60 Transmission Control Protocol—TCP

Aleksander Malinowski
Bradley University

Bogdan M. Wilamowski
Auburn University

60.1 Introduction **60**-1
60.2 Protocol Operation **60**-3
TCP Segment • Port Number Assignments • Connection Establishment • Maintaining the Open Connection • Flow Control and Sliding Window Protocol • Improving Flow Control • Error Control • Congestion Control • Connection Termination
60.3 State Diagram **60**-14
60.4 Programming Samples **60**-14
References **60**-17

60.1 Introduction

Transmission control protocol (TCP) is a part of TCP/Internet protocol (IP) suite [STD7,C02-1,F10, GW03,PD07]. It provides full transport layer services to applications. It belongs to the transport layer in the TCP/IP suite model as shown in Figure 60.1. TCP is more complicated than user datagram protocol (UDP) and provides a reliable connection between both ends of transmission. This connection needs to be set up, maintained, and torn down. The protocol divides a stream of data into units of limited size called segments. These segments are reassembled in a reliable way at the other end. Reliability is achieved by assembling segments by preserving their original sequence, and arranging for retransmission of lost or corrupted data. While the underlying IP provides logical addressing of segments and is responsible for their transmission through networks [STD5], TCP provides additional addressing that allows setting up connections for multiple applications and services on the same end. Each application seeking TCP connection is assigned one or more numbers (one number per connection) that are called port numbers. Figure 60.2 illustrates the concept of port numbers.

TCP provides a logical full duplex connection between two application layer processes. The connection is perceived as a connection-oriented, reliable, in-sequence byte-stream service. The protocol also allows the recipient of data to control the rate at which the sender transmits data to avoid buffer overflow.

- *Connection-oriented* shows that the connection needs to be established and its parameters negotiated between the two processes. This negotiation is performed by sending transmission control blocks (TCB). At the end of the transmission, the connection must be torn down. TCP terminates each direction of the connection separately, which allows the data flow to continue even after the other direction has been closed.
- *Reliable* indicates that either all data are delivered or the connection is broken. TCP is designed to work over the IP and thus it does not assume that the underlying network services are reliable. To ensure reliability, each chunk of data are numbered, and a selective automatic repeat request (ARQ) is employed to request retransmission of lost or corrupt data chunk.

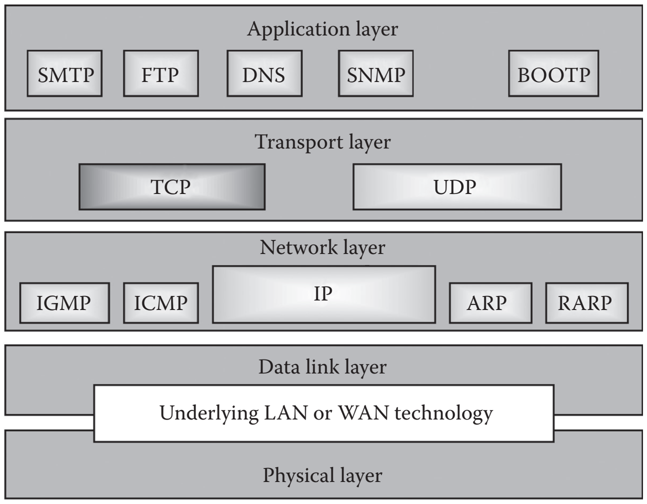


FIGURE 60.1 UDP and TCP/IP in the TCP/IP protocol suite model.

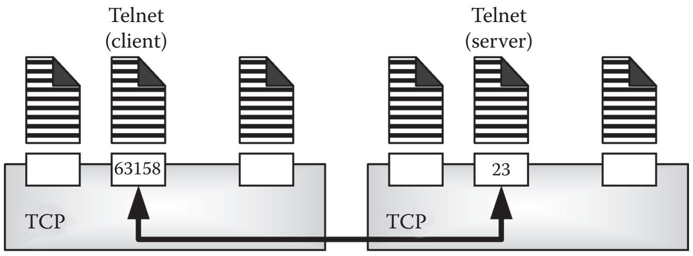


FIGURE 60.2 Process to process communication.

- *In-sequence* shows that all data are guaranteed to be received by the end receiving process in the same order that they were sent. Sequencing is implemented by the same mechanism that is used for missing data chunk detection. All received data are placed in a receiving buffer in location corresponding to the data chunk number. The process that receives data reads it from that buffer in a correct sequence.
- *Byte-stream* oriented implies that the connection can be perceived as continuous stream of data regardless of the fact that the underlying network services may transmit data in chunks. Those data chunks may be of various sizes and may have maximum size limit. The sending process, however, may send data either continuously or in bursts. All data to be sent are placed in the sending buffer. Data from the buffer are partitioned into chunks, which are called segments according to algorithms of TCP, and then they are sent over the network. On the receiving side, the data are placed in the receiving buffer in an appropriate sequence as mentioned earlier. The receiving process may read data either as chunks or as byte after byte.
- *Full duplex* indicates that each party involved in the transmission has control over it. For example, any side can send data at any time or terminate the connection.

The TCP is standardized by *Internet Society*, which is an independent international nonprofit organization. A complete information about TCP-related standards is published by *RFC Editor* [RFC]. Several relevant RFC articles as well as further reading materials are listed at the end of this article.

60.2 Protocol Operation

Before data transfer begins, TCP establishes a connection between two application processes by setting up the protocol parameters. These parameters are stored in a structure variable called TCB. Once the connection is established, the data transfer may begin. During the transfer, parameters may be modified to improve efficiency of transmission and prevent congestion. In the end, the connection is closed. Under certain circumstances, connection may be terminated without formal closing.

60.2.1 TCP Segment

Although TCP is byte-stream oriented, connection and buffering take care of the disparities among the speed of preparing data, sending data, and consuming of data at the other end, one more step is necessary before data are sent. The underneath IP layer that takes care of sending data through the entire network sends data in packets, not as a stream of bytes. Sending one byte in a packet would be very inefficient due to the large overhead caused by headers. Therefore, data are grouped into larger chunks called segments. Each segment is encapsulated in an IP datagram and then transmitted. Each TCP segment follows the same format that is shown in Figure 60.3 [STD7]. Each segment consists of a header that is followed by options and data. The meaning of each header field will be discussed now, and its importance is illustrated in the following sections, which show phases of a typical TCP connection. AQ1

- *Source port address*: This is a 16 bit field that contains a port number of the process that sends options or data in this segment.
- *Destination port address*: This is a 16 bit field that contains a port number of the process that is supposed to receive options or data carried by this segment.

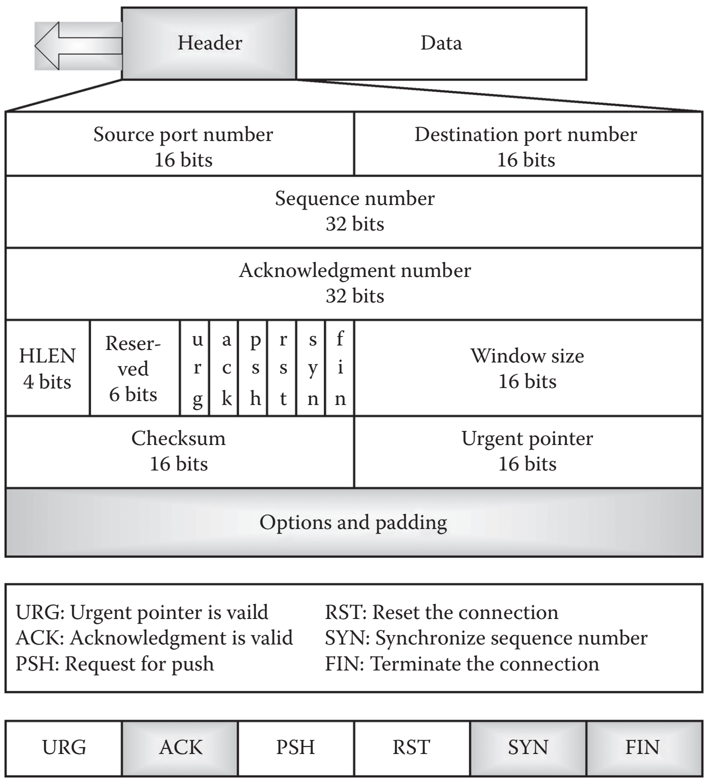

Header
Data
Source port number 16 bits
Destination port number 16 bits
Sequence number 32 bits
Acknowledgment number 32 bits
HLEN 4 bits
Reser-ved 6 bits
urg
ack
psh
rst
syn
fin
Window size 16 bits
Checksum 16 bits
Urgent pointer 16 bits
Options and padding
URG: Urgent pointer is vaild
ACK: Acknowledgment is valid
PSH: Request for push
RST: Reset the connection
SYN: Synchronize sequence number
FIN: Terminate the connection
URG
ACK
PSH
RST
SYN
FIN

FIGURE 60.3 TCP segment format.

- *Sequence number*: This is a 32 bit field that contains the number assigned to the first byte of data carried by this segment. During the establishment of connection each side generates its own random initial sequence number (ISN). The sequence number is then increased by one with every one byte of data sent. For example, if a previous segment had a sequence number 700 and had 200 bytes of data then sequence number of the next segment is 900.
- *Acknowledgment number*: This is a 32 bit field that contains the sequence number for the next segment that is expected to be received from the other party. For example, if the last received segment had a sequence number 700 and had 200 bytes of data, then the acknowledgment number is 900.
- *Header length*: This 4 bit field contains the total size of the segment header divided by 4. Since headers may be 20–60 bytes long, the value is in the range 5–15.
- *Reserved*: This 6 bit field is reserved for future use.
- *Control*: This 6 bit field contains transmission control flags.
- *URG*: Urgent. If the bit is set, then the urgent pointer is valid.
- *ACK*: Acknowledgment. If the bit is set then the acknowledgment number is valid.
- *PSH*: Push. If the bit is set, then the other party should not buffer the outgoing data but send it immediately, as it becomes available, without buffering.
- *RST*: Reset. If the bit is set, then the connection should be aborted immediately due to abnormal condition.
- *SYN*: Synchronize. The bit is set in the connection request segment when the connection is being established.
- *FIN*: Finalize. If the bit is set, it means that the sender does not have anything more to send, and that the connection may be closed. The process will still accept data from the other party until the other party also sends FIN.
- *Window size*: This 16 bit field contains information about the size of the transmission window that the other party must maintain. Window size will be explained and illustrated later.
- *Checksum*: This 16 bit field contains the checksum. The checksum is calculated by
  - Initially filling it with 0's
  - Adding a pseudoheader with information from IP, as illustrated in Figure 60.4.
  - Treating the whole segment with the pseudoheader prepended as a stream of 16 bit numbers. If the number of bytes is odd, 0 is appended at the end.

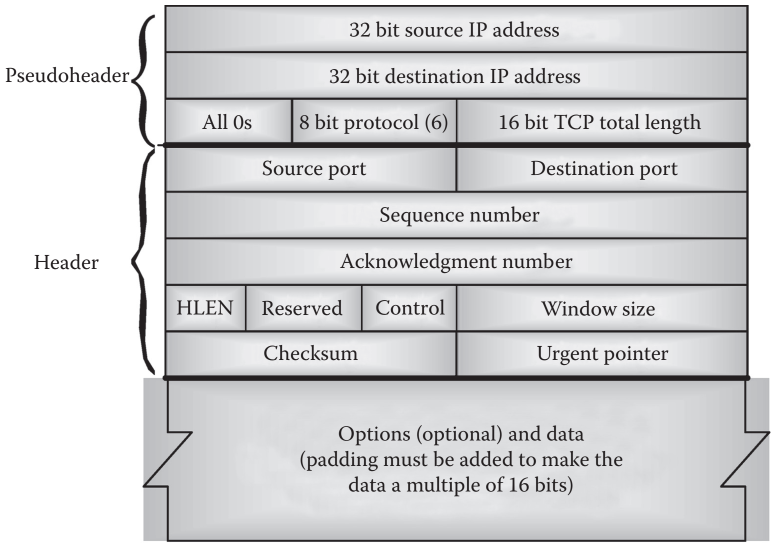


FIGURE 60.4 Pseudoheader added to the TCP datagram.

  - Adding all 16 bit numbers using 1's complement binary arithmetic.
  - Complementing the result. This complemented result is inserted into the checksum field.
- *Checksum verification*: The receiver calculates the new checksum for the received packet, which includes the original checksum, after adding the pseudoheader (see Figure 60.4). If the new checksum is nonzero then the packet is corrupt and is discarded.
- *Urgent pointer*: This 16 bit field is used only if the URG flag is set. It defines the location of the last urgent byte in the data section of the segment. Urgent data are those that are sent out-of-sequence. For example, telnet client sends a break character without queuing it in the sending buffer.
- *Options*: There can be up to 40 bytes of options appended to the header. The options are padded inside with 0's so that the total number of bytes in the header is divisible by 4. The possible options are listed below and are illustrated in Figure 60.5:

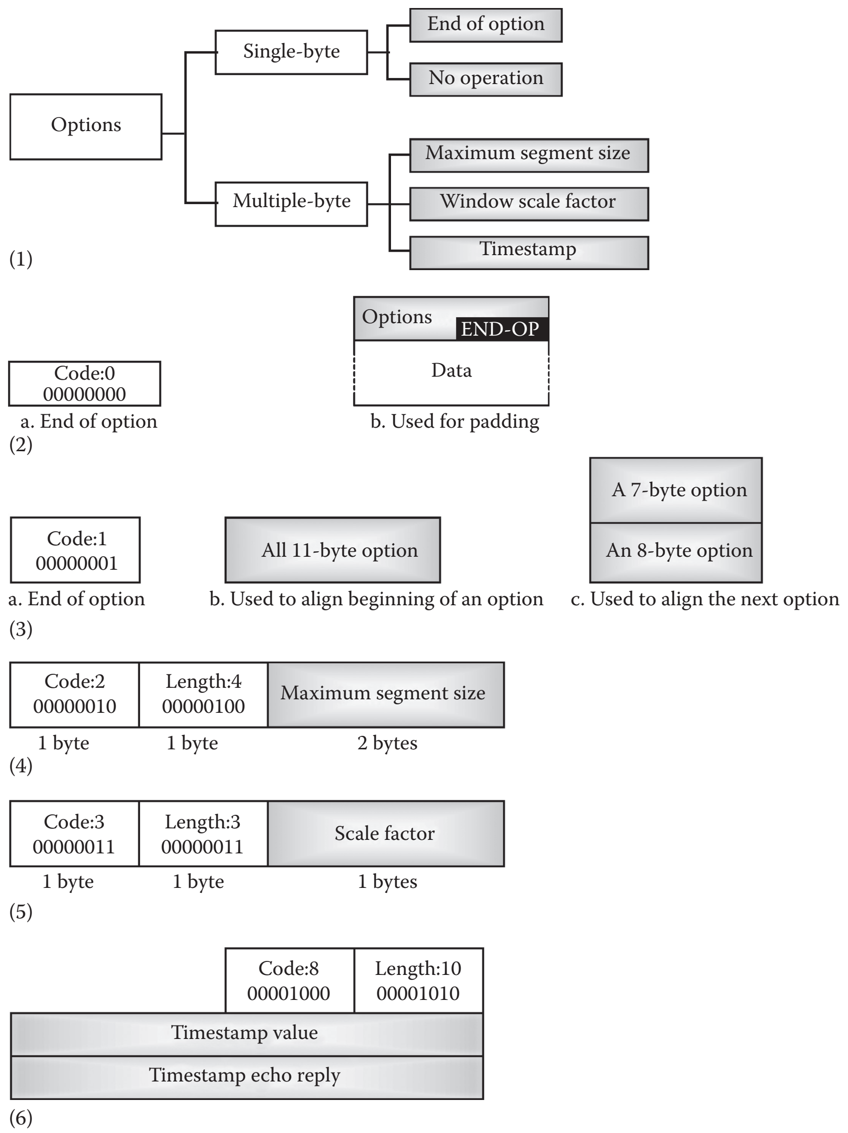


FIGURE 60.5 TCP options: 1. classification; 2. end of options; 3. no operation (used for padding); 4. setting maximum segment size; 5. setting window scale factor; 6. time stamp.

- *No operation*: 1 byte option used as filler between the options and as padding.
- *End of options*: 1 byte option used as padding at the end of options filed. Data portion of the segment follows immediately.
- *Maximum segment size*: Option defines the maximum size of a chunk of data that can AQ2 be received. IT refers to the size of data not the whole segment. It is used only during the establishment of the connection. The receiving party defines the maximum size of the datagram sent by the sender. If not specified, the default maximum is 536. Because the length of the datagram is not specified in the TCP header, there is no length limitation AQ3 for an individual TCP packet. However, the MSS value that is negotiated at the beginning of the connection limits the largest TCP packet that can be sent, and the Urgent Pointer cannot reference data beyond 65,535 bytes. MSS value of 65,535 is treated as infinity [RFC2675] [RFC2147].
- *Window scale factor*: For high throughput channels the window size from the segment header (16 bit, maximum 65,535) may not be sufficient. This 3 byte option defines the multiplication factor for the window size. The actual window size is computed using the following formula

$$\text{Actual window size} = \text{Window size} \cdot 2^{\text{window scale factor}}$$

- *Time stamp*: This is a 10 byte option that holds the time stamp value of transmitted segment. When an acknowledgment is sent, the same value is used. This allows the original sender to determine the optimal retransmission time based on the packet round trip time.

60.2.2 Port Number Assignments

In order to provide a platform for an independent process addressing on a host, each connection of the process to the network is assigned a 16 bit number. There are three categories of port numbers: well-known (0–1,023), registered (1,024–49,151), and ephemeral (49,152–6,553). *Well-known ports* have been historically assigned to common services. Some operating systems require that the process that utilizes those ports must have administrative privileges. This requirement was historically created to avoid hackers running server imposters on multiuser systems. Well-known ports are registered and controlled by Internet Assigned Numbers Authority (IANA) [IANA,STD2]. Table 60.1 shows well-known ports commonly used by TCP. *Registered ports* are also registered by IANA to avoid possible conflicts among different applications attempting to use the same port for listening to incoming connections. *Ephemeral* ports are also called dynamic ports. These ports are used by outgoing connections that are typically assigned to the first available port above 49,151. Many operating systems may not follow IANA AQ4 recommendations and treat the registered ports range as ephemeral. For example, BSD uses ports 1,024 through 4,999 as ephemeral ports, many Linux kernels use 32,768–61,000, Windows use 1,025–5,000, while FreeBSD follows IANA port range.

For example, a Web Server usually uses well-known port 80, a proxyserver may typically use registered port 8,000, 8,008, or 8,080, a Web browser uses multiple ephemeral ports, one for each new connection open to download a component of a Web page. Ephemeral ports are eventually recycled.

60.2.3 Connection Establishment

Before data transfer begins, TCP establishes a connection between two application processes by setting up the protocol parameters. These parameters are stored in a structure variable called TCB.

TABLE 60.1 Common Well-Known Ports Used with TCP

Port	Protocol	Description	RFC/STD #
7	Echo	Echoes a received data back to its sender	STD20
9	Discard	Discards any data that are received	STD21
13	Daytime	Returns the date and the time in ASCII string format	STD25
19	Chargen	Sends back arbitrary characters until connection closed	STD22
20	FTP Data	File Transfer Protocol (data transmission connection, unencrypted)	STD9
21	FTP Control	File Transfer protocol (control connection, unencrypted)	STD9
22	SSH	Secure Shell (SSH)—used for secure logins, file transfers (scp, sftp), and port forwarding	RFC4250-4256
23	Telnet	Telnet (unencrypted)	STD8, STD27-32
25	SMTP	Simple Mail Transfer Protocol (sending email)	STD10
37	Time	Returns the current time in seconds since 1/1/1900 in a 32 bit format	STD26
53	Nameserver	Domain Name Service	STD13
80	HTTP	Hypertext Transfer Protocol	RFC2616
110	POP3	Post Office Protocol 3 (email download)	STD53
111	RPC	SUN Remote Procedure Call	RFC5531
143	IMAP	Interim Mail Access Protocol v2 (email access)	RFC2060
443	SHTTP	Secure Hypertext Transfer Protocol	RFC2660
995	POP3	POP3 over TLS/SSL	RFC2595

Before the two parties—called here as host A and host B—can start sending data, four actions must be performed:

1. Host A sends a segment announcing that it wants to establish connection. This segment includes initial information about traffic from A to B.
2. Host B sends a segment acknowledging the request received from A.
3. Host B sends a segment with information about traffic from B to A.
4. Host A sends a segment acknowledging the request received from B.

The second and the third steps can be combined into one segment. Therefore, the connection establishment described is also called three-way handshaking. Figure 60.6 illustrates an example of segment

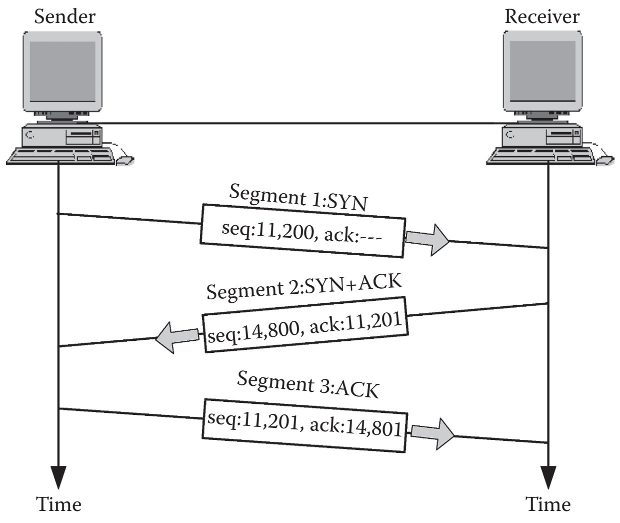


FIGURE 60.6 Three-way handshaking during connection establishment.

exchange during connection establishment. Among the two programs that are involved in the connection, the server program is the one that passively waits for another party to connect. It will be called shortly as the server. It tells the underlying TCP library in its system that it is ready to accept a connection. The program that actively seeks connection is called a client program. It will be called shortly as the client. The client tells the underlying library in its system that it needs to connect to a particular server that is defined by the computer IP address and the destination port number.

1. The client sends the first segment that includes the source and destination port with SYN bit set. The segment also contains the initial value of Sequence Number, which is also called ISN that will be used by the client in this connection. The client may also send some options as necessary, for example, to alter the window size, or maximum segment size in order to change their values from default.
2. The server sends the second segment that has both ACK and SYN bit sets. This is done in order to both acknowledge receiving the first segment from the client (ACK) and to initialize the sequence number in the direction from the server to the client (SYN). The Acknowledgment number is set to the client's ISN + 1. The server ISN is set to a random number. The option to define client window size must also be sent.
3. In reply to the segment from the server, the client sends as segment with ACK bit set and the segment number set to its ISN + 1. The Acknowledgment number is set to the server ISN + 1.
4. In very rare situation, the two processes may want to start connection at the same time. In that case, both will issue an active open segment with SYN and ISN, and then both would send the ACK segment. A situation when one of the segments is lost during the transmission and needs to be resent after time-out may also be analyzed.

60.2.4 Maintaining the Open Connection

After connection is established, each process can send data. Each segment has its sequence number increased by the number of bytes sent in the previous segment, with the previous sequence number. The number rolls over at 2^{32}. The process expects to receive segments with ACK bit set and acknowledgment numbers that confirm that data are received. Acknowledgment number is not merely the repetition of the received sequence number. It indicates the next expected sequence number of a segment to be received by the acknowledging process. Acknowledgments must be received within certain time frame, not every segment that is send needs to be acknowledged. Sending acknowledgment of a certain segment also acknowledges receiving all previous segments. More segments are usually sent ahead before the sender expects to receive an acknowledgment. Sending an acknowledgment can be combined with sending data in the other direction (piggybacking).

60.2.5 Flow Control and Sliding Window Protocol

The amount of data may be sent before waiting for an acknowledgment from the destination. In extreme situations, the process may wait for an acknowledgment for each segment but that would make the long distance transmission very slow due to long latency. On the other hand, the process may send all data without waiting for confirmation. This is also impractical because high volume of data may need to be retransmitted in case a segment is lost in the middle of the transmission.

Sliding window protocol is a solution in-between the two extreme situations described above that TCP implements in order to speed up the transmission. TCP sends as much data as allowed by a so-called sliding window before waiting for an acknowledgment. After receiving an acknowledgment, it moves (slides) the window ahead in the outgoing data buffer, as illustrated in Figure 60.7.

On the receiving side, incoming data are stored in the receiving buffer before data are consumed by the communicating process. Since the size of the buffer is limited, the number of bytes that can be received at a particular time is limited. The size of the unused portion of the buffer is called the *receiver window*. In order to maintain the flow control, the sender must be prevented from sending the excessive

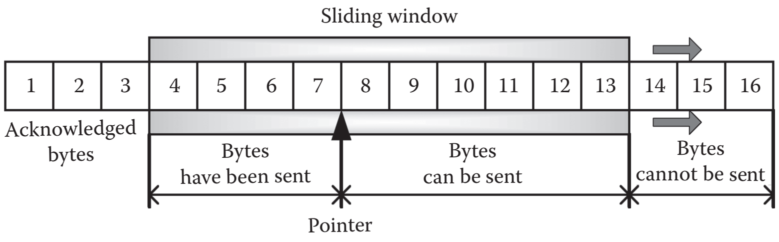


FIGURE 60.7 Sender buffer and sliding window protocol.

amount of data and overflowing the incoming data buffer. This is implemented by controlling the timing when the acknowledgments are sent.

If the receiving process consumes data at a faster rate than they arrive, then the receiver window expands. This information is passed to the sender as an option included in a segment so that the sender may synchronize the size if its *sender window*. This allows more data to be sent without receiving an acknowledgment. If the receiving process consumes data slower than they arrive, then the receiver buffer shrinks. In that case, the receiver must inform the sender to shrink the sender window as well. This causes transmission to wait for more frequent acknowledgments and prevents data from being sent without possibility of storing. If the receiving buffer is full, then the receiver may close (set to zero) the receiver window. The sender cannot send any more data until the receiver reopens the window.

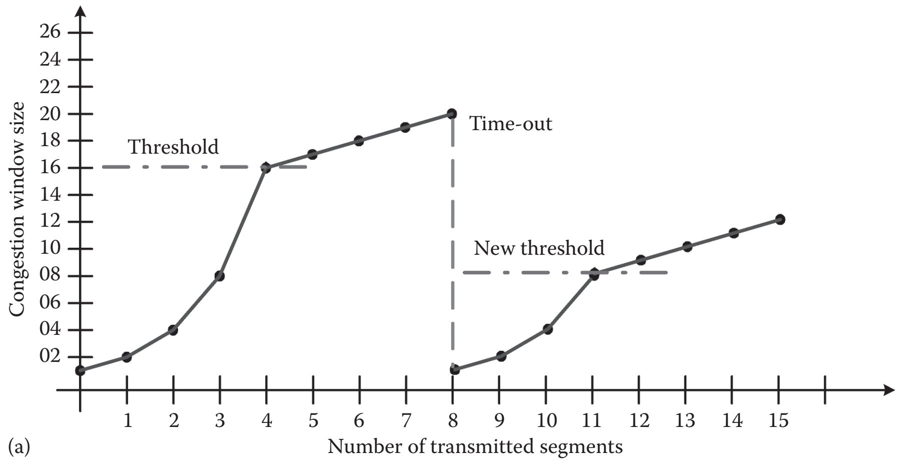
60.2.6 Improving Flow Control

In case when one or both communicating processes process the data slowly, then a problem leading to significant bandwidth waste may arise. In case of slow transmission caused by the receiving processes, the sliding window may be reduced even as low as to one byte. In case of slow transmission caused by the sender, data may be sent as soon as it is produced even with a large window, one byte at a time, as well. This causes the segments carrying small data payload, and lowers the ration of carried data size to the total segment size, and to the total packet size in the underlying protocols. Extensive overview of these and other problems that may develop is provided by [RFC2525] and [RFC2757]. Due to size limitation of this article only two most commonly known problems are discussed and their classic solutions are provided here. AQ5

To prevent this situation, data should not be sent immediately as and when it becomes available, but combined in larger chunks, provided that this approach is suitable for a particular application. Nagle found a very good and simple solution for the sender, which is called *Nagle's Algorithm* [RFC896]. The algorithm can be described by the following three steps:

- The first piece of data are sent immediately even if it is only one byte.
- After sending the first segment, the sender waits for the acknowledgment segment or until enough data are accumulated to fill in the segment of the maximum segment size, whichever happens first. Then, it sends the next segment with data.
- The previous step is repeated until the end of transmission.

On the receiving side, two different solutions are frequently used. Clark proposed to send the acknowledgment as soon as the incoming segment arrives, but to set the window to zero if the receiving buffer is full (Figure 60.8b) [RFC813]. Then, to increase the buffer only after a maximum size of a segment can be received or half of the buffer becomes empty. The other solution is to delay the acknowledgment and thus prevent the sender from sending more segments with data. The latter of the two solutions creates less network traffic but it may cause unnecessary retransmission of segment, if the delay is too long.



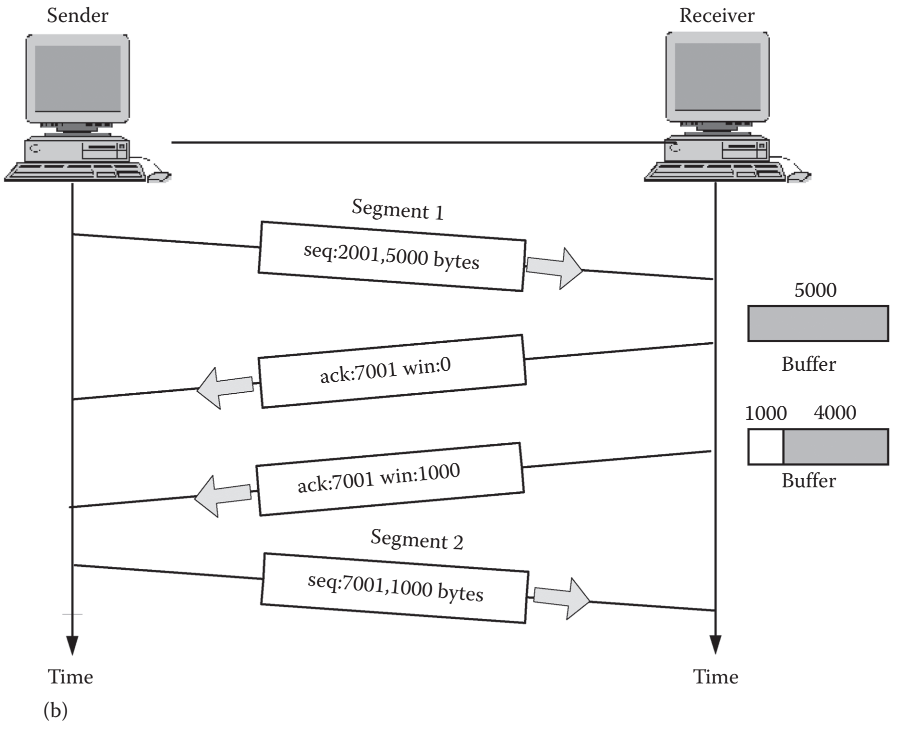


FIGURE 60.8 Congestion Control Window Size. Sender side (a) (upper/left), receiver side control (b) (lower/right).

60.2.7 Error Control

TCP as a reliable protocol that uses unreliable underlying IP must implement error control. This error control includes detecting lost segments, out-of-order segments, corrupted segments, and duplicated segments. After detection, the error must be corrected. The error detection is implemented by using three features: segment checksum, segment acknowledgment, and time-out. Segments with *incorrect checksum* (Figure 60.9a) are discarded. Since TCP uses only positive acknowledgment, no negative acknowledgment can be sent, and both the corrupted segment error and the *lost segment* (Figure 60.9b) error are detected by time-out for receiving an acknowledgment by the sender.

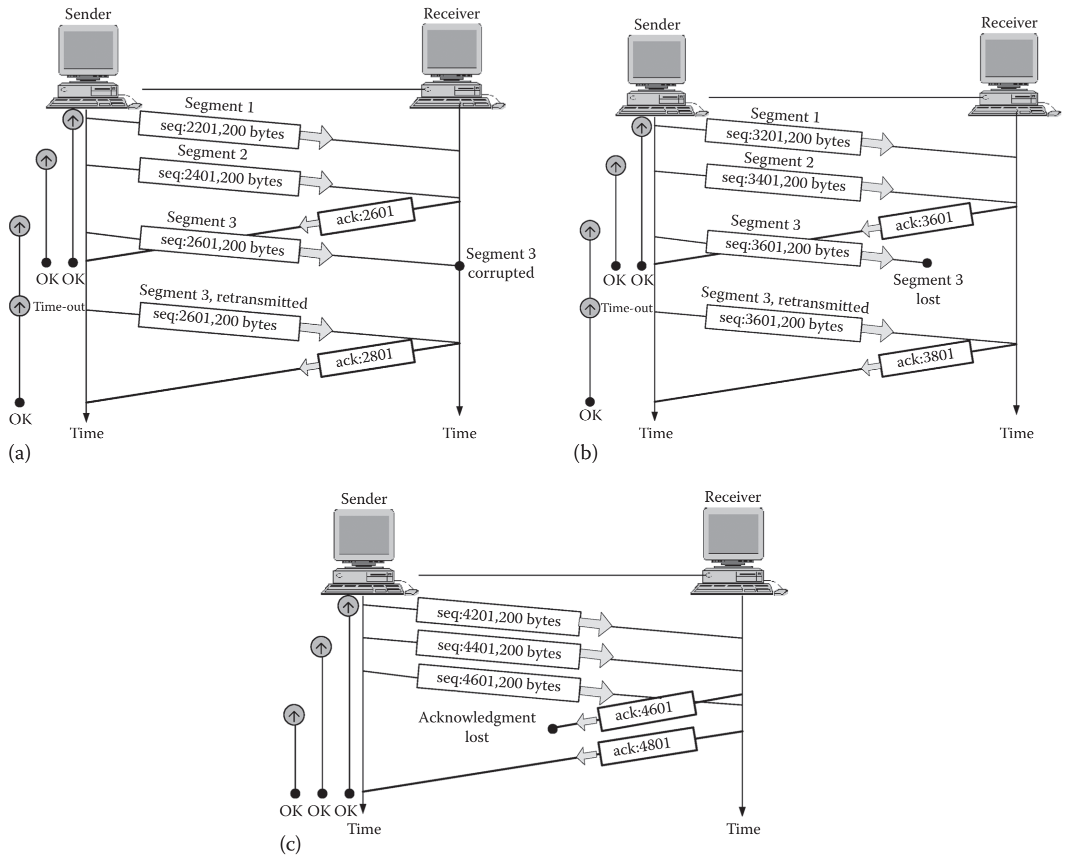


FIGURE 60.9 Corrupted segment (a), lost segment (b), and lost acknowledgment (c).

The error correction is implemented by establishing a *time-out counter* for each segment sent. If the acknowledgment is received, then the particular and all preceding counters and the segments are disposed. If the counter reaches time-out, then the segment is resent and the timer is reset.

A *duplicate segment* can be created if the acknowledgment segment is lost or is delayed too much. Since data are reassembled into a byte-stream at the destination data from the duplicate segment is already in the buffer. The duplicate segment is discarded. No acknowledgment is sent. AQ6

An *out-of-order segment* is stored but not acknowledged before all segments that precede it are received. This may cause resending the segments. The duplicates are discarded.

In case of *lost acknowledgment* (Figure 60.9c), the error is corrected when the acknowledgment for the next segment is sent, since an acknowledgment to the next segment also acknowledges all previous segments. Therefore, a lost acknowledgment may not be noticed at all. This could generate a deadlock in case if the acknowledgment is sent on the last segment and for a time when there is no more segments to send. This deadlock is corrected by a sender that uses a so-called *persistence timer*. If no acknowledgment comes within a certain time frame, then the sender sends a 1 byte segment called probe and resets the persistence counter. With each attempt, the persistence timer time-out is doubled until 60 s time is reached. After receiving an acknowledgment, the persistence timer time-out is reset to the initial value.

Keep-alive timer is used to prevent a long idle connection. If no segments are exchanged for a given period of time, usually 2 h, the server send a probe. If there is no response after 10 probes the connection is reset and thus terminated.

Time-waited timer is used during the connection termination procedure that is described later. If there is no acknowledgment to the FIN segment, the FIN segment is resent after time-waited timer expires. Duplicate FIN segments are discarded by the destination.

60.2.8 Congestion Control

In case of the congestion on the network, packets could be dropped by routers. TCP assumes that the cause of a lost segment is due to the network congestion. Therefore, an additional mechanism is built in into TCP to prevent prompt resending of packets and creating even more congestion. This mechanism is implemented by additional control of the sender window, as described below and illustrated in Figure 60.8a.

- The sender window size is always set to the smaller of the two values—the receiver window size and the *congestion window size*.
- The congestion window size starts at the size of two maximum segments.
- With each received acknowledgment, the congestion window size is doubled until certain threshold. After reaching the threshold the further increase is additive.
- With each acknowledgment timed out, the threshold is reduced by half and the congestion window size is reset to the initial size of two maximum segment size.

60.2.9 Connection Termination

Any of the two application processes that established a TCP connection can close that connection. When a connection in one direction is terminated, the other direction is still functional, until the second process also terminates it. Therefore, four actions must be performed to close the connection in both directions:

1. Host A (either client or server) sends a segment announcing that it wants to terminate the connection.
2. Host B sends a segment acknowledging the request of A. At this moment, the connection from A to B is closed, but from B to A is still open.
3. When host B is ready to terminate the connection, for example, after sending the remaining data, it sends a segment announcing that it wants to terminate the connection as well.
4. Host A sends a segment acknowledging the request of B. At this moment, the connection in both directions is closed.

Unlike in case of opening the connection, none of the steps can be combined into one segment. Therefore, the connection termination described is also called *four-way handshaking*. Figure 60.10 illustrates an example of segment exchange during connection termination. Because it does not matter which host is a server and which is a client, they are denoted only as A and B.

1. Host A sends a segment with FIN bit set.
2. Host B sends a segment with ACK bit set to confirm receiving the FIN segment from host A.
3. Host B may continue to send data and expect to receive acknowledgment messages with no data from host A. Eventually, when there is nothing more to send, it sends a segment with FIN bit set.
4. Host A sends the last segment in the transmission with ACK bit set to confirm receiving the FIN segment from host B.

The connection termination procedure described above is so called as graceful termination. In certain cases, the connection may be terminated abruptly by resetting. This may happen in the following cases:

- The client process requested a connection to a nonexistent port. The TCP library, on the other side, may send a segment with the RST bit set to indicate connection refusal. AQ7

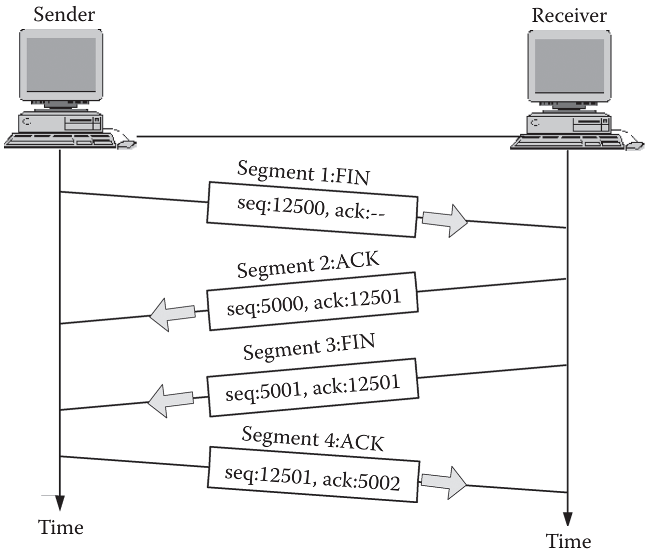


FIGURE 60.10 Four-way handshaking during connection termination.

- One of the processes wants to abandon the connection because of an abnormal situation. It can send the RST segment to destroy the connection.
- One of the processes determines that the other side is idle for a long time. The other system is considered idle if it does not send any data, any acknowledgments, or keep-alive segments. The awaiting process may send the RST segment to destroy the connection.

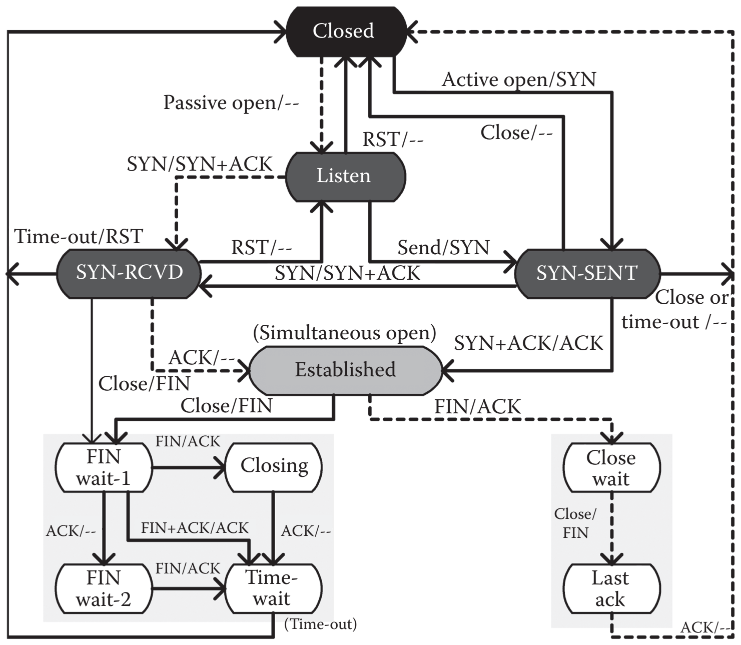


FIGURE 60.11 TCP State Transition Diagram.

60.3 State Diagram

The state transition diagram for software that implements TCP is presented. Figure 60.11 shows the state transition diagram for the server and for the client. The dashed lines indicate transitions in a server. The solid lines indicate transitions in a client.

60.4 Programming Samples

Figures 60.12 and 60.13 contain two C programs that demonstrate the use of TCP for communication AQ8 using client–server paradigm [C02-3]. Socket API is a *de facto* standard. Programs use conditional compilation directives that include minor variations in the library headers needed for operation under

```
/* * server.c - code for example server program that uses TCP * */
#ifndef unix
#include <winsock2.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <string.h>

#define PORT 1200             /* server TCP port number             */
#define QLEN 3                /* size of connection request queue   */

int main()
{
    SOCKET  sd, sd2;         /* socket descriptors - (integers)     */
    struct  protoent *ptrp;  /* pointer to a protocol table entry   */
    struct  sockaddr_in sad; /* structure to hold server's address  */
    struct  sockaddr_in cad; /* structure to hold client's address  */
    int     alen;            /* length of address                   */

#ifdef WIN32
    WSADATA wsaData;
    if(WSAStartup(0x0101, &wsaData)!=0)
    { fprintf(stderr, "Windows Socket Init failed: %d\n", GetLastError()); exit(1); }
#endif

    /* prepare the server socket information */
    memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
    sad.sin_family = AF_INET;           /* set family to Internet   */
    sad.sin_addr.s_addr = INADDR_ANY;   /* set the local IP address */
    sad.sin_port = htons((u_short)PORT);/* use the defined port num */

    /* Map TCP transport protocol name to protocol number */
    ptrp = getprotobyname("tcp");
    if ( ptrp == 0) {
        fprintf(stderr, "cannot map \"tcp\" to protocol number\n"); exit(1); }

    /* Create a socket */
    sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
    if (sd < 0) { fprintf(stderr, "socket creation failed\n"); exit(1); }

    /* Bind a local address to the socket */
    if (bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0)
        { fprintf(stderr, "bind failed\n"); exit(1); }

    /* Specify size of request queue */
    if (listen(sd, QLEN) < 0) { fprintf(stderr, "listen failed\n"); exit(1); }
```
(a)

AQ9 **FIGURE 60.12** Sample C program that illustrates a TCP server.

```
    { /* Run the sample server application */
        char buf[1000];      /* buffer for string the server sends  */
        char name[80];       /* small buffer for received user data */
        int  visits = 0;     /* counts client connections           */

        while (1) { /* Main server loop - accept and handle requests */
            alen = sizeof(cad);
            sd2=accept(sd, (struct sockaddr *)&cad, &alen);
            if ( sd2<0) { fprintf(stderr, "accept failed\n"); continue; }

            /* This block could be run in a separate thread
               to allow for servicing concurrent clients */
            sprintf(buf,"Welcome to the server.\r\nWhat is your name please?\r\n");
            send(sd2,buf,strlen(buf),0);

            { /* collect one line of data from the user */
                int len=0; char bch;
                while (1) {
                    if (recv(sd2,&bch,1,0) < 0) break; /* client disconnected */
                    else if (bch=='\n') break;      /* done, received a line of text */
                    else if (bch=='\r')             /* ignore \r */ ;
                    else { name[len]=bch; len++; }  /* received another cahracter */
                    if (len+1>=sizeof(name)) break; // exit on exhausted data buffer
                }
                name[len]='\0';
            }
            fprintf(stdout, "%s has just contacted us.\n", name);

            visits++;
            sprintf(buf,"Nice to meet you %s.\r\nYou are the %dth connection today\r\n",
                name, visits);
            send(sd2,buf,strlen(buf),0);
            closesocket(sd2);
            /* End of the block that could be run in a separate thread */
        }
    }

#ifdef WIN32
    WSACleanup();                      /* release use of winsock.dll */
#endif
    return(0);
}
```

(b)

AQ9 **FIGURE 60.12 (continued)**

Unix/Linux and Windows. Function calls should be modified if desired to run under other operating systems or with proprietary operating systems for embedded devices. Minor modifications need to be performed when utilizing underlying IPv6 API, as described in [RFC3493]. Figure 60.12 shows the server program that waits for the incoming connection from the client program listed in Figure 60.13. To allow maximum compatibility, the sample server does not use multithreading when servicing client connections, thus allowing only one connection at a time. However, such modification can be easily made by relocating code for servicing a client (marked by comments) into a function run in a separate thread, each time a new client is connected. Programs show all steps necessary to set-up, use, and tear down a TCP connection.

Internet socket is an end point of a bidirectional interprocess communication flow across an IP-based computer network. Interface to the socket library is standardized only on Unix/Linux and Windows. Other operating systems, including various embedded systems, may employ different functions to carry out the same functionality. Programs of the same functionality implemented in other languages may combine several steps into calling a single function. This is especially true for the socket initialization and connection set up.

```
/* * code for example client program that uses TCP * */
#ifndef unix
#include <winsock2.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <string.h>

#define PORT 1200             /* server TCP port number             */
char    hostname[] = "localhost";    /* server host name            */

int main()
{
    SOCKET  sd;              /* socket descriptor - (integer)       */
    struct  hostent  *ptrh;  /* pointer to a host table entry       */
    struct  protoent *ptrp;  /* pointer to a protocol table entry   */
    struct  sockaddr_in sad; /* structure to hold an IP address     */
    char    host[256];       /* pointer to host name                */

#ifdef WIN32
    WSADATA wsaData;
    if(WSAStartup(0x0101, &wsaData)!=0)
    { fprintf(stderr, "Windows Socket Init failed: %d\n", GetLastError()); exit(1); }
#endif

    /* prepare the client socket information */
    memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
    sad.sin_family = AF_INET;         /* set family to Internet     */
    strcpy(host,hostname);

    /* Convert host name to equivalent IP address and copy to sad. */
    ptrh = gethostbyname(host);
    if ( ((char *)ptrh) == NULL )
        { fprintf(stderr, "invalid host: %s\n", host); exit(1); }
    memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);
    sad.sin_port = htons((u_short)PORT); /* server port number on remote host */

    /* Map TCP transport protocol name to protocol number. */
    ptrp = getprotobyname("tcp");
    if ( ptrp == 0) { fprintf(stderr, "cannot map \"tcp\" to number\n"); exit(1); }

    /* Create a socket. */
    sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
    if (sd < 0) { fprintf(stderr, "soclet creation failed\n"); exit(1); }

    /* Connect the socket to the specified server. */
    if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0)
        { fprintf(stderr, "connection failed\n"); exit(1); }

    /* Note: May want/need to read the initial greeting sent to us by the server */
    {   char buf[1000];      /* buffer for data sent to the server  */
        sprintf(buf,"Automated Client\r\n"); send(sd, buf, strlen(buf),0);
    }

    {   int  n;              /* number of characters read           */
        char buf[1000];      /* buffer for data from the server     */
        /* Repeatedly read data from socket and write to user's screen. */
        n = recv(sd, buf, sizeof(buf), 0);
        while (n > 0) { fwrite(buf, n, 1, stdout); n = recv(sd, buf, sizeof(buf), 0); }
 /* Note: data may come in chunks of different size than chunks sent by the server */
    }
    closesocket(sd);        /* close as soon as no longer needed    */

#ifdef WIN32
    WSACleanup();                      /* release use of winsock.dll */
#endif
    return(0);
}
```

FIGURE 60.13 Sample C program that illustrates a TCP client.

References

[C02–1] D. Comer, *Internetworking with TCP/IP*, Vol. 1: *Principles, Protocols, and Architecture*, 5th edn., Prentice Hall, Upper Saddle River, NJ, 2005.

[C02-3] D. Comer et al., *Internetworking with TCP/IP*, Vol. III: *Client-Server Programming and Applications, Linux/Posix Sockets Version*, Prentice Hall, Upper Saddle River, NJ, 2000.

[F10] B.A. Forouzan, *TCP/IP Protocol Suite*, 4th edn., McGraw-Hill, New York, 2010.

[GW03] A. Leon-Garcia, I. Widjaja, *Communication Networks*, 2nd edn., McGraw-Hill, New York, 2003.

[IANA] Web Site for IANA—Internet Assigned Numbers Authority, http://www.iana.org/

[PD07] L.L. Peterson, B.S. Davie, *Computer Networks: A System Approach*, 4th edn., Morgan Kaufmann Publishers, San Francisco, CA, 2007.

[RFC] Web Site for *RFC Editor*, http://www.rfc-editor.org/

[RFC813] D. Clark, Window and Acknowledgement Strategy in TCP, July 1982, http://www.rfc-editor.org/

[RFC896] J. Nagle, Congestion Control in IP/TCP Internetworks, January 1984, http://www.rfc-editor.org/

[RFC2147] D. Borman et al., TCP and UDP over IPv6 Jumbograms, May 1997, http://www.rfc-editor.org/

[RFC2525] V. Paxson et al., Known TCP Implementation Problems, March 1999, http://www.rfc-editor.org/

[RFC2675] D. Borman et al., IPv6 Jumbograms, August 1999, http://www.rfc-editor.org/

[RFC2757] G. Montenegro et al., Long Thin Networks, January 2000, http://www.rfc-editor.org/

[RFC3493] R. Gilligan et al., Basic Socket Interface Extensions for IPv6, February 2003, http://www.rfc-editor.org/

[STD2] J. Reynolds, J. Postel, Assigned Numbers, October 1994, http://www.rfc-editor.org/

[STD5] J. Postel, Internet Protocol, September 1981, http://www.rfc-editor.org/

[STD7] RFC0793/STD0007, J. Postel (ed.), Transmission Control Protocol DARPA Internet Program Protocol Specification, September 1981, http://www.rfc-editor.org/