be accepted, and the second argument of SOMAXCONN specifies the maximum number of pending connections the kernel will queue for the socket.

SOMAXCONN is defined in the <sys/socket.h> header. Historically its value has been 5, although some newer systems define it to be 10. But busy servers (e.g., systems providing a World Wide Web server) have found a need for even higher values, say 256 or 1024. We talk about this more in Section 14.5.

accept a connection and process request

18–28 The server blocks in the call to accept until a connection is established by the client's connect. The new socket descriptor returned by accept, sockfd, refers to the connection to the client. The client's request is read by read_stream (Figure 1.6) and the reply is returned by write.

This server is an *iterative* server: it processes each client's request completely before looping around to accept the next client connection. A *concurrent* server is one that handles multiple clients concurrently (i.e., at the same time). A common technique for implementing concurrent servers on Unix hosts is for the server to fork a child process after accept returns, letting the child handle the client's request, allowing the parent to accept another client connection immediately. Another technique is for the server to create a thread (called a lightweight process) to handle each request. We show the iterative server to avoid complicating the example with process control functions that don't affect the networking aspects of the example. (Chapter 8 of [Stevens 1992] discusses the fork function. Chapter 4 of [Stevens 1990] discusses iterative versus concurrent servers.)

A third option is a *pre-forked* server. Here the server calls fork a fixed number of times when it starts and each child calls accept on the same listening descriptor. This approach saves a call to fork for each client request, which can be a big savings on a busy server. Some HTTP servers use this technique.

Figure 1.8 shows the time line for the TCP client–server transaction. The first thing we notice, compared to the UDP time line in Figure 1.4, is the increased number of network packets: nine for the TCP transaction, compared to two for the UDP transaction. With TCP the client's measured transaction time is *at least* 2×RTT + SPT. Normally the middle three segments from the client to the server—the ACK of the server's SYN, the request, and the client's FIN—are spaced closely together, as are the later two segments from the server to the client—the reply and the server's FIN. This makes the transaction time closer to 2×RTT + SPT than it might appear from Figure 1.8.

The additional RTT in this example is from the establishment of the TCP connection: the first two segments that we show in Figure 1.8. If TCP could combine the establishment of the connection with the client's data and the client's FIN (the first four segments from the client in the figure), and then combine the server's reply with the server's FIN, we would be back to a transaction time of RTT + SPT, the same as we had with UDP. Indeed, this is basically the technique used by T/TCP.

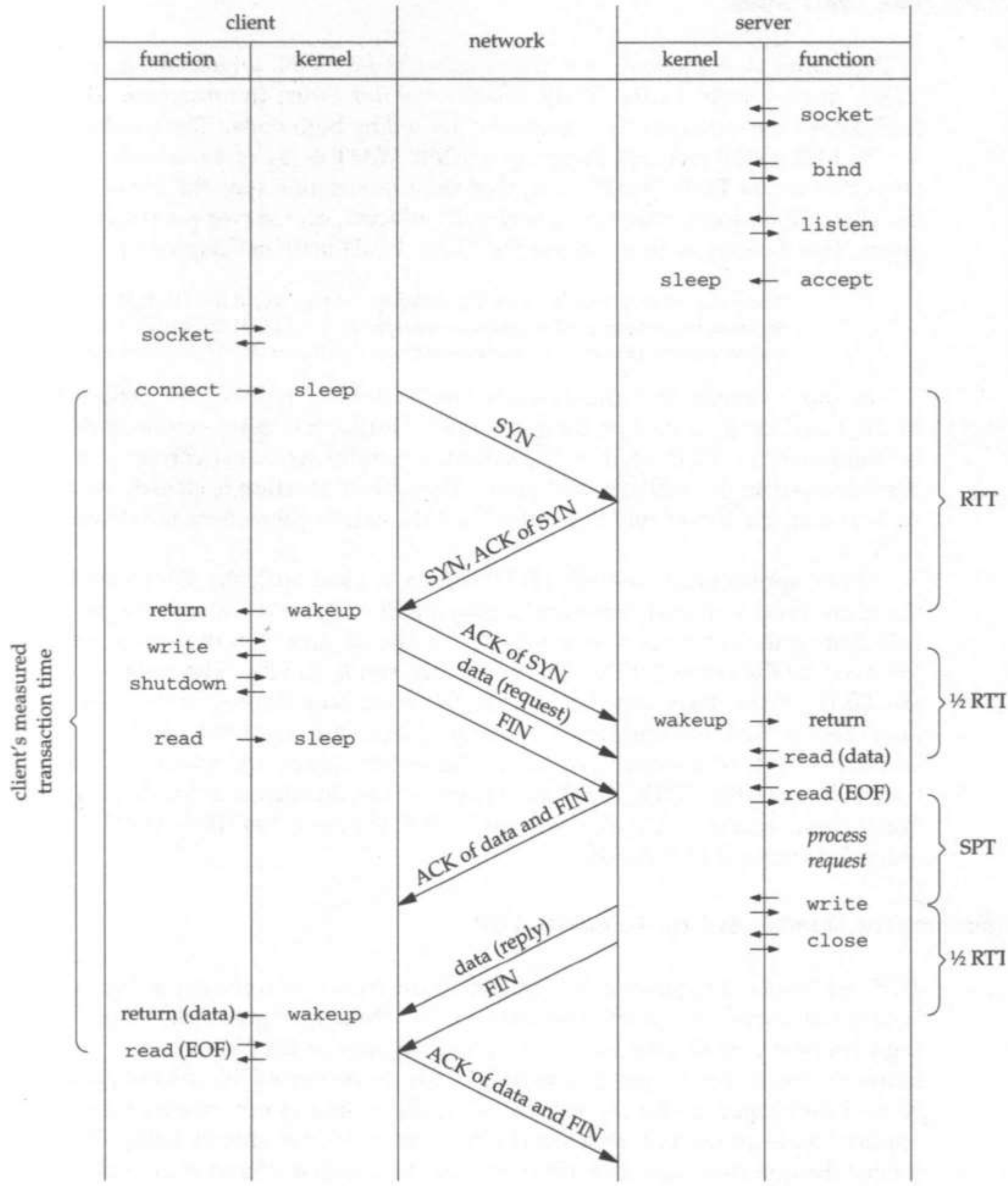

Figure 1.8 Time line of TCP client–server transaction.