example

```
sun % sock -s 140.252.1.29 8888
```

restricts this server to connections arriving on the SLIP interface (140.252.1.29). The `netstat` output reflects this:

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.8888     *.*                    LISTEN
```

If we connect to this server across the SLIP link, from the host `solaris`, it works.

```
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp        0      0  140.252.1.29.8888     140.252.1.32.34614     ESTABLISHED
tcp        0      0  140.252.1.29.8888     *.*                    LISTEN
```

But if we try to connect to this server from a host on the Ethernet (140.252.13), the connection request is not accepted by the TCP module. If we watch it with `tcpdump` the SYN is responded to with an RST, as we show in Figure 18.21.

```
1  0.0                  bsdi.1026 > sun.8888: S 3657920001:3657920001(0)
                                                win 4096 <mss 1024>
2  0.000859 (0.0009)    sun.8888 > bsdi.1026: R 0:0(0) ack 3657920002 win 0
```

Figure 18.21 Rejection of a connection request based on local IP address of server.

The server application never sees the connection request—the rejection is done by the kernel's TCP module, based on the local IP address specified by the application.

Restricting Foreign IP Address

In Section 11.12 we saw that a UDP server can normally specify the foreign IP address and foreign port, in addition to specifying the local IP address and local port. The interface functions shown in RFC 793 allow a server doing a passive open to have either a fully specified foreign socket (to wait for a particular client to issue an active open) or a unspecified foreign socket (to wait for any client).

Unfortunately, most APIs don't provide a way to do this. The server must leave the foreign socket unspecified, wait for the connection to arrive, and then examine the IP address and port number of the client.

Figure 18.22 summarizes the three types of address bindings that a TCP server can establish for itself. In all cases, *lport* is the server's well-known port and *localIP* must be the IP address of a local interface. The ordering of the three rows in the table is the order that the TCP module applies when trying to determine which local end point receives an incoming connection request. The most specific binding (the first row, if supported) is tried first, and the least specific (the last row with both IP addresses wildcarded) is tried last.

Local Address	Foreign Address	Description
localIP, *lport*	*foreignIP*, *fport*	restricted to one client (normally not supported)
localIP, *lport*	*, *	restricted to connections arriving on one local interface: *localIP*
*, *lport*	*, *	receives all connections sent to *lport*

Figure 18.22 Specification of local and foreign IP addresses and port number for TCP server.

Incoming Connection Request Queue

A concurrent server invokes a new process to handle each client, so the listening server should always be ready to handle the next incoming connection request. That's the underlying reason for using concurrent servers. But there is still a chance that multiple connection requests arrive while the listening server is creating a new process, or while the operating system is busy running other higher priority processes. How does TCP handle these incoming connection requests while the listening application is busy?

In Berkeley-derived implementations the following rules apply.

1. Each listening end point has a fixed length queue of connections that have been accepted by TCP (i.e., the three-way handshake is complete), but not yet accepted by the application.

   Be careful to differentiate between TCP accepting a connection and placing it on this queue, and the application taking the accepted connection off this queue.

2. The application specifies a limit to this queue, commonly called the backlog. This backlog must be between 0 and 5, inclusive. (Most applications specify the maximum value of 5.)

3. When a connection request arrives (i.e., the SYN segment), an algorithm is applied by TCP to the current number of connections already queued for this listening end point, to see whether to accept the connection or not. We would expect the backlog value specified by the application to be the maximum number of queued connections allowed for this end point, but it's not that simple. Figure 18.23 shows the relationship between the backlog value and the real maximum number of queued connections allowed by traditional Berkeley systems and Solaris 2.2.

Backlog value	Max # of queued connections	
	Traditional BSD	Solaris 2.2
0	1	0
1	2	1
2	4	2
3	5	3
4	7	4
5	8	5

Figure 18.23 Maximum number of accepted connections allowed for listening end point.

Keep in mind that this backlog value specifies only the maximum number of queued connections for one listening end point, all of which have already been accepted by TCP and are waiting to be accepted by the application. This backlog has no effect whatsoever on the maximum number of established connections allowed by the system, or on the number of clients that a concurrent server can handle concurrently.

> The Solaris values in this figure are what we expect. The traditional BSD values are (for some unknown reason) the backlog value times 3, divided by 2, plus 1.

4. If there is room on this listening end point's queue for this new connection (based on Figure 18.23), the TCP module ACKs the SYN and completes the connection. The server application with the listening end point won't see this new connection until the third segment of the three-way handshake is received. Also, the client may think the server is ready to receive data when the client's active open completes successfully, before the server application has been notified of the new connection. (If this happens, the server's TCP just queues the incoming data.)
5. If there is not room on the queue for the new connection, TCP just ignores the received SYN. Nothing is sent back (i.e., no RST segment). If the listening server doesn't get around to accepting some of the already accepted connections that have filled its queue to the limit, the client's active open will eventually time out.

We can see this scenario take place with our `sock` program. We invoke it with a new option (`-O`) that tells it to pause after creating the listening end point, before accepting any connection requests. If we then invoke multiple clients during this pause period, it should cause the server's queue of accepted connections to fill, and we can see what happens with `tcpdump`.

```
bsdi % sock -s -v -q1 -O30 7777
```

The `-q1` option sets the backlog of the listening end point to 1, which for this traditional BSD system should allow two pending connection requests (Figure 18.23). The `-O30` option causes the program to sleep for 30 seconds before accepting any client connections. This gives us 30 seconds to start some clients, to fill the queue. We'll start four clients on the host `sun`.

Figure 18.24 shows the `tcpdump` output, starting with the first SYN from the first client. (We have removed the window size advertisements and MSS announcements. We have also marked the client port numbers in bold when the TCP connection is established—the three-way handshake.)

The first client's connection request from port 1090 is accepted by TCP (segments 1–3). The second client's connection request from port 1091 is also accepted by TCP (segments 4–6). The server application is still asleep, and has not accepted either connection yet. Everything has been done by the TCP module in the kernel. Also, the two clients have returned successfully from their active opens, since the three-way handshakes are complete.

```
 1   0.0                  sun.1090 > bsdi.7777: S 1617152000:1617152000(0)
 2   0.002310 ( 0.0023)   bsdi.7777 > sun.1090: S 4164096001:4164096001(0)
                                                  ack 1617152001
 3   0.003098 ( 0.0008)   sun.1090 > bsdi.7777: . ack 1

 4   4.291007 ( 4.2879)   sun.1091 > bsdi.7777: S 1617792000:1617792000(0)
 5   4.293349 ( 0.0023)   bsdi.7777 > sun.1091: S 4164672001:4164672001(0)
                                                  ack 1617792001
 6   4.294167 ( 0.0008)   sun.1091 > bsdi.7777: . ack 1

 7   7.131981 ( 2.8378)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
 8  10.556787 ( 3.4248)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
 9  12.695916 ( 2.1391)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
10  16.195772 ( 3.4999)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
11  24.695571 ( 8.4998)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)

12  28.195454 ( 3.4999)   sun.1093 > bsdi.7777: S 1618688000:1618688000(0)
13  28.197810 ( 0.0024)   bsdi.7777 > sun.1093: S 4167808001:4167808001(0)
                                                  ack 1618688001
14  28.198639 ( 0.0008)   sun.1093 > bsdi.7777: . ack 1

15  48.694931 (20.4963)   sun.1092 > bsdi.7777: S 1618176000:1618176000(0)
16  48.697292 ( 0.0024)   bsdi.7777 > sun.1092: S 4170496001:4170496001(0)
                                                  ack 1618176001
17  48.698145 ( 0.0009)   sun.1092 > bsdi.7777: . ack 1
```

Figure 18.24 `tcpdump` output for backlog example.

We try to start a third client in segment 7 (port 1092), and a fourth in segment 8 (port 1093). TCP ignores both SYNs since the queue for this listening end point is full. Both clients retransmit their SYNs in segments 9, 10, 11, 12, and 15. The fourth client's third retransmission is accepted (segments 12–14) because the server's 30-second pause is over, causing the server to remove the two connections that were accepted, emptying its queue. (The reason it appears this connection was accepted by the server at the time 28.19, and not at a time greater than 30, is because it took a few seconds to start the first client [segment 1, the starting time point in the output] after starting the server.) The third client's fourth retransmission is then accepted (segments 15–17). The fourth client connection (port 1093) is accepted by the server before the third client connection (port 1092) because of the timing interactions between the server's 30-second pause and the client's retransmissions.

> We would expect the queue of accepted connections to be passed to the application in FIFO (first-in, first-out) order. That is, after TCP accepts the connections on ports 1090 and 1091, we expect the application to receive the connection on port 1090 first, and then the connection on port 1091. But a bug has existed for years in many Berkeley-derived implementations causing them to be returned in a LIFO (last-in, first-out) order instead. Vendors have recently started fixing this bug, but it still exists in systems such as SunOS 4.1.3.

TCP ignores the incoming SYN when the queue is full, and doesn't respond with an RST, because this is a soft error, not a hard error. Normally the queue is full because the application or the operating system is busy, preventing the application from servicing incoming connections. This condition could change in a short while. But if the server's TCP responded with a reset, the client's active open would abort (which is what we saw

happen if the server wasn't started). By ignoring the SYN, the server forces the client TCP to retransmit the SYN later, hoping that the queue will then have room for the new connection.

A subtle point in this example, which is found in most TCP/IP implementations, is that TCP accepts an incoming connection request (i.e., a SYN) if there is room on the listener's queue, without giving the application a chance to see who it's from (the source IP address and source port number). This is not required by TCP, it's just the common implementation technique (i.e., the way the Berkeley sources have always done it). If an API such as TLI (Section 1.15) gives the application a way to learn when a connection request arrives, and then allows the application to choose whether to accept the connection or not, be aware that with TCP, when the application is supposedly told that the connection has just arrived, TCP's three-way handshake is over! Other transport layers may be implemented to provide this separation to the application between arrival and acceptance (i.e., the OSI transport layer) but not TCP.

> Solaris 2.2 provides an option that prevents TCP from accepting an incoming connection request until the application says so (`tcp_eager_listeners` in Section E.4).

This behavior also means that a TCP server has no way to cause a client's active open to fail. When a new client connection is passed to the server application, TCP's three-way handshake is over, and the client's active open has completed successfully. If the server then looks at the client's IP address and port number, and decides it doesn't want to service this client, all the server can do is either close the connection (causing a FIN to be sent) or reset the connection (causing an RST to be sent). In either case the client thought everything was OK when its active open completed, and may have already sent a request to the server.

18.12 Summary

Before two processes can exchange data using TCP, they must establish a connection between themselves. When they're done they terminate the connection. This chapter has provided a detailed look at how connections are established using a three-way handshake, and terminated using four segments.

We used `tcpdump` to show all the fields in the TCP header. We've also seen how a connection establishment can time out, how resets are sent, what happens with a half-open connection, and how TCP provides a half-close, simultaneous opens, and simultaneous closes.

Fundamental to understanding the operation of TCP is its state transition diagram. We've followed through the steps involved in connection establishment and termination, and the state transitions that take place. We also looked at the implications of TCP's connection establishment on the design of concurrent TCP servers.

A TCP connection is uniquely defined by a 4-tuple: the local IP address, local port number, foreign IP address, and foreign port number. Whenever a connection is terminated, one end must maintain knowledge of the connection, and we saw that the TIME_WAIT state handles this. The rule is that the end that does the active close enters this state for twice the implementation's MSL.

Exercises

18.1 In Section 18.2 we said that the initial sequence number (ISN) normally starts at 1 and is incremented by 64,000 every half-second and every time an active open is performed. This would imply that the low-order three digits of the ISN would always be 001. But in Figure 18.3 these low-order three digits are 521 in each direction. What's going on?

18.2 In Figure 18.15 we typed 12 characters and saw 13 bytes sent by TCP. In Figure 18.16 we typed eight characters but TCP sent 10 bytes. Why was 1 byte added in the first case, but 2 bytes in the second case?

18.3 What's the difference between a half-open connection and a half-closed connection?

18.4 If we start our `sock` program as a server, and then terminate it (without having a client connect to it), we can immediately restart the server. This implies that it doesn't go through the 2MSL wait state. Explain this in terms of the state transition diagram.

18.5 In Section 18.6 we showed that a client cannot reuse the same local port number while that port is part of a connection in the 2MSL wait. But if we run our `sock` program twice in a row as a client, connecting to the daytime server, we can reuse the same local port number. Additionally, we're able to create a new incarnation of a connection that should be in the 2MSL wait. What's going on?

```
sun % sock -v bsdi daytime
connected on 140.252.13.33.1163 to 140.252.13.35.13
Wed Jul  7 07:54:51 1993
connection closed by peer
sun % sock -v -b1163 bsdi daytime      reuse same local port number
connected on 140.252.13.33.1163 to 140.252.13.35.13
Wed Jul  7 07:55:01 1993
connection closed by peer
```

18.6 At the end of Section 18.6 when describing the FIN_WAIT_2 state, we mentioned that many implementations move a connection from this state into the CLOSED state if the application did a complete close (not a half-close) after just over 11 minutes. If the other end (in the CLOSE_WAIT state) waited 12 minutes before issuing its close (i.e., sending its FIN), what would its TCP get in response to the FIN?

18.7 Which end of a telephone conversation does the active open, and which does the passive open? Are simultaneous opens allowed? Are simultaneous closes allowed?

18.8 In Figure 18.6 we don't see an ARP request or an ARP reply. Obviously the hardware address for host `svr4` must be in the ARP cache on `bsdi`. What would change in this figure if this ARP cache entry was not present?

18.9 Explain the following `tcpdump` output. Compare it with Figure 18.13.

```
1  0.0                 solaris.32990 > bsdi.discard: S 40140288:40140288(0)
                                                     win 8760 <mss 1460>
2  0.003295 (0.0033)   bsdi.discard > solaris.32990: S 4208081409:4208081409(0)
                                                     ack 40140289 win 4096
                                                     <mss 1024>
3  0.419991 (0.4167)   solaris.32990 > bsdi.discard: P 1:257(256) ack 1 win 9216
4  0.449852 (0.0299)   solaris.32990 > bsdi.discard: F 257:257(0) ack 1 win 9216
5  0.451965 (0.0021)   bsdi.discard > solaris.32990: . ack 258 win 3840
6  0.464569 (0.0126)   bsdi.discard > solaris.32990: F 1:1(0) ack 258 win 4096
7  0.720031 (0.2555)   solaris.32990 > bsdi.discard: . ack 2 win 9216
```

18.10 Why doesn't the server in Figure 18.4 combine the ACK of the client's FIN with its own FIN, reducing the number of segments to three?

18.11 In Figure 18.16 why is the sequence number of the RST 263680027

18.12 Does TCP's querying the link layer for the MTU violate the spirit of layering?

18.13 Assume in Figure 14.16 that each DNS query is issued using TCP instead of UDP. How many packets are exchanged?

18.14 With an MSL of 120 seconds, what is the maximum at which a system can initiate new connections and then do an active close?

18.15 Read RFC 793 to see what happens when an end point that is in the TIME_WAIT state receives a duplicate of the FIN that placed it into this state.

18.16 Read RFC 793 to see what happens when an end point that is in the TIME_WAIT state receives an RST.

18.17 Read the Host Requirements RFC to obtain the definition of a *half-duplex* TCP close.

18.18 In Figure 1.8 (p. 11) we said that incoming TCP segments are demultiplexed based on the destination TCP port number. Is that correct?