transport protocols

7.1 Introduction

As we saw earlier in Section 1.2, although the range of Internet applications (and the different types of access network used to support them) are many and varied, the protocol suites associated with the different application/network combinations have a common structure. As we saw in Section 1.2.2, the different types of network operate in a variety of modes – circuit-switched or packet-switched, connection-oriented or connectionless – and hence each type of network has a different set of protocols for interfacing to it. Above the network-layer protocol, however, all protocol suites comprise one or more application protocols and a number of what are called application-support protocols.

For example, in order to mask the application protocols from the services provided by the different types of network protocols, all protocol suites have one or more transport protocols. These provide the application protocols with a network-independent information interchange service and, in the case of the TCP/IP suite, they are the **transmission control protocol** (**TCP**) and the **user datagram protocol** (**UDP**). TCP provides a connection-oriented (reliable) service and UDP a connectionless (best-effort) service. Normally, both protocols are present in the suite and the choice of protocol used is determined by the requirements of the application. In addition, when the

application involves the transfer of streams of audio and/or video in real time, the timing information required by the receiver to synchronize the incoming streams is provided by the **real-time transport protocol** (**RTP**) and its associated **real-time transport control protocol** (**RTCP**). There is also a version of TCP for use with wireless networks. We describe the operation of these five protocols and the services they provide to application protocols.

7.2 TCP/IP protocol suite

Before describing the various protocols, it will be helpful to illustrate the position of each protocol relative to the others in the TCP/IP suite. This is shown in Figure 7.1. Normally, the IP protocols and network-dependent protocols below them are all part of the operating system kernel with the various application protocols implemented as separate programs/processes. The two transport protocols, TCP and UDP, are then implemented to run either within the operating system kernel, as separate programs/processes, or in a library package linked to the application program.

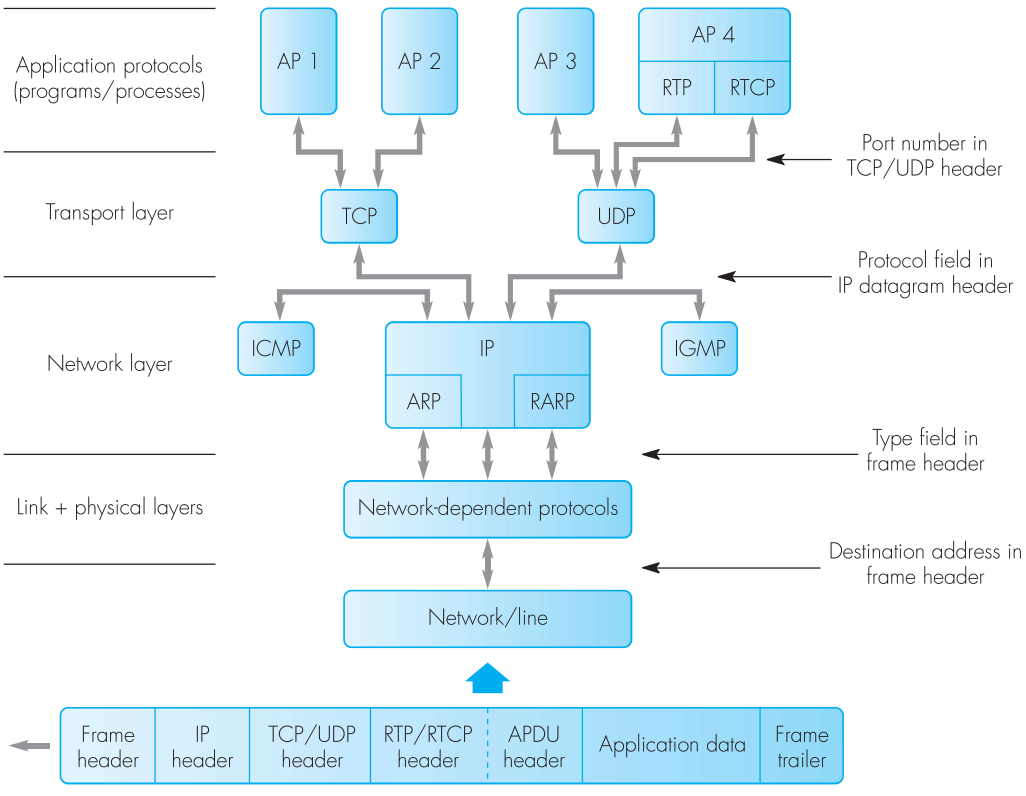


Figure 7.1 TCP/IP protocol suite and interlayer address selectors.

With most networked applications, the client–server paradigm is used. The client application protocol/program runs in one computer – typically a PC or workstation – and this communicates with a similar (peer) application program that runs (normally continuously) in a remote server computer. Examples of applications of this type are file transfers and the messages associated with electronic mail, both of which require a reliable service; that is, the transferred information should be free of transmission errors and the messages delivered in the same sequence that they were submitted. Hence applications of this type use the reliable service provided by TCP.

Thus the role of TCP is to convert the best-effort service provided by IP into a reliable service. For other applications, a simple best-effort service is acceptable and hence they use UDP as the transport protocol. Examples of applications of this type are interpersonal applications that involve the transfer of streams of (compressed) audio and/or video in real time. Clearly, since new information is being received and output continuously, it is inappropriate to request blocks that are received with errors to be retransmitted. Also, it is for applications of this type that RTP and RTCP are used. Other applications that use UDP are application protocols such as HTTP and SNMP, both of which involve a single request-response message exchange.

As we saw in Figure 6.1 and its accompanying text, all message blocks – protocol data units (PDUs) – relating to the protocols that use the services of the IP layer are transferred in an IP datagram. Hence, as we can deduce from Figure 7.1, since there are a number of protocols that use the services of IP – TCP, UDP, ICMP and IGMP – it is necessary for IP to have some means of identifying the protocol to which the contents of the datagram relate. As we saw in Section 6.2, this is the role of the *protocol* field in each IP datagram header. Similarly, since a number of different application protocols may use the services of both TCP and UDP, it is also necessary for both these protocols to have a field in their respective PDU header that identifies the application protocol to which the PDU contents relate. As we shall see, this is the role of the source and destination *port numbers* that are present in the header of the PDUs of both protocols. In addition, since a server application receives requests from multiple clients, in order for the server to send the responses to the correct clients, both the source port number and the source IP address from the IP datagram header are sent to the application protocol with the TCP/UDP contents.

In general, within the client host, the port number of the source application protocol has only local significance and a new port number is allocated for each new transfer request. Normally, therefore, client port numbers are called **ephemeral ports** as they are short-lived. The port numbers of the peer application protocol in the server application protocols are fixed and are known as **well-known port numbers**. Their allocation is managed by ICANN and they are in the range 1 through to 1023. For example, the well-known port number of the server-side of the file transfer (application) protocol (FTP) is 21. Normally, ephemeral port numbers are allocated in the range 1024 through to 5000.

As we can see in Figure 7.2, all the protocols in both the application and transport layers communicate directly with a similar peer protocol in the remote host computer (end system). The protocols in both these layers are said, therefore, to communicate on an end-to-end basis. In contrast, the IP protocols present in each of the two communicating hosts are network-interface protocols. These, together with the IP in each intermediate gateway/router involved, carry out the transfer of the datagram across the internetwork. The IP protocol in each host is said to have local significance and the routing of each datagram is carried out on a hop-by-hop basis.

7.3 TCP

The transmission control protocol (TCP) provides two communicating peer application protocols – normally one in a client computer and the other in a server computer – with a two-way, reliable data interchange service. Although the APDUs associated with an application protocol have a defined structure, this is transparent to the two communicating peer TCP protocol entities which treat all the data submitted by each local application entity as a stream of bytes. The stream of bytes flowing in each direction is then transferred (over the underlying network/internet) from one TCP entity to the other in a reliable way; that is, to a high probability, each byte in the stream flowing in

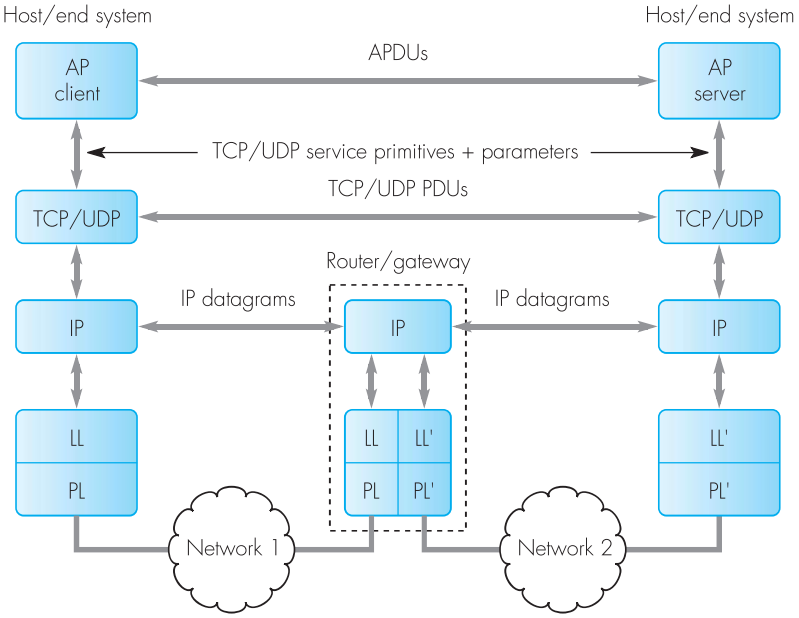


Figure 7.2 TCP/IP protocol suite interlayer communications.

each direction is free of transmission errors, with no lost or duplicate bytes, and the bytes are delivered in the same sequence as they were submitted. The service provided by TCP is known, therefore, as a **reliable stream service**.

As we explained in Section 6.2, the service provided by IP is an unreliable best-effort service. Hence in order to provide a reliable service, before any data is transferred between the two TCP entities, a logical connection is first established between them in order for the sequence numbers in each TCP entity – which are required for error correction and flow control purposes – to be initialized. Also, once all data has been transferred in both directions, the logical connection is closed.

During the actual data transfer phase, in order for the receiving TCP to detect the presence of transmission errors, each TCP entity divides the submitted stream of bytes into blocks known as **segments**. For interactive applications involving a user at a terminal, a segment may contain just a single byte, while for large file transfers a segment may contain many bytes. There is an agreed **maximum segment size** (**MSS**) used with a connection that is established by the two peer TCP entities during the setting up of the connection. This is such that an acceptable proportion of segments are received by the destination free of transmission errors. The default MSS is 536 bytes although larger sizes can be agreed. Normally, the size chosen is such that no fragmentation is necessary during the transfer of a segment over the network/internet and hence is determined by the path MTU. The TCP protocol then includes a retransmission procedure in order to obtain error-free copies of those segments that are received with transmission errors.

In addition, the TCP protocol includes a flow control procedure to ensure no data is lost when the TCP entity in a fast host – a large server for example – is sending data to the TCP in a slower host such as a PC. It also includes a congestion control procedure which endeavors to control the rate of entry of segments into the network/internet to the rate at which segments are leaving.

In the following subsections we discuss firstly the user services provided by TCP, then selected aspects of the operation of the TCP protocol, and finally the formal specification of the protocol. Collectively these are defined in RFCs 793, 1122 and 1323.

7.3.1 User services

The most widely used set of user service primitives associated with TCP are the **socket primitives** used with Berkeley Unix. Hence, although there are a number of alternative primitives, in order to describe the principles involved, we shall restrict our discussion to these. They are operating system calls and collectively form what is called an **application program interface** (**API**) to the underlying TCP/IP protocol stack. A typical list of primitives is given in Table 7.1 and their use is shown in diagrammatic form in Figure 7.3.

Table 7.1 List of socket primitives associated with TCP and their parameters.

Primitive	*Parameters*
socket ( )	service type, protocol, address format, return value = socket descriptor or error code
bind ( )	socket descriptor, socket address (= host IP address + port number), return value = success or error code
listen ( )	socket descriptor, maximum queue length, return value = success or error code
accept ( )	socket descriptor, socket address, return value = success or error code
connect ( )	socket descriptor, local port number, destination port number, destination IP address, precedence, optional data (for example a user name and a password), return value = success or error code
send ( )	socket descriptor, pointer to message buffer containing the data to send, data length (in bytes), push flag, urgent flag, return value = success or error code
receive ( )	socket descriptor, pointer to a message buffer into which the data should be put, length of the buffer, return value = success or end of file (EOF) or error code
close ( )	socket descriptor, return value = success or error code
shutdown ( )	socket descriptor, return value = success or error code

Each of the two peer user application protocols/processes (APs) first creates a communications channel between itself and its local TCP entity. This is called a **socket** or **endpoint** and, in the case of the server AP, involves the AP issuing a sequence of primitive (also known as system of function) calls each with a defined set of parameters associated with it: *socket( )*, *bind( )*, *listen( )*, *accept( )*. Each call has a return value(s) or an error code associated with it.

The parameters associated with the *socket( )* primitive include the service required (reliable stream service), the protocol (TCP), and the address format (Internet). Once a socket has been created – and send/receive memory buffers allocated – a **socket descriptor** is returned to the AP which it then uses with each of the subsequent primitive calls. The AP then issues a

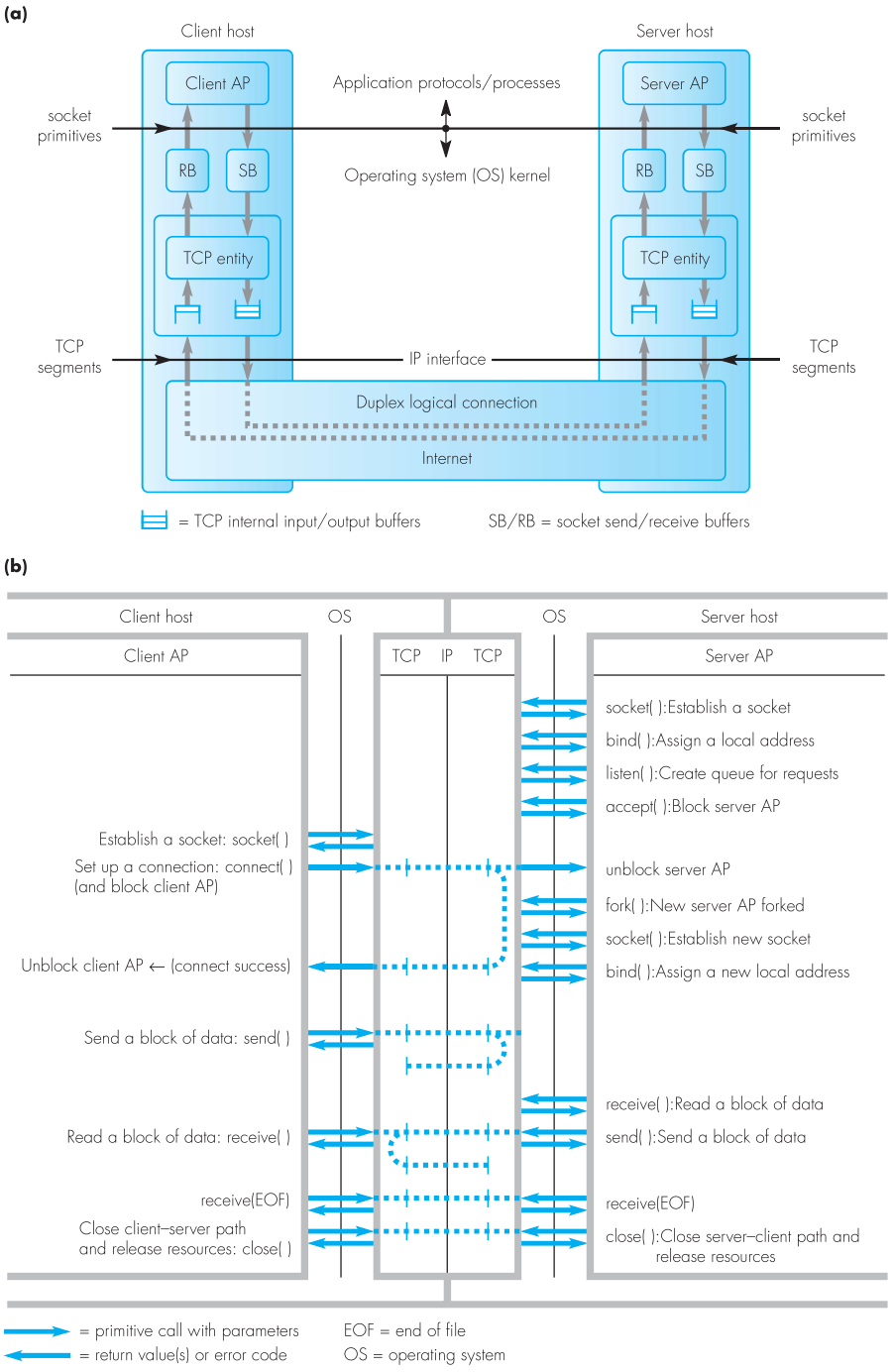


Figure 7.3 TCP socket primitives: (a) socket interface; (b) primitives and their use.

bind( ) primitive, which, in addition to the socket descriptor, has an address parameter associated with it. This is the address the AP wishes to be assigned to the newly created socket and is called the **socket address**. This comprises the Internet-wide IP address of the host and, in the case of the server AP, the 16-bit well-known port number associated with this type of application protocol (FTP and so on).

The *listen( )* primitive call results in the local TCP entity creating a queue (whose maximum length is given as a parameter) to hold incoming connection requests for the server AP. The *accept( )* primitive is then used to put the AP in the blocked state waiting for an incoming connection request to be received from a client TCP entity. Collectively, this sequence of four primitives forms what is called a **passive-open**.

In the case of a client AP, since it can only set up a single TCP connection at a time and the socket address has only local significance, it simply issues a *socket( )* primitive to create a new socket with the same parameters as those used by the server (AP). This is then followed by a *connect( )* primitive, which, in addition to the locally allocated socket descriptor, has parameters that contain the IP address of the remote (server) host, the well-known port number of the required server AP, the local port number that has been assigned to this socket by the client AP, a precedence value, and an optional item of data such as a user name and a password.

The local port number, together with the host IP address, forms the address to be assigned to this socket. The precedence parameter is a collection of parameters that enable the IP protocol to specify the contents of the *type of service* field in the header of the IP datagram that is used to transfer the segments associated with the connection over the Internet. We identified the contents of this field in Figure 6.2 when we discussed the operation of the IP protocol. Note that the IP address of the remote (server) host and the precedence parameters are used by the IP protocol and not TCP. They are examples of what are called **pass-through parameters**, that is, a parameter that is passed down from one protocol layer to another without modification.

Once the *connect( )* call has been made, this results in the calling AP being put into the blocked state while the local TCP entity initiates the setting up of a logical connection between itself and the TCP entity in the server. Collectively, these two primitives form what is called an **active-open**.

The TCP entity in a client host may support multiple concurrent connections involving different user APs. Similarly, the TCP entity in a server may support multiple connections involving different clients. Hence in order for the two TCP entities to relate each received segment to the correct connection, when each new connection is established, both TCP entities create a **connection record** for it. This is a data structure that contains a *connection identifier* (comprising the pair of socket addresses associated with the connection), the agreed *MSS* for the connection, the *initial sequence number* (associated with the acknowledgment procedure) to be used in each direction, the *precedence value*, the *size of the window* associated with the TCP flow control procedure, and a number of fields associated with the operation of the protocol entity including *state variables* and the current state of the protocol entity.

At the server side, when a new connection request (PDU) is received, the server AP is unblocked and proceeds to create a new instance of the server AP to service this connection. Typically, this is carried out using the Unix *fork primitive*. A new socket between the new AP and the local TCP entity is then created and this is used to process the remaining primitives associated with this connection. The parent server AP then either returns to the blocked state waiting for a new connection request to arrive or, if one is already waiting in the server queue, proceeds to process the new request. Once a new instance of the server AP has been created and linked to its local TCP entity by a socket, both the client and server APs can then initiate the transfer of blocks of data in each direction using the *send( )* and *receive( )* primitives.

Associated with each socket is a *send buffer* and a *receive buffer*. The send buffer is used by the AP to transfer a block of data to its local TCP entity for sending over the connection. Similarly, the receive buffer is used by the TCP entity to assemble data received from the connection ready for reading by its local AP. The *send( )* primitive is used by an AP to transfer a block of data of a defined size to the send buffer associated with the socket ready for reading by its local TCP entity. The parameters include the local socket descriptor, a pointer to the memory buffer containing the block of data, and also the length (in bytes) of the block of data. With TCP there is no correlation between the size of the data block(s) submitted (by an AP to its local TCP entity for sending) and the size of the TCP segments that are used to transfer the data over the logical connection. As we saw in Section 7.2, normally the latter is determined by the path MTU and, in many instances, this is much smaller than the size of the data blocks submitted by an AP.

With some applications, however, each submitted data block may be less than the path MTU. For example, in an interactive application involving a user at a keyboard interacting with a remote AP, the input data may comprise just a few bytes/characters. So in order to avoid the local TCP entity waiting for more data to fill an MTU, the user AP can request that a submitted block of data is sent immediately. This is done by setting a parameter associated with the *send( )* primitive called the *push flag*. A second parameter called the *urgent flag* can also be set by a user AP. This again is used with interactive applications to enable, for example, a user AP to abort a remote computation that it has previously started. The (urgent) data – string of characters – associated with the abort command are submitted by the source AP with the urgent flag set. The local TCP entity then ceases waiting for any further data to be submitted and sends what is outstanding, together with the urgent data, immediately. On receipt of this, the remote TCP entity proceeds to interrupt the peer user AP, which then reads the urgent data and acts upon it.

Finally, when a client AP has completed the transfer of all data blocks associated with the connection, it initiates the release of its side of the connection by issuing a *close( )* – or sometimes a *shutdown( )* – primitive. When the server AP is informed of this (by the local TCP entity), assuming it also has finished sending data, it responds by issuing a *close( )* primitive to release the other side of the connection. Both TCP entities then delete their connection records

and also the server AP that was forked to service the connection. As we shall expand upon later, the *shutdown( )* primitive is used when only half of the connection is to be closed.

7.3.2 Protocol operation

As we can see from the above, the TCP protocol involves three distinct operations: setting up a logical connection between two previously created sockets, transferring blocks of data reliably over this connection, and closing down the logical connection. In practice, each phase involves the exchange of one or more TCP segments (PDUs) and, since all segments have a common structure, before describing the three phases we first describe the usage of the fields present in each segment header.

Segment format

All segments start with a common 20-byte header. In the case of acknowledgment and flow control segments, this is all that is present. In the case of connection-related segments, an options field may be present and a data field is present when data is being transferred over a connection. The fields making up the header are shown in Figure 7.4(a).

The 16-bit *source port* and *destination port* fields are used to identify the source and destination APs at each end of a connection. Also, together with the 32-bit source and destination IP addresses of the related hosts, they form the 48-bit socket address and the 96-bit connection identifier. Normally, the port number in a client host is assigned by the client AP while the port number in a server is a well-known port.

The *sequence number* performs the same function as the send sequence number in the HDLC protocol and the *acknowledgment number* the same function as the receive sequence number. Also, as with HDLC, a logical connection involves two separate flows, one in each direction. Hence the *sequence number* in a segment relates to the flow of bytes being transmitted by a TCP entity and the *acknowledgment number* relates to the flow of bytes in the reverse direction. However, with the TCP, although data is submitted for transfer in blocks, the flow of data in each direction is treated simply as a stream of bytes for error and flow control purposes. Hence the *sequence* and *acknowledgment numbers* are both 32-bits in length and relate to the position of a byte in the total session stream rather than to the position of a message block in the sequence. The *sequence number* indicates the position of the first byte in the *data* field of the segment relative to the start of the byte stream, while the *acknowledgment number* indicates the byte in the stream flowing in the reverse direction that the TCP entity expects to receive next.

The presence of an *options* field in the segment header means that the header can be of variable length. The *header length* field indicates the number of 32-bit words in the header. The 6-bit *reserved* field, as its name implies, is reserved for possible future use.

All segments have the same header format and the validity of selected fields in the segment header is indicated by the setting of individual bits in

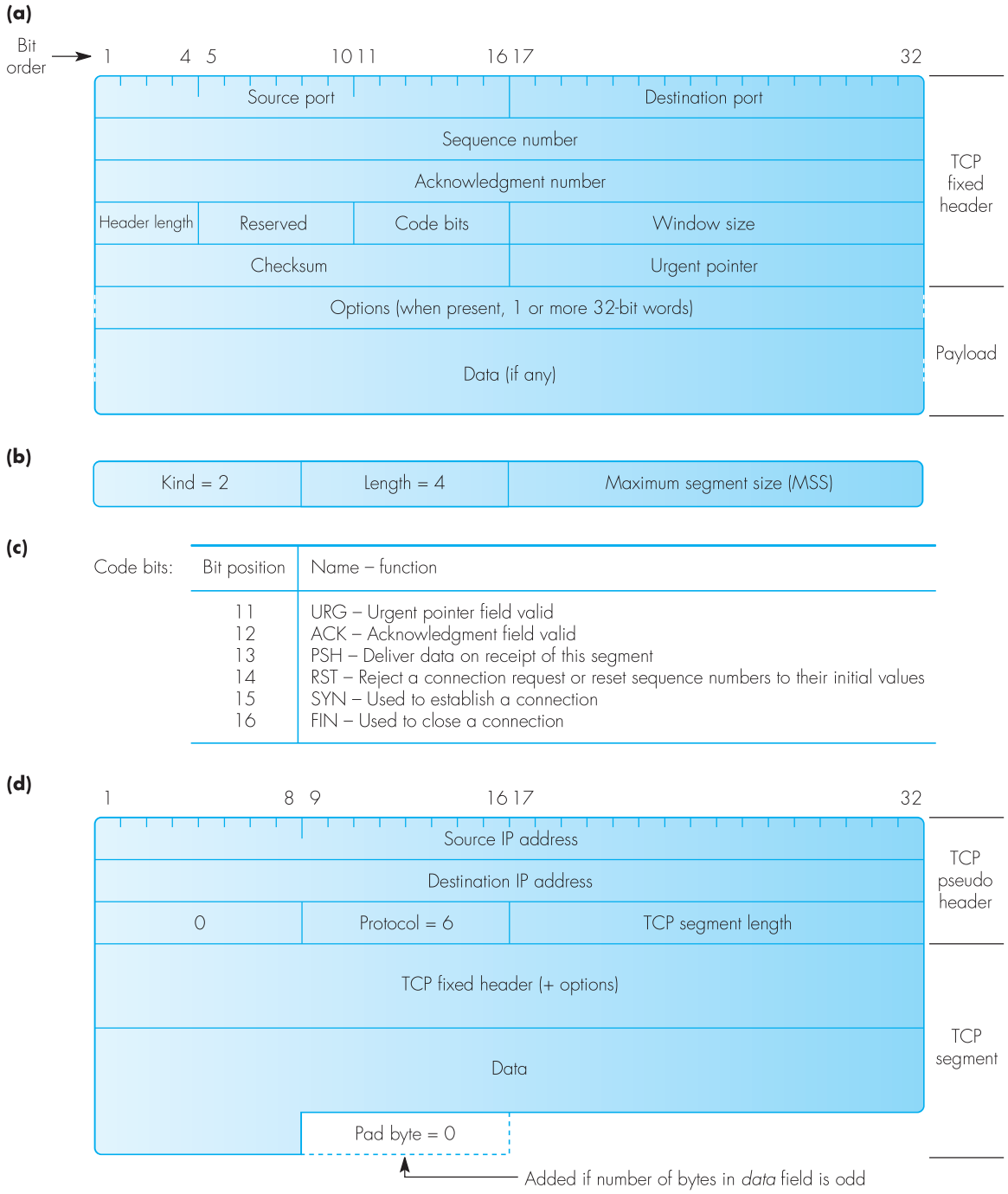


Figure 7.4 TCP segment format: (a) header fields; (b) MSS option format; (c) code bit definitions; (d) pseudo header fields.

the 6-bit *code* field; if a bit is set (a binary 1), the corresponding field is valid. Note that multiple bits can be set in a single segment. The bits have the meaning shown in Figure 7.4(c).

The *window size* field relates to a sliding window flow control scheme the principles of which we considered in Section 1.4.4. The number in the window size indicates the number of data bytes (relative to the byte being acknowledged in the *acknowledgment* field) that the receiver is prepared to accept. It is determined by the amount of unused space in the receive buffer the remote TCP entity is using for this connection. The maximum size of the receive buffer – and hence the maximum size of the window – can be different in each direction and has a default value which, typically, is 4096, 8192 or 16 384 bytes.

As we saw in Section 6.2, the checksum field in the header of each IP datagram applies only to the fields in the IP header and not the datagram contents. Hence the *checksum* field in the TCP segment header covers the complete segment; that is, header plus contents. In addition, since only a simple checksum is used to derive the checksum value in the IP header, in order to add an additional level of checking, some selected fields from the IP header are also included in the computation of the TCP checksum. The fields used form what is called the (TCP) **pseudo header** and these are identified in Figure 7.4(d).

As we can see, these are the source and destination IP addresses and the protocol value (=6 for TCP) from the IP header, plus the total byte count of the TCP segment (header plus contents). The computation of the checksum uses the same algorithm as that used by IP. As we saw in Section 6.2, this is computed by treating the complete datagram as being made up of a string of 16-bit words that are then added together using 1s complement arithmetic. Since the number of bytes in the original TCP segment *data* field may be odd, in order to ensure the same checksum is computed by both TCP entities, a **pad byte** of zero is added to the data field whenever the number of bytes in the original *data* field is odd. As we can deduce from this, the byte count of the TCP segment must always be an even integer.

When the URG (urgent) flag is set in the *code* field, the *urgent pointer* field is valid. This indicates the number of bytes in the *data* field that follow the current *sequence number*. This is known as **urgent data** – or sometimes **expedited data** – and, as we mentioned earlier, it should be delivered by the receiving TCP entity immediately it is received.

The *options* field provides the means of adding extra functionality to that covered by the various fields in the segment header. For example, it is used during the connection establishment phase to agree the maximum amount of data in a segment each TCP entity is prepared to accept. During this phase, each indicates its own preferred maximum size and hence can be different for each direction of flow. As we indicated earlier, this is called the maximum segment size (MSS) and excludes the fixed 20-byte segment header. If one of the TCP entities does not specify a preferred maximum size then a default value of 536 bytes is chosen. The TCP entity in all hosts connected to the Internet must accept a segment of up to 556 bytes – 536 plus a 20-byte header – and all IPs must accept a datagram of 576 bytes – 556 bytes plus a further

20-byte (IP) header. Hence the default MSS of 536 bytes ensures the datagram (with the TCP segment in its payload) will be transferred over the Internet without fragmentation. The format of the MSS option is shown in Figure 7.4(b). Also, although not shown, if this is the last or only option present in the header, then a single byte of zero is added to indicate this is the end of the option list.

Connection establishment

On receipt of a *connect( )* primitive (system call) from a client AP, the local TCP entity attempts to set up a logical connection with the TCP entity in the server whose IP address (and port number) are specified in the parameters associated with the primitive. This is achieved using a three-way exchange of segments. Collectively, this is known as a **three-way handshake** procedure and the segments exchanged for a successful *connect( )* call are shown in Figure 7.5(a).

As we indicated in the previous section, the flow of data bytes in each direction of a connection is controlled independently. The TCP entity at each end of a connection starts in the CLOSED state and chooses its own *initial sequence number* (*ISN*). These are both non-zero and change from one connection to another. This ensures that any segments relating to a connection that get delayed during their transfer over the internet – and hence arrive at the client/server after the connection has been closed – do not interfere with the segments relating to the next connection. Normally, each TCP entity maintains a separate 32-bit counter that is incremented at intervals of either 4 or 8 μs. Then, when a new ISN is required, the current contents of the counter are used.

- To establish a connection, the TCP at the client first reads the ISN to be used (from the counter) and makes an entry of this in the *ISN* and *send sequence variable* fields of the connection record used for this connection. It then sends a segment to the TCP in the server with the SYN code bit on, the ACK bit off, and the chosen ISN (X) in the sequence (number) field. Note that since no window or MSS option fields are present, then the receiving TCP assumes the default values. The TCP entity then transfers to the SYN_SENT state.
- On receipt of the SYN, if the required server AP – as determined by the destination port and IP address – is not already in the LISTEN state, the server TCP declines the connection request by returning a segment with the RST code bit on. The client TCP entity then aborts the connection establishment procedure and returns an error message with a reason code to the client AP. Alternatively, if the server AP is in the LISTEN state, the server TCP makes an entry of the ISN (to be used in the client–server direction and contained within the received segment) in its own connection record – in both the *ISN* and *receive sequence variable* fields – together with the ISN it proposes to use in the return direction. It then proceeds to create a new segment with the SYN bit on and the chosen ISN(Y) in the sequence field. In addition, it sets the ACK bit on

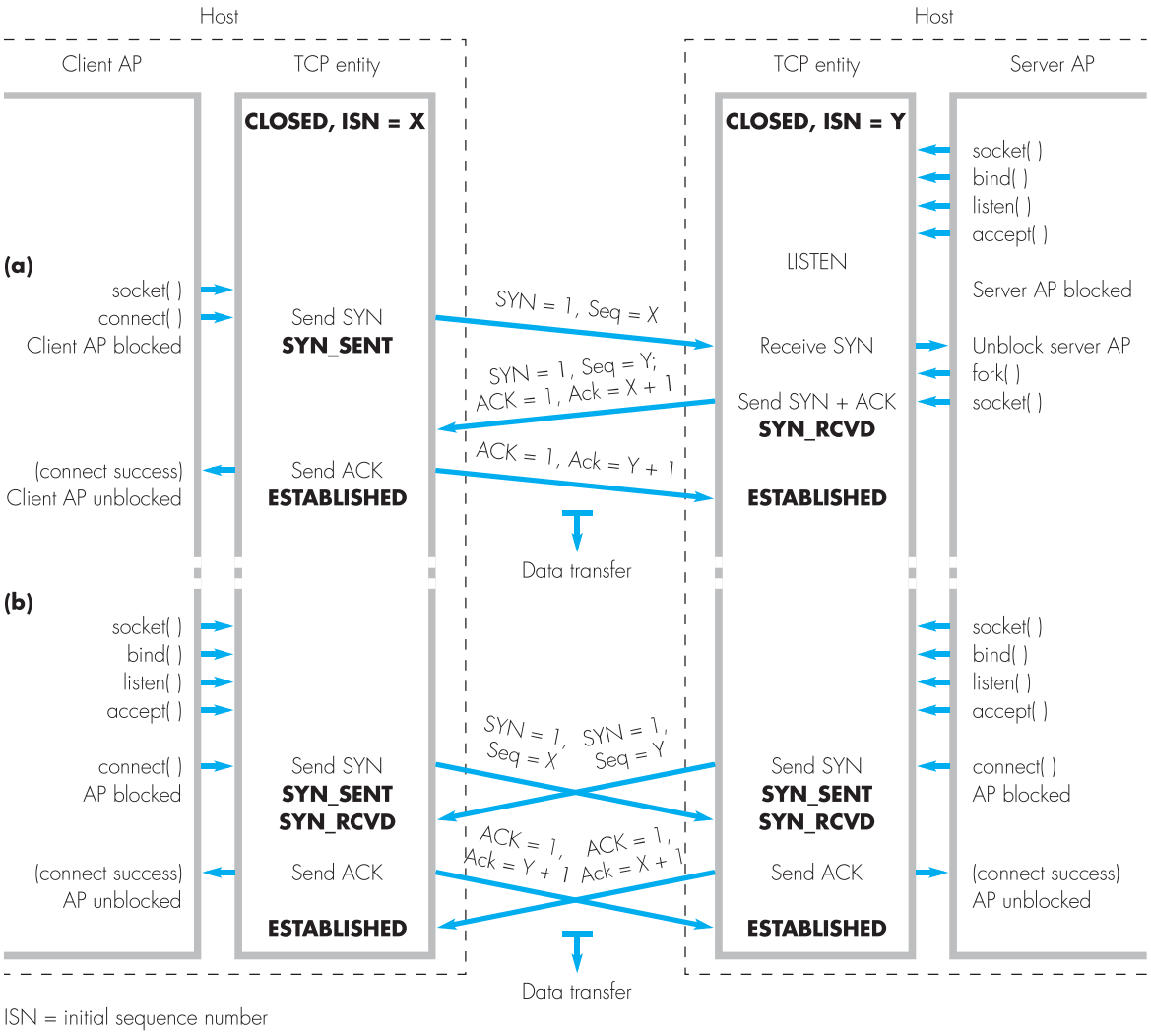


Figure 7.5 TCP connection establishment examples: (a) client–server; (b) connection collision.

and returns a value of X+1 in the acknowledgment field to acknowledge receipt of the client's SYN. It then sends the segment to the client and enters the SYN_RCVD state.

- On receipt of the segment, the client TCP makes an entry of the ISN to be used in the server–client direction in its connection record – in both the *ISN* and *receive sequence variable* fields – and increments the send sequence variable in the record by 1 to indicate the SYN has been acknowledged. It then acknowledges receipt of the SYN by returning a

segment with the ACK bit on and a value of Y+1 in the acknowledgment field. The TCP entity then enters the (connection) ESTABLISHED state.

- On receipt of the ACK, the server TCP increments the send sequence variable in its own connection record by 1 and enters the ESTABLISHED state. As we can deduce from this, the acknowledgment of each SYN segment is equivalent to a single data byte being transferred in each direction. At this point, both sides are in the ESTABLISHED state and ready to exchange data segments.

Although in a client–server application the client always initiates the setting up of a connection, in applications not based on the client–server model the two APs may try to establish a connection at the same time. This is called a **simultaneous open** and the sequence of segments exchanged in this case is as shown in Figure 7.5(b).

As we can see, the segments exchanged are similar to those in the client–server case and, since both ISNs are different, each side simply returns a segment acknowledging the appropriate sequence number. However, since the connection identifier is the same in both cases, only a single connection is established.

Data transfer

The error control procedure associated with the TCP protocol is similar to that used with the HDLC protocol. The main difference is that the sequence and acknowledgment numbers used with TCP relate to individual bytes in the total byte stream whereas with HDLC the corresponding send and receive sequence numbers relate to individual blocks of data. Also, because of the much larger round-trip time of an internet (compared with a single link), with TCP the window size associated with the flow control procedure is not derived from the sequence numbers. Instead, a new window size value is included in each segment a TCP entity sends to inform the other TCP entity of the maximum number of bytes it is willing to receive from it. This is known also as a **window size advertisement**.

In addition, because the TCP protocol may be operating (on an end-to-end basis) over a number of interconnected networks rather than a single line, it includes a congestion control procedure. This endeavors to regulate the rate at which the sending TCP entity sends data segments into the internet to the rate segments are leaving the internet. We shall discuss the main features of the different procedures that are used by means of examples.

Small segments

In order to explain the features of the protocol that relate to the exchange of small segments – that is, all the segments contain less than the MSS – we shall consider a typical data exchange relating to a networked interactive application. An example application protocol of this type is Telnet. Typically, this involves a user at the client side typing a command and the server AP in a remote host responding to it. An example set of segments is shown in Figure 7.6(a).

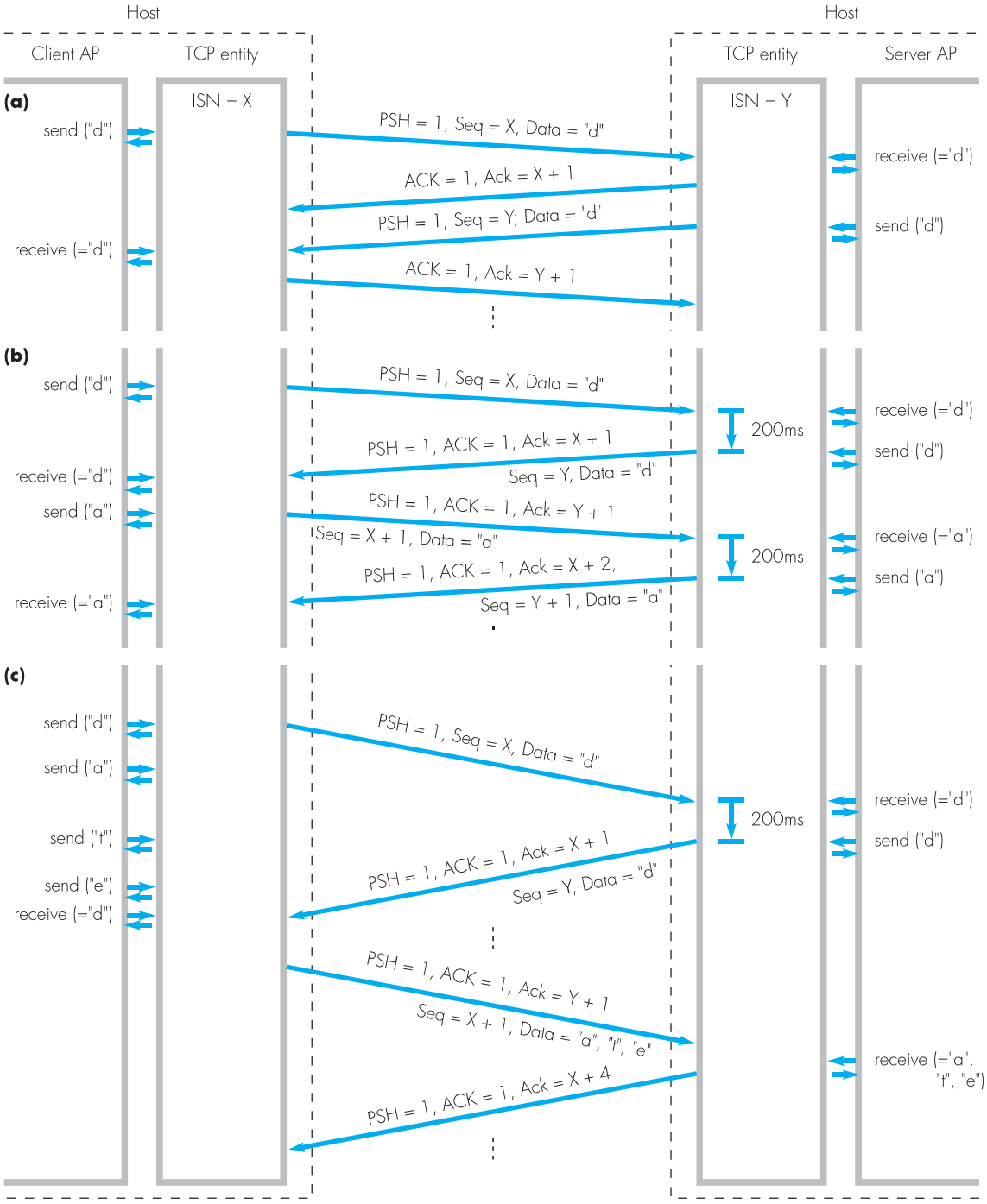


Figure 7.6 Small segment data transfers: (a) immediate acknowledgments; (b) delayed acknowledgments; (c) Nagle algorithm.

With interactive applications involving a user at a keyboard, each character typed is sent directly by the client AP to the server side. The server AP then reads the character from the receive buffer and immediately echoes the character back to the client side by writing it into the send buffer. On receipt of the character, the client AP displays it on the host screen. Hence each typed character is sent directly in a separate segment with the PSH flag on. Similarly, the echoed character is also sent in a separate segment with the PSH flag on. In addition to these two segments, however, each TCP entity returns a segment with the ACK bit on to acknowledge receipt of the segment containing the typed/echoed character. This means that for each character that is typed, four segments are sent, each with a 20-byte header and a further 20-byte IP header.

In practice, in order to reduce the number of segments that are sent, a receiving TCP entity does not return an ACK segment immediately it receives an (error-free) data segment. Instead, it waits for up to 200 ms to see if any data is placed in the send buffer by the local AP. If it is, then the acknowledgment is piggybacked in the segment that is used to send the data. This procedure is called **delayed acknowledgments** and, as we can see in Figure 7.6(b), with interactive applications it can reduce significantly the number of segments that are required.

Although this mode of working is acceptable when communicating over a single network such as a LAN, when communicating over an internet which has a long round-trip time (RTT), the delays involved in waiting for each echoed character can be annoying to the user. Hence a variation of the basic delayed acknowledgment procedure is often used. This is called the **Nagle algorithm** and is defined in RFC 896. When the algorithm is enabled, each TCP entity can have only a single small segment waiting to be acknowledged at a time. As a result, in interactive applications, when the client TCP entity is waiting for the ACK for this segment to be received, a number of characters may have been typed by the user. Hence when the ACK arrives, all the waiting characters in the send buffer are transmitted in a single segment. A sequence diagram showing this is given in Figure 7.6(c).

In these examples, the window size has not been shown since, in general, when small segments are being exchanged it has no effect on the flow of the segments. Also, the Nagle algorithm is not always enabled. For example, when the interactions involve a mouse, each segment may contain a collection of mouse movement data and, when echoed, the movement of the cursor can appear erratic. An example application of this type is X-Windows.

Error control

As we saw in Figure 7.4(a), each TCP segment contains only a single acknowledgment number. Hence should a segment not arrive at the destination host, the receiving TCP can only return an acknowledgment indicating the next in-sequence byte that it expects. Also, since the packets relating to a message are being transmitted over an internet, when the path followed has alternative routes, packets may arrive at the destination host out of sequence. Hence the

receiving TCP simply holds each out-of-sequence segment that it receives in temporary storage and returns an ACK indicating the next in-sequence byte – and hence segment – that it expects. Normally, the out-of-sequence segment arrives within a short time interval at which point the receiving TCP returns an ACK that acknowledges all the segments now received including those held in temporary storage.

At the sending side, the TCP receives an ACK indicating a segment has been lost but, within a short time interval, it receives an ACK for a segment that it transmitted later (so acknowledging receipt of all bytes up to and including the last byte in the later segment). Hence, since this is a relatively frequent occurrence, the sending TCP does not initiate a retransmission immediately it receives an out-of-sequence ACK. Instead, it only retransmits a segment if it receives three duplicate ACKs for the same segment – that is, three consecutive segments with the same acknowledgment number in their header – since it is then confident that the segment has been lost rather than simply received out of sequence.

In addition, to allow for the possibility that the sending TCP has no further segments ready for transmission, when a loss is detected it starts a *retransmission timer* for each new segment it transmits. A segment is then retransmitted if the TCP does not receive an acknowledgment for it before the timeout interval expires. An example illustrating the two possibilities is shown in Figure 7.7. In the example we assume:

- there is only a unidirectional flow of data segments;
- the sending AP writes a block of data – a message – comprising 3072 bytes into the send buffer using a *send( )* primitive;
- the MSS being used for the connection is 512 bytes and hence six segments are required to send the block of data;
- the size of the receive buffer is 8192 bytes and hence the sending TCP can send the complete set of segments without waiting for an acknowledgment;
- an ACK segment is returned on receipt of each error-free data segment;
- segments 2 and 6 are both lost – owing to transmission errors for example – as is the final ACK segment.

To follow the transmission sequence shown in the figure we should note the following:

- The sending TCP has a send sequence variable, V(S), in its connection record, which is the value it places in the sequence number field of the next new segment it sends. Also it has a *retransmission list* to hold segments waiting to be acknowledged. Similarly, the receiving TCP has a receive sequence variable, V(R), in its connection record (which is the sequence number it expects to receive next) and a *receive list* to hold segments that are received out of sequence.

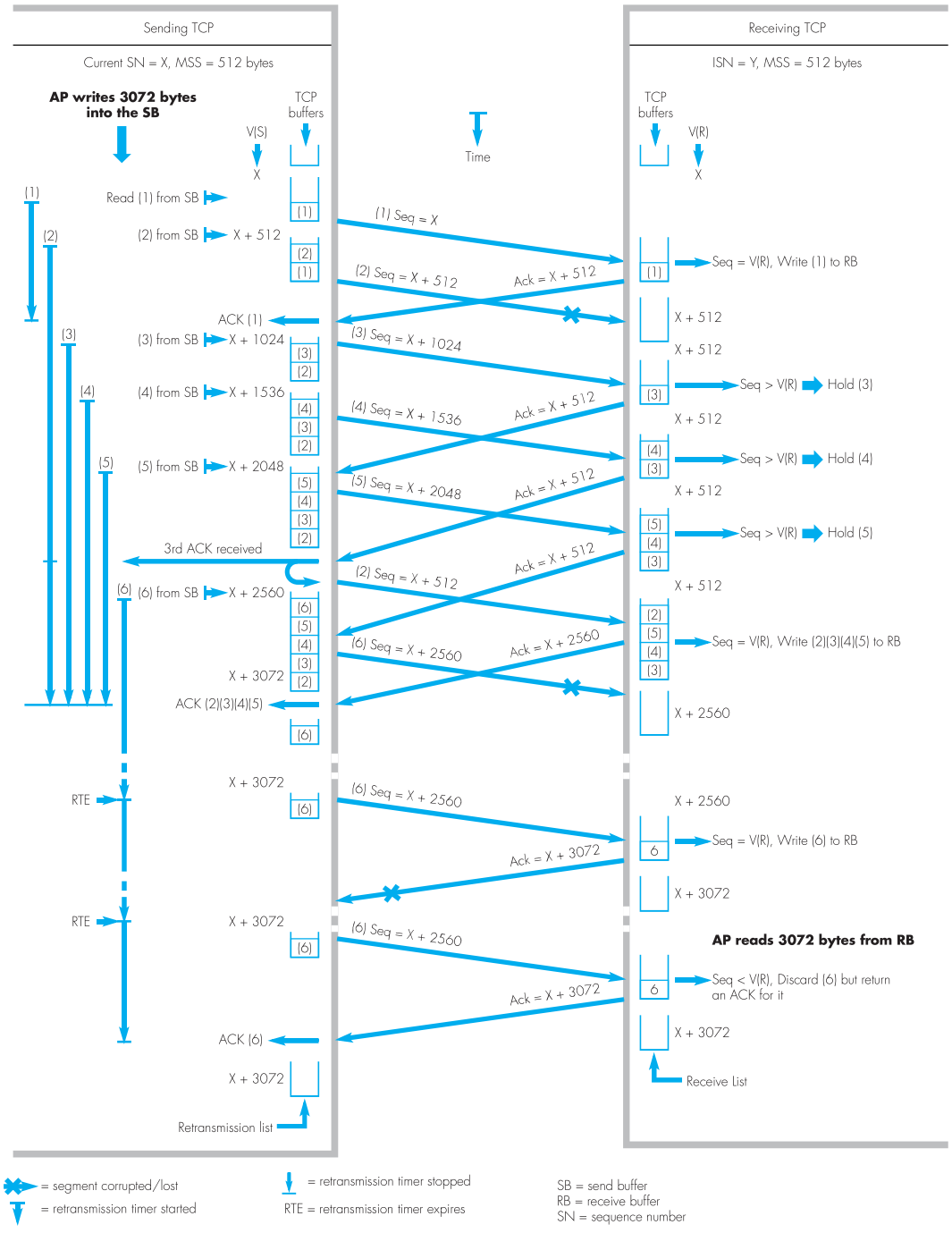


Figure 7.7 Example segment sequence showing TCP error control procedure.

- Segment (1) is received error-free and, since its sequence number (Seq = X) is equal to V(R), the 512 bytes of data it contains are transferred directly into the receive buffer (ready for reading by the destination AP), V(R) is incremented to X + 512, and an ACK (with Ack = X + 512) is returned to the sending side.
- On receipt of the ACK, the sending TCP stops the retransmission timer for (1) and, in the meantime, segment (2) has been sent.
- Since segment (2) is corrupted, no ACK for it is returned and hence its timer continues running.
- The sending TCP continues to send segments (3), (4) and (5), all of which are received error-free. However, since they are out of sequence – segment (2) is missing – the receiving TCP retains them in its receive list and returns an ACK with Ack = X + 512 in each segment for each of them to indicate to the sending TCP that segment (2) is missing.
- On receipt of the third ACK with an ACK = X + 512, the sending TCP retransmits segment (2) without waiting for the retransmission timer to expire. As we indicated earlier, this is done since if three or more ACKs with the same acknowledgment number are received one after the other, it is assumed that the segment indicated has been lost rather than received out of sequence. Because the retransmission occurs before the timer expires, this procedure is called **fast retransmit**.
- This time segment (2) is received error-free and, as a result, the receiving TCP is able to transfer the contents of segments (2), (3), (4) and (5) to the receive buffer – ready for reading by the AP – and returns an ACK with Ack = X + 2560 to indicate to the sending TCP that all bytes up to and including byte 2560 have been received and their timers can be stopped.
- In the meantime, segment (6) has been transmitted but is corrupted. Hence, since no other segments are waiting to be sent, its timer continues running until it expires when the segment is retransmitted.
- This time the segment is received error-free and so its contents are passed to the receive buffer directly and an ACK for it is returned. Also, it is assumed that at this point the receiving AP reads the accumulated 3072 bytes from the receive buffer using a *receive( )* primitive.
- The ACK for segment (6) is corrupted and hence the timer for the segment expires again and the segment is retransmitted. However, since the sequence number is less than the current V(R), the receiving TCP assumes it is a duplicate. Hence it discards it but returns an ACK to stop the sending TCP from sending another copy.

As we can see from this example, a key parameter in the efficiency of the error control procedure is the choice of the **retransmission timeout** (**RTO**) **interval**. With a single data link the choice of an RTO is straightforward since the worst-case round-trip time – the time interval between sending a packet/frame and receiving an ACK for it – can be readily determined. The

RTO is then set at a value slightly greater than this. With an internet, however, the RTT of a TCP connection can vary considerably over relatively short intervals as the traffic levels within routers build up and subside. Hence choosing an RTO when the internet is lightly loaded can lead to a significant number of unnecessary retransmissions, while choosing it during heavy load conditions can lead to unnecessary long delays each time a segment is corrupted/lost. The choice of RTO, therefore, must be dynamic and vary not only from one connection to another but also during a connection.

The initial approach used to derive the RTO for a connection was based on an **exponential backoff** algorithm. With this an initial RTO of 1.5 seconds is used. Should this prove to be too short – that is, each segment requires multiple retransmission attempts – the RTO is doubled to 3 seconds. This doubling process then continues until no retransmissions occur. To allow for network problems, a maximum RTO of 2 minutes is used at which point a segment with the RST flag bit on is sent to abort the connection/session.

Although very simple to implement, a problem with this method is that when an ACK is received, it is not clear whether this is for the last retransmission attempt or an earlier attempt. This is known as the **retransmission ambiguity problem** and was identified by Karn. Because of this, a second approach was proposed by Jacobson and is defined in RFC 793. With this method, the RTO is computed from actual RTT measurements. The RTO is then updated as each new RTT measurement is made. In this way, the RTO for each connection is continuously being updated as each new estimate of the RTT is determined.

As we indicated earlier, when each data segment is sent, a separate retransmission timer is started. A connection starts with a relatively large RTO. Then, each time an ACK segment is received before the timer expires, the actual RTT is determined by subtracting the initial start time of the timer from the time when the ACK was received. The current estimate of the RTT is then updated using the formula

$$\text{RTT} = \alpha\,\text{RTT} + (1-\alpha)\,M$$

where M is the measured RTT and α is a smoothing factor that determines by how much each new M influences the computation of the new RTT relative to the current value. The recommended value for α is 0.9. The new RTO is then set at twice the updated RTT to allow for a degree of variance in the RTT.

Although this method performed better than the original method, a refinement of it was later introduced. This was done because by using a fixed multiple of each updated RTT ($\times 2$) to compute the new RTO, it was found that it did not handle well the wide variations that occurred in the RTT. Hence in order to obtain a better estimate, Jacobson proposed that each new RTO should be based not just on the mean of the measured RTT but also on the variance. In the proposed algorithm, the mean deviation of the RTT measurements, D, is used as an estimate of the variance. It is computed using the formula:

$$D = \alpha\,D + (1-\alpha)|\text{RTT} - M|$$

where α is a smoothing factor and $|RTT - M|$ is the modulus of the difference between the current estimate of the RTT and the new measured RTT (M). This is also computed for each new RTT measurement and the new estimate of the RTO is then derived using the formula:

$$RTO = RTT + 4D$$

As we indicated earlier, to overcome the retransmission ambiguity problem, the RTT is not measured/updated for the ACKs relating to retransmitted segments.

Note also that although in the above example an ACK is returned on receipt of each data segment, this is not always the case. Indeed, in most implementations, providing there is a steady flow of data segments, the receiving TCP only returns an ACK for every other segment it receives correctly. On sending each ACK, a timer – called the **delayed ACK timer** – is started and, should a second segment not be received before it expires, then an ACK for the first segment is sent. Note, however, that when a single ACK is sent for every other segment, since the V(R) is incremented on receipt of each segment, then the acknowledgment number within the (ACK) segment acknowledges the receipt of all the bytes in both segments.

Flow control

As we indicated earlier, the value in the *window size* field of each segment relates to a sliding window flow control scheme and indicates the number of bytes (relative to the byte being acknowledged in the *acknowledgment* field) that the receiving TCP is able to accept. This is determined by the amount of free space that is present in the receive buffer being used by the receiving TCP for the connection. Recall also that the maximum size of the window is determined by the size of the receive buffer. Hence when the sending TCP is running in a fast host – a large server for example – and the receiving TCP in a slow host, the sending TCP can send segments faster than, firstly, the receiving TCP can process them and, secondly, the receiving AP can read them from the receive buffer after they have been processed. The window flow control scheme, therefore, is present to ensure that there is always the required amount of free space in the receive buffer at the destination before the source sends the data. An example showing the sequence of segments that are exchanged to implement the scheme is given in Figure 7.8. In the example, we assume:

- there is only a unidirectional flow of data segments;
- the sizes of both the send and the receive buffers at the sending side are 4096 bytes and those at the receiving side 2048 bytes. Hence the maximum size of the window for the direction of flow shown is 2048 bytes;
- associated with each direction of flow the sending side maintains a *send window variable*, W_S, and the receiving side a *receive window variable*, W_R;

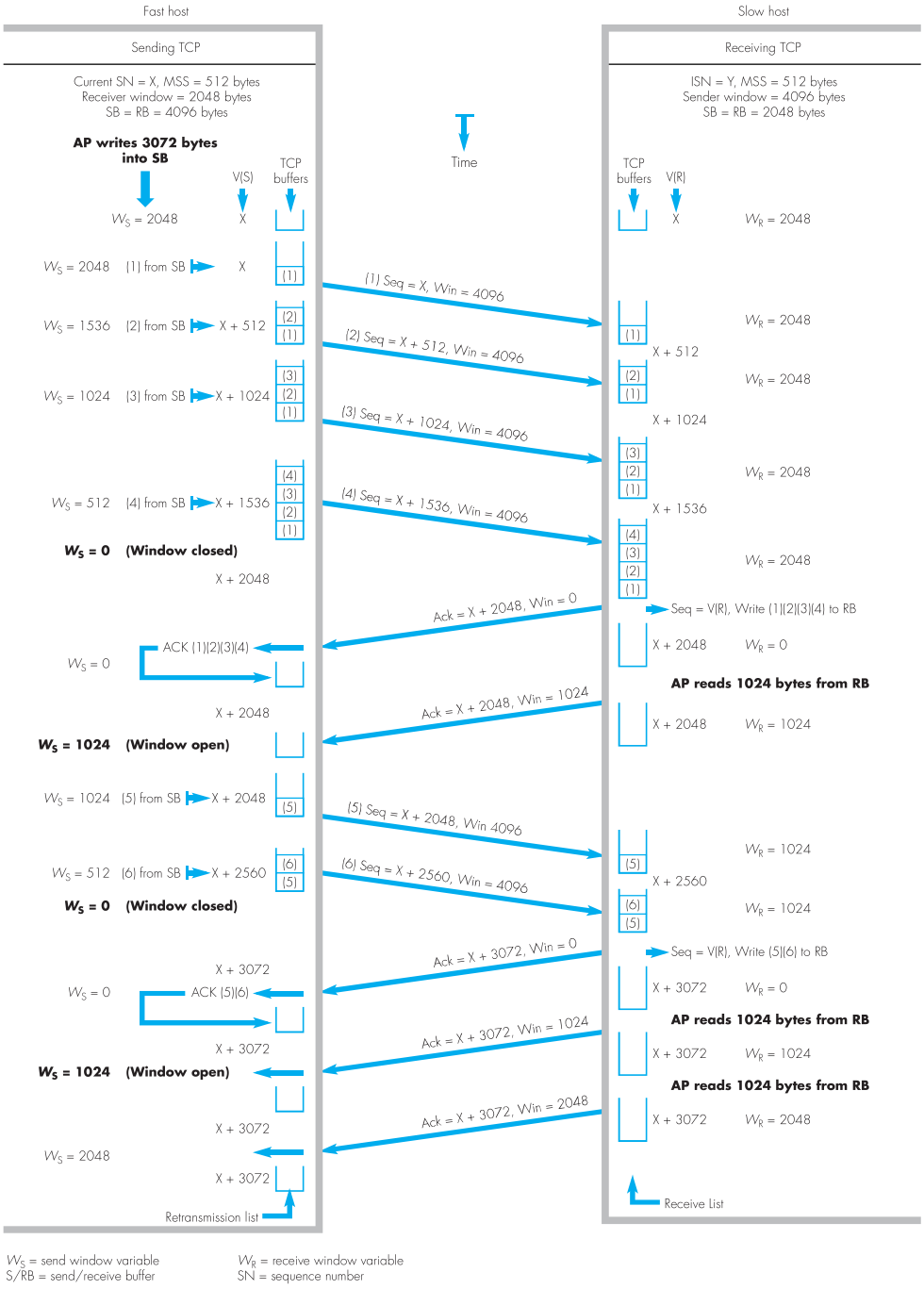


Figure 7.8 Example segment sequence showing TCP flow control procedure.

- the source AP can write bytes into the send buffer up to the current value of W_S and, providing W_S is greater than zero, the sending TCP can read data bytes from its send buffer up to the current value of W_S and initiate their transmission. Flow is stopped when $W_S = 0$ and the window is then said to be closed;
- at the destination, the receiving TCP, on receipt of error-free data segments, transfers the data they contain to the receive buffer and increments W_R by the number of bytes transferred. W_R is then decremented when the destination AP reads bytes from the receive buffer and, after each read operation, the receiving TCP returns a segment with the number of bytes of free space now available in the window size field of the segment.

The following should be noted when interpreting the sequence:

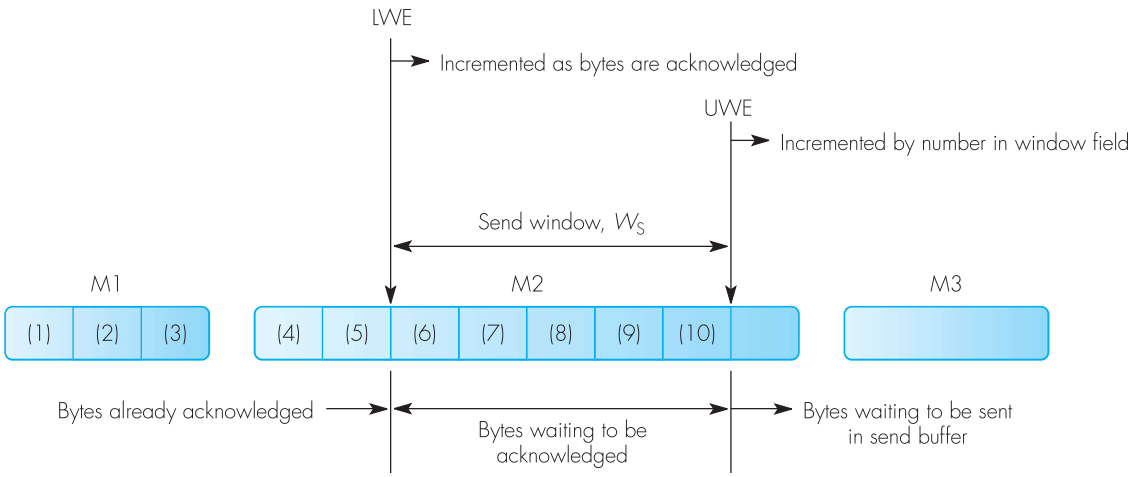
- The flow of segments starts with the AP at the sending side writing 3072 bytes into the send buffer using a *send( )* primitive.
- Since the sending host is much faster than the receiving host, the sending TCP is able to send a full window of 2048 bytes (in four 512-byte segments) before the receiving TCP is able to start processing them. The sending TCP must then stop as W_S is now zero.
- When the receiving TCP is scheduled to run, it finds four segments in its receive list and, since the first segment – segment (1) – has a sequence number equal to V(R), it transfers its contents to the receive buffer. It then proceeds to process and transfer the contents of segments (2), (3) and (4) in the same way and, after it has done this, it returns a single ACK segment to acknowledge receipt of these four segments but with a window size field of zero.
- On receipt of the ACK, the sending TCP deletes segments (1), (2), (3) and (4) from its retransmission list but leaves $W_S = 0$.
- When the receiving AP is next scheduled to run, we assume it reads just 1024 bytes from the receive buffer. On detecting this, the TCP returns a second ACK with the same acknowledgment number but with a window size of 1024. The second ACK is known, therefore, as a **window update**.
- At this point, since its sending window is now open, the sending TCP proceeds to send segments (5) and (6) at which point W_S again becomes zero.
- At the receiving side, when the TCP is next scheduled to run it finds segments (5) and (6) in the receive list and, since their sequence numbers are in sequence, it transfers their contents to the receive buffer. It then returns a single ACK for them but with a window size of zero.
- On receipt of the ACK, the sending TCP deletes segments (5) and (6) from its retransmission list but leaves $W_S = 0$.

- At some point later, the receiving AP is scheduled to run and we assume it again reads 1024 bytes from the receive buffer. Hence when the TCP next runs it returns a window update of 1024.
- Finally, after the receiving AP reads the last 1024 bytes from the RB, the TCP – some time later – returns a window update of 2048. After this has been received, both sides are back to their initial state.

We should note that there are a number of different implementations of TCP and hence the sequence shown in Figure 7.8 is only an example. For example, the receiving TCP may return two ACKs – one for segments (1) and (2) and the other for segments (3) and (4) – rather than a single ACK. In this case there would be a different distribution of segments between the TCP buffers and the receive buffer. Nevertheless, providing the size of the TCP buffers in the receiver is the same as the receive buffer, then the window procedure ensures there is sufficient buffer storage to hold all received segments. A schematic diagram summarizing the operation of the sliding window procedure is given in Figure 7.9.

Congestion control

A segment may be discarded during its transfer across an internet either because transmission errors are detected in the packet containing the



LWE = lower window edge, initialized to (ISN + 1)
UWE = upper window edge, initialized to (ISN + 1) + advertised window
W_S = (UWE – LWE), flow stopped when W_S = 0
M1, 2, 3 = sending AP messages
(1), (2) etc. = segments sent by TCP entity
Note: TCP chooses the size of segments it sends

Figure 7.9 TCP sliding window.

segment or because a router or gateway along the path being followed becomes congested; that is, during periods of heavy traffic it temporarily runs out of buffer storage for packets in the output queue associated with a line. However, the extensive use of optical fiber in the transmission network means that the number of lost packets due to transmission errors is relatively small. Hence the main reason for lost packets is congestion within the internet.

To understand the reason for congestion, it should be remembered that the path followed through an internet may involve a variety of different transmission lines some of which are faster – have a higher bit rate – than others. In general, therefore, the overall speed of transmission of segments over the path being followed is determined by the bit rate of the slowest line. Also, congestion can arise at the sending side of this line as the segments relating to multiple concurrent connections arrive at a faster rate than the line can transmit them. Clearly, if this situation continues for even a relatively short time interval, the number of packets in the affected router output queue builds up until the queue becomes full and packets have to be dropped. This also affects the ACKs within the lost segments and, as we have just seen, this can have a significant effect on the overall time that is taken to transmit a message.

In order to reduce the likelihood of lost packets occurring, the TCP in each host has a congestion control/avoidance procedure that, for each connection, uses the rate of arrival of the ACKs relating to a connection to regulate the rate of entry of data segments – and hence IP packets – into the internet. This is in addition to the window flow control procedure, which, as we have just seen, is concerned with controlling the rate of transmission of segments to the current capacity of the receive buffer in the destination host. Hence in addition to a send window variable, W_S, associated with the flow control procedure, each TCP also has a **congestion window** variable, W_C, associated with the congestion control/avoidance procedure. Both are maintained for each connection and the transmission of a segment relating to a connection can only take place if both windows are in the open state.

As we can see from the above, under lightly loaded network conditions the flow of segments is controlled primarily by W_S and, under heavily loaded conditions, it is controlled primarily by W_C. However, when the flow of segments relating to a connection first starts, because no ACKs have been received, the sending TCP does not know the current loading of the internet. So to stop it from sending a large block of segments – up to the agreed window size – the initial size of the congestion window, W_C, is set to a single segment which, because W_C has a dimension of bytes, is equal to the agreed MSS for the connection.

As we show in Figure 7.10, the sending TCP starts the data transfer phase of a connection by sending a single segment of up to the MSS. It then starts the retransmission timer for the segment and waits for the ACK to be received. If the timer expires, the segment is simply retransmitted. If the ACK is received before the timer expires, W_C is increased to two segments, each equal to the MSS. The sending TCP is then able to send two segments and,

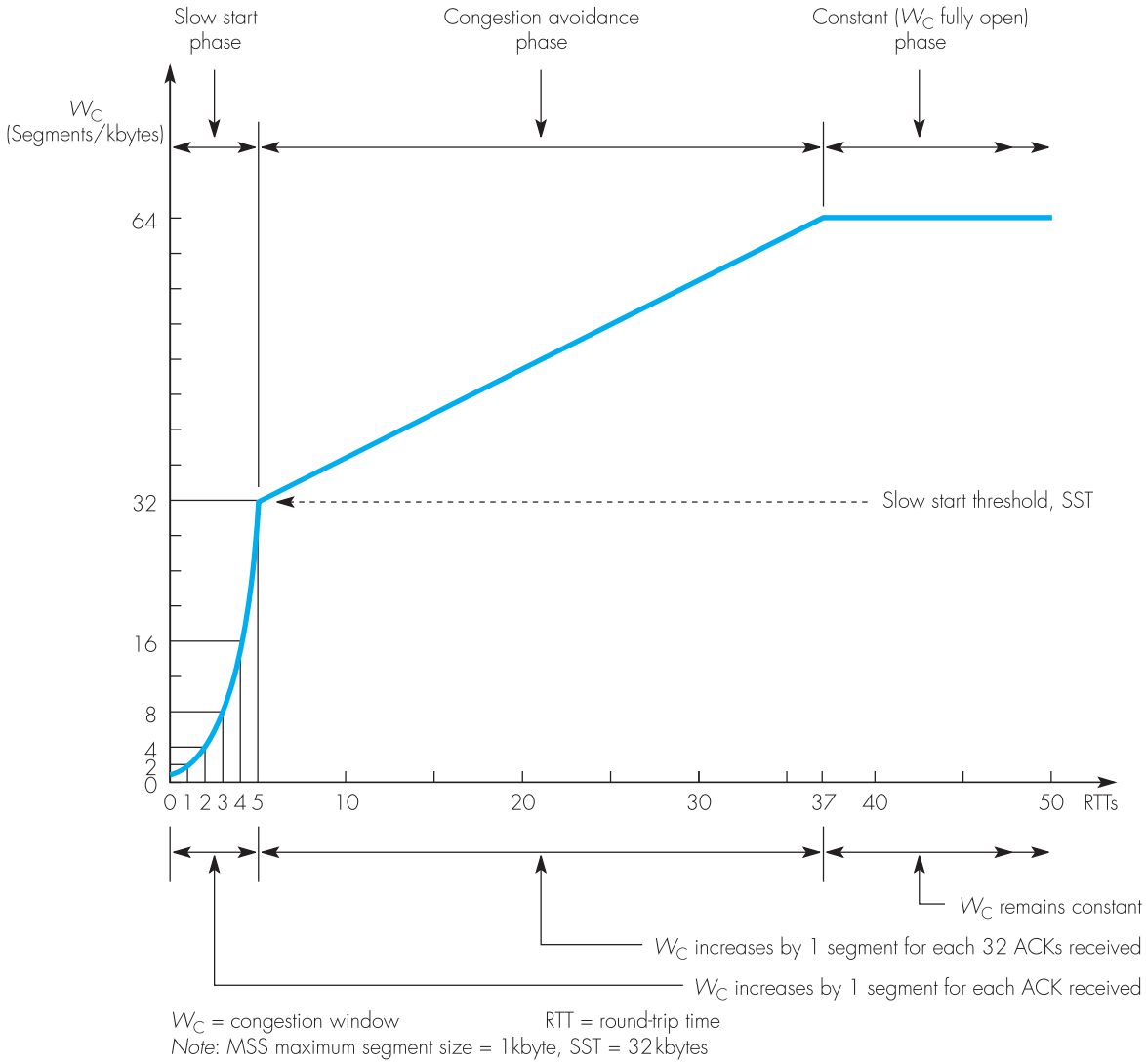


Figure 7.10 TCP congestion control window procedure.

for each of the ACKs it receives for these segments, W_C is increased by one segment (MSS). Hence, the sending TCP can now send four segments and, as we can see, W_C increases exponentially. Even though W_C increases rapidly, this phase is called **slow start** since it builds up from a single segment. It continues until a timeout for a lost segment occurs, or a duplicate ACK is received, or an upper threshold is reached. This is called the **slow start threshold** (**SST**) and, for each new connection, it is set to 64 kbytes. In the example, however, it is assumed to be initialized to 32 kbytes, which, because

the MSS is 1 kbyte, is equal to 32 segments. Assuming the SST is reached, this is taken as an indication that the path is not congested. Hence the connection enters a second phase during which, instead of W_C increasing by 1 segment (MSS) for each ACK it receives, it increases by $1/W_C$ segments for each ACK received. Hence, as we can see, W_C now increases by 1 segment for each set of W_C ACKs that are received. This is called the **congestion avoidance** phase and, during this phase, the increase in W_C is additive. It continues until a second threshold is reached and, in the example, this is set at 64 kbytes. On reaching this, W_C remains constant at this value.

The profile shown in Figure 7.10 is typical of a lightly loaded internet in which none of the lines making up the path through the internet is congested. Providing W_C remains greater than the maximum flow control window, the flow of segments relating to the connection is controlled primarily by W_S. During these conditions all segments are transferred with a relatively constant transfer delay and delay variation. As the number of connections using the internet increases, however, so the traffic level increases up to the point at which packet (and hence segment) losses start to occur and, when this happens, the TCP controlling each connection starts to adjust its congestion window in a way that reflects the level of congestion.

The steps taken depend on whether a lost packet is followed by duplicate ACKs being received or the retransmission timer for the segment expiring. In the case of the former, as we saw earlier in Figure 7.7, the receipt of duplicate ACKs is indicative that segments are still being received by the destination host. Hence the level of congestion is assumed to be light and, on receipt of the third duplicate ACK relating to the missing segment – fast retransmit – the current W_C value is halved and the congestion avoidance procedure is invoked starting at this value. This is called **fast recovery** and an example is shown in Figure 7.11(a).

In this example it is assumed that the first packet loss occurs when W_C is at its maximum value of 64 segments, which, with an MSS of 1 kbyte, is equal to 64 kbytes. Hence on receipt of the third duplicate ACK, the lost segment is retransmitted and W_C is immediately reset to 32 segments/kbytes. The W_C is then incremented back up using the congestion avoidance procedure. However, when it reaches 34 segments, a second segment is lost. It is assumed that this also is detected by the receipt of duplicate ACKs and hence W_C is reset to 17 segments before the congestion avoidance procedure is restarted.

In the case of a lost segment being detected by the retransmission timer expiring, it is assumed that the congestion has reached a level at which no packets/segments relating to the connection are now getting through. As we show in the example in Figure 7.11(b), when a retransmission timeout (RTO) occurs, irrespective of the current W_C, it is immediately reset to 1 segment and the slow start procedure is restarted. Thus, when the level of congestion reaches the point at which RTOs start to occur, the flow of segments is controlled primarily by W_C.

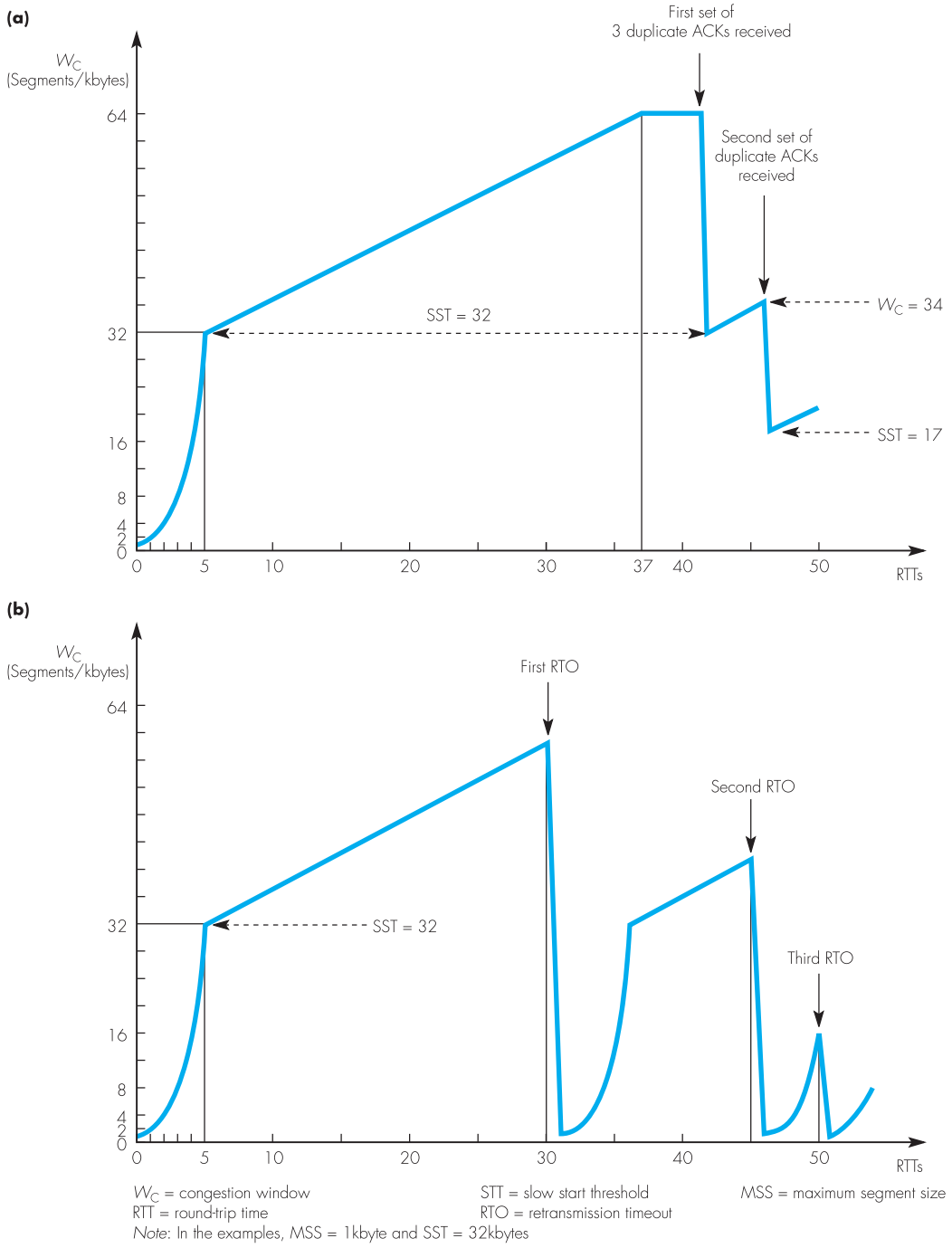


Figure 7.11 TCP congestion window adjustments: (a) on receipt of duplicate ACKs; (b) on expiry of a retransmission timer.

Connection termination

As we indicated earlier, each TCP connection is duplex and hence data can be transferred in both directions simultaneously. To support this, each TCP entity maintains separate send and receive sequence and window variables and a separate state variable for each direction of flow. When a connection is terminated, each direction of flow is closed separately. In practice, there are a number of ways this is carried out and four examples are given in Figure 7.12.

In most cases, the TCP entity at each end of a connection goes through a different sequence of states. In order to discriminate between the two ends, the AP that issues the first *close( )* – and hence the TCP which sends the first FIN segment – performs what is called an **active close** procedure and the other side, a **passive close**. As we can see in the first example – part (a) – each procedure involves a slightly different sequence:

- In all the examples, both the forward and return paths of the TCP connection are currently in the ESTABLISHED (EST) – data transfer – state.
- The connection termination is started by one of the APs – normally the client – issuing a *close( )* primitive. The TCP entity at this side then goes through the active close procedure and the TCP at the other side the passive close.
- On receipt of the *close( )* primitive, the active TCP entity sends a FIN segment – that is, a segment with the FIN bit on – with a sequence number equal to the current V(S), X. It then enters the FIN_WAIT1 state to indicate it is waiting for an ACK for the first FIN.
- On receipt of the FIN, the passive TCP first returns an ACK indicating correct receipt of the FIN and, when its local AP does the next *receive( )*, an end-of-file (EOF) is returned to indicate that no more data will be coming from the other side for this connection. It then enters the CLOSE_WAIT state to indicate it is waiting for a *close( )* from its local AP. Note that the ack number in the ACK is equal to X + 1 since a FIN consumes a byte of the byte stream.
- On receipt of the ACK, the TCP performing the active close simply enters the FIN_WAIT2 state to indicate it is now waiting for a FIN from the passive side.
- At some time later the AP in the passive side issues a *close( )* primitive and, as a result, the local TCP sends a FIN to indicate the closure of the connection in the reverse direction. It then enters the LAST_ACK state to indicate it is now waiting for the last ACK.
- On receipt of the FIN at the active side, the TCP first returns an ACK for the segment. It then starts a timer called the **2MSL timer** and enters the TIMED_WAIT state. When configuring each TCP entity a parameter called the **maximum segment lifetime** (**MSL**) is entered. This is the maximum time duration a segment can exist in the internet before being

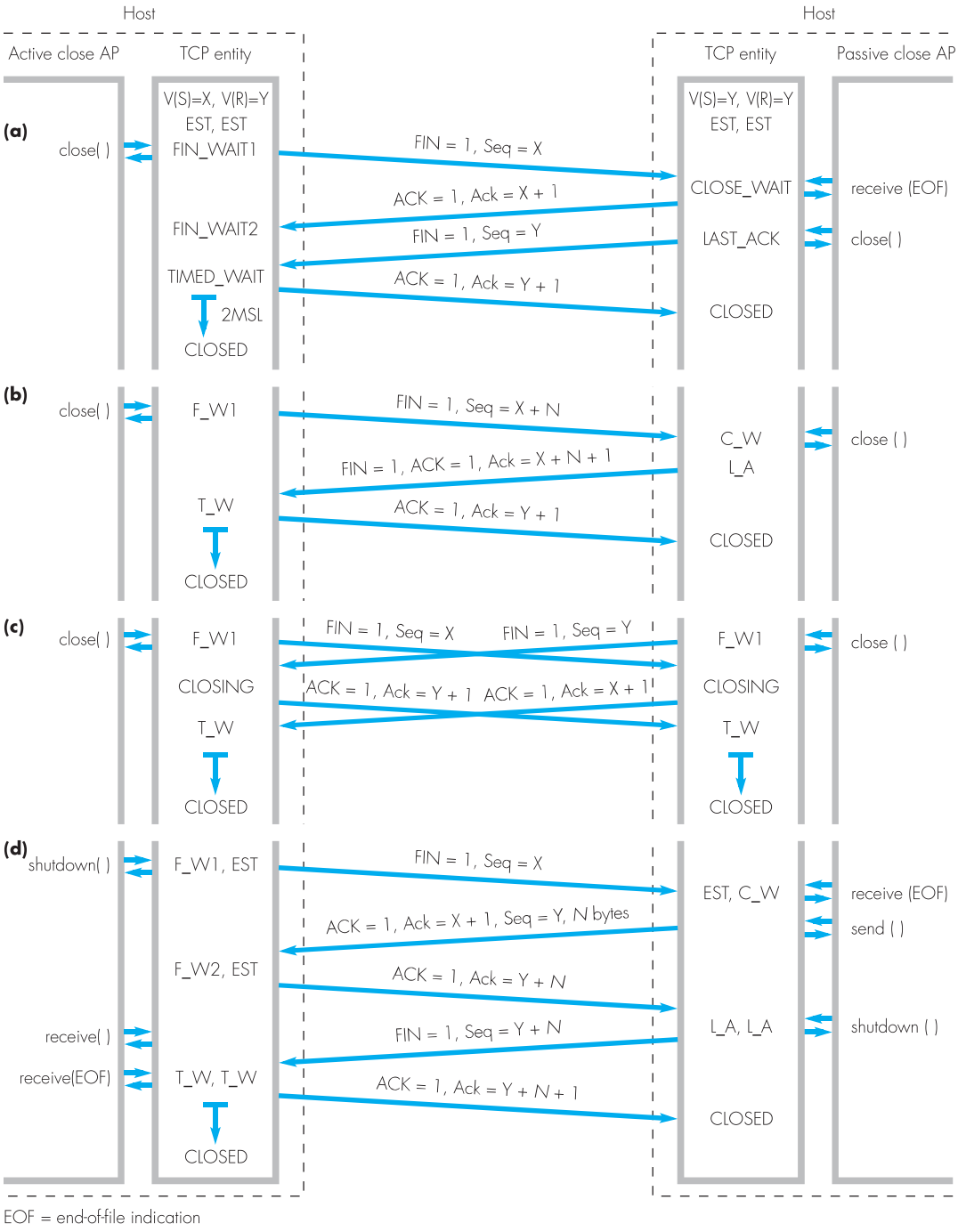


Figure 7.12 Connection close examples: (a) normal (4-way) close; (b) 3-way close; (c) simultaneous close; (d) half-close.

discarded. In practice, therefore, it is related to the time-to-live value used in each IP packet header. Typical values for the MSL are between 30 seconds and 2 minutes. This means that should the ACK for the second FIN be lost, the active TCP is still able to receive the retransmitted FIN.

- Finally, on receipt of the ACK, the TCP at the passive side deletes the connection record relating to this connection and, when the 2MSL timer expires, the TCP at the active side does the same.

The segment sequence shown in Figure 7.12(b) is similar to that in the first example except that at the passive side data is still waiting to be acknowledged when the *close( )* is received. Hence the passive TCP piggybacks the ACK for the data in the same segment that carries the FIN in the reverse direction. This occurs when the *close( )* is received before the FIN arrives and, as we can see, this reduces the standard closure to a three-way segment exchange rather than a four-way exchange. Note that with this, however, data may be lost at the passive side if both sides of the connection are closed on receipt of the *close( )* primitive.

The segment sequence shown in Figure 7.12(c) illustrates what happens when the AP in both hosts issue a *close( )* simultaneously. As we can see, in this case the TCP at both sides carries out the active close sequence. Here, however, the intermediate state CLOSING is entered on receipt of the FIN from the opposite side. Then, as before, on receipt of the related ACK segment, both sides enter the TIMED_WAIT state to wait for the 2MSL timer to expire before closing down.

When an AP initiates the termination of a connection with a *close( )* primitive, this indicates to its local TCP that it has now completed sending the data/messages relating to the session and expects the remote AP to do the same. As a result, both TCPs proceed to close the (duplex) connection using one of the segment exchanges shown in the first three examples. With some applications, however, although the AP in the active side has completed sending data, it still expects to receive further data from the correspondent AP. Clearly, since both directions of a TCP connection are managed separately, this is possible. Hence to enable its local TCP to discriminate between this and a normal close, the AP issues a *shutdown( )* call. The local TCP initiates the closure of its side of the connection but leaves the other side in the ESTABLISHED state. This is known as a **half-close** and an example showing a typical segment sequence is shown in Figure 7.12(d). The following points should be noted:

- In the figure, we show the state of both the forward and return paths of the connection at both sides and, as before, initially these are both in the ESTABLISHED (EST) state.
- On receipt of the *shutdown( )* call, the local TCP leaves the return path in the EST state but proceeds to close the forward path. Hence it sends a FIN segment and enters the FIN_WAIT1 state.

- On receipt of this, an EOF is returned to the correspondent AP in response to the next *receive( )* to indicate that it will not receive any further data. It then changes the state of this side of the connection to the CLOSE_WAIT state. The AP then receives some data to send in the return direction and, since this path is still open, it sends the data and, in the same segment, piggybacks the acknowledgment of the FIN segment.
- On receipt of this, the other TCP returns an ACK for the data and, in response to the ACK for the FIN it sent, changes the state of this side of the connection to the FIN_WAIT2 state. Also, at some time later the AP reads the data using a *receive( )* call.
- After the ACK for the data is received, a *shutdown( )* is received from the AP. This results in a FIN segment being sent for the return path and the state of both paths being set into the LAST_ACK state.
- On receipt of the FIN, an EOF is returned to the local AP in response to the next *receive( )* to inform it that no further data will be coming. An ACK for the FIN is then returned and the state of both sides of the connection changed to the TIMED_WAIT state.
- On receipt of the ACK, the TCP deletes the connection record relating to this connection and, when the 2MSL timer expires, the TCP at the other side does the same.

The sequences shown in all four examples relate to what is called an **orderly release** since both FINs are sent either with or after all outstanding data relating to the session has been sent. In some instances, however, a connection is closed abruptly by the TCP at one side sending a segment with the RST bit on. This results in both sides of the connection being closed immediately and hence any data currently being held by either TCP will be lost. This is called an **abortive release** and an example of its use is when the sequence numbers relating to the connection become unsynchronized.

7.3.3 Additional features

The various examples we used in the data transfer section were chosen to explain the main procedures relating to this part of the TCP protocol. In addition, however, there are a number of details relating to these procedures that the examples did not show. In this section we identify a number of these.

Persist timer

In the example shown in Figure 7.8, it was assumed that the duplicate ACK containing the window update – Ack = X + 2048, Win = 1024 – was received by the sending TCP error free. In practice, of course, the packet containing this segment may have been corrupted or lost. If this had occurred, since the TCP entity at the receiving side has initiated the reopening of the window, it is now waiting to receive further segments. However, because the ACK is not

received, the send window of the sending TCP is still zero and so it continues to assume that it cannot send any more segments. Although the data within segments is acknowledged, acknowledgments are not. Hence if the window update was not received, deadlock would occur with each side waiting for the other. Figure 7.13 shows how this can be avoided.

As we can see, the example relates directly to the sequence we showed earlier in Figure 7.8. Whenever the sending TCP sets its send window, W_S, to

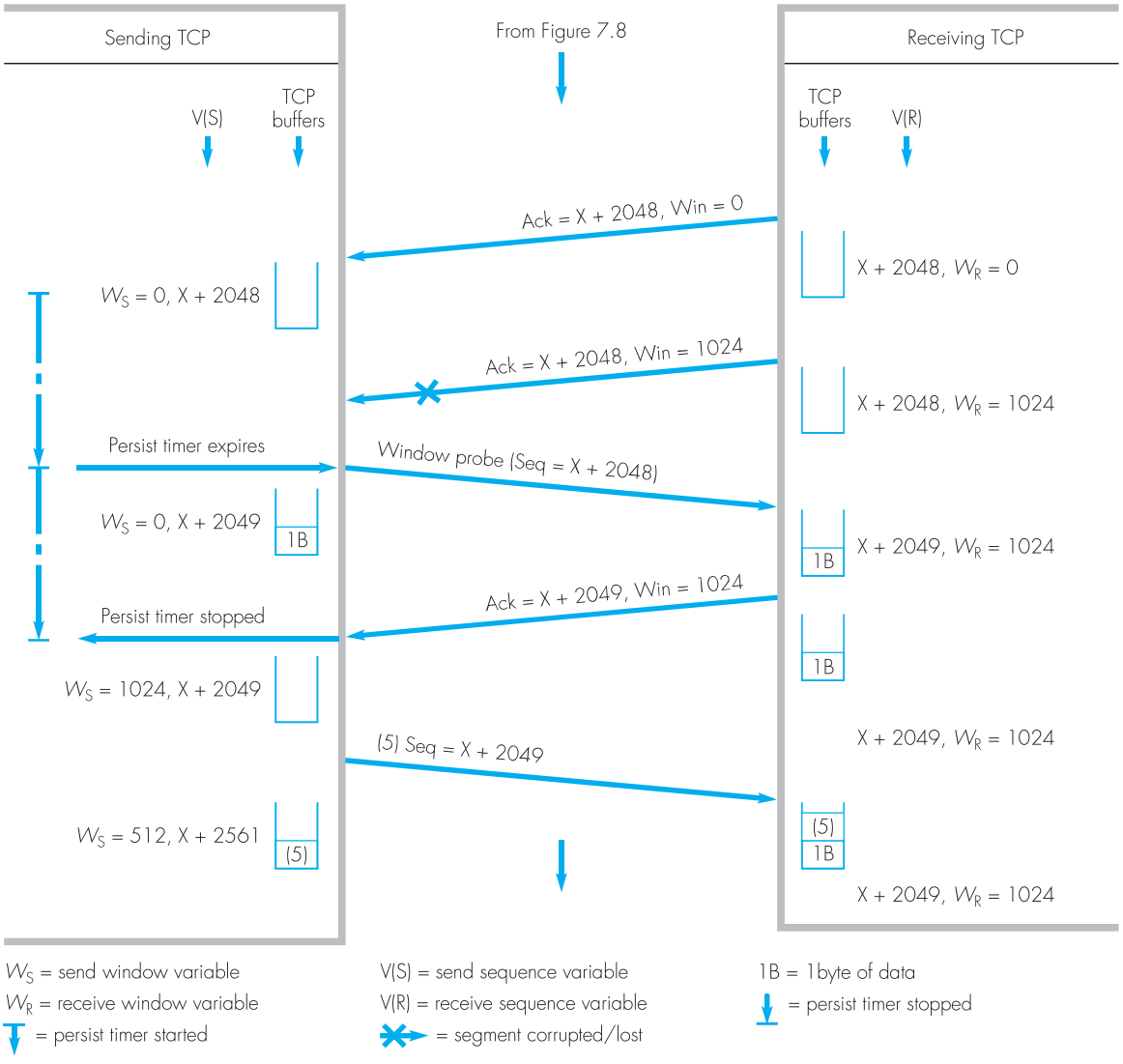


Figure 7.13 Persist timer: application and operation.

zero, it starts a timer. This is known as the **persist timer**, and if a segment containing a window update is not received before the timer expires, the sending TCP sends what is called a **window probe** segment. Even though the send window is zero, a TCP can always send a single byte of data. Hence the probe segment contains the first byte of the remaining data and, as we can see, if the ACK for this has a nonzero window value, then the flow of data segments can be resumed. Alternatively, should the value still be zero, then the timer is restarted and the procedure repeats. This is repeated at 60 s intervals until either a nonzero window value is received or the connection is terminated.

Keepalive timer

Once a connection between two TCP entities has been set up, it remains in place until the connection is terminated by the two communicating APs. As we saw in the previous section, this involves the two APs initiating the closure of their side of the connection. In most client–server applications, however, if the client host is simply switched off (instead of going through the normal log off procedure) then the connection from the server to the client will remain in place even though the client host is no longer responding. To overcome this, although not part of the TCP specification, many TCP implementations in servers include a timer known as the **keepalive timer**. The way this is used is shown in the example in Figure 7.14.

A separate timer is kept by the server TCP for all the connections – of which there can be many – that are currently in place. The default value of the keepalive timer is 2 hours and, should no data segments be exchanged over a connection during this time interval, the TCP in the server sends a probe segment to the client and sets the timer this time to 75 s. The probe segment has no data and has a *sequence number* of one less than the current V(S) of the server–client side of the connection. If the client host is still switched on – and the TCP connection is still in place – the client TCP simply returns an ACK for this with its current V(R) within it and, on receipt of this, the server TCP restarts the keepalive timer at 2 hours. If the client TCP does not respond, then the timer expires and the TCP in the server sends a second probe with the timer again set to 75 s. This procedure is repeated and, if no reply is received after 10 consecutive probes, the client is assumed to be switched off (or unreachable) and the server terminates the connection.

Silly window syndrome

The combined IP and TCP packet/segment headers are at least 40 bytes. Hence the larger the number of bytes in the data field of each segment, the smaller are the overheads associated with each transfer and the higher is the mean end-to-end data transfer rate. In the flow control example shown earlier in Figure 7.8, it was assumed that the AP at the receiving side read large (1024-byte) blocks of data from the receive buffer which, in turn, enabled the sending TCP to operate efficiently by always sending segments containing large amounts of data. In some interactive applications, however, a condition can arise that results in a very small number of bytes being sent in

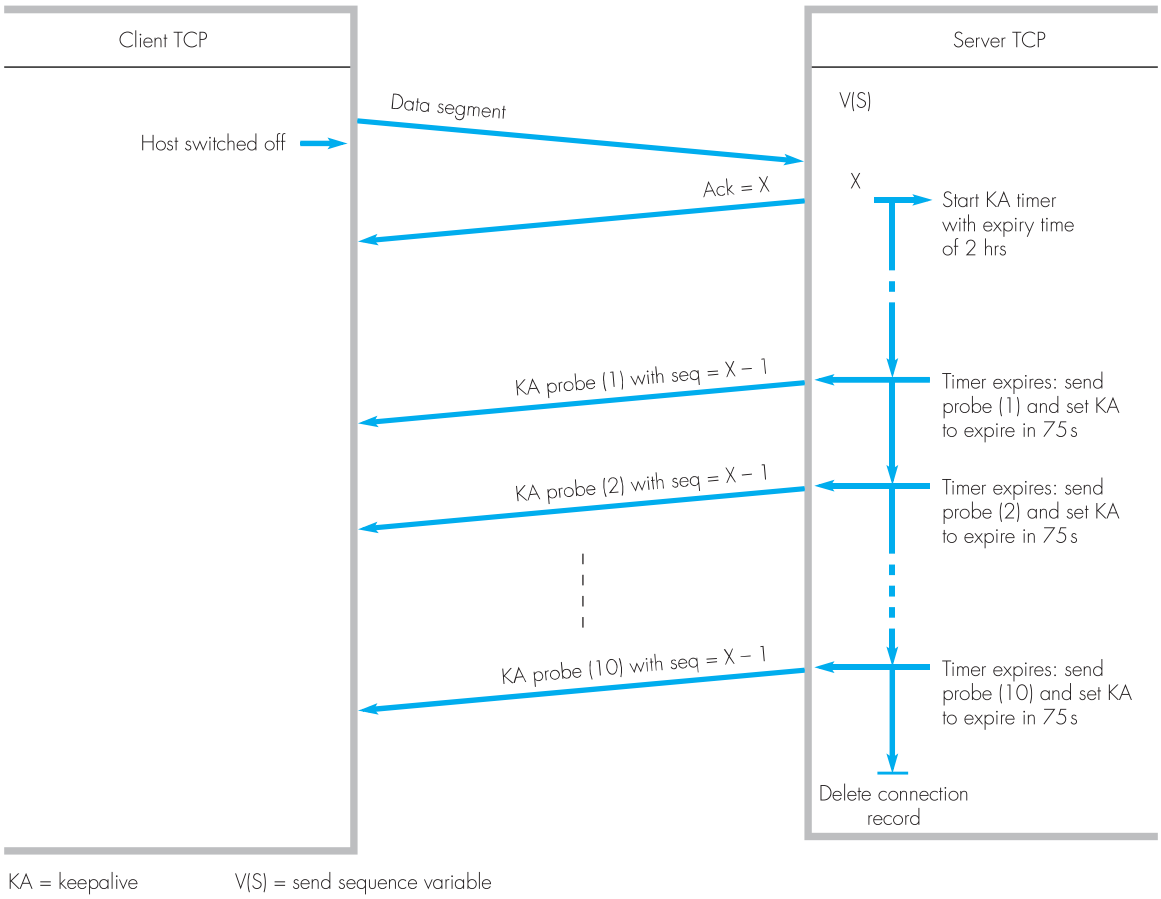


Figure 7.14 Keepalive timer: application and operation.

each segment. It is known as the **silly window syndrome** (**SWS**) and can arise either at the receiving side or the sending side.

To avoid this occurring, an added feature defined by Clark is incorporated into the basic flow control procedure we discussed earlier. At the receiving side this forbids a receiving TCP entity from advertising a small window and, at the sending side, it forbids a sending TCP entity from sending segments containing small amounts of data. Note that both features are complementary to the Nagle algorithm we discussed earlier and, in many instances, both operate together.

To illustrate how the problem can arise at the receiver, assume that in an interactive application a server AP initiates the transfer of a large character file to a client AP but the latter proceeds to read it (from the receive buffer of the receiving TCP) a single character at a time. On receipt of the first block

of characters/bytes sent by the TCP in the server – up to the current maximum window size – the receiving TCP in the client would first return an ACK for the complete block but with a window value of zero. Then, assuming the procedure we showed in the earlier example in Figure 7.8, each time the client AP reads a character from the TCP receive buffer, the client TCP will return a window update to inform the TCP at the server side that it can now send a further character/byte. This it duly does and, on its receipt, the client TCP returns an ACK for it but again with a window value of zero. Similarly, when the next character is read by the AP, the client TCP returns a second window update of one byte/character and the procedure repeats until the complete file has been transferred in this way.

To avoid this happening, a receiving TCP is prevented from sending a window update until there is sufficient space in its buffer either for a segment equal to the maximum segment size in use or for one half of the maximum buffer capacity – and hence window capacity – whichever is the smaller. An example showing this is given in Figure 7.15. Part (a) illustrates the problem and part (b) shows how the solution attributed to Clark avoids the problem occurring.

In the example it is assumed that the window size in use is 1024 bytes and the MSS is also 1024 bytes. Hence, as we can deduce from part (a), after the first 1024-byte block has been transferred, without the Clark extension, each of the remaining 1024 bytes is transferred in a separate segment. Moreover, each of these may require an ACK and a window update segment. With the Clark extension, however, as we can see from part (b), after the first 1024 bytes have been transferred the remaining bytes are transferred in just two 512-byte segments, the choice of 512 being half of the buffer capacity.

As we indicated earlier, the same problem can arise at the sending side. For example, in an interactive application in which a user enters a string of characters at a keyboard, the sending AP may write each character into the send buffer as it is entered. In the absence of the Clark extension, the sending TCP may then proceed to send each character in a single segment. With the Clark extension, however, the sending TCP is made to wait until it has accumulated enough bytes/characters to fill a segment equal to either the MSS in use with the connection or to, one half of the buffer capacity of the receiving TCP. The latter is determined from the maximum window update value the sending TCP has received from the receiving TCP.

Window scale option

During our discussion of the HDLC protocol in Section 1.4.9, we saw that for transmission links that have a large bandwidth/delay product, the sizes of the send and receive sequence number fields in the frame header are extended in order to allow a larger window size to be used. As we saw in Example 1.6, this improves the utilization of the available link bandwidth. This can also be a requirement with TCP when the transmission path followed through the Internet covers a large distance – and hence has a large signal propagation delay – and a high mean bit rate.

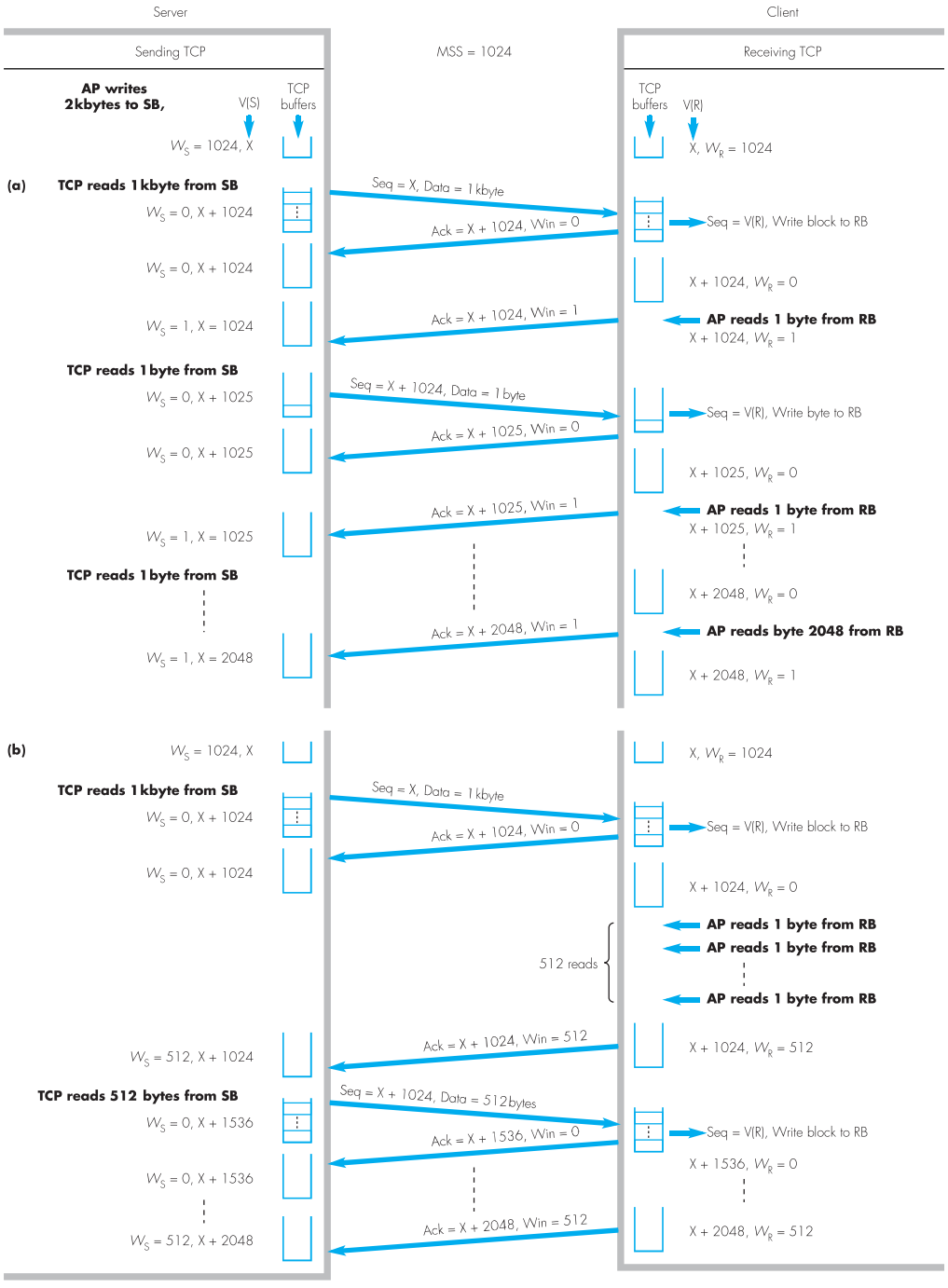


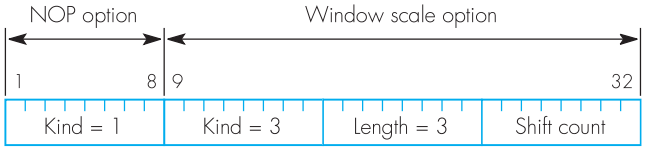
Figure 7.15 Silly window syndrome example: (a) the problem; (b) Clark's solution.

As we saw in Figure 7.4, the *window size* field in the segment header is 16 bits which allows a maximum window size of 65 535 bytes. However, when the path involves intercontinental lines, for example, a typical propagation delay is 40 ms. Also, since optical fiber is now widely used, mean bit rates in excess of 155 Mbps are common. This means that, in order to fully utilize the available transmission capacity, a window size in excess of 775 000 bytes is required. Hence, even though a single connection may only use a portion of the available bit rate, it can be seen that, for connections that span large distances, a larger maximum window size is required.

In order to achieve this without changing the format of the segment header, an option has been defined that enables a scaling factor to be applied to the value specified in the *window size* field. This is called the **window scale option** and is defined in **RFC 1323**. The format of this option is shown in Figure 7.16.

Normally, the option is included in the SYN segment when a connection is being established and, since the actual window size is determined by the size of the receive buffer, a different scaling factor can be agreed for use in each direction. Also, since the option is only three bytes in length, it is always preceded by a single byte containing a value of 1. This is known as a **no operation** (**NOP**) **option** and they are used as pad bytes so that all options are multiples of four bytes.

The scaling factor is defined in the *shift count* field of the option. As we can see, this is a one-byte field and the count value can be between 0 – no scaling in use – and 14. Although the *window size* field in each segment header is only 16 bits, the send and receive window variables kept by each TCP entity are both 32-bit values. The actual window value is then computed by first writing the 16-bit value from the *window size* field into the corresponding window variable and then shifting this left – and hence multiplying it by 2 – by the number in the *shift count* field. A shift count of 1, therefore, means that the maximum window size can be up to $65\,535 \times 2^1 = 131\,070$ bytes, and a shift count of 14 means that the maximum size can be $65\,535 \times 2^{14} = 1\,073\,725\,440$ bytes. As we indicated earlier, however, the maximum window size that can be used in each direction is determined by the size of the corresponding receive



NOP = no operation option: used to pad the window scale option to 4 bytes. The value in the *shift count* field is the power of 2 multiples of the value in the *window size* field. The maximum count value is 14.
Shift count = 0 , no scaling: maximum window = 65 535 (bytes)
Shift count = 1 , multiply by 2^1: maximum window = 131 070
Shift count = 14, multiply by 2^{14}: maximum window = 1 073 725 440

Figure 7.16 Window scale option format.

buffer. Normally, therefore, the shift count to be used for each direction is chosen by the TCP entity since it knows the amount of memory that has been allocated for the receive buffer.

Time-stamp option

During our discussion of the error control procedure, it was assumed that the measurement procedure used to estimate the worst-case round-trip time (RTT) – used by a TCP to compute the retransmission timeout (RTO) interval – is carried out for every data segment that is sent. Although this is the case in some implementations, in others a measurement update occurs only for one segment per window. In general, this is sufficient for transfers involving a small window size – and hence small number of segments per window – but, with a large window, the use of a single segment per window can lead to a poor estimate of the RTT. As we indicated, this can result in many unnecessary retransmissions. Hence to obtain a more accurate estimate of the RTT, the **time-stamp option** is used with these implementations when a large window size is detected. The option is defined in **RFC 1323** and allows the sending TCP to obtain an estimate of the RTT with each ACK it receives.

The option is requested by the TCP that performs an active open, by including the time-stamp option in the SYN segment. The request is then accepted if the receiving TCP includes a time-stamp option in the SYN segment that it returns. Once accepted, both TCPs can then include a time-stamp option in every data segment that they send. The format of the option is shown in Figure 7.17(a) and, as we can see, since it is 10 bytes long, the option is preceded by two NOP option bytes.

The TCP at each side of a connection keeps a 32-bit *timeout timer* that it uses to estimate the RTT for its own direction of transmission. The timer at each side is independent – not synchronized – and is incremented at intervals of between 1 ms and 1 s, a typical interval being 500 ms. For every data segment it sends, the sending TCP reads the current time from the timer and writes this into the *time-stamp value* field. Then, when the TCP returns an ACK for a segment, it writes the same time-stamp value in the *time-stamp echo reply* field. Hence on receipt of each ACK segment, the sending TCP can estimate the RTT for its direction of the connection by computing the difference between the current time in its timer and the time-stamp value contained in the option field of the ACK.

This procedure will work providing the receiving TCP returns an ACK for each data segment it receives. In many implementations, however, as we showed earlier in Figure 7.8, an ACK may be returned only after multiple data segments have been received. Indeed, following the widespread introduction of optical fiber, most TCP implementations now return an ACK for every other data segment received. In such cases, the question arises as to which of the time-stamps – one from each of the data segments that it has received – does the receiving TCP return in the *echo reply* field of the ACK. The answer is the time-stamp from the first in-sequence segment that the receiving TCP received after it returned the last ACK.

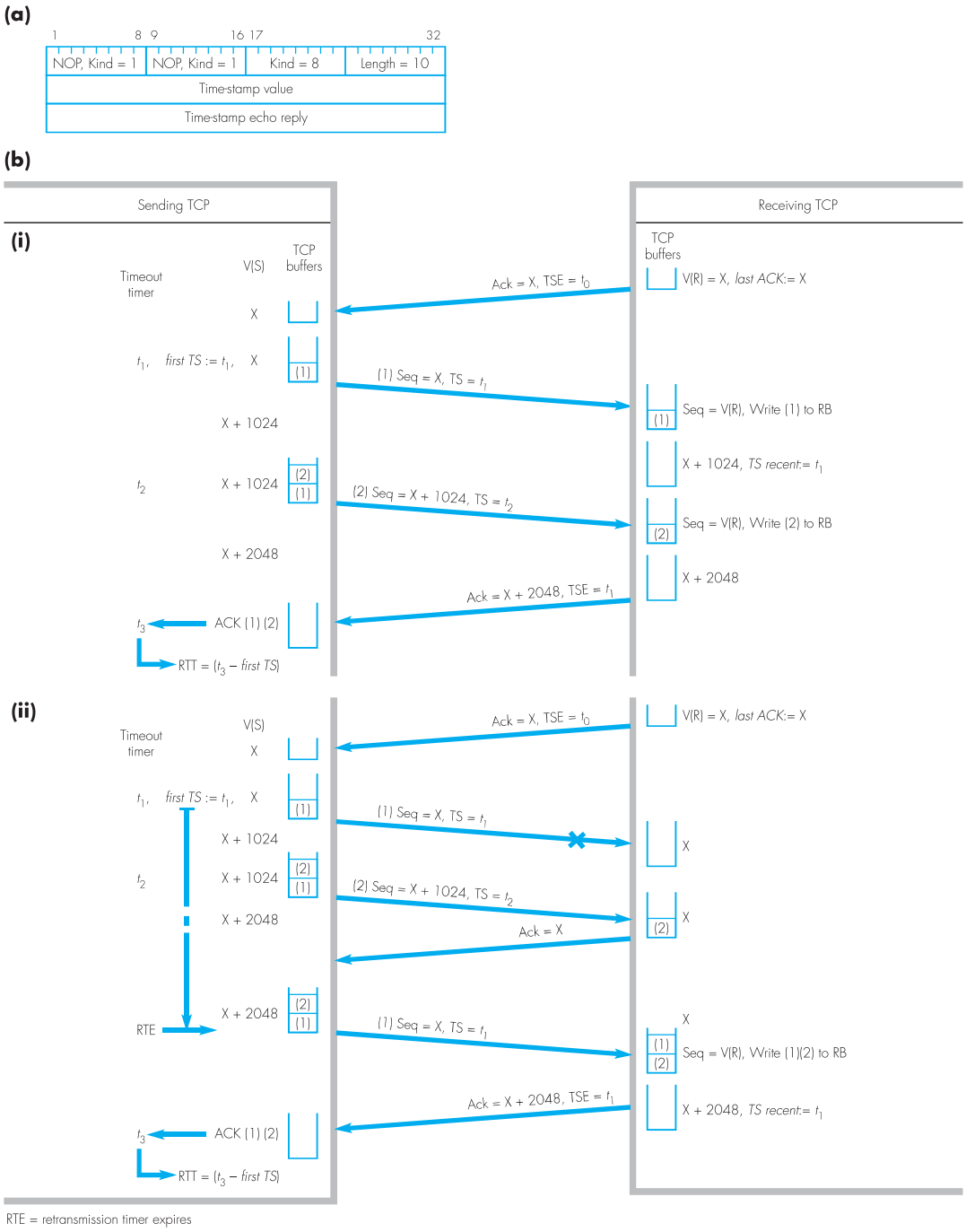


Figure 7.17 Time-stamp option: (a) option format; (b) two examples.

To implement this, the receiving TCP maintains two variables in its connection record: *lastACK* and *TSrecent*. Each time the receiving TCP returns an ACK, it keeps a record of the V(R) that it sent in the ACK in *lastACK*. Then, when the first data segment arrives after it has returned the ACK, if the sequence number in the segment header equals that stored in *lastACK*, it keeps a record of the time-stamp value it contains in *TSrecent*. As each subsequent data segment arrives, the TCP simply processes it in the normal way and, when it returns an ACK, includes the current value in *TSrecent* in the *echo reply* field of the option. Thus, the RTT computed by the sending TCP will reflect that the receiving TCP is only returning an ACK on receipt of multiple data segments.

If the sequence number in a segment is not that expected – that is, the sequence number is greater than V(R) – this indicates that a segment has been lost. In this case, the receiving TCP simply returns an ACK and proceeds to wait for the missing segment to be received. Then when it arrives, the time-stamp from this is echoed, not the time-stamp from the out-of-sequence segment. To allow for the possibility of the first segment in a new sequence being corrupted or lost, the sending TCP keeps a record of the time it first sent the segment in a variable called *firstTS*. Should the segment need retransmitting, the value in *firstTS* is used as the *time-stamp value*. In this way, the corrupted RTT also includes the time to retransmit the segment. Although this is an overestimate of the RTT, it is considered to be better than an underestimate, as would have been the case if the later time had been used. Two examples of sequences that show the two alternatives are shown in Figure 7.17(b).

In the first example, it is assumed that no lost segments occur and an ACK is returned on receipt of every other data segment. The time the first segment is sent, t_1, is stored in *firstTS* and, at the receiving side, since it is the segment expected, *TSrecent* becomes equal to the time-stamp value from the segment, t_1. Then, when the ACK is returned after the second segment is received, the value in the *time-stamp echo* field is set to t_1. The RTT is then computed as the difference between the time when the ACK was received and the value in *firstTS*, t_1.

In the second example, it is assumed that the first data segment in the new sequence is corrupted/lost but the second is received error free. In this case, *TSrecent* is not updated until the second copy of the retransmitted first segment is received. However, since this has the same *time-stamp value* as the corrupted first segment, the *time-stamp echo* is the same as before. Hence, as we can see, the computed RTT includes the time taken to retransmit the first segment.

SACK-permitted option

As we indicated earlier, for connections that involve paths through the Internet that span large distances, the propagation delay – also referred to as latency – of the path can be several tens of milliseconds. Hence in addition to influencing the choice of window size, it also has an impact on the efficiency of the error control scheme. As we showed in the example in Figure 7.7, to allow for segments being received out of sequence, a segment is retransmitted

only after three duplicate ACKs have been received. Clearly, with connections that have a large RTT associated with them, the delays involved each time a packet/segment is lost or corrupted can be large. In order to reduce this delay, the alternative selective repeat/acknowledgment error control scheme can be used. This is defined in **RFC 2018** and is requested by including the **SACK-permitted option** in the SYN segment header. It is then accepted if the receiving TCP includes the same option in the SYN segment that it returns.

We explained the principle of operation of the selective repeat error control scheme in Section 1.4.3 and gave two example frame sequences showing how the protocol overcomes both a corrupted I-frame and a corrupted ACK-frame. A similar protocol is used with TCP SACK except in this, the SACK segment returned by a receiving TCP contains a list of the data segments that are missing in a specified window of data. In this way, all the missing segments are retransmitted in a single RTT.

Protection against wrapped sequence numbers

When very large amounts of data are being transferred between two hosts using a high bit rate LAN, for example, to speed up the transfer, the window scale option is often used with the maximum shift count of 14. An example is performing a backup of the contents of a disk over a LAN where, since this may involve many gigabytes, it is possible for the sequence numbers to wrap around during the backup. This means that a segment that is lost during one pass through the sequence numbers may be retransmitted and received during a later pass through the numbers so ruining the integrity of the transfer.

To overcome this possibility, the **PAWS** – protection against wrapped sequence numbers – algorithm is often used. This involves the time-stamp option being selected and both sides using the current 32-bit time-stamp as an extension to the 32-bit sequence number. Effectively, this produces 64-bit sequence numbers, which overcomes the problem.

7.3.4 Protocol specification

To finish our discussion of TCP we shall illustrate how the segment sequences shown in the various examples, coupled with the socket primitives shown in Table 7.1, relate directly to the formal specification of the TCP protocol entity. Recall from Section 1.4.7 that protocol specifications are carried out in a number of ways. One of the most widely used methods is to use a combination of a state transition diagram and an extended event–state table. A state transition diagram illustrates the various sequences in a pictorial form and hence the various examples relate directly to this. The extended event–state table method, however, is better for implementing a protocol since, as we saw in the example in Section 1.4.7, program code can be derived directly from this. To avoid too much detail, we shall consider only the specification of the connection establishment and termination phases of a basic client TCP and the related server TCP since, as we saw in Figure 1.33, the data transfer phase contains many state variables and predicates.

Using the methodology we established in Section 1.4.7, the various incoming events, protocol states, outgoing events, and specific (internal) actions for both the client and server are shown in parts (a) and (b) of Table 7.2 respectively. As we can see, these relate directly to the segment sequence shown in Figure 7.5(a) – connection establishment – and Figure 7.12(a) – connection termination. The specifications of both the client TCP and the server TCP protocols in both state transition diagram and extended event–state diagram forms are then shown in parts (a) and (b) of Figure 7.18 respectively.

Table 7.2 Abbreviated names used in the specification of the TCP protocol: (a) client TCP; (b) server TCP.

(a) Incoming events

Name	Interface	Meaning
connect ( )	AP_user	Initiate the setting up of a connection
SYN+ACK	IP_provider	SYN+ACK received from server (TCP)
DATA	IP_provider	Block of data received from server
close( )	AP_user	Initiate the closure of a connection
ACK	IP_provider	ACK received from server
2MSL	Timer	2MSL timer expires
FIN	IP_provider	FIN received from server
RST	IP_provider	Abort connection

States

Name	Meaning
CLOSED	No connection in place
SYN_SENT	SYN sent to server (TCP)
ESTABLISHED	SYN received from server, connection now in place
FIN_WAIT1	FIN sent to server
FIN_WAIT2	ACK to FIN received from server
TIMED_WAIT	FIN received from server

Outgoing events

Name	Interface	Meaning
SendSYN	IP provider	Send SYN to server (TCP)
SendACK	IP provider	Send ACK to server
SendDATA	IP provider	Send block of data to server
SendFIN	IP provider	Send FIN to server

▶

Table 7.2 Continued.

Specific actions

[1] Block client AP	[3] Write EOF to receive buffer (RB)
[2] Unblock client AP	[4] Start 2MSL timer

(b) Incoming events

Name	Interface	Meaning
listen ( )	AP_user	Create queue for connection requests
accept ( )	AP_user	Block server AP
SYN	IP_provider	SYN received from client
RST	IP_provider	RST received from client
ACK	IP_provider	ACK received from client
DATA	IP_provider	DATA received from client
FIN	IP_provider	FIN received from client
RST	IP_provider	Abort connection

States

Name	Meaning
CLOSED	No connection in place
LISTEN	Server AP blocked, TCP waiting for a SYN
SYN_RCVD	SYN received from a client AP
ESTABLISHED	ACK for own SYN received, connection now in place
CLOSE_WAIT	FIN received from client TCP
LAST_ACK	FIN sent and waiting for an ACK

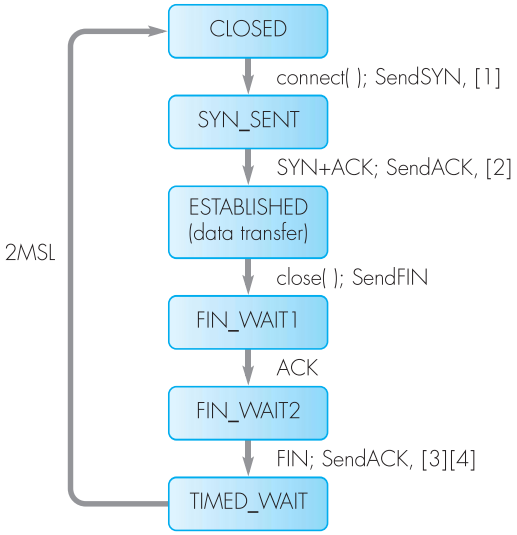
Outgoing events

Name	Interface	Meaning
SendSYN+ACK	IP_provider	Send SYN+ACK to client (TCP)
SendDATA	IP_provider	Send block of data to client
SendFIN	IP_provider	Send FIN to client
SendACK	IP_provider	Send ACK to client

Specific actions

[1] Create queue for connection requests	[3] Unblock server AP
[2] Block server AP	[4] Write EOF to receive buffer (RB)

(a)



Present state \ Incoming event	connect( )	SYN+ACK	close( )	ACK	FIN	2MSL
CLOSED	1	0	0	0	0	0
SYN_SENT	0	2	0	0	0	0
ESTABLISHED	0	0	3	0	0	0
FIN_WAIT1	0	0	0	4	0	0
FIN_WAIT2	0	0	0	0	5	0
TIMED_WAIT	0	0	0	0	0	6

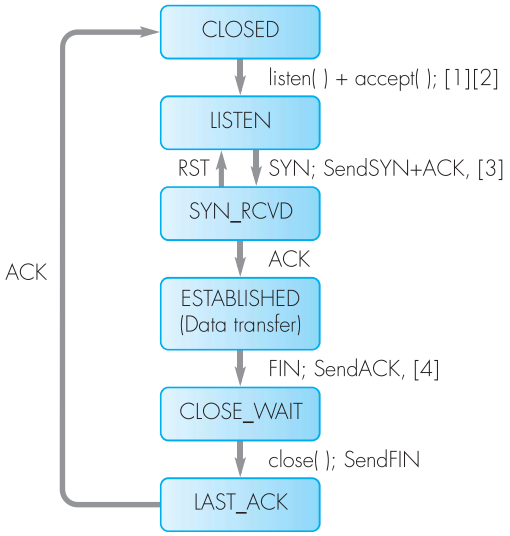
0 = SendRST, CLOSED
1 = SendSYN, [1], SYN_SENT
2 = SendACK, [2], ESTABLISHED
3 = SendFIN, FIN_WAIT1
4 = FIN_WAIT2
5 = SendACK, [3][4], TIMED_WAIT
6 = CLOSED

(*continued overleaf*)

Figure 7.18 TCP protocol specifications: (a) client; (b) server.

It should be stressed that the specifications are only simplified versions of the real specifications. For instance, they do not include the actions taken in the event of simultaneous connection requests or simultaneous closes occurring. Also, when in the ESTABLISHED state, many predicates – for example to check whether sequence numbers are valid and within the current window – and state variables are used when determining the required action and new state. Again, a good insight into the latter can be obtained from the example sequences associated with the data transfer phase.

(b)



Present state \ Incoming event	listen( ) + accept( )	SYN	RST	ACK	FIN	close( )
CLOSED	1	0	0	0	0	0
LISTEN	0	2	0	0	0	0
SYN_RCVD	0	0	3	4	0	0
ESTABLISHED	0	0	0	0	5	0
CLOSE_WAIT	0	0	0	0	0	6
LAST_ACK	0	0	0	7	0	0

0 = SendRST, CLOSED
1 = [1][2], LISTEN
2 = SendSYN+ACK, [3], SYN_RCVD
3 = LISTEN
4 = ESTABLISHED
5 = SendACK, [4], CLOSE_WAIT
6 = SendFIN, LAST_ACK
7 = CLOSED

Figure 7.18 Continued.

7.4 UDP

Recall that with TCP there is no correlation between the size of the messages/blocks of data submitted by a user AP and the amount of data in each TCP segment that is used to transfer the messages. Typically, as we saw in Section 7.2, the latter is determined by the path MTU to avoid fragmentation of each segment occurring.

In contrast, with UDP each message/block of data that is submitted by a user AP is transferred directly in a single IP datagram. On receipt of the message, the source UDP simply adds a short header to it to form what is called a **UDP datagram**. This is then submitted to the IP layer for transfer over the internet using, if necessary, fragmentation. At the destination, the IP first determines from the *protocol* field in the datagram header that the destination protocol is UDP, and then passes the contents of the (IP) datagram to the UDP.

The latter first determines the intended user AP from a field in the UDP datagram header and then passes the contents of the (UDP) datagram to the peer user AP for processing. There are no error or flow control procedures involved and hence no connection setup is required. The service offered by UDP to a user AP, therefore, is simply an extension of the service provided by IP. Hence in addition to two-party calls, multicast group calls can be supported. Nevertheless, the set of service primitives and the protocol are both simpler than those of TCP. As with TCP, we shall discuss the services and the protocol separately.

7.4.1 User services

As with TCP, the most widely used set of user service primitives associated with UDP are the Berkeley Unix socket primitives. With most applications that use UDP, the two user APs either exchange messages on a request-response basis or simply initiate the transfer of blocks of data as these are generated. A typical list of service primitives – system/function calls – is given in Table 7.3 and their use is shown in diagrammatic form in Figure 7.19.

Table 7.3 List of socket primitives associated with UDP and their parameters.

Primitive	*Parameters*
socket( )	service type, protocol, address format, return value = socket descriptor or error code
bind( )	socket descriptor, socket address (= host IP address + port number) return value = success or error code
sendto( )	socket descriptor, local port number, destination port number, destination IP address, precedence, pointer to message buffer containing the data to send, data length (in bytes), return value = success or error code
receive( )	socket descriptor, pointer to message buffer into which the data should be put, length of the buffer, return value = success/error code or end-of-file (EOF)
shutdown( )	socket descriptor, return value = success or error code

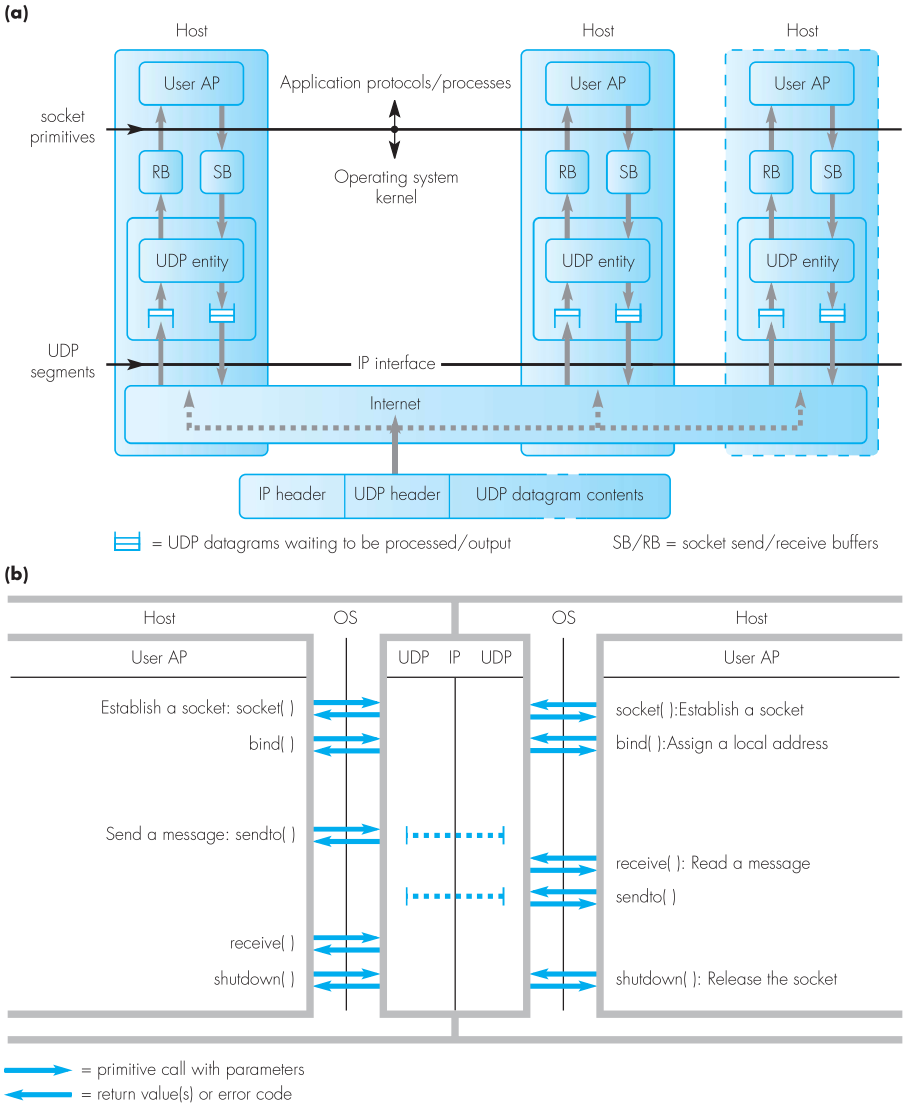


Figure 7.19 UDP socket primitives: (a) socket interface; (b) primitives and their use.

As we can see in Figure 7.19(b), prior to exchanging any messages, each of the user APs involved in the call must first establish a socket between itself and its local UDP. The parameters associated with the *socket( )* primitive include the service required (datagram service), the protocol (UDP), and the address format (Internet). Once a socket has been created – and send/receive memory buffers allocated – a socket descriptor is returned to the AP which it

then uses with each of the subsequent primitive calls. If this is the only AP that is running in the host, the AP can now start to send and receive messages. If not – for example a server is involved – the AP then issues a *bind( )* primitive which, in addition to the socket descriptor, has an address parameter. This comprises the IP address of the host plus the 16-bit port number the AP wishes to be assigned to the socket. In the case of a server AP, for example, this will be the related well-known port number. When a *bind( )* is not used, the port number will be an ephemeral port number and assigned locally.

Once a socket has been created, the user AP can start to send and receive messages. However, since no connection is involved – and hence no connection record has been created – in addition to the message, the AP must specify the IP address of the destination host – or the IP multicast address in the case of multiple destinations – and the port number of the destination socket/AP. Also, if required, it must specify a precedence value to be sent in the *type of service* field of the IP datagram header. Hence, as we can see in Table 7.3, the *sendto( )* primitive includes each of these fields in its set of parameters. Finally, when all the data transfers associated with the call/session have been carried out, the socket is released by issuing a *shutdown( )* call.

7.4.2 Protocol operation

The format of each UDP datagram is shown in Figure 7.20(a). The *source port* is the port number of the sending application protocol/socket and the *destination port* is that of the peer (receiving) application protocol(s). Both are 16-bit integers. The value in the *length* field is the number of bytes in the complete (UDP) datagram and includes the 8-byte header and the contents of the *data* field. The *checksum* covers the complete datagram, header plus contents. In addition, as with TCP, since only a simple checksum is used to compute the checksum value in the IP header, in order to add an additional level of checking, some selected fields from the IP header are also included in the computation of the UDP checksum. The fields used form what is called the **UDP pseudo header**. These are identified in Figure 7.20(b) and, as we can see, they are the source and destination IP addresses and the protocol value (=17 for UDP) from the IP header, plus the value from the length field in the UDP header.

The computation of the UDP checksum uses the same algorithm as that used by IP. As we saw in Section 6.2, this is computed by treating the complete datagram as being made up of a string of 16-bit words that are added together using 1s complement arithmetic. Using 1s complement number representation, a value of zero is represented as either all 1s or all 0s. With UDP, however, if the computed 1s complement sum is all 0s, then the checksum is sent as all 1s. This is done because the use of a checksum with UDP is optional and, if the value in the checksum field is all 0s, this indicates the sender has not sent a checksum.

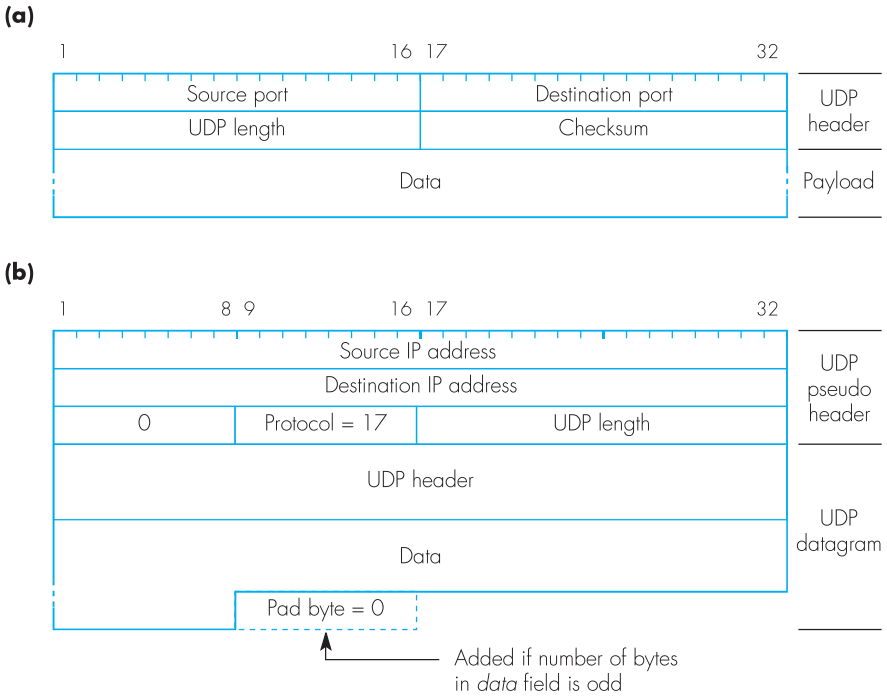


Figure 7.20 UDP datagram format: (a) UDP header fields; (b) fields used in pseudo header for computation of checksum.

Since the number of bytes in the original UDP data field – and hence submitted application protocol data unit – may be odd, in order to ensure the same checksum is computed by both UDPs, a pad byte of zero is added to the data field whenever the number of bytes in the original data field is odd. Thus the value in the length field must always be an even integer. Also, since the UDP datagram is carried in a single IP datagram, as we saw in Section 6.3, in order to avoid fragmentation, the size of each submitted application protocol data unit must be limited to that dictated by the MTU of the path followed through the Internet by the IP datagram. For example, assuming a path MTU of 1500 bytes, allowing for the 8 bytes in the UDP header and 20 bytes in the IP header, the maximum submitted application PDU should be limited to 1472 bytes if fragmentation is to be avoided.

Finally, although the maximum theoretical size of a UDP datagram – as determined by the maximum size of an IP datagram, which is 65 535 (64K–1) bytes – is 65 507 (65 535–20–8) bytes, the maximum value supported by most implementations is 8192 bytes or less.