2.6 TCP Connection Establishment and Termination

To aid in our understanding of the `connect`, `accept`, and `close` functions and to help us debug TCP applications using the `netstat` program, we must understand how TCP connections are established and terminated, and TCP's state transition diagram.

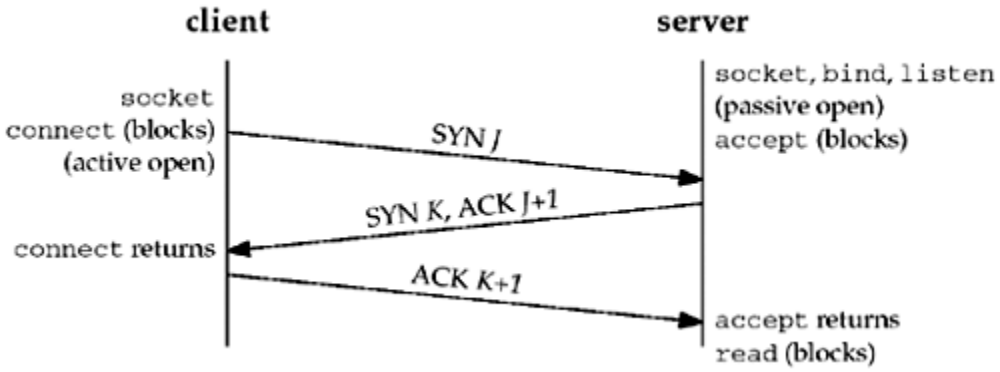
Three-Way Handshake

The following scenario occurs when a TCP connection is established:

1. The server must be prepared to accept an incoming connection. This is normally done by calling `socket`, `bind`, and `listen` and is called a *passive open*.
2. The client issues an *active open* by calling `connect`. This causes the client TCP to send a "synchronize" (SYN) segment, which tells the server the client's initial sequence number for the data that the client will send on the connection. Normally, there is no data sent with the SYN; it just contains an IP header, a TCP header, and possible TCP options (which we will talk about shortly).
3. The server must acknowledge (ACK) the client's SYN and the server must also send its own SYN containing the initial sequence number for the data that the server will send on the connection. The server sends its SYN and the ACK of the client's SYN in a single segment.
4. The client must acknowledge the server's SYN.

The minimum number of packets required for this exchange is three; hence, this is called TCP's *three-way handshake*. We show the three segments in Figure 2.2.

Figure 2.2. TCP three-way handshake.



We show the client's initial sequence number as *J* and the server's initial sequence number as *K*. The acknowledgment number in an ACK is the next expected sequence number for the end sending the ACK. Since a SYN occupies one byte of the sequence number space, the acknowledgment number in the ACK of each SYN is the initial sequence number plus one. Similarly, the ACK of each FIN is the sequence number of the FIN plus one.

> An everyday analogy for establishing a TCP connection is the telephone system [Nemeth 1997]. The `socket` function is the equivalent of having a telephone to use. `bind` is telling other people your telephone number so that they can call you. `listen` is turning on the ringer so that you will hear when an incoming call arrives. `connect` requires that we know the other person's phone number and dial it. `accept` is when the person being called answers the phone. Having the client's identity returned by `accept` (where the identify is the client's IP address and port number) is similar to having the caller ID feature show the caller's phone number. One difference, however, is that `accept` returns the client's identity only after the connection has been established, whereas the caller ID feature shows the caller's phone number before we choose whether to answer the phone or not. If the DNS is used (Chapter 11), it provides a service analogous to a telephone book. `getaddrinfo` is similar to looking up a person's phone number in the phone book. `getnameinfo` would be the equivalent of having a phone book sorted by telephone numbers that we could search, instead of a book sorted by name.

TCP Options

Each SYN can contain TCP options. Commonly used options include the following:

- MSS option. With this option, the TCP sending the SYN announces its *maximum segment size*, the maximum amount of data that it is willing to accept in each TCP segment, on this connection. The sending TCP uses the receiver's MSS value as the maximum size of a

segment that it sends. We will see how to fetch and set this TCP option with the `TCP_MAXSEG` socket option (Section 7.9).

- Window scale option. The maximum window that either TCP can advertise to the other TCP is 65,535, because the corresponding field in the TCP header occupies 16 bits. But, high-speed connections, common in today's Internet (45 Mbits/sec and faster, as described in RFC 1323 [Jacobson, Braden, and Borman 1992]), or long delay paths (satellite links) require a larger window to obtain the maximum throughput possible. This newer option specifies that the advertised window in the TCP header must be scaled (left-shifted) by 0–14 bits, providing a maximum window of almost one gigabyte ($65,535 \times 2^{14}$). Both end-systems must support this option for the window scale to be used on a connection. We will see how to affect this option with the `SO_RCVBUF` socket option (Section 7.5).

  To provide interoperability with older implementations that do not support this option, the following rules apply. TCP can send the option with its SYN as part of an active open. But, it can scale its windows only if the other end also sends the option with its SYN. Similarly, the server's TCP can send this option only if it receives the option with the client's SYN. This logic assumes that implementations ignore options that they do not understand, which is required and common, but unfortunately, not guaranteed with all implementations.

- Timestamp option. This option is needed for high-speed connections to prevent possible data corruption caused by old, delayed, or duplicated segments. Since it is a newer option, it is negotiated similarly to the window scale option. As network programmers there is nothing we need to worry about with this option.

These common options are supported by most implementations. The latter two are sometimes called the "RFC 1323 options," as that RFC [Jacobson, Braden, and Borman 1992] specifies the options. They are also called the "long fat pipe options," since a network with either a high bandwidth or a long delay is called a *long fat pipe*. Chapter 24 of TCPv1 contains more details on these options.

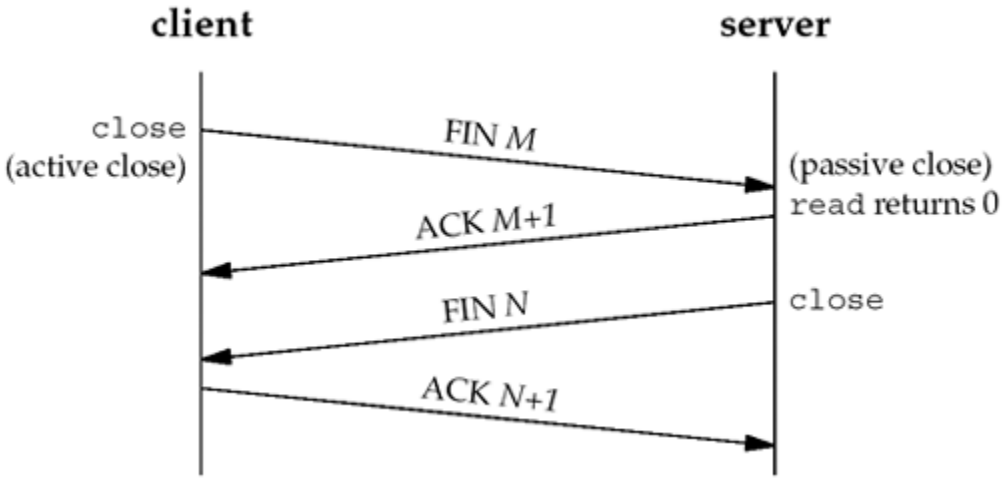
TCP Connection Termination

While it takes three segments to establish a connection, it takes four to terminate a connection.

1. One application calls `close` first, and we say that this end performs the *active close*. This end's TCP sends a FIN segment, which means it is finished sending data.
2. The other end that receives the FIN performs the *passive close*. The received FIN is acknowledged by TCP. The receipt of the FIN is also passed to the application as an end-of-file (after any data that may have already been queued for the application to receive), since the receipt of the FIN means the application will not receive any additional data on the connection.
3. Sometime later, the application that received the end-of-file will `close` its socket. This causes its TCP to send a FIN.
4. The TCP on the system that receives this final FIN (the end that did the active close) acknowledges the FIN.

Since a FIN and an ACK are required in each direction, four segments are normally required. We use the qualifier "normally" because in some scenarios, the FIN in Step 1 is sent with data. Also, the segments in Steps 2 and 3 are both from the end performing the passive close and could be combined into one segment. We show these packets in Figure 2.3.

Figure 2.3. Packets exchanged when a TCP connection is closed.



A FIN occupies one byte of sequence number space just like a SYN. Therefore, the ACK of each FIN is the sequence number of the FIN plus one.

Between Steps 2 and 3 it is possible for data to flow from the end doing the passive close to the end doing the active close. This is called a *half-close* and we will talk about this in detail with the `shutdown` function in Section 6.6.

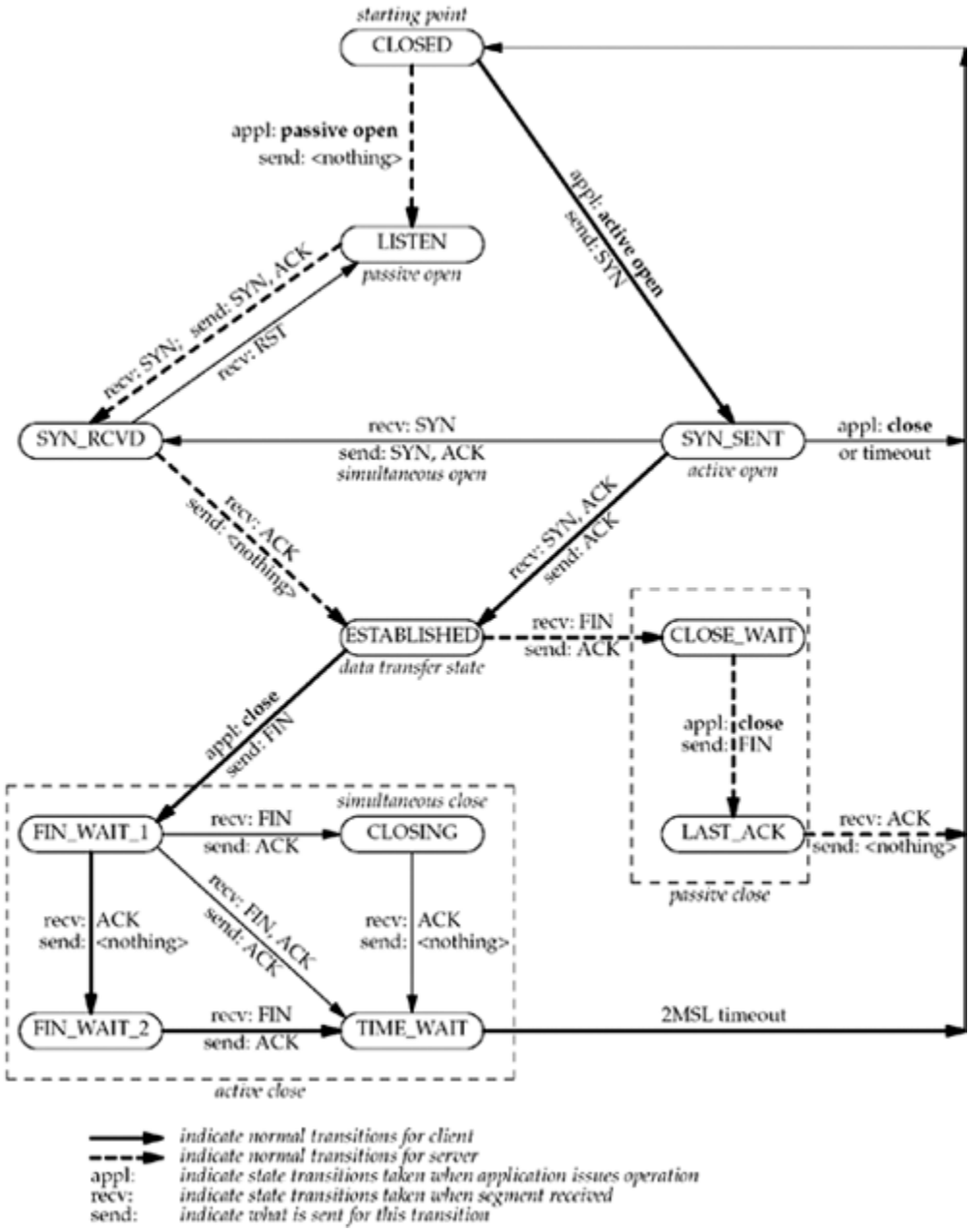
The sending of each FIN occurs when a socket is closed. We indicated that the application calls `close` for this to happen, but realize that when a Unix process terminates, either voluntarily (calling `exit` or having the `main` function return) or involuntarily (receiving a signal that terminates the process), all open descriptors are closed, which will also cause a FIN to be sent on any TCP connection that is still open.

Although we show the client in Figure 2.3 performing the active close, either end—the client or the server—can perform the active close. Often the client performs the active close, but with some protocols (notably HTTP/1.0), the server performs the active close.

TCP State Transition Diagram

The operation of TCP with regard to connection establishment and connection termination can be specified with a *state transition diagram*. We show this in Figure 2.4.

Figure 2.4. TCP state transition diagram.



There are 11 different states defined for a connection and the rules of TCP dictate the transitions from one state to another, based on the current state and the segment received in that state. For example, if an application performs an active open in the CLOSED state, TCP sends a SYN and the new state is SYN_SENT. If TCP next receives a SYN with an ACK, it sends an ACK and the new state is ESTABLISHED. This final state is where most data transfer occurs.

The two arrows leading from the ESTABLISHED state deal with the termination of a connection. If an application calls `close` before receiving a FIN (an active close), the transition is to the FIN_WAIT_1 state. But if an application receives a FIN while in the ESTABLISHED state (a passive close), the transition is to the CLOSE_WAIT state.

We denote the normal client transitions with a darker solid line and the normal server transitions with a darker dashed line. We also note that there are two transitions that we have not talked about: a simultaneous open (when both ends send SYNs at about the same time and the

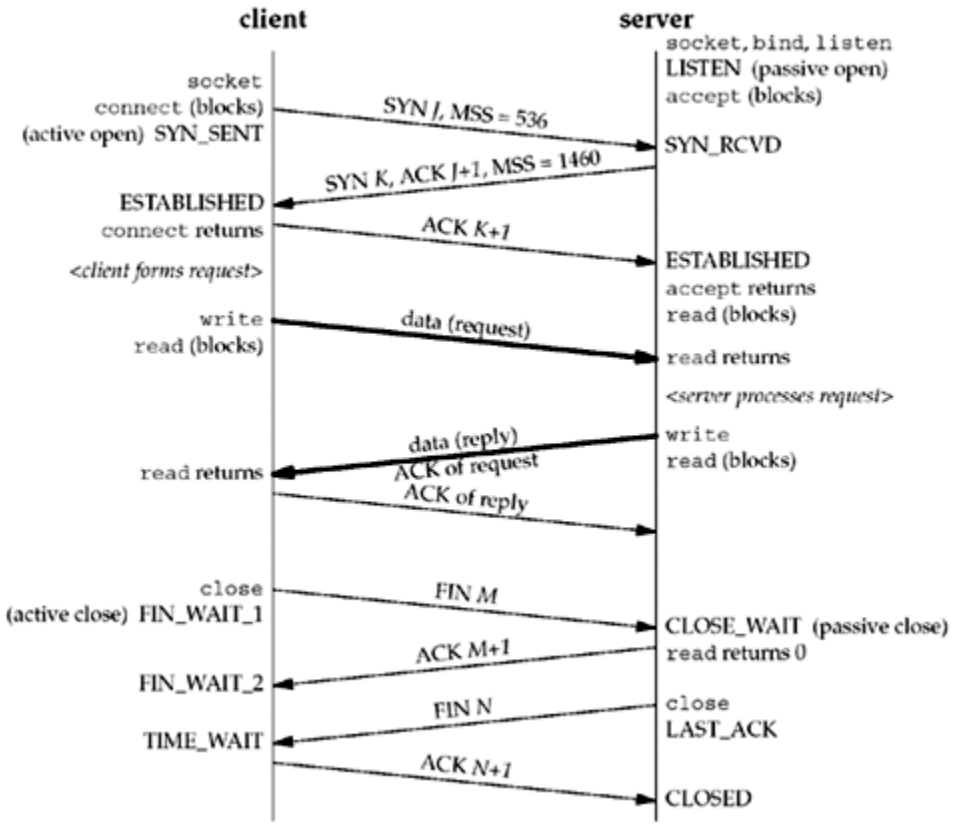
SYNs cross in the network) and a simultaneous close (when both ends send FINs at the same time). Chapter 18 of TCPv1 contains examples and a discussion of both scenarios, which are possible but rare.

One reason for showing the state transition diagram is to show the 11 TCP states with their names. These states are displayed by `netstat`, which is a useful tool when debugging client/server applications. We will use `netstat` to monitor state changes in Chapter 5.

Watching the Packets

Figure 2.5 shows the actual packet exchange that takes place for a complete TCP connection: the connection establishment, data transfer, and connection termination. We also show the TCP states through which each endpoint passes.

Figure 2.5. Packet exchange for TCP connection.



The client in this example announces an MSS of 536 (indicating that it implements only the minimum reassembly buffer size) and the server announces an MSS of 1,460 (typical for IPv4 on an Ethernet). It is okay for the MSS to be different in each direction (see Exercise 2.5).

Once a connection is established, the client forms a request and sends it to the server. We assume this request fits into a single TCP segment (i.e., less than 1,460 bytes given the server's announced MSS). The server processes the request and sends a reply, and we assume that the reply fits in a single segment (less than 536 in this example). We show both data segments as bolder arrows. Notice that the acknowledgment of the client's request is sent with the server's reply. This is called *piggybacking* and will normally happen when the time it takes the server to process the request and generate the reply is less than around 200 ms. If the server takes longer, say one second, we would see the acknowledgment followed later by the reply. (The dynamics of TCP data flow are covered in detail in Chapters 19 and 20 of TCPv1.)

We then show the four segments that terminate the connection. Notice that the end that performs the active close (the client in this scenario) enters the TIME_WAIT state. We will discuss this in the next section.

It is important to notice in Figure 2.5 that if the entire purpose of this connection was to send a one-segment request and receive a one-segment reply, there would be eight segments of overhead involved when using TCP. If UDP was used instead, only two packets would be exchanged: the request and the reply. But switching from TCP to UDP removes all the reliability that TCP provides to the application, pushing lots of these details from the transport layer (TCP) to the UDP application. Another important feature provided by TCP is congestion control, which must then be handled by the UDP application. Nevertheless, it is important to understand that many applications are built using UDP because the application exchanges small amounts of data and UDP avoids the overhead of TCP connection establishment and connection termination.