more sensitive to delay and jitter than the packets in the first category and hence should be transmitted before them during periods of congestion. Also, within the second category, packets containing compressed video are more sensitive to packet loss than packets containing just audio and hence are given a higher priority.

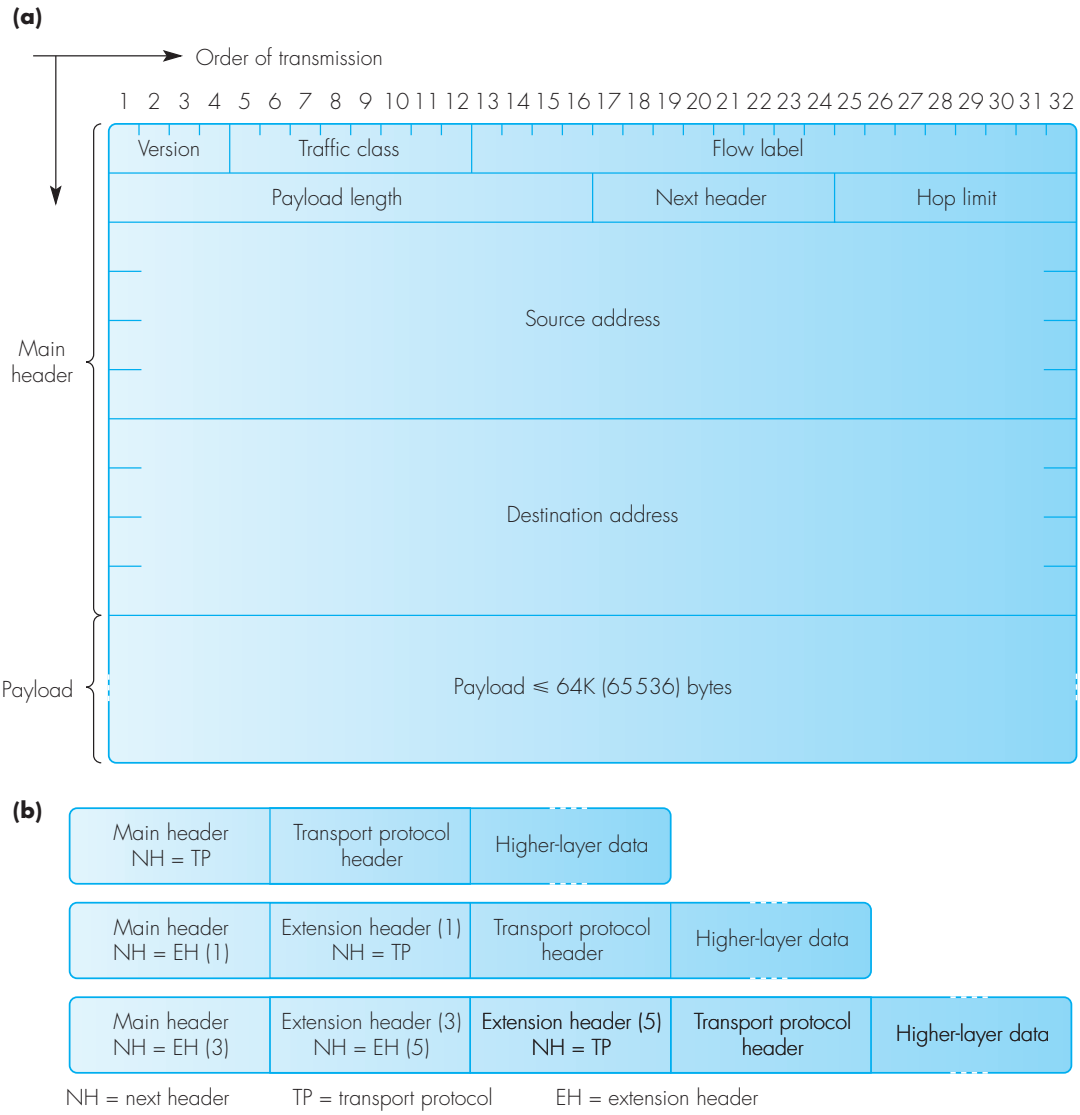


Figure 6.41 IPv6: (a) main header fields and format; (b) position and order of extension headers.

Flow label

This is a new field and is closely linked to the *traffic class* field. It is set to zero in best-effort packets and, in packets in the second category, it is used to enable a router to identify the individual packets relating to the same call/session. As we saw earlier in Section 6.7.1, one approach to handling packets containing real-time streams is to reserve resources – for example transmission bandwidth – for such calls in advance of sending the packets relating to the call. During the reservation procedure, the call is allocated a *flow label* by the source. Also, each router along the reserved path keeps a record of this, together with the source and destination IP addresses, in a table. Routers then use the combined *flow label* and source IP address present in each packet header to relate the packet to a specific call/flow. The related routing information is then retrieved from the routing table and this is used, together with the *traffic class* value, to ensure the QoS requirements of the call/flow are met during the forwarding procedure.

Payload length

This indicates the number of bytes that follow the basic 40-byte header in the datagram. The minimum length of a basic datagram is 536 bytes and the maximum length is 64K bytes. The *payload length* is slightly different from the *total length* field used in the header of an IPv4 datagram since, as we explained in Section 6.2, the *total length* includes the number of bytes in the datagram header.

Next header

As we show in Figure 6.41(b), a basic IPv6 datagram comprises a main header followed by the header of the peer transport layer protocol (TCP/UDP) and, where appropriate, the data relating to the higher layers. With a basic datagram, therefore, the *next header* field indicates the type of transport layer protocol (header) that follows the basic header. If required, however, a number of what are called ***extension headers*** can be inserted between the main header and the transport protocol header. Currently, there are six types of extension header defined and, when present, each extension header starts with a new *next header* field which indicates the type of header that follows. The *next header* field in the last extension header always indicates the type of transport protocol header that follows. Thus, the *next header* field in either the main header or the last extension header plays the same role as the *protocol* field in an IPv4 datagram header.

Hop limit

This is similar to the *time-to-live* parameter in an IPv4 header except the value is a hop count instead of a time. In practice, as we explained in Section 6.2, most IPv4 routers also use this field as a hop count so the change in the field's name simply reflects this. The initial value in the *hop limit* field is set by the source and is decremented by 1 each time the packet/datagram is forwarded. The packet is discarded if the value is decremented to zero.

Source address and destination address

As we indicated earlier, these are 128-bit addresses that are used to identify the source of the datagram and the intended recipient. In most cases this will be the destination host but, as we shall explain later, it might be the next router along a path if source routing is being used. Unlike IPv4 addresses, an IPv6 address is assigned to the (physical) interface, not to the host or router. Hence in the case of routers (which have multiple interfaces) these are identified using any of the assigned interface addresses.

6.8.2 Address structure

As we showed in Figure 6.19, the various networks and internetworks that make up the global Internet are interconnected in a hierarchical way with the access networks at the lowest level in the hierarchy and the global backbone network at the highest level. However, the lack of structure in the netid part of IPv4 addresses means that the number of entries in the routing tables held by each gateway/router increases with increasing height in the hierarchy. At the lowest level, most access gateways associated with a single site LAN have a single netid, while at the highest level, most backbone routers/gateways have a routing table containing many thousands of netids.

In contrast, the addresses associated with telephone networks are hierarchical with, for example, a country, region, and exchange code preceding the local number. This has a significant impact on the size of the routing tables held by the switches since, at a particular level, all calls with the same preceding code are routed in the same way. This is known as **address aggregation**.

As we explained earlier in Section 6.4.3, classless inter-domain routing is a way of introducing a similar structure with IPv4 addresses and reduces considerably the size of the routing tables held by the routers/gateways in the global backbone. From the outset, therefore, IPv6 addresses are hierarchical. Unlike telephone numbers, however, the hierarchy is not constrained just to a geographical breakdown. The large address space available means that a number of alternative formats can be used. For example, to help interworking with existing IPv4 hosts and routers, there is a format that allows IPv4 addresses to be embedded into an IPv6 address. Also, since the majority of access networks are now Internet service provider (ISP) networks, there is a format that allows large blocks of addresses to be allocated to individual providers. The particular format being used is determined by the first set of bits in the address. This is known as the **prefix format** (**PF**) and a list of the prefixes that have been assigned – together with their usage – is given in Figure 6.42(a).

Unicast addresses

As we can see, addresses starting with a prefix of 0000 0000 are used to carry existing IPv4 addresses. There are two types, the formats of which are shown

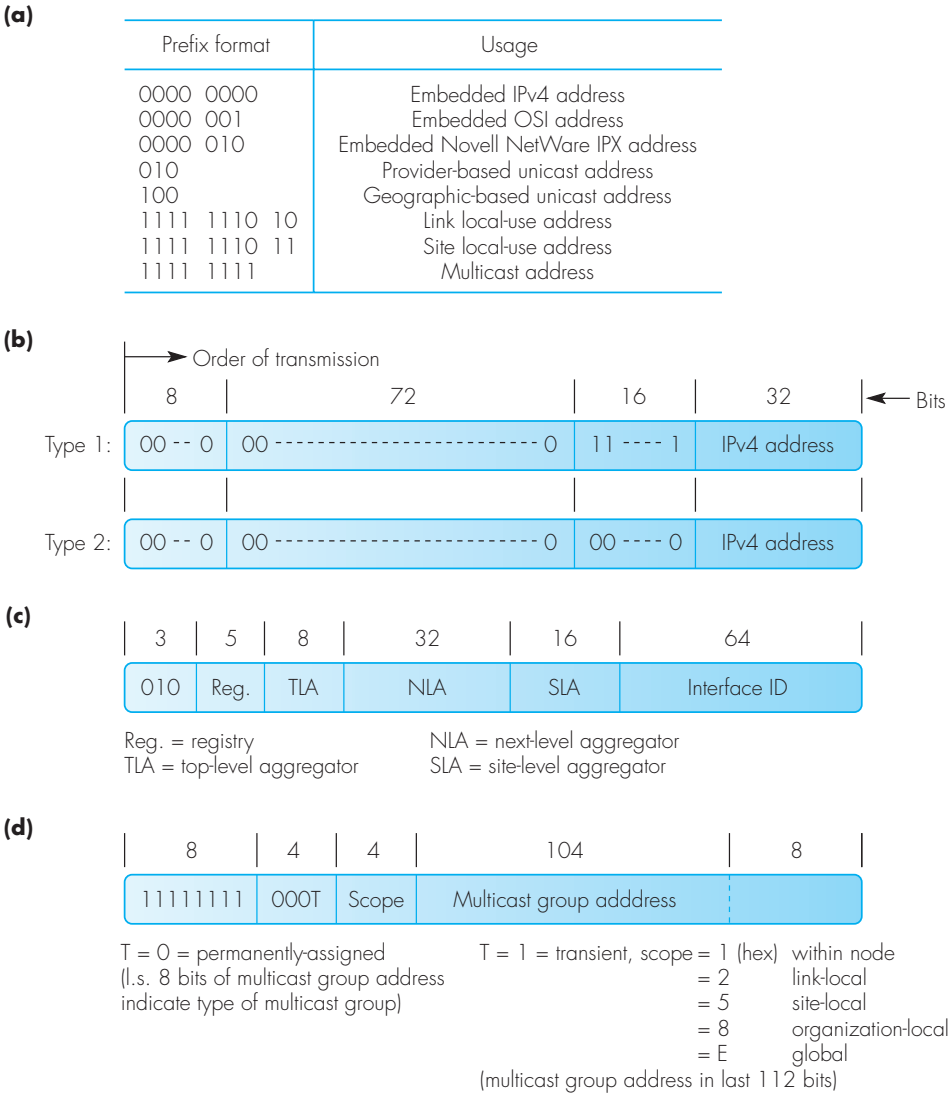


Figure 6.42 IPv6 addresses: (a) prefix formats and their use; (b) IPv4 address types; (c) provider-based unicast address format; (d) multicast address format.

in Figure 6.42(b). As we shall expand upon in Section 6.9, a common requirement during the transition from IPv4 to IPv6 is to tunnel the IPv6 packets being generated by the two communicating IPv6 hosts – often written **V6 hosts** – over an existing IPv4 network/internetwork. Hence to simplify the routing of the IPv4 packet – containing the IPv6 packet within it – the IPv6

address contains the IPv4 address of the destination gateway embedded within it. The second type is to enable a V4 host to communicate with a V6 host. The IPv4 address of the V4 host is then preceded by 96 zeros. In addition, two addresses in this category are reserved for other uses. An address comprising all zeros indicates there is no address present. As we shall expand upon in Section 6.8.4, an example of its use is for the source address in a packet relating to the autoconfiguration procedure. An address with a single binary 1 in the least significant bit position is reserved for the loopback address used during the test procedure of a host protocol stack.

The OSI and NetWare address prefixes have been defined to enable a host connected to one of these networks to communicate directly with a V6 host. The most widely used is the provider-based prefix as this format reflects the current structure of the Internet. A typical format of this type of address is shown in Figure 6.42(c). As we saw in Figure 6.19, the core backbone of the Internet consists of a number of very high bandwidth lines that interconnect the various continental backbones together. The routers that perform this function are owned by companies known as **top-level aggregators** (**TLAs**). Each TLA is allocated a large block of addresses by what is called a **registry**, the identity of which is in the field immediately following the 010 prefix. Examples include the North American registry, the European registry, the Asia and Pacific registry, and so on.

From their allocation, the TLAs allocate blocks of addresses to large Internet service providers and global enterprises. These are known as **next-level aggregators** (**NLAs**) and, in the context of Figure 6.19, operate at the continental backbone and national and regional levels. The various NLAs allocate both single addresses to individual subscribers and blocks of addresses to large business customers. The latter are known as **site-level aggregators** (**SLAs**) and include ISPs that operate at the regional and national levels. The 64-bit **interface ID** is divided locally into a fixed subnetid part and a hostid part. Typically, the latter is the 48-bit MAC address of the host and hence 16 bits are available for subnetting.

As we can deduce from Figure 6.42(c), the use of hierarchical addresses means that each router in the hierarchy can quickly determine whether a packet should be routed to a higher-level router or to another router at the same level simply by examining the appropriate prefix. Also, the routers at each level can route packets directly using the related prefix. The same overall description applies to the processing of geographic-based addresses.

As their names imply, the two types of **local-use addresses** are for local use only and have no meaning in the context of the global Internet. As we shall expand upon in Section 6.8.4, *link local-use addresses* are used in the autoconfiguration procedure followed by hosts to obtain an IPv6 address from a local router. The router only replies to the host on the same link the request was received and hence this type of packet is not forwarded beyond the router.

The *site local-use addresses* are used, for example, by organizations that are not currently connected to the Internet but wish to utilize the technology

associated with it. Normally, the 64-bit interface ID part is subdivided and used for routing purposes within the organization. In this way, should the organization wish to be connected to the Internet at a later date, it is only necessary to change the site local-use prefix with the allocated subscriber prefix.

Multicast addresses

The format of an IPv6 multicast address is shown in Figure 6.42(d). As we can see, following the multicast prefix are two additional fields that have been introduced to limit the geographic scope of the related multicast operation. The first is known as the *flags* field and is used to indicate whether the multicast is a permanently-assigned (reserved) address (0000) or a temporary (transient) address (0001). In the case of a permanently-assigned address, the least significant 8 bits of the *multicast group address* field identify the type of the multicast operation, while for a transient address, the full 112 bits identify the multicast group address.

In both cases, the 4-bit *scope* field defines the geographic scope of the multicast packet. The various alternatives are identified in the figure and mrouters use this field to determine whether the (multicast) packet should be forwarded further or discarded.

Anycast addresses

In addition to unicast and multicast addresses, a new address type known as an ***anycast group address*** has been defined. These are allocated from the unicast address space and are indistinguishable from a unicast address. With an anycast address, however, a group of hosts or routers can all have the same (anycast) address. A common requirement in a number of applications is for a host or router to send a packet to any one of a group of hosts or routers, all of which provide the same service. For example, a group of servers distributed around a network may all contain the same database. Hence in order to avoid all clients needing to know the unique address of its nearest server, all the servers can be members of the same anycast group and hence have the same address. In this way, when a client makes a request, it uses the assigned anycast address of the group and the request will automatically be received by its nearest server. Similarly, if a single network/internetwork has a number of gateway routers associated with it, they can all be allocated the same anycast address. As a result, the shortest-path routes from all other networks/internetworks will automatically use the gateway nearest to them.

In order to perform the routing function, although an anycast address has the same format as a unicast address, when an anycast address is assigned to a group of hosts or routers, it is necessary for each host/router to be explicitly informed – by network management for example – that it is a member of an anycast group. In addition, each is informed of the common part of the address prefixes which collectively identify the topological region in which all the hosts/routers reside. Within this region, all the routers then maintain a separate entry for each member of the group in its routing table.

Address representation

A different form of representation of IPv6 addresses has been defined. Instead of each 8-bit group being represented as a decimal number (with a dot/period between them), groups of 16 bits are used. Each 16-bit group is then represented in its hexadecimal form with a colon between each group. An example of an IPv6 address is:

FEDC:BA98:7654:3210:0000:0000:0000:0089

In addition, a number of abbreviations have been defined:

- One or more consecutive groups of all zeros can be replaced by a pair of colons.
- Leading zeros in a group can be omitted.

Hence the preceding address can also be written as:

FEDC:BA98:7654:3210::89

Also, for the two IPv4 embedded address types, the actual IPv4 address can remain in its dotted decimal form. Hence assuming a dotted decimal address of 15.10.0.6, the two embedded forms are:

:: 150.10.0.6 (IPv4 host address)
:: FFFF:150.10.0.6 (IPv4 tunnel address)

Example 6.6

Derive the hexadecimal form of representation of the following link-local multicast addresses:

(i) a permanently-assigned multicast group address of 67,

(ii) a transient multicast group address of 317.

Answer:

The formats of the two types of multicast addresses were shown in Figure 6.42(d). Hence:

The most significant 16 bits are FF02 = permanently-assigned, link-local
and FF12 = transient, link-local

(i) A permanently-assigned multicast group address of 67 = 0043 (hex)
Hence IPv6 address = FF02 :: 67

(ii) A transient multicast group address of 317 = 013D (hex)
Hence IPv6 address = FF12 :: 13D

6.8.3 Extension headers

As we have just indicated, if required, a number of extension headers can be added to the main header to convey additional information, either to the routers visited along the path followed or to the destination host. The six types of extension header currently defined are:

- **hop-by-hop options**: information for the routers visited along a path;
- **routing**: list of routers relating to source routing information;
- **fragment**: information to enable the destination to reassemble a fragmented message;
- **authentication**: information to enable the destination to verify the identity of the source;
- **encapsulating security payload**: information to enable the destination to decrypt the payload contents;
- **destination options**: optional information for use by the destination.

The two options headers can contain a variable number of option fields, each possibly of a variable length. For these, each option field is encoded using a **type-length-value** (**TLV**) format. The *type* and *length* are both single bytes and indicate the option type and its length (in bytes) respectively. The *value* is then found in the following number of bytes indicated by the *length*. The option type identifiers have been chosen so that the most significant two bits specify the action that must be taken if the type is not recognized. These are:

00 ignore this option and continue processing the other option fields in the header;

01 discard the complete packet;

10 discard the packet and return an ICMP error report to the source indicating a parameter problem and the option type not recognized;

11 same as for 10 except the ICMP report is only returned if the destination address is not a multicast address.

Note also that since the type of extension header is indicated in the preceding *next header* field, the related decoder that has been written to decode the contents of the header is invoked automatically as each header is processed. Some examples of each header type now follow.

Hop-by-hop options

This type of header contains information that must be examined by all the gateways and routers the packet visits along its route. The *next header* value for this is 0 and, if this header is present, it must follow the main header. Hence the *next header* field in the main header is set to 0. An example of its use is for a host to send a datagram that contains a payload of more than 64K bytes. This is

particularly useful, for example, when a host is transferring many very large files over a path that supports a maximum transmission unit (MTU) significantly greater than 64 kbytes. Datagrams that contain this header are known as **jumbograms** and the format of the header is shown in Figure 6.43(a).

As we can see, this type of header is of fixed length and comprises two 32-bit words (8 bytes). The *header extension length* field indicates the length of the header in multiples of 8 bytes, excluding the first 8 bytes. Hence in this case the field is 0. This option contains only one option field, the *jumbo payload length*. As we indicated earlier, this is encoded in the TLV format. The *type* for this option is 194 (11000010) and the *length* is 4 (bytes). The *value* in the *jumbo payload length* is the length of the packet in bytes, excluding the main header but including the 8 bytes in the extension header. This makes the *payload length* in the main header redundant and hence this is set to zero.

Routing

This plays a similar role to the (strict) *source routing* and *loose source routing* optional headers used in IPv4 datagrams. The *next header* value for this type of

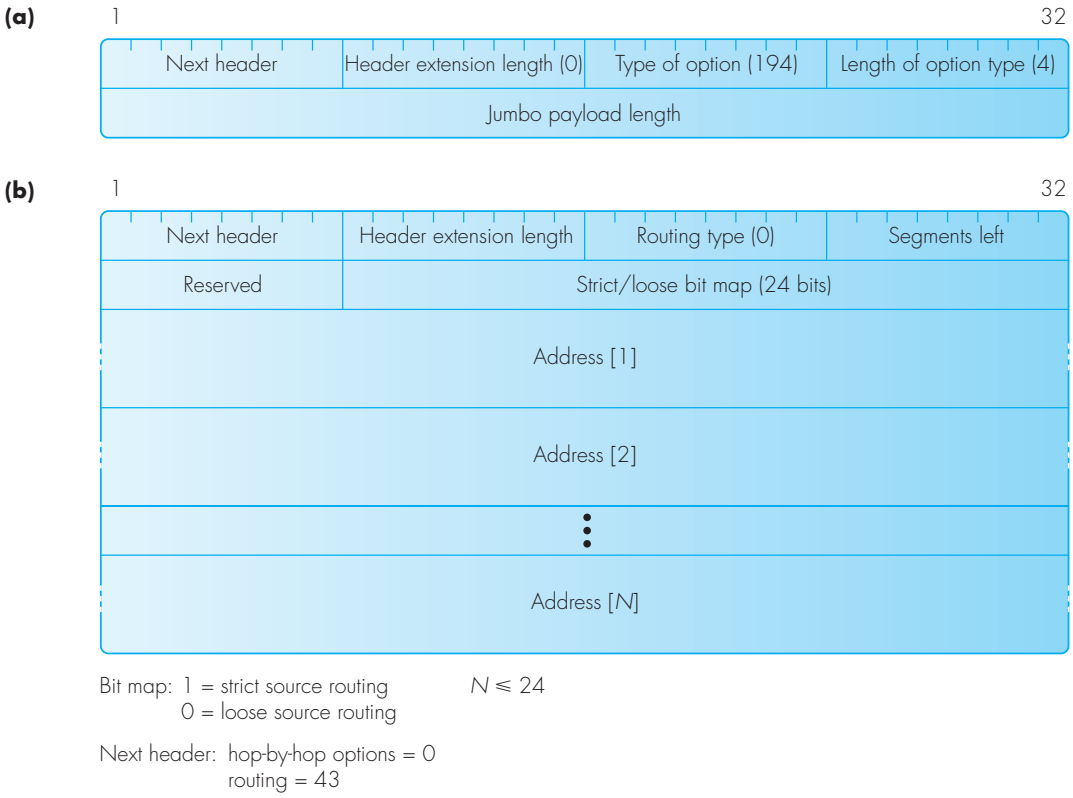


Figure 6.43 Extension header formats: (a) hop-by-hop options; (b) routing.

header is 43. Currently only one type of routing header has been defined, the format of which is shown in Figure 6.43(b).

The *header extension length* is the length of the header in multiples of 8 bytes, excluding the first 8 bytes. Hence, as we can deduce from the figure, this is equal to two times the number of (16-byte) addresses present in the header. This must be an even number less than or equal to 46. The *segments left* field is an index to the list of addresses that are present. It is initialized to 0 and is then incremented by the next router in the list as it is visited. The maximum therefore is 23.

The second 4-byte word contains a *reserved* byte followed by a 24-bit field called the *strict/loose bit map*. This contains one bit for each potential address present starting at the leftmost bit. If the related bit is a 1, then the address must be that of a directly attached (neighbor) router – that is, strict source routing. If the bit is a 0, then the address is that of a router with possibly several other routers in between – loose source routing. The latter is used, for example, when tunneling is required to forward the datagram. In the case of strict source routing, at each router the destination address in the main header is changed to that obtained from the list of addresses before the index is incremented. In the case of loose source routing, the destination address will be that of an attached neighbor which is on the shortest-path route to the specified address.

Example 6.7

A datagram is to be sent from a source host with an IPv6 address of A to a destination host with an IPv6 address of B via a path comprising three IPv6 routers. Assuming the addresses of the three routers are R1, R2, and R3 and strict source routing is to be used, (i) state what the contents of the initial values in the various fields in the extension header will be and (ii) list the contents of the source and destination address fields in the main header and the *segments left* field in the extension header as the datagram travels along the defined path.

Answer:

(i) Extension header initial contents:

Next header = Transport layer protocol
Header extension length = $2 \times 3 = 6$
Routing type = 0
Segments left = 0
Strict/loose bit map = 11100000 00000000 00000000
List of addresses = R1, R2, R3 (each of 16 bytes)

(ii) Contents of main header fields:

At source	SA = A	DA = R1	Segments left = 0
At R1	SA = A	DA = R2	Segments left = 1
At R2	SA = A	DA = R3	Segments left = 2

Fragment

This header is present if the original message submitted by the transport layer protocol exceeds the MTU of the path/route to be used. The *next header* value for a *fragment* extension header is 44. The fragmentation and reassembly procedures are similar to those used with IPv4 but, in the case of IPv6, the fragmentation procedure is carried out only in the source host and not by the routers/gateways along the path followed by the packet(s). As we saw in Figure 6.41(a), there is no don't fragment (D) bit in the IPv6 main header since, in order to speed up the processing/routing of packets, IPv6 routers do not support fragmentation. Hence, as we explained in Section 6.6.9, either the minimum MTU size of 576 bytes must be used or the *path MTU discovery* procedure is used to determine if the actual MTU size is greater than this. In either case, if the submitted message (including the transport protocol header) exceeds the chosen MTU (minus the 40 bytes for the IPv6 main header), then the message must be sent in multiple packets, each with a main header and a *fragment* extension header. The various fields and the format of each *fragment* extension header are shown in Figure 6.44(a). An example is shown in Figure 6.44(b).

Each packet contains a main header – plus, if required, a *hop-by-hop options* header and a *routing* header – followed by a *fragment* extension header and the fragment of the message being transmitted. Thus the maximum size of the payload (and hence message fragment) in each packet will be the MTU size being used minus the number of 8-byte fields required for the main header and any extension headers that are present. The *payload length* field in the main header of the first packet indicates the total number of bytes in the message being transmitted – including the IP header – plus the number of bytes that are required for the other extension headers that are being used. The *payload length* in the main header of the remaining packets indicates the number of bytes in the packet following the main header.

The various fields in each *fragment* header have similar functions to those used with IPv4. The *fragment offset* indicates the position of the first byte of the fragment contained within the packet relative to the start of the complete message being transmitted. Its value is in units of 8-bytes. The *M-bit* is the *more fragments bit*; it is a 1 if more fragments follow and a 0 if the packet contains the last fragment. Similarly, the value in the *identification* field is used by the destination host, together with the source address, to relate the data fragments contained within each packet to the same original message. Normally, the source uses a 32-bit counter (that is incremented by 1 for each new message transmitted) to keep track of the next value to use.

Authentication and encapsulating security payload

As we shall expand upon in Section 10.7.1, authentication and the related subject of encryption are both mechanisms that are used to enhance the security of a message during its transfer across a network. In the case of authentication, this enables the recipient of a message to validate that the

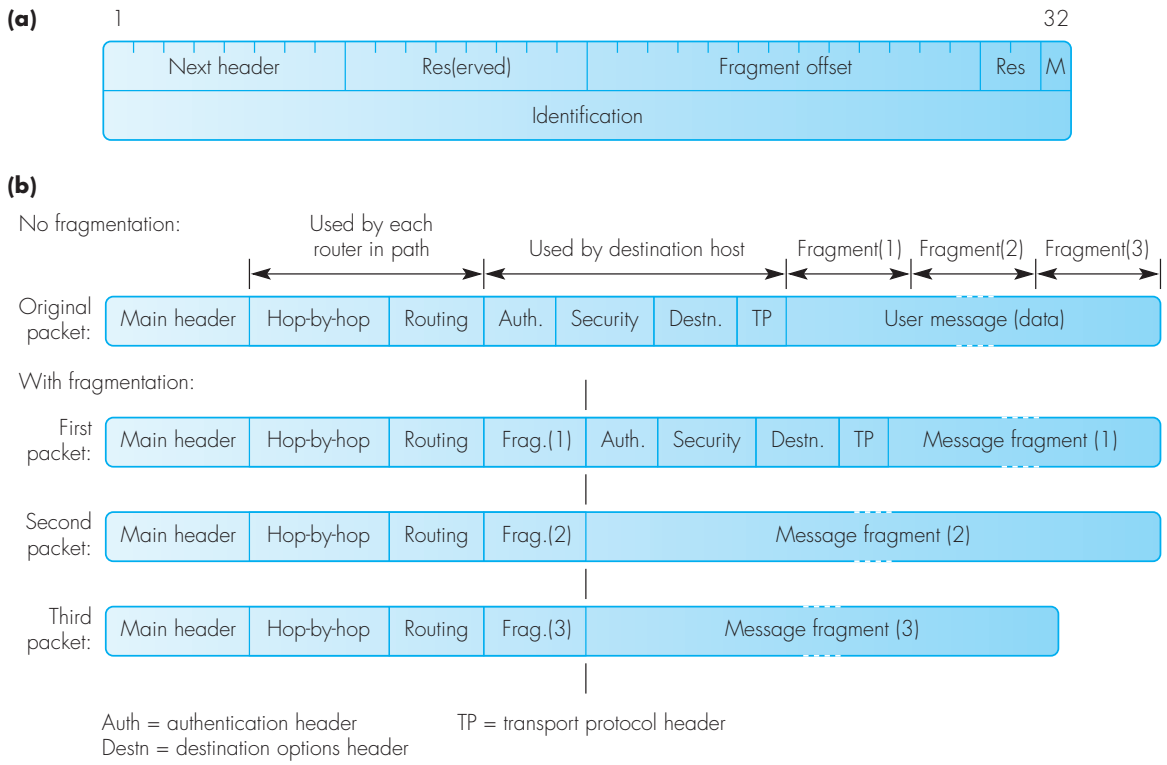


Figure 6.44 IPv6 fragmentation: (a) fragment header fields and format; (b) example.

message was indeed sent by the source address present in the packet/datagram header and not by an impostor. Encryption is concerned with ensuring that the contents of a message can only be read by – and hence have meaning to – the intended recipient. The *authentication* and *encapsulating* security payload (ESP) extension headers are present when both these features are being used at the network layer.

When using IPv6 authentication, prior to any information (packets) being exchanged, the two communicating hosts first use a secure algorithm to exchange secret keys. An example is the MD5 algorithm we describe later in Section 10.3. Then, for each direction of flow, the appropriate key is used to compute a checksum on the contents of the entire datagram/packet. The computed checksum is then carried in the authentication header of the packet. The same computation is repeated at the destination host, and only if the computed checksum is the same as that carried in the authentication header is it acknowledged that the packet originated from the source host address indicated in the main header and also that the packet contents have

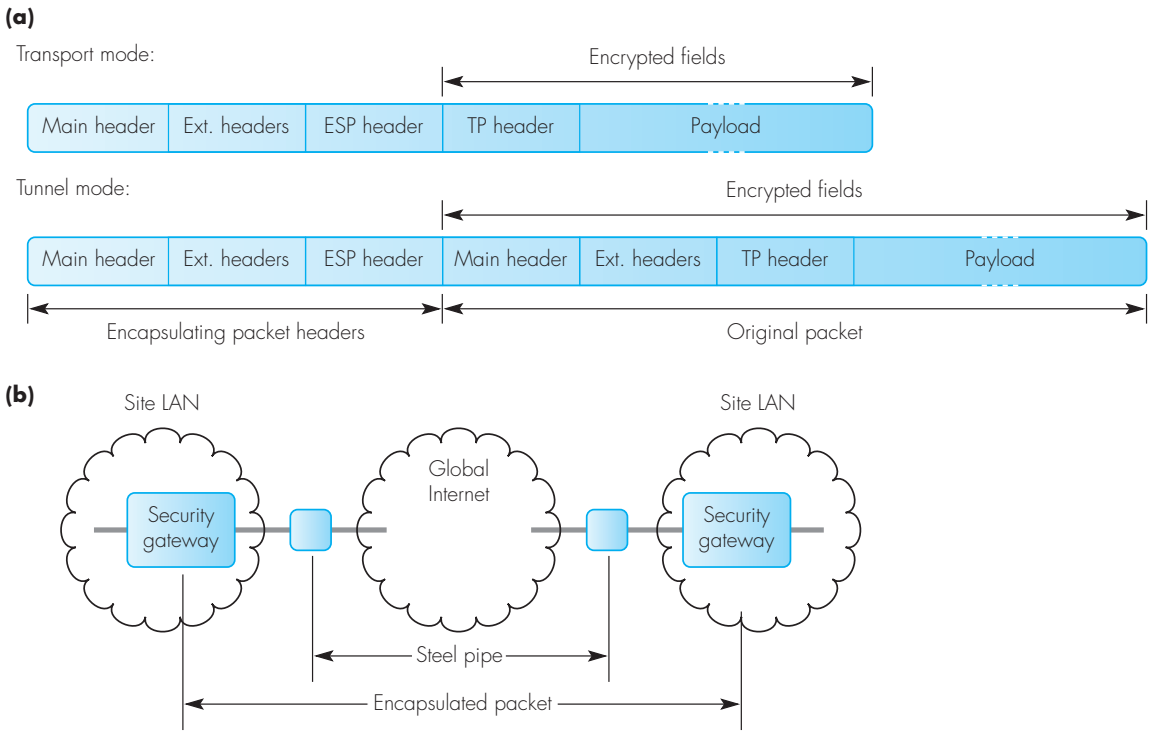


Figure 6.45 Encapsulating security payload: (a) transport and tunnel modes; (b) example application of tunnel mode.

not been modified during its transfer over the network. We discuss the subject of authentication further in Section 10.4.

The encryption algorithm used with the ESP is also based on the use of an agreed secret key. An example is the DES algorithm we describe later in Section 10.2.3. The agreed key is used to encrypt either the transport protocol header and payload parts of each packet or, in some instances, the entire packet including the main header and any extension headers present. In the case of the latter, the encrypted packet is then carried in a second packet containing a completely different main and, if necessary, extension headers. The first is known as **transport mode encryption** and the second **tunnel mode encryption**. Figure 6.45(a) illustrates the principle of operation of both modes.

As we can deduce from the figure, the overheads associated with the tunnel mode are significantly higher than those of the transport mode. The extra security obtained with the tunnel mode is that the information in the main and extension headers of the original packet cannot be interpreted by a person passively monitoring the transmissions on a line. An example use of this mode is in

multisite enterprise networks that use the (public) Internet to transfer packets from one site to another. The general scheme is shown in Figure 6.45(b).

As we will show in Figure 9.4(a) and explain in the accompanying text, associated with each site is a security gateway through which all packet transfers to and from the site take place. Hence to ensure the header information (especially the routing header) of packets is not visible during the transfer of the packet across the Internet, the total packet is encrypted and inserted into a second packet by the IP in the security gateway with the IPv6 address of the two communicating gateways in the source and destination address fields of the main header. The path through the Internet connecting the two gateways is referred to as a **steel pipe**.

Destination options

These are used to convey information that is examined only by the destination host. As we indicated earlier, one of the ways of encoding options is to use the type-length-value format. Hence in order to ensure a header that uses this format comprises a multiple of 8 bytes, two *destination options* have been defined. These are known as Pad1 and Pad*N*, the first to insert one byte of padding and the second two or more bytes of padding. Currently these are the only two options defined.

6.8.4 Autoconfiguration

As we described earlier in Section 6.1, the allocation of the IP addresses for a new network involves a central authority to allocate a new netid – the ICANN – and a local network administrator to manage the allocation of hostids to each attached host/end system. Thus, the allocation, installation and administration of IPv4 addresses can entail considerable effort and expenditure. To alleviate this, IPv6 supports an autoconfiguration facility that enables a host to obtain an IP address dynamically via the network and, in the case of mobile hosts, use it just for the duration of the call/session.

Two types of autoconfiguration are supported. The first involves the host communicating with a local (site) router using a simple (stateless) request-response protocol. The second involves the host communicating with a site (or enterprise) address server using an application protocol known as the **dynamic host configuration protocol** (**DHCP**). The first is suitable for small networks that operate in the broadcast mode (such as an Ethernet LAN) and the second for larger networks in which the allocation of IP addresses needs to be managed.

With the first method, a simple protocol known as **neighbor discovery** (**ND**) is used. As we show in Figure 6.46, this involves the host broadcasting a *router solicitation* packet/message on the subnet/network and the router responding with a *router advertisement* message. Both messages are ICMPv6 messages and hence are carried in an IPv6 packet. The latter is then broadcast over the LAN in a standard frame.

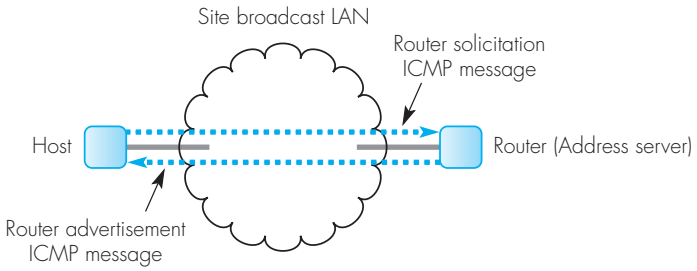


Figure 6.46 Neighbor discovery protocol messages.

The main header of the IPv6 packet containing the router solicitation message has an IPv6 source address created by the host. This is made up of the (standard) link-local address prefix and the 48-bit MAC address of the host's LAN interface. As we indicated earlier, with IPv6 a number of permanent multicast group addresses have been defined including an all-routers group address. Hence the destination address in the packet main header is set to this and, since the packet is broadcast over the LAN, it is received by the ICMP in all the routers that are attached to the LAN.

A single router is selected to process this type of ICMP message and this responds to a router solicitation message with a route advertisement message containing the Internet-wide address prefix for the subnet/network. On receipt of this, the ICMP in the host proceeds to create its own IPv6 address by adding its 48-bit MAC address to the prefix. As we can deduce from this, in relation to IPv4, this is equivalent to the router providing the netid of the site and the host using its own MAC address as the hostid. Also, the same procedure can be used by mobile hosts.

With the second method, the host requests an IPv6 address from the site address server using the DHCP. The **DHCP address server** first validates the request and then allocates an address from the managed list of addresses the server contains. Alternatively if a site does not have its own address server – for example if the site LAN is part of a larger enterprise network — then a designated router acts as a **DHCP relay agent** to forward the request to the DHCP address server.

6.9 IPv6/IPv4 interoperability

The widespread deployment of IPv4 equipment means that the introduction of IPv6 is being carried out in an incremental way. Hence a substantial amount of the ongoing standardization effort associated with IPv6 is concerned with the interoperability of the newer IPv6 equipment with existing IPv4 equipment. Normally, when a new network/internetwork is created, it is based on the IPv6 protocol and, in the context of the existing

(IPv4) Internet, it is referred to as an **IPv6 island**. It is necessary to provide a means of interworking between the two types of network at both the address level and the protocol level. In this section, we identify a number of situations where interoperability is required and describe a selection of the techniques that are used to achieve this.

6.9.1 Dual protocols

Dual stacks are already widely used in networks that use dissimilar protocol stacks, for example a site server that supports IPX on one port and IP on a second port. In a similar way, dual protocols can be used to support both IPv4 and IPv6 concurrently. An example is shown in Figure 6.47.

In this example, the site has a mix of hosts, some that use IPv4 and others IPv6. In order to be able to respond to requests originating from both types of host, the server machine has both an IPv4 and an IPv6 protocol at the network layer. The value in the version field of the datagram header is then used by the link layer protocol to pass a datagram to the appropriate IP. In this way the upper layer protocols are unaware of the type of IP being used at the network layer.

6.9.2 Dual stacks and tunneling

A common requirement is to interconnect two IPv6 islands (networks/internetworks) through an intermediate IPv4 network/internetwork. To achieve this, the gateway/router that connects each IPv6 island to the IPv4 network

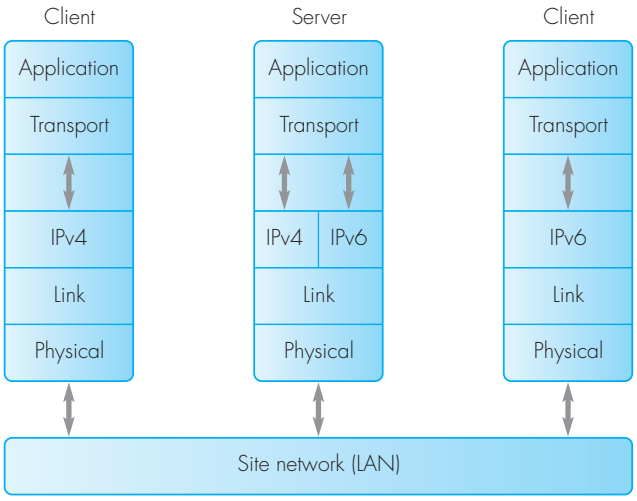


Figure 6.47 IPv6/IPv4 interoperability using dual (IPv6/IPv4) protocols.

must have dual stacks, one that supports IPv6 and the other IPv4. The IPv6 packets are then transferred over the IPv4 network using tunneling. The general approach is illustrated in Figure 6.48(a) and the protocols involved in Figure 6.48(b).

As we showed earlier in Figure 6.16 and explained in the accompanying text, tunneling is used to transfer a packet relating to one type of network layer protocol across a network that uses a different type of network layer protocol. Hence in the example shown in Figure 6.48, each IPv6 packet is transferred from one (IPv6/IPv4) edge gateway to the other edge gateway within an IPv4 packet. As we show in the figure, in order to do this, the two

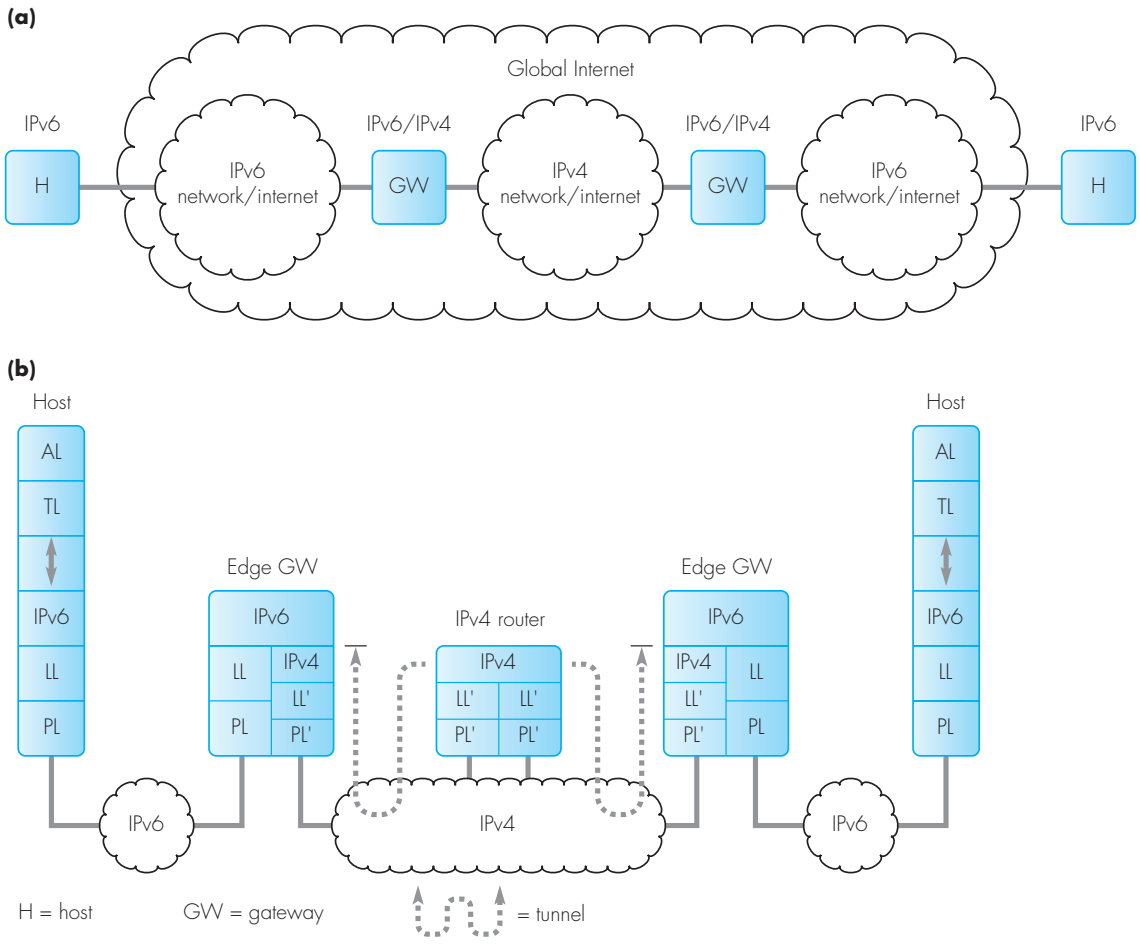


Figure 6.48 IPv6/IPv4 interoperability using dual stacks and tunneling: (a) schematic; (b) protocols.

edge gateways contain dual stacks each of which has a related IPv6/IPv4 address associated with it. Normally, entered by network management, the routing table entry for the remote destination IPv6 host is the IPv4 address of the remote edge gateway.

The IPv6 in each gateway, on determining from its routing table that an (IPv6) packet should be forwarded to a remote IPv6 network via an IPv4 tunnel, passes the packet to the IPv4 protocol together with the IPv4 address of the remote gateway. The IPv4 protocol first encapsulates the IPv6 packet in an IPv4 datagram/packet with the IPv4 address of the remote gateway in the destination address field. It then uses this address to obtain the (IPv4) address of the next-hop router from its own (IPv4) routing table and proceeds to forward the packet over the IPv4 network/internetwork. On receipt of the packet, the IPv4 in the remote gateway, on detecting from its own routing table that it is a tunneled packet, strips off the IPv4 header and passes the payload – containing the original IPv6 packet – to the IPv6 layer. The latter then forwards the packet to the destination host identified in the packet header in the normal way.

6.9.3 Translators

A third type of interoperability requirement is for a host attached to an IPv6 network – and hence having an IPv6 address and using the IPv6 protocol – to communicate with a host that is attached to an IPv4 network – hence having an IPv4 address and using the IPv4 protocol. In this case, both the addresses and the packet formats are different and so a translation operation must be carried out by any intermediate routers/gateways. As we show in Figure 6.49, the translations can be performed at either the network layer – part (a) – or the application layer – part (b).

Using the first approach, on receipt of an IPv6/IPv4 packet, this is converted into a semantically equivalent IPv4 /IPv6 packet. This involves a **network address translator** (**NAT**) and a **protocol translator** (**PT**). The intermediate gateway is then known as a **NAT-PT gateway**. As we explained earlier in Section 6.8.2, an IPv4 address can be embedded in an IPv6 address. Hence to send a datagram/packet from a host with an IPv6 address to a host with an IPv4 address, the NAT in the gateway can readily obtain the destination IPv4 address from the destination address in the main header of the IPv6 packet. The issue is what the source address in the IPv4 packet header should be.

A proposed solution is for the NAT to be allocated a block of hostids for the destination IPv4 network. These, together with the netid of the destination network, then form a block of unique IPv4 addresses. For each new call/session, the NAT allocates an unused IPv4 address from this block for the duration of the call/session. It then makes an entry in a table containing the IPv6 address of the V6 host and its equivalent (temporary) IPv4 address. The NAT translates between one address and the other as the

packet is relayed. A timeout is applied to the use of such addresses and, if no packets are received within the timeout interval, the address is transferred back to the free address pool.

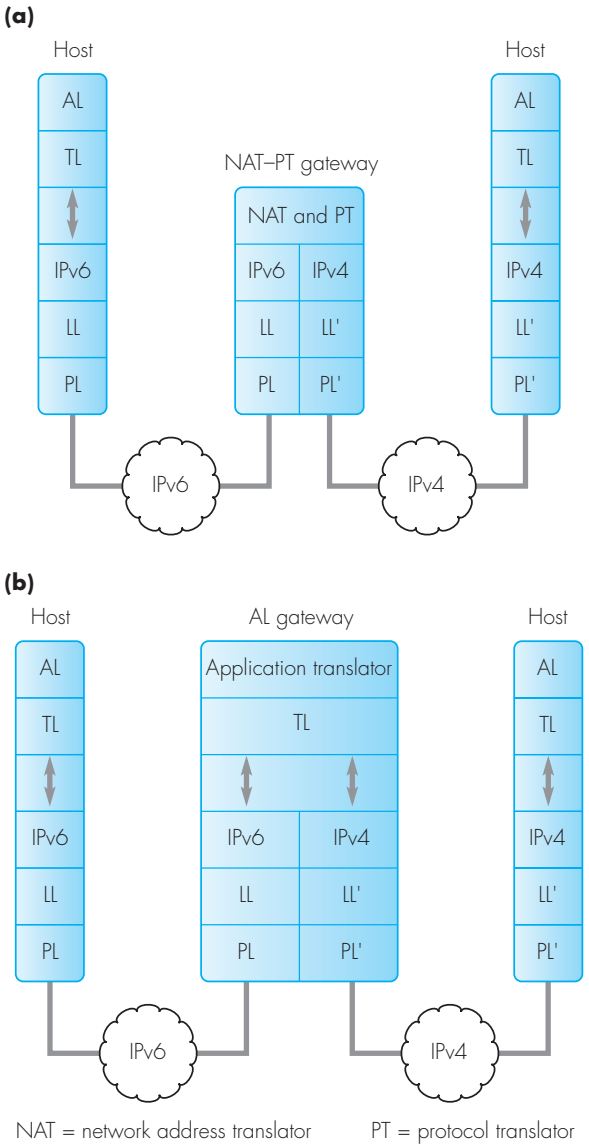


Figure 6.49 IPv6/IPv4 interoperability using translators: (a) network level; (b) application level.

The protocol translation operation is concerned with translating the remaining fields in the packet header and, in the case of ICMP messages, converting ICMPv4 messages into and from ICMPv6 messages. As we indicated in Section 6.8.1, most of the fields in the IPv6 main header have the same meaning as those in the IPv4 header and hence their translation is relatively straightforward. In general, however, there is no attempt to translate the fields in the options part. Similarly, since ICMPv6 messages have different *type* fields, the main translation performed is limited to changing this field. For example, the two ICMPv4 query messages have *type* values of 8 and 0 and the corresponding ICMPv6 messages are 128 and 129.

The use of a NAT-PT gateway works providing the packet payload does not contain any network addresses. Although this is the case for most application protocols, a small number do. The FTP application protocol, for example, often has IP addresses embedded within its protocol messages. In such cases, therefore, the translation operation must be carried out at the application layer. The associated gateway is then known as an **application level gateway** (**ALG**). This requires a separate translation program for each application protocol. Normally, therefore, most translations are performed at the network layer – using a NAT and a PT – and only the translations relating to application protocols such as FTP are carried out in the application layer.

Summary

A summary of the topics discussed in this chapter is given in Figure 6.50.

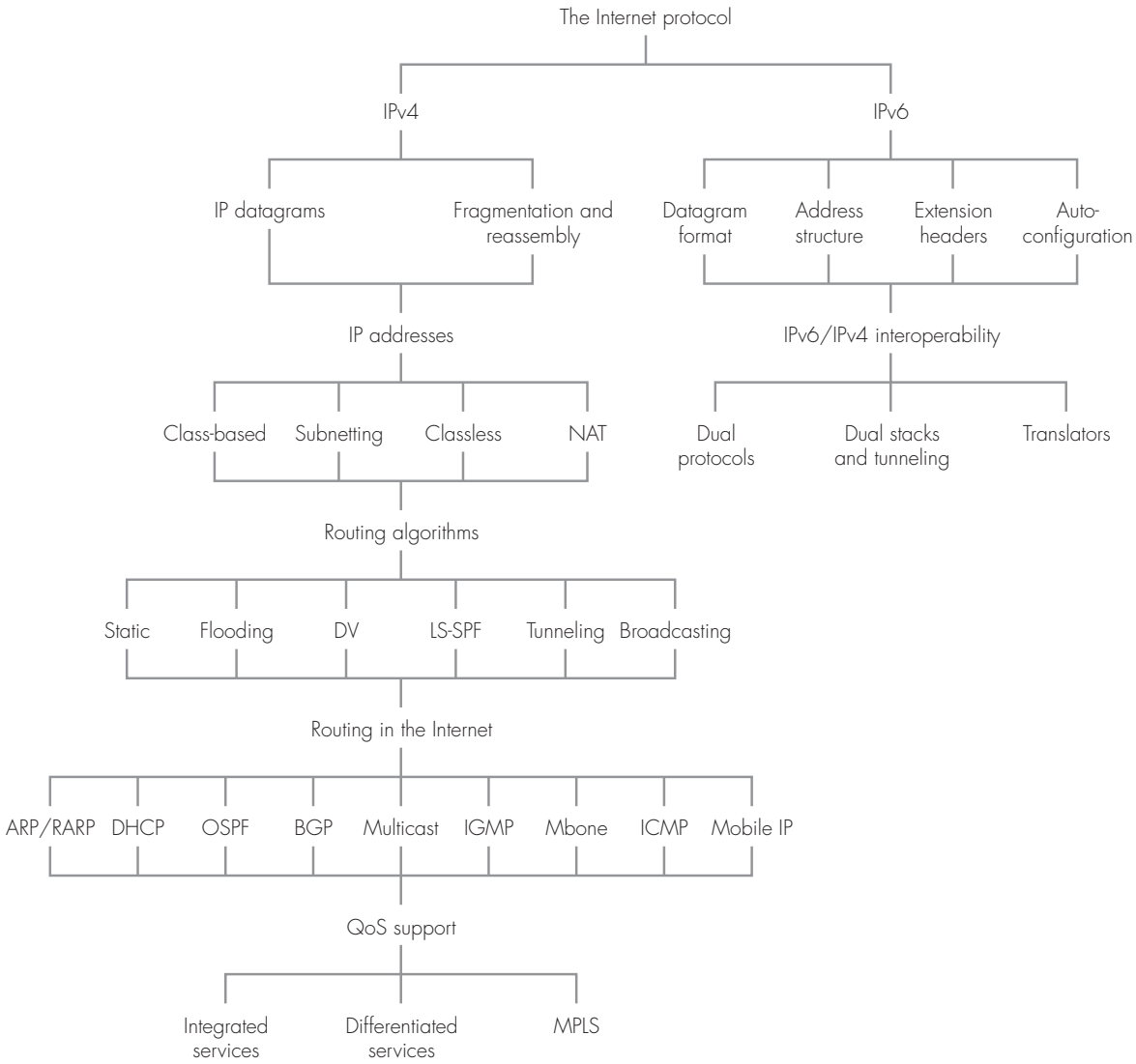


Figure 6.50 Internet protocol summary.

Exercises

Section 6.1

6.1 With the aid of the network schematic shown in Figure 6.1, explain briefly the role of the following network components and protocols:
(i) access network gateway,
(ii) routing gateway,
(iii) internet protocol (IP),
(iv) datagram/packet.

6.2 With the aid of the protocol stack shown in Figure 6.2, explain briefly the meaning of the term "adjunct protocol" and how the IP in the destination host determines to which protocol the contents/payload of a received IP datagram should be passed.

Section 6.2

6.3 In relation to the IP datagram/packet format shown in Figure 6.3, explain the role of the following header fields:
(i) IHL,
(ii) TOS,
(iii) Total length and Identification,
(iv) flag bits,
(v) Fragment offset,
(vi) Time-to-live,
(vii) Protocol,
(viii) Header checksum,
(ix) Options.

Section 6.3

6.4 Assume a message block of 7000 bytes is to be transferred from one host to another as shown in the example in Figure 6.4. In this instance, however, assume the token ring LAN has an MTU of 3000 bytes. Compute the header fields in each IP packet shown in the figure as it flows
(i) over the token ring LAN,
(ii) over the Ethernet LAN.

6.5 State why fragmentation is avoided whenever possible and the steps followed within each host IP to achieve this.

Section 6.4

6.6 List the four types of IP address schemes that are in use and give a brief summary of the use of each scheme.

6.7 Explain the meaning of the term "IP address class" and why these classes were created. Hence, with the aid of the three (unicast) address classes identified in Figure 6.5, identify a particular application for each class.

6.8 State the meaning of the following addresses:
(i) an address with a netid of all 0s,
(ii) an address with a netid of all 1s,
(iii) an address with a hostid of all 0s,
(iv) an address of all 1s.

6.9 Explain the meaning of the term "dotted decimal". Hence derive the netid and hostid for the following IP addresses expressed in dotted decimal notation:
(i) 13.0.0.15,
(ii) 132.128.0.148,
(iii) 220.0.0.0,
(iv) 128.0.0.0,
(v) 224.0.0.1.

6.10 With the aid of the example in Figure 6.6, explain why subnetting was introduced. Hence state the meaning of a subnet router and an address mask.

6.11 A site with a netid of 127.0 uses 20 subnet routers. Suggest a suitable address mask for the site which allows for a degree of expansion in the future. Give an example of a host IP address at this site.

6.12 With the aid of Example 6.3, explain the classless inter-domain routing (CIDR) scheme.

6.13 With the aid of Example 6.4, explain how, with CIDR, multiple address matches are possible with a network that has been allocated a large block of addresses. State which of these matches is chosen and why.

6.14 Explain why the network address translation (NAT) scheme is now used with most new access networks.

6.15 In relation to the NAT scheme shown in Figure 6.7(a), determine the number of host addresses/interfaces that have been declared for private use.

6.16 In relation to the outline operation of NAT shown in Figure 6.7(b), explain:
(i) how the four IP addresses are used and the role of the NAT table,
(ii) how the source and destination addresses in the related IP and TCP fields are derived.

Section 6.5

6.17 State the meaning of the following terms relating to the routing of packets over an internet:
(i) line cost,
(ii) path cost,
(iii) hopcount,
(iv) routing metric,
(v) shortest path.

6.18 With the aid of the routing table entries shown in Figure 6.9, explain the meaning of the terms:
(i) static routing tables,
(ii) next-hop routing,
(iii) optionality principle,
(iv) alternative paths.

6.19 With the aid of the broadcast diagram shown in Figure 6.10, explain:
(i) why the broadcast following the route via R2 is assumed to arrive first
(ii) how duplicate copies of a packet are determined by R3
(iii) how the number of copies of the packet produced is limited
(iv) why flooding is an example of an adaptive/dynamic routing algorithm.

6.20 In relation to the distance vector algorithm, with the aid of the example shown in Figure 6.11, explain:
(i) the meaning of the term "connectivity/adjacency table" and how the table's contents are obtained,
(ii) how the final routing table entries for R3 are built up,
(iii) how a packet from a host attached to netid3 is routed to a host attached to netid1,
(iv) the limitations of the algorithm including how looping may arise.

6.21 Assuming the connectivity/adjacency tables given in Figure 6.12(a), show how the overall network topology is built up by router R3 using the link state algorithm. Hence derive the contents of the netid location table for R3.

6.22 Assuming the initial network topology shown in Figure 6.13(a), use the Dijkstra algorithm to derive the shortest paths from R3 to each other router. State the meaning of the terms "tentative" and "permanent" relating to the algorithm and the implications of alternative paths/routers.

6.23 Using the set of link state, routing, and connectivity tables for R1, R2 and R3 in Figure 6.15, explain how a packet received by R3 from G3 with a destination netid of 1 is routed using hop-by-hop routing.

6.24 Explain how a packet received by R3 from G3 with a destination netid of 1 is routed using source routing. Include how the routing tables you use are derived.

6.25 In relation to the link-state algorithm, explain why each link-state message contains a sequence number and a timeout value. How are these used?

6.26 Explain the term "tunneling" and when it is used. Hence with the aid of the schematic diagram shown in Figure 6.16, explain how the host on the left of the diagram sends an IP datagram/packet to a host attached to the Internet. Include in your explanation the role of the two multiprotocol routers.

State an application of tunneling IP packets over an IP network.

6.27 What are the aims of both the reverse path forwarding algorithm and the spanning tree broadcast algorithm?

6.28 Use the final routing tables and broadcast sequence relating to the reverse path forwarding algorithm shown in Figure 6.17 to explain why only the (broadcast) packet received by SR6 from SR3 is broadcast at the fourth stage. What is the number of duplicate broadcasts that occur?

6.29 Assuming the network topology shown in Figure 6.18(a) and that SR3 is an access gateway, determine the spanning tree derived by each subnet router. Use this to derive the broadcast sequence.

Section 6.6

6.30 With the aid of the generalized Internet architecture shown in Figure 6.19, give an example of the type of network that is used at each tier in the hierarchy and the routing method that is used with it.

6.31 Define the terms "IP address", "MAC address", and "hardware/physical address". Also explain the terms "address-pair" and "ARP cache".

6.32 In relation to the simple network topology shown in Figure 6.20, explain why:
(i) on receipt of an ARP request message, each host retains a copy of the IP/MAC address-pair of the source host in its ARP cache,
(ii) on receipt of an ARP reply message, the ARP in the source host makes an entry of the IP/MAC address-pair in its own cache,
(iii) the Lan port of the gateway keeps a copy of the IP/MAC address-pair from each ARP request and reply message that it receives.

6.33 Explain the role of a proxy ARP. Hence explain how an IP packet sent by a host at one site is routed to a host at a different site. Also explain how the reply packet is returned to the host that sent the first packet.

6.34 Explain how the reverse ARP is used to enable a diskless host to determine its own IP address from its local server.

6.35 With the aid of the two frame formats shown in Figure 6.21, explain:
(i) how the MAC sublayer in the receiver determines whether a received frame is in the Ethernet format or IEEE802.3,
(ii) the number of pad bytes required with each frame type.

6.36 State the role of the dynamic host configuration protocol (DHCP).

6.37 Use the example network topology shown in Figure 6.22(a) to describe the DHCP message exchange sequence to obtain an IP address.

6.38 In relation to the simplified Internet structure shown in Figure 6.23, explain the function of the four types of router.

6.39 With the aid of the simplified autonomous system (AS) topology shown in Figure 6.24(a), describe the operation of the OSPF algorithm. First describe how the directed graph is derived and then the derivation of the SPF tree for R7.

6.40 List the message types used with OSPF and explain their function when routing within an AS.

6.41 List the four message types that are used in the border gateway protocol (BGP) and explain their function.

6.42 Given the example backbone topology shown in Figure 6.25, give an example of an update message that changes the route followed between a pair of boundary routers.

6.43 In relation to multicasting over a LAN, describe how ICANN controls the allocation of

multicast addresses. Also explain how the 48-bit MAC address and 28-bit IP address of a host are derived from the allocated address. Hence with the aid of the schematic diagram shown in Figure 6.27(b), describe how a host joins a multicast session that is taking place over the LAN. Include the role of the multicast address table and group address table held by each member of the group.

6.44 What is the meaning of the term "multicast router"? Outline the sequence of steps that are followed to route an IP packet with a multicast address over the Internet.

6.45 Assume the same topology, multicast address table contents, routing table contents, and routing table entries as shown in Figure 6.28. Assuming the DVMRP, explain how a packet arriving from one of its local networks with a multicast address of C is routed by MR3 to all the other MRs that have an interest in this packet.

6.46 Repeat Exercise 6.45 but this time using the MOSPF routing protocol and the spanning tree shown in Figure 6.29(b).

6.47 What is the role of the IGMP protocol?

With the aid of the example shown in Figure 6.30, explain how a host that is attached to a local network/subnet of an MR joins a multicast session. Include in your explanation the table entries retained by both the host and the MR and how multicast packets relating to the session are then routed to the host.

6.48 With the aid of the example shown in Figure 6.30, explain the procedure followed when a host that is attached to a local network/subnet of an MR leaves a multicast session.

6.49 The multicast backbone (M-bone) network shown in Figures 6.28 and 6.29 comprised a set of multicast routers interconnected by single links. In practice, these are logical links since each may comprise multiple interconnected routers that do not take part in multicast routing. Explain how a multicast packet is sent from one mrouter to another using IP tunneling.

6.50 Explain briefly the role of the ICMP protocol and the different procedures associated with it. Hence explain how the path MTU discovery procedure is used to determine the MTU of a path/route prior to sending any datagrams.

6.51 In the simplified network architecture shown in Figure 6.32, explain the use of the following terms:

(i) home agent,
(ii) foreign agent.

Discuss the issues to be resolved when Host A wants to communicate with Host B.

6.52 In relation to the example shown in Figure 6.33, use a sequence diagram to illustrate the exchange of mobile IP messages in order to register a mobile host (Host B) with its HA and FA. List the main addresses held by both the HA and the FA after the registration procedure is complete.

6.53 Using the example shown in Figure 6.34, describe the indirect routing method used to route packets between Host A and Host B once Host B has been registered with the HA and FA.

Section 6.7

6.54 Discuss the reasons why improved levels of QoS support are now being used within the Internet.

6.55 Describe the role and principle of operation of the following control mechanisms used within Internet routers:

(i) token bucket filter,
(ii) weighted fair queuing,
(iii) random early detection.

6.56 Define the three different classes of service used with the IntServ scheme. With the aid of the network topology shown in Figure 6.35(a) describe the operation of the resource reservation protocol (RSVP). Include in your

description the meaning/role of the following:
(i) path, reserve, and path-tear messages,
(ii) path-state table,
(iii) cleanup timer,
(iv) soft-state.

6.57 Define the usage of the type of service (ToS) field in each packet header with the DiffServ scheme including the meaning of the term "DS packet codepoint".

6.58 With the aid of the general architecture shown in Figure 6.35(b), describe the operation of the DiffServ scheme. Include in your description the meaning/role of the following components of an ingress router:
(i) behavior aggregate,
(ii) traffic meter module,
(iii) MF classifier,
(iv) marker module,
(v) shaper/dropper.

6.59 Use the schematic diagram of a router in Figure 6.36 to explain the packet forwarding procedure in a conventional router.

6.60 Explain the terms traffic engineering, class-based queuing, shaping and grooming in an MPLS network.

6.61 In relation to the MPLS network architecture shown in Figure 6.37, explain where and why the various QoS mechanisms are located.

6.62 Using the example topology shown in Figure 6.38, insert two further sets of table entries to illustrate the label switching procedure.

6.63 Use the schematic diagram of the area border router/label edge router shown in Figure 6.39 to explain the MPLS forwarding procedure. Include in your explanation the role of the packet classifier, the LSP table, the output queue interfaces and the associated scheduling rules.

6.64 Explain how alternative routes to those computed using OSPF are used with the constraint-routed label distribution protocol (CR-LDP). Describe how the alternative routing tables are downloaded after the CR analysis.

Also explain the meaning/role of the following components of a core router:
(i) BA classifier,
(ii) per-hop behavior,
(iii) expedited forwarding,
(iv) assured forwarding.

Section 6.8

6.65 Discuss the reasons behind the definition of IP version 6, IPv6/IPng, including the main new features associated with it.

6.66 With the aid of the frame format shown in Figure 6.41(a), explain the role of the following fields in the IPv6 packet header:
(i) traffic class,
(ii) flow label,
(iii) payload length (and how this differs from the total length in an IPv4 packet header),
(iv) next header,
(v) hop limit,
(vi) source and destination addresses.

6.67 In relation to IPv6 addresses, with the aid of the prefix formats shown in Figures 6.42(a) and (b), explain the meaning/use of:
(i) address aggregation,
(ii) prefix formats,
(iii) embedded IPv4 addresses.

6.68 With the aid of the frame format shown in Figure 6.42(c), explain the meaning/use of the following IPv6 fields:
(i) registry,
(ii) top-level aggregators,
(iii) next-level agrregators,
(iv) site-level aggregators,
(v) interface ID.
Comment on the implications of adopting a hierarchical address structure.

6.69 With the aid of examples, explain the use of a link local-use address and a site local-use address.

6.70 Explain the format and use of
(i) a multicast address,
(ii) an anycast address.

6.71 With the aid of examples, show how an IPv6 address can be represented:
(i) in hexadecimal form,
(ii) with leading zeros removed,
(iii) when it contains an IPv4 embedded address.

6.72 Explain the role of the extension headers that may be present in an IPv6 packet. List the six types of extension header and state their use. Also, with the aid of examples, state the position and order of the extension headers in relation to the main header.

6.73 The fields in an options extension header are encoded using a type-length-value format. Use the hop-by-hop options header as an example to explain this format.

6.74 In relation to the routing extension header, explain:
(i) the difference between strict and loose source routing, and the associated bit map,
(ii) the use of the segments left field.

6.75 In relation to the packet formats shown in Figure 6.44, explain:
(i) the meaning and use of the identification field and the M-bits in each extension header,
(ii) why the hop-by-hop and routing headers are present in each fragment packet.

6.76 In relation to the encapsulating security payload header, with the aid of diagrams, explain:
(i) the difference between transport mode and tunnel mode encryption,
(ii) the meaning and use of the term "steel pipe".

6.77 State the aim of the autoconfiguration procedure used with IPv6 and the application domain of:
(i) the neighbor discovery (ND) protocol,
(ii) the dynamic host configuration protocol (DHCP).

6.78 With the aid of Figure 6.46, explain the operation of the ND protocol. Include the role of the router solicitation and router advertisement messages and how a host creates its own IP address.

6.79 Explain how an IPv6 address is obtained:
(i) using a DHCP address server,
(ii) using a DHCP relay agent.

Section 6.9

6.80 With the aid of Figure 6.47, explain how a LAN server can respond to requests from both an IPv4 and an IPv6 client using dual protocols.

6.81 With the aid of Figure 6.48, explain how two hosts, each of which is attached to a different IPv6 network, communicate with each other if the two IPv6 networks are interconnected using an IPv4 network. Include the addresses that are used in each message transfer.

6.82 State the meaning of the terms "network address translation" (NAT) and "protocol translation" (PT). Hence, with the aid of the schematic diagram shown in Figure 6.49(a), explain the role and operation of a NAT-PT gateway. Include what the source address in each IPv6 packet should be.

6.83 Identify when the use of a NAT-PT gateway is not practical. Hence, with the aid of the schematic diagram shown in Figure 6.49(b), explain the role of an application level gateway.

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.

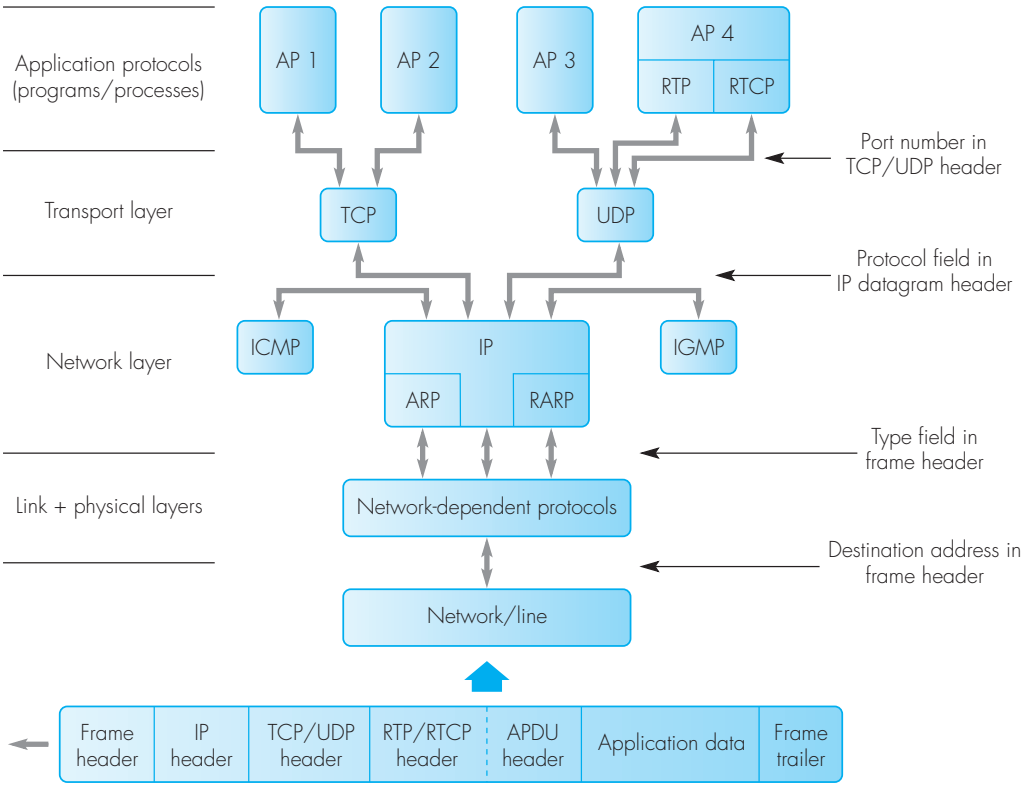


Figure 7.1 TCP/IP protocol suite and interlayer address selectors.

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

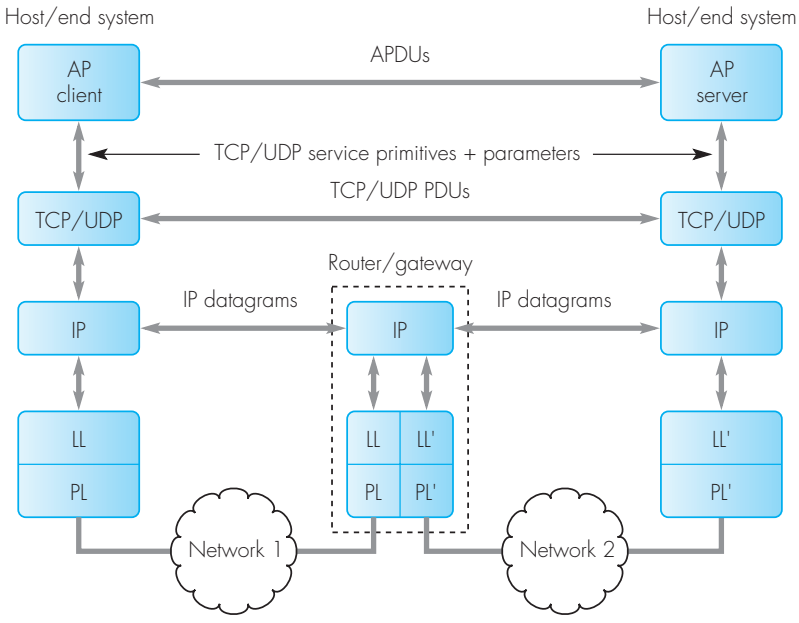


Figure 7.2 TCP/IP protocol suite interlayer communications.

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

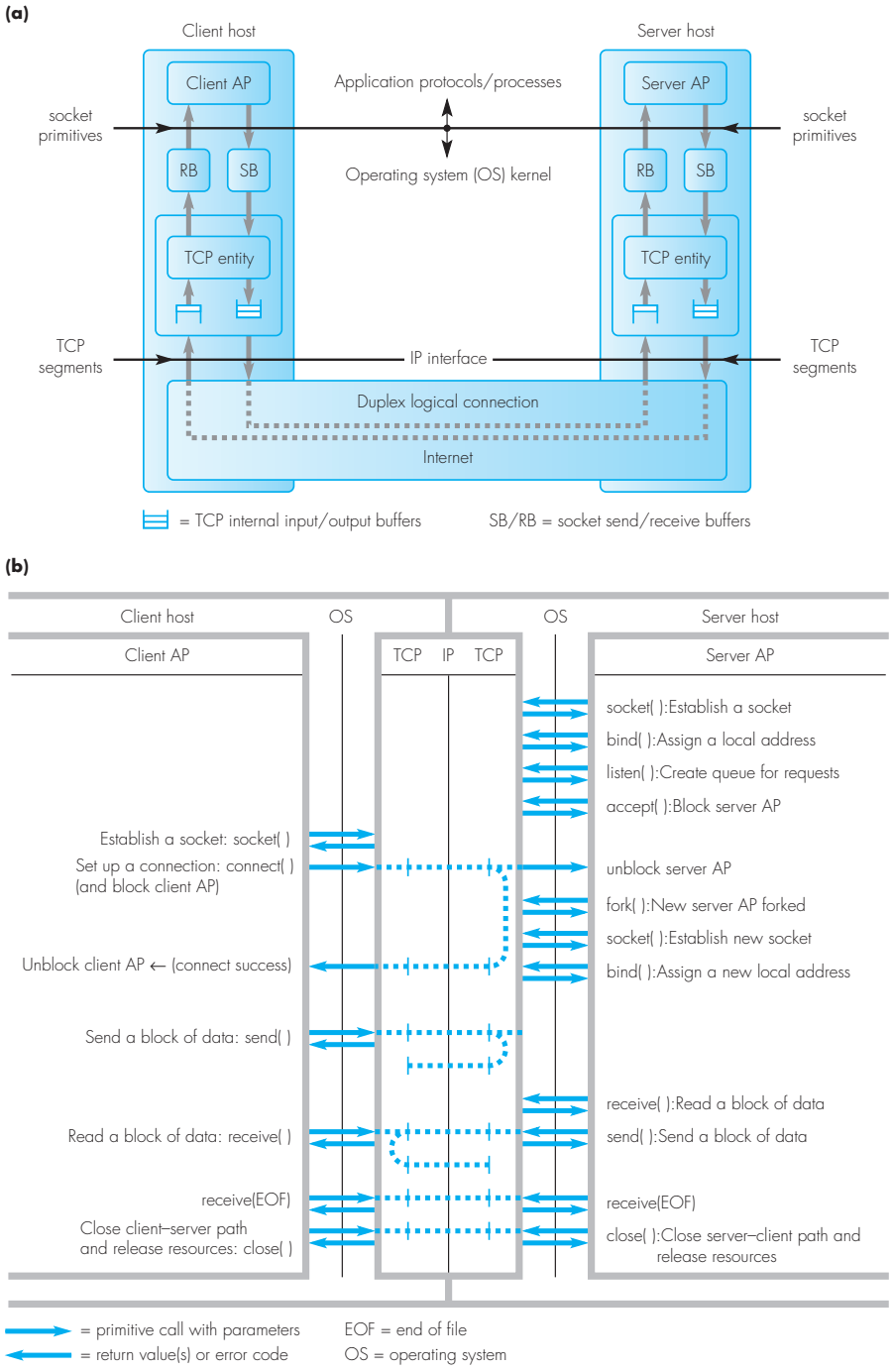


Figure 7.3 TCP socket primitives: (a) socket interface; (b) primitives and their use.

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

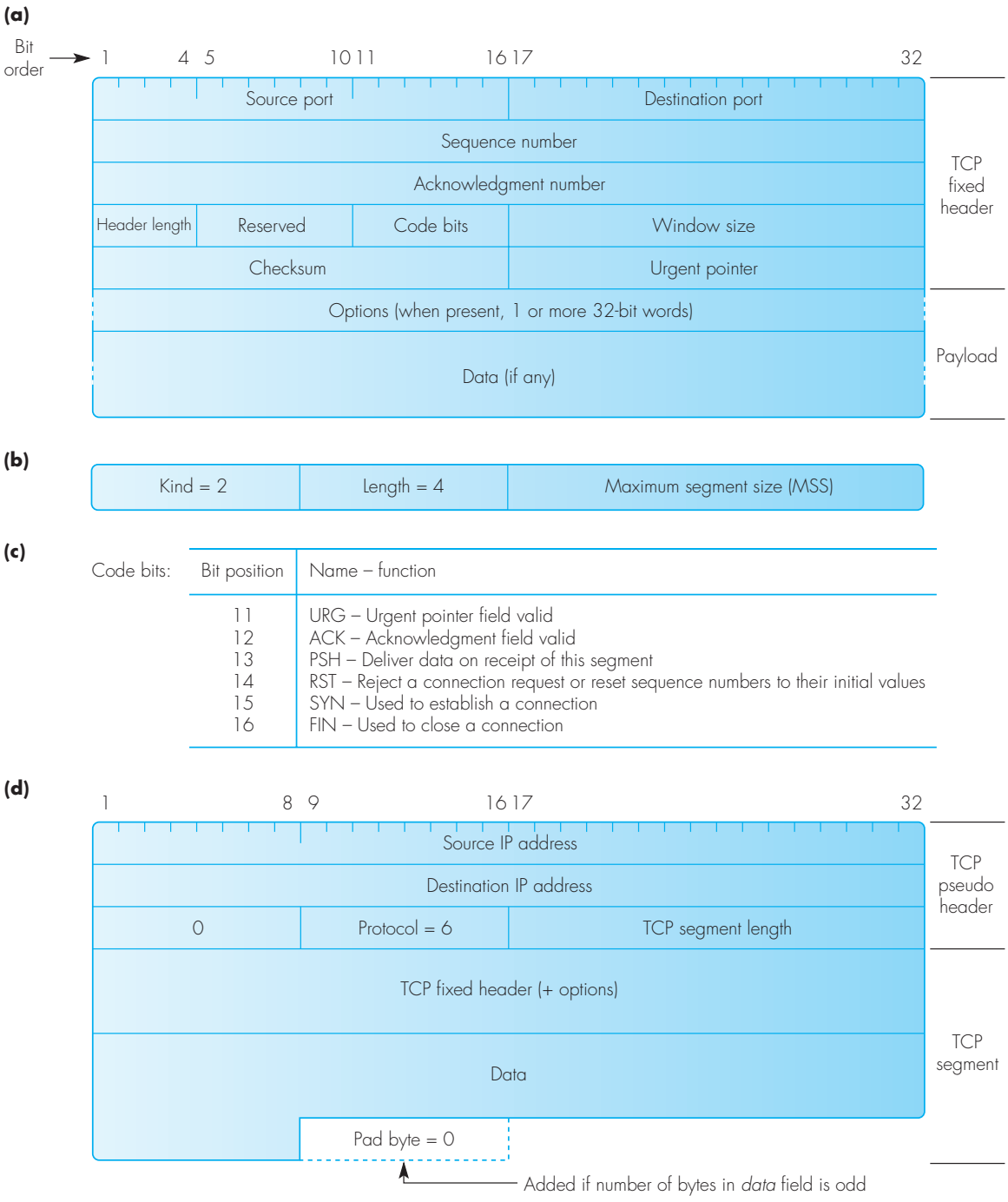


Figure 7.4 TCP segment format: (a) header fields; (b) MSS option format; (c) code bit definitions; (d) pseudo header fields.

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 16384 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

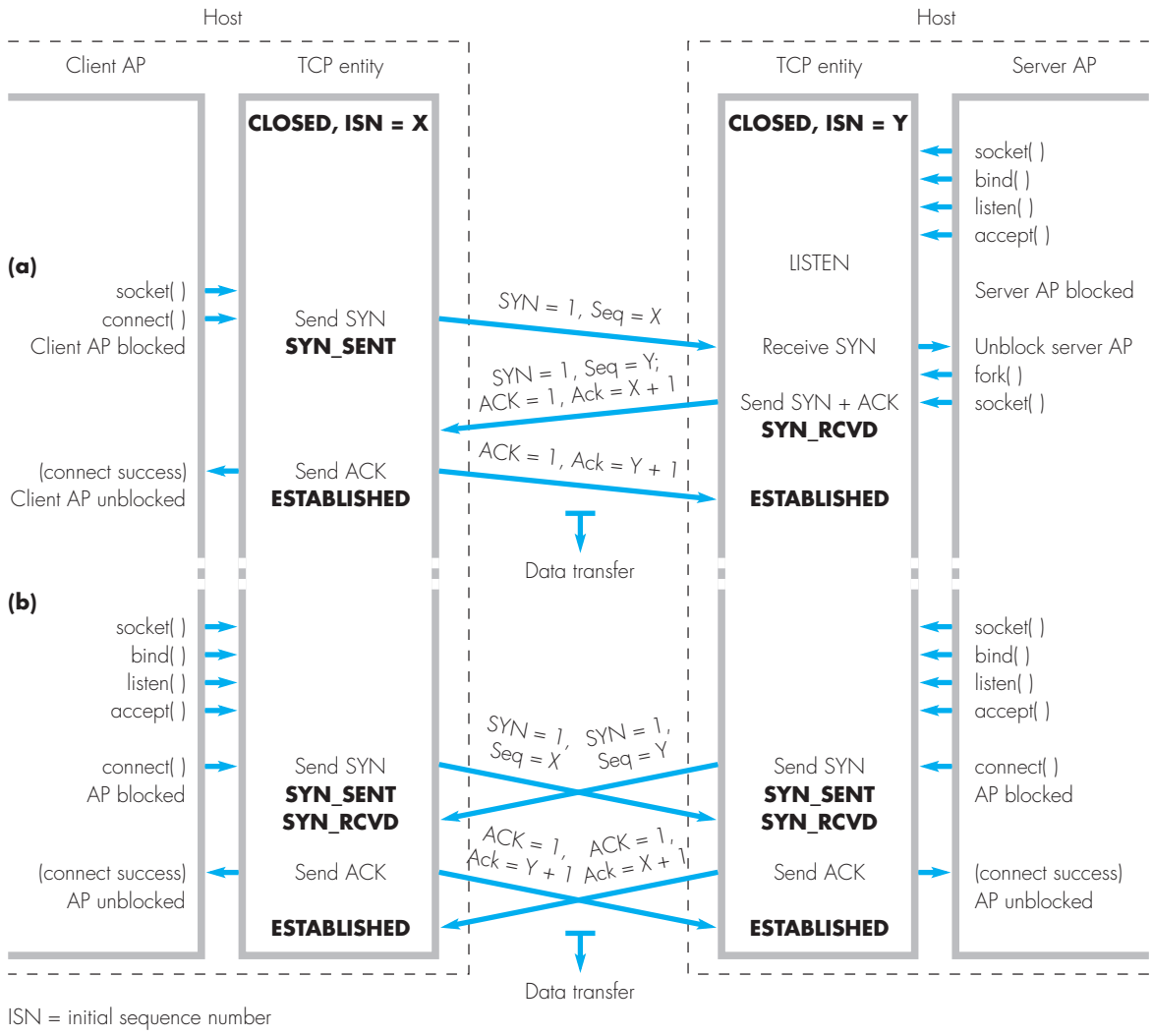


Figure 7.5 TCP connection establishment examples: (a) client–server; (b) connection collision.

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).

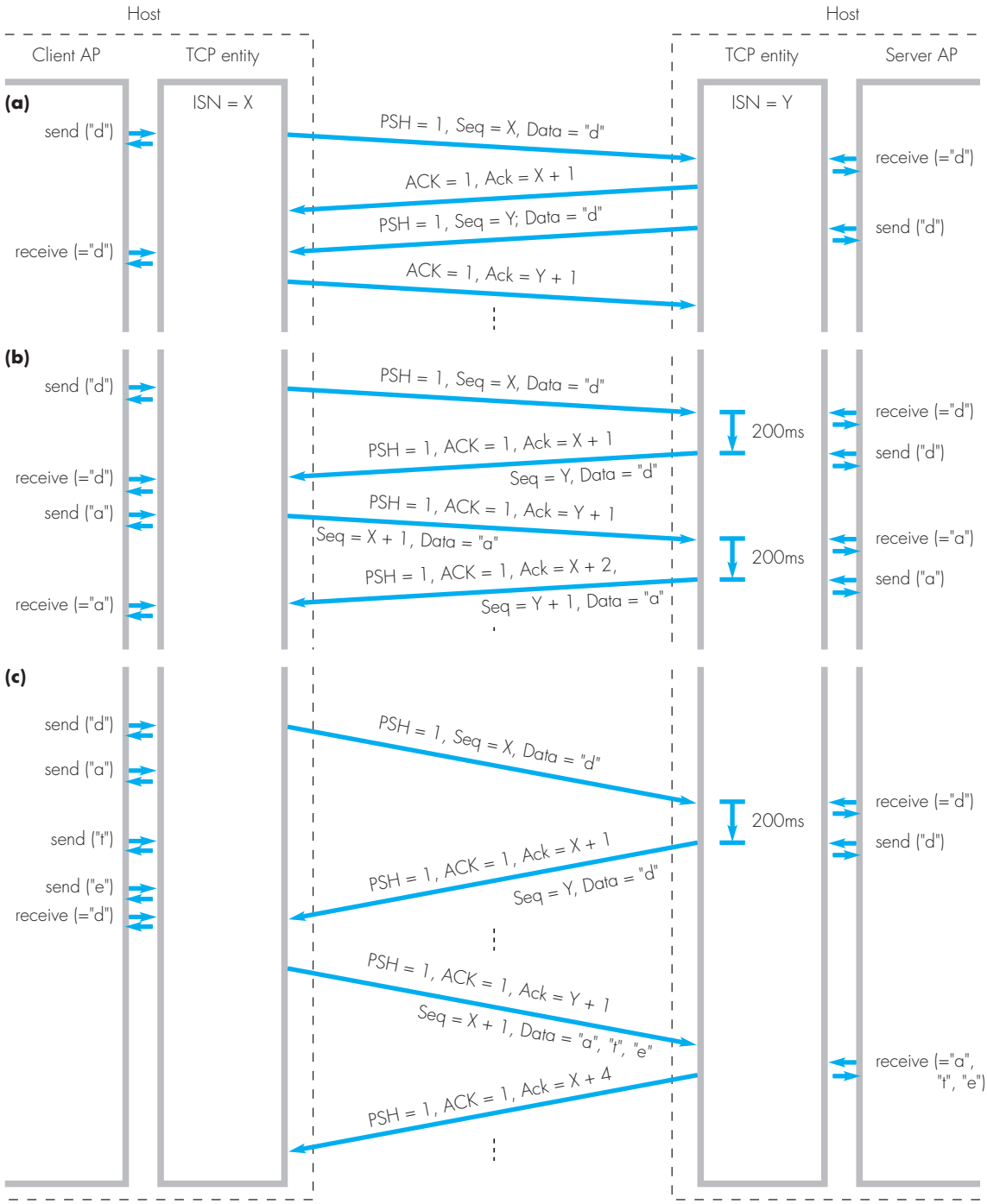


Figure 7.6 Small segment data transfers: (a) immediate acknowledgments; (b) delayed acknowledgments; (c) Nagle algorithm.

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.

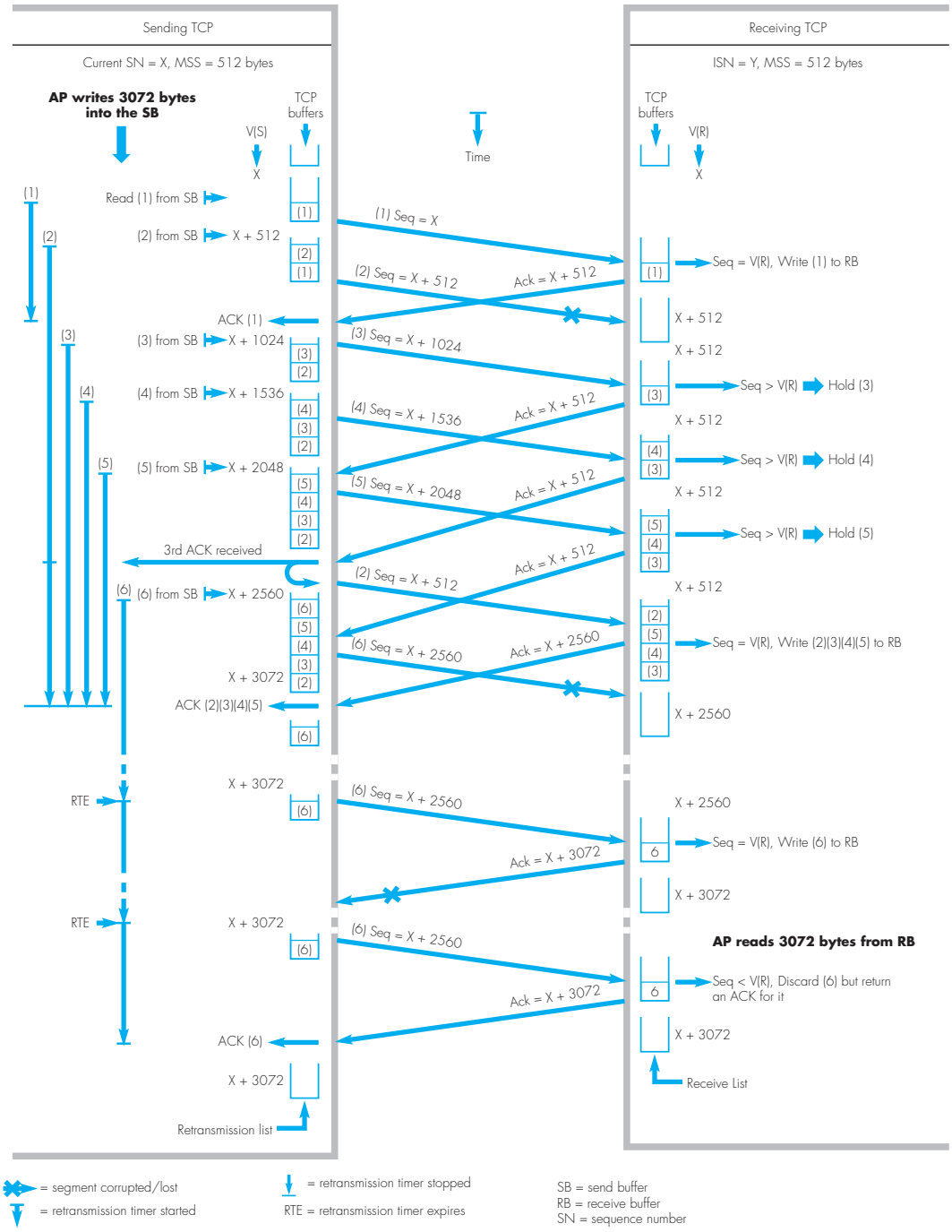


Figure 7.7 Example segment sequence showing TCP error control procedure.

- 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;

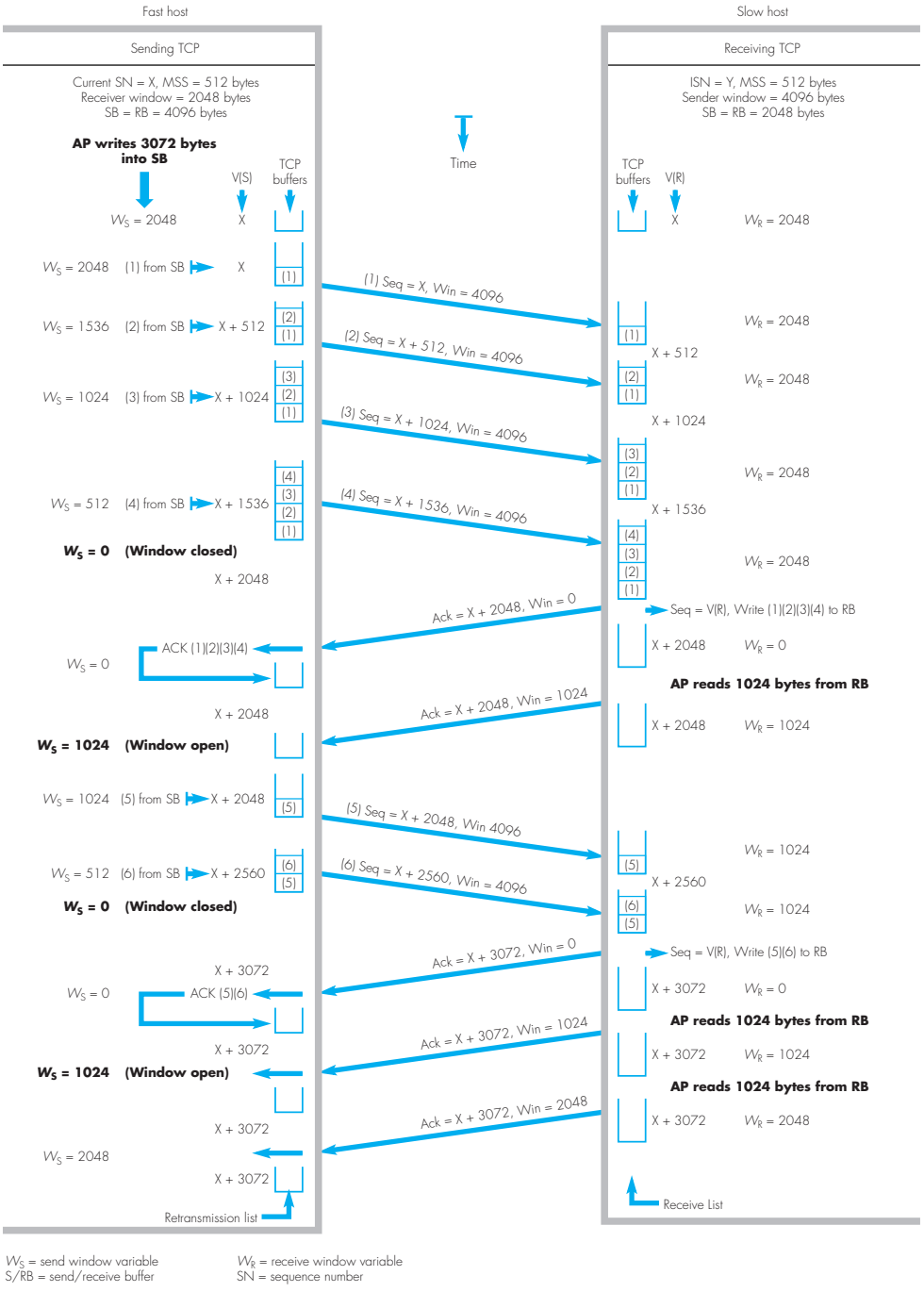


Figure 7.8 Example segment sequence showing TCP flow control procedure.

- 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:

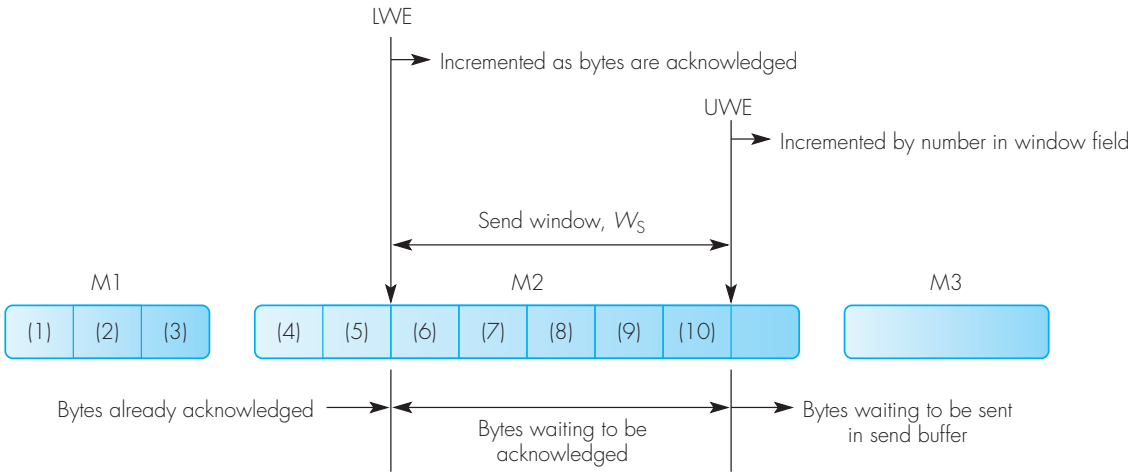
- The flow of segments starts with the AP at the sending side writing 3072 bytes into the send buffer using a *send( )* primitive.
- Since the sending host is much faster than the receiving host, the sending TCP is able to send a full window of 2048 bytes (in four 512-byte segments) before the receiving 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,

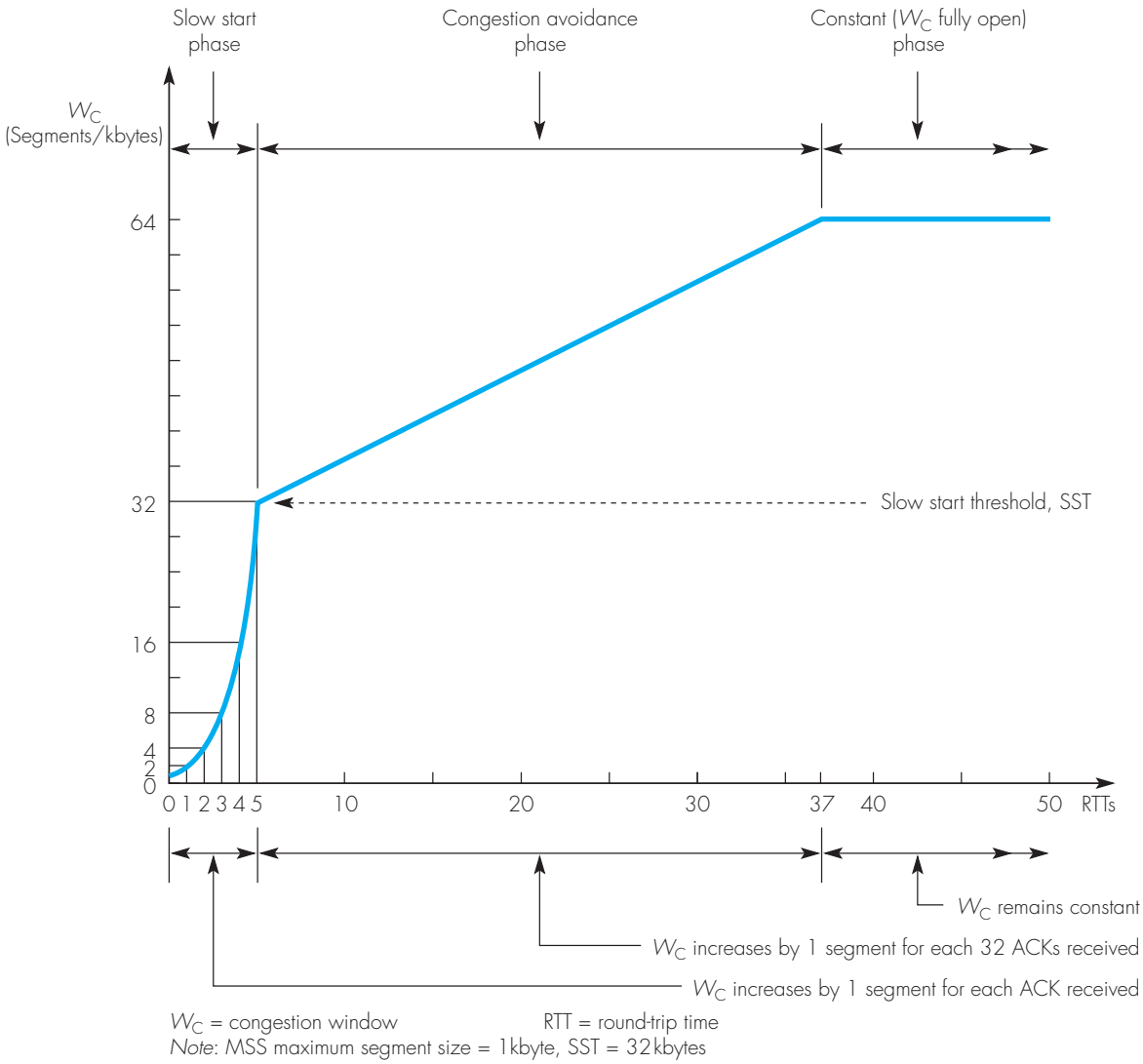


Figure 7.10 TCP congestion control window procedure.

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.

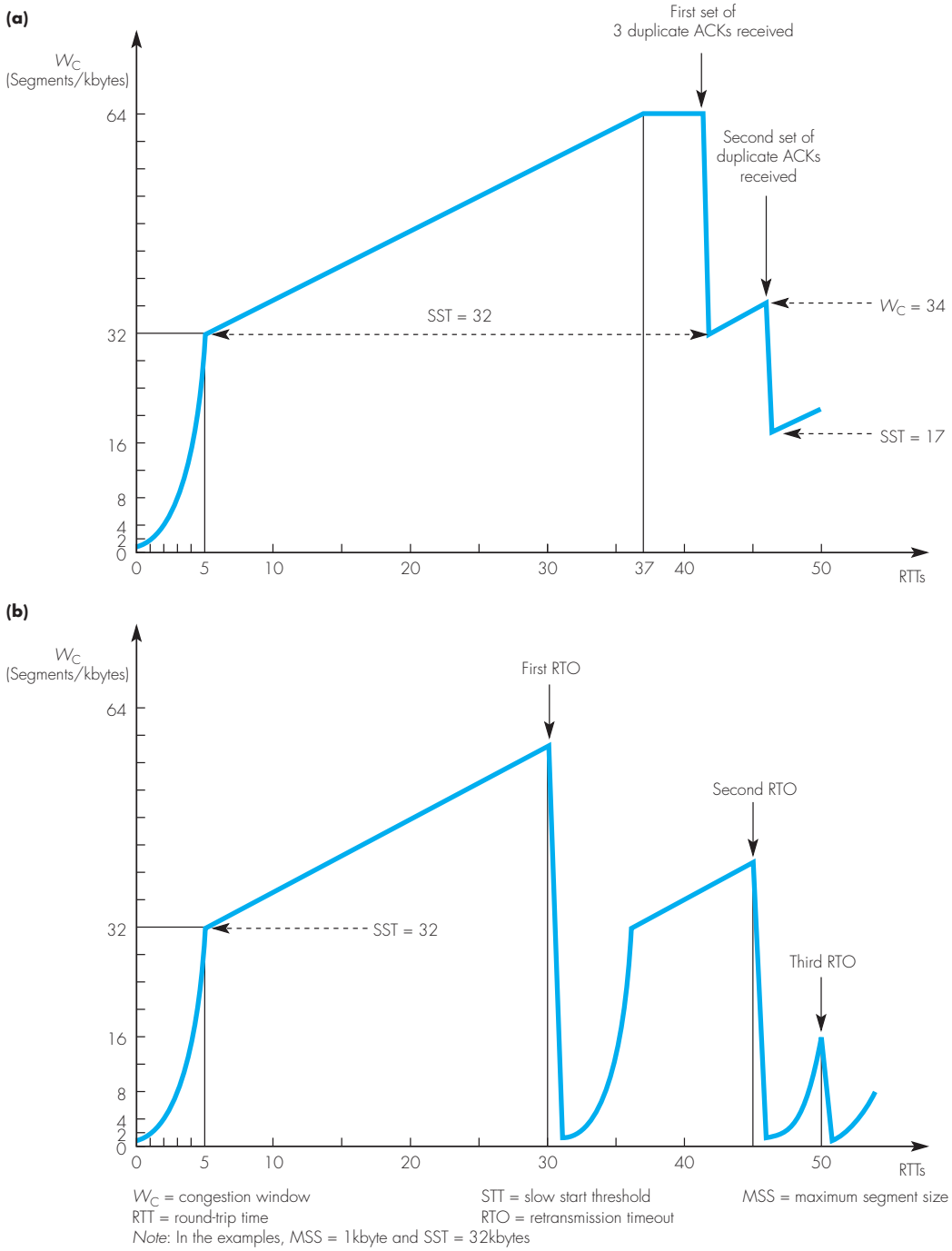


Figure 7.11 TCP congestion window adjustments: (a) on receipt of duplicate ACKs; (b) on expiry of a retransmission timer.