Computer Network
Unit-V

Transport Layer

By:- Dr. D.R.Patil

• Outline

- The Transport Service: Port Addressing, Transport Service Primitives, Berkeley Sockets, Connection Management (Handshake, Teardown),
- UDP, TCP, TCP State Transition, TCP Timers, TCP Flow Control (Sliding Window),
- TCP Congestion Control: Slow Start.

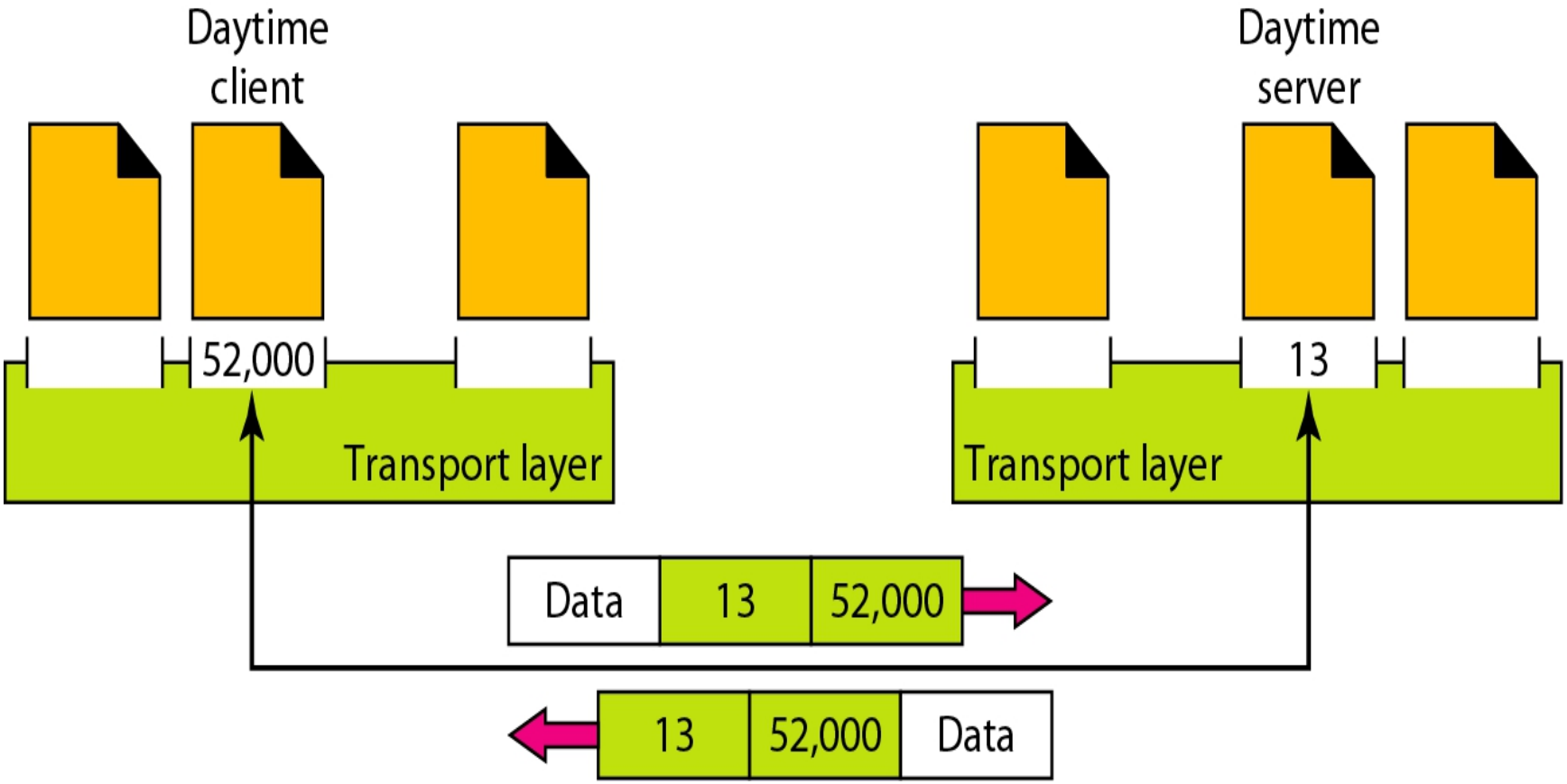
- **The Transport Service**
  - **Addressing**
    - –**At the data link laye**r, we need a MAC address to choose one node among several nodes if the connection is not point-to-point. A frame in the data link layer needs a destination MAC address for delivery and a source address for the next node's reply.
    - –**At the network layer**, we need an IP address to choose one host among millions.
    - –A datagram in the network layer needs a destination IP address for delivery and a source IP address for the destination's reply.

–**At the transport layer**, we need a transport layer address, called a port number, to choose among multiple processes running on the destination host.

–The destination port number is needed for delivery; the source port number is needed for the reply.

- In the Internet model, the port numbers are 16-bit integers between 0 and 65,535. The client program defines itself with a port number, chosen randomly by the transport layer software running on the client host.
- **This is the ephemeral port number.**
- The server process must also define itself with a port number.
- This port number, however, cannot be chosen randomly.
- Of course, one solution would be to send a special packet and request the port number of a specific server, but this requires more overhead.
- **The Internet has decided to use universal port numbers for servers; these are called well-known port numbers.**

Figure 23.2 *Port numbers*



- **IANA Ranges**
  - The lANA (Internet Assigned Number Authority) has divided the port numbers into three ranges:
    - well known,
    - registered, and
    - dynamic (or private), as shown in Figure.

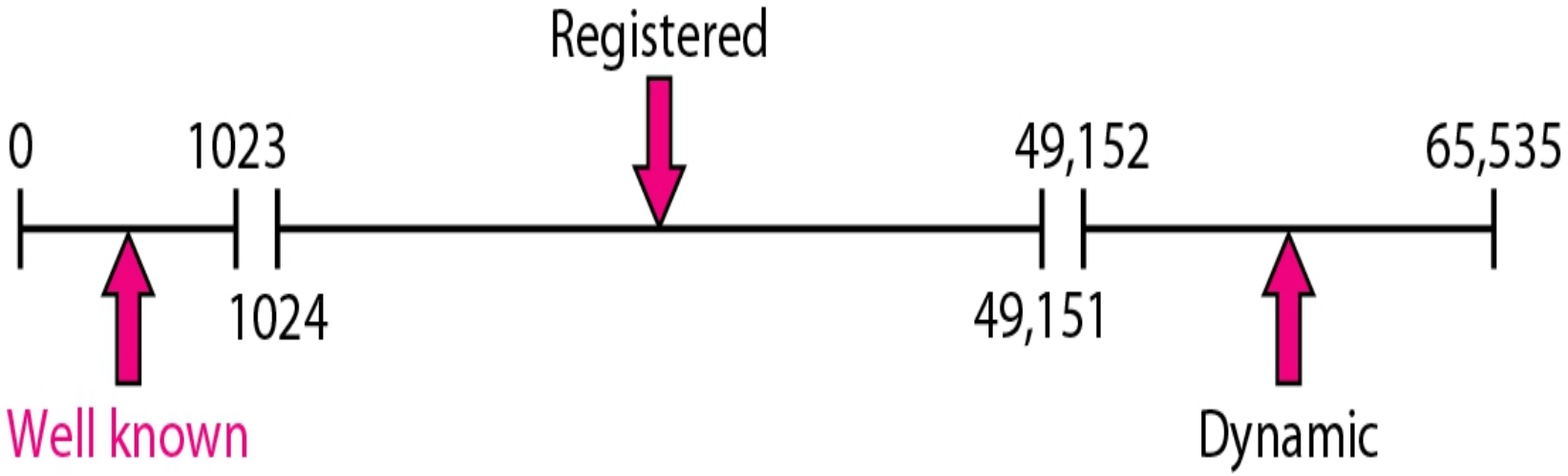
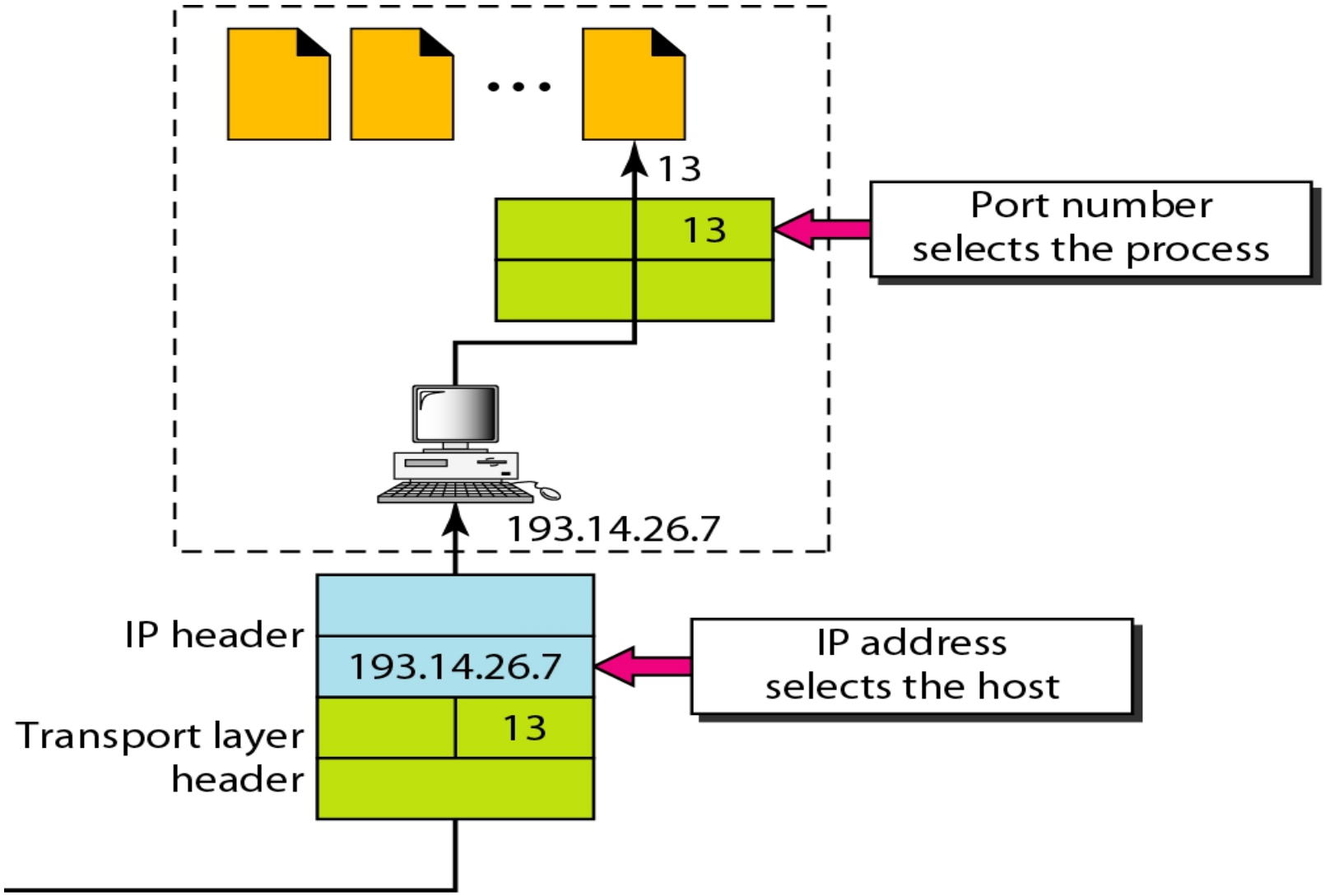


Figure 23.4 ***IANA ranges***

- **Well-known ports.**
  - The ports ranging from 0 to 1023 are assigned and controlled by lANA.
  - These are the well-known ports.
- **Registered ports.**
  - The ports ranging from 1024 to 49,151 are not assigned or controlled by lANA.
  - They can only be registered with lANA to prevent duplication.
- **Dynamic ports.**
  - The ports ranging from 49,152 to 65,535 are neither controlled nor registered.
  - They can be used by any process.
  - These are the ephemeral ports.

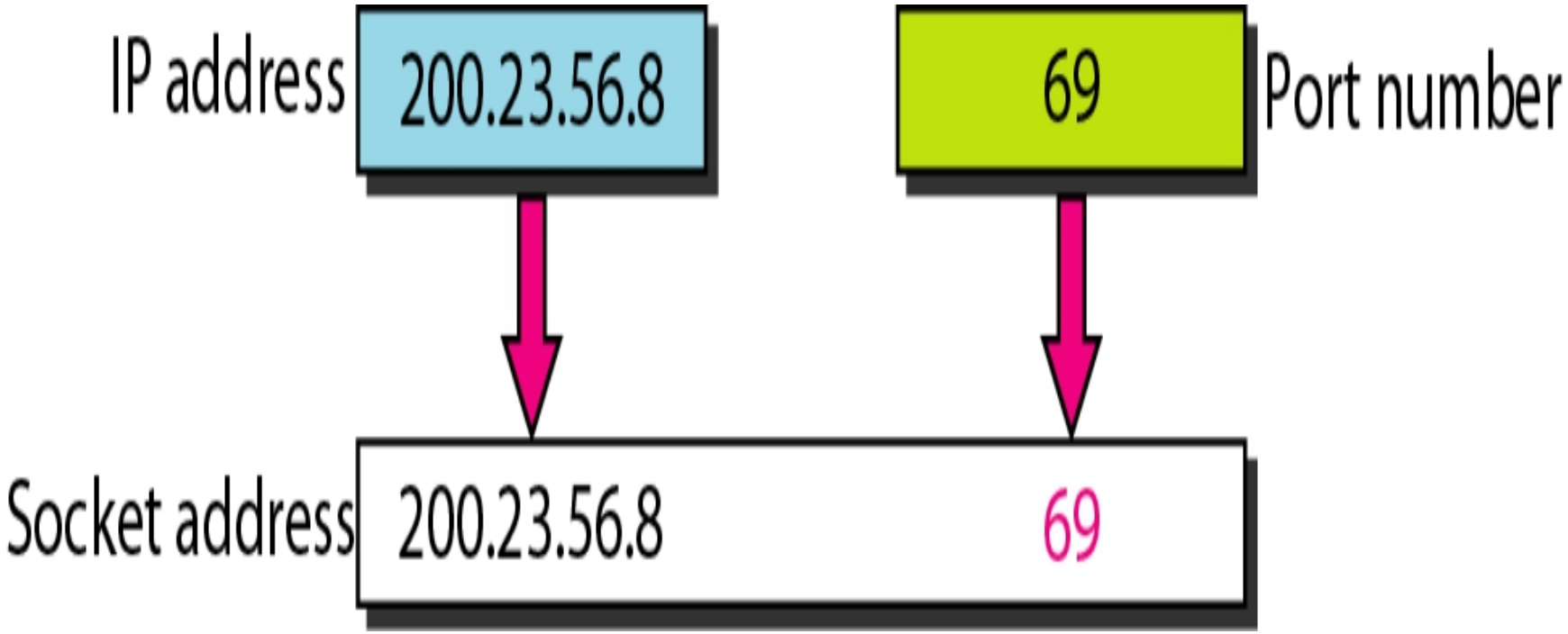
Figure 23.3 *IP addresses versus port numbers*



- **Socket Addresses**
  - –Process-to-process delivery needs two identifiers, IP address and the port number, at each end to make a connection.
  - –**The combination of an IP address and a port number is called a socket address.**
  - –The client socket address defines the client process uniquely just as the server socket address defines the server process uniquely.
  - –A transport layer protocol needs a pair of socket addresses: the client socket address and the server socket address.
  - –These four pieces of information are part of the IP header and the transport layer protocol header.

–The IP header contains the IP addresses; the UDP or TCP header contains the port numbers.

Figure 23.5 *Socket address*

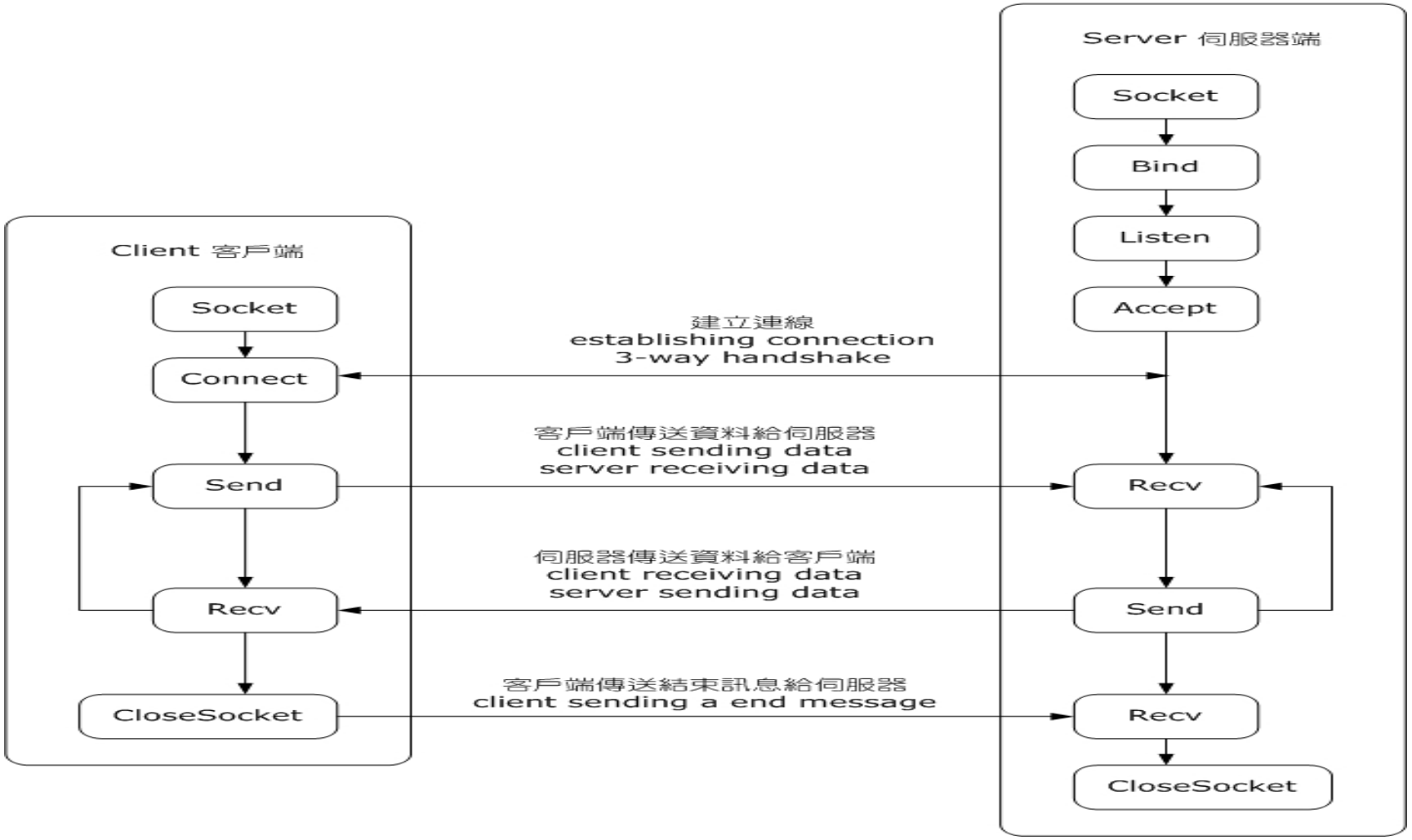


- **Berkeley Sockets**
- Berkeley sockets is an application programming interface (API) for Internet sockets and Unix domain sockets, used for inter-process communication (IPC).
- It is commonly implemented as a library of linkable modules.
- It originated with the 4.2BSD Unix released in 1983.
- A socket is an abstract representation (handle) for the local endpoint of a network communication path.
- The Berkeley sockets API represents it as a file descriptor (file handle) in the Unix philosophy that provides a common interface for input and output to streams of data.

- **Berkeley Sockets**
- Berkeley sockets evolved with little modification from a de facto standard into a component of the POSIX specification.
- Therefore, the term POSIX sockets is essentially synonymous withBerkeley sockets.
- They are also known as BSD sockets, acknowledging the first implementation in the Berkeley Software Distribution.

•Socket API functions

TCP Socket 基本流程圖
TCP Socket flow diagram



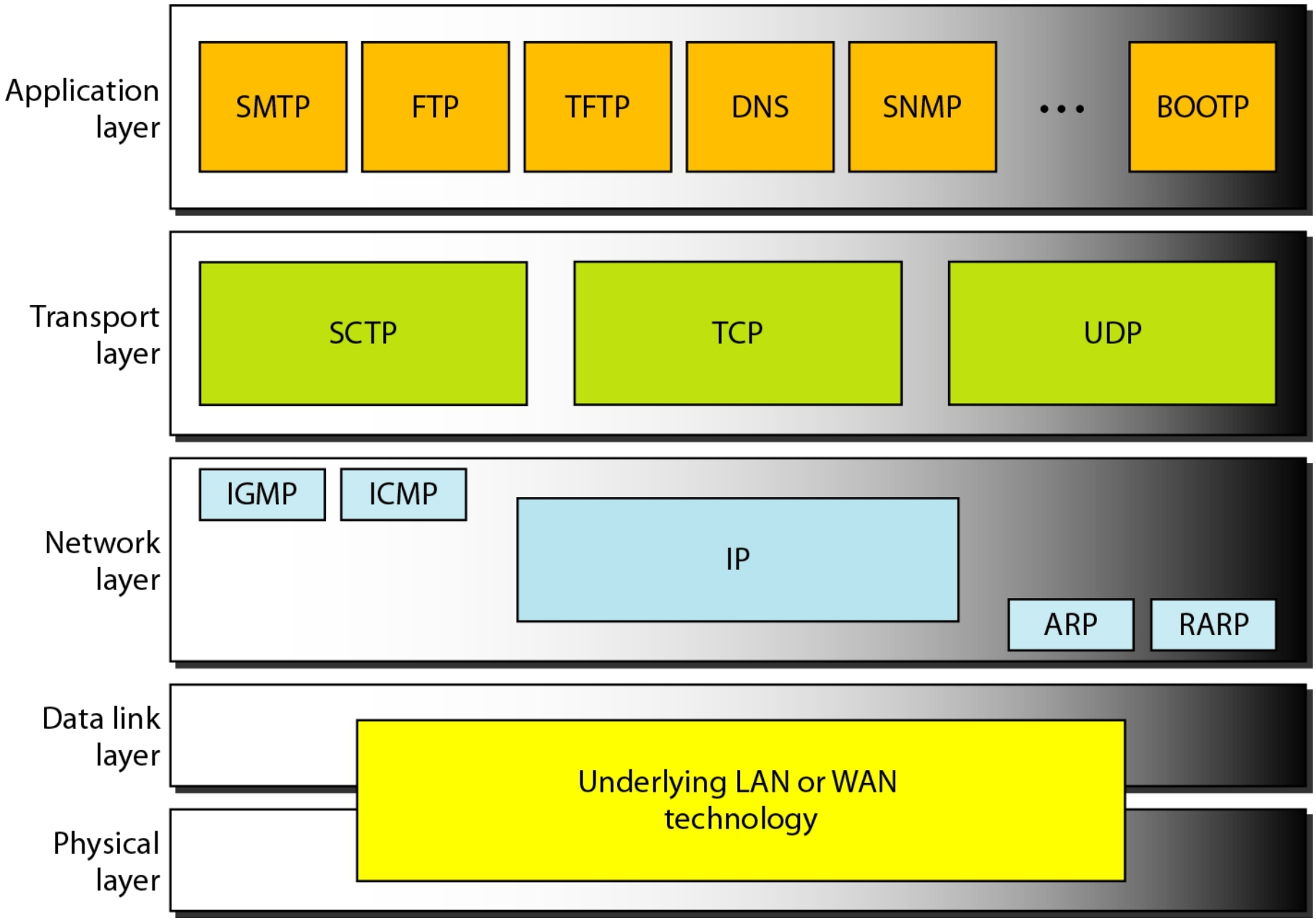
- **Socket API functions**
- This list is a summary of functions or methods provided by the Berkeley sockets API library:
- **socket()** creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.
- **bind()** is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.
- **listen()** is used on the server side, and causes a bound TCP socket to enter listening state.

- **Socket API functions**
- **connect()** is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.
- **accept()** is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.
- **send()** and **recv()**, or **write()** and **read()**, or **sendto()** and **recvfrom(),** are used for sending and receiving data to/from a remote socket.

- **Socket API functions**
- **close()** causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.
- **gethostbyname()** and **gethostbyaddr()** are used to resolve host names and addresses. IPv4 only.
- **select()** is used to suspend, waiting for one or more of a provided list of sockets to be ready to read, ready to write, or that have errors.
- **poll()** is used to check on the state of a socket in a set of sockets. The set can be tested to see if any socket can be written to, read from or if an error occurred.

- **Socket API functions**
- **getsockopt()** is used to retrieve the current value of a particular socket option for the specified socket.
- **setsockopt()** is used to set a particular socket option for the specified socket.

Figure 23.8 *Position of UDP, TCP, and SCTP in TCP/IP suite*



- **User Datagram Protocol (UDP):**
  - The User Datagram Protocol (UDP) is called a connectionless, unreliable transport protocol.
  - It does not add anything to the services of IP except to provide process-to-process communication instead of host-to-host communication.
  - Also, it performs very limited error checking.
  - If UDP is so powerless, why would a process want to use it?
  - With the disadvantages come some advantages.
  - UDP is a very simple protocol using a minimum of overhead.
  - If a process wants to send a small message and does not care much about reliability, it can use UDP.
  - Sending a small message by using UDP takes much less interaction between the sender and receiver than using TCP or SCTP.

- **Well-Known Ports for UDP**
  - Table shows some well-known port numbers used by UDP.
  - Some port numbers can be used by both UDP and TCP.

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Nameserver	Domain Name Service
67	BOOTPs	Server port to download bootstrap information
68	BOOTPc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
111	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

- **User Datagram**
  - UDP packets, called user datagrams, have a fixed-size header of 8 bytes.
  - Figure shows the format of a user datagram.

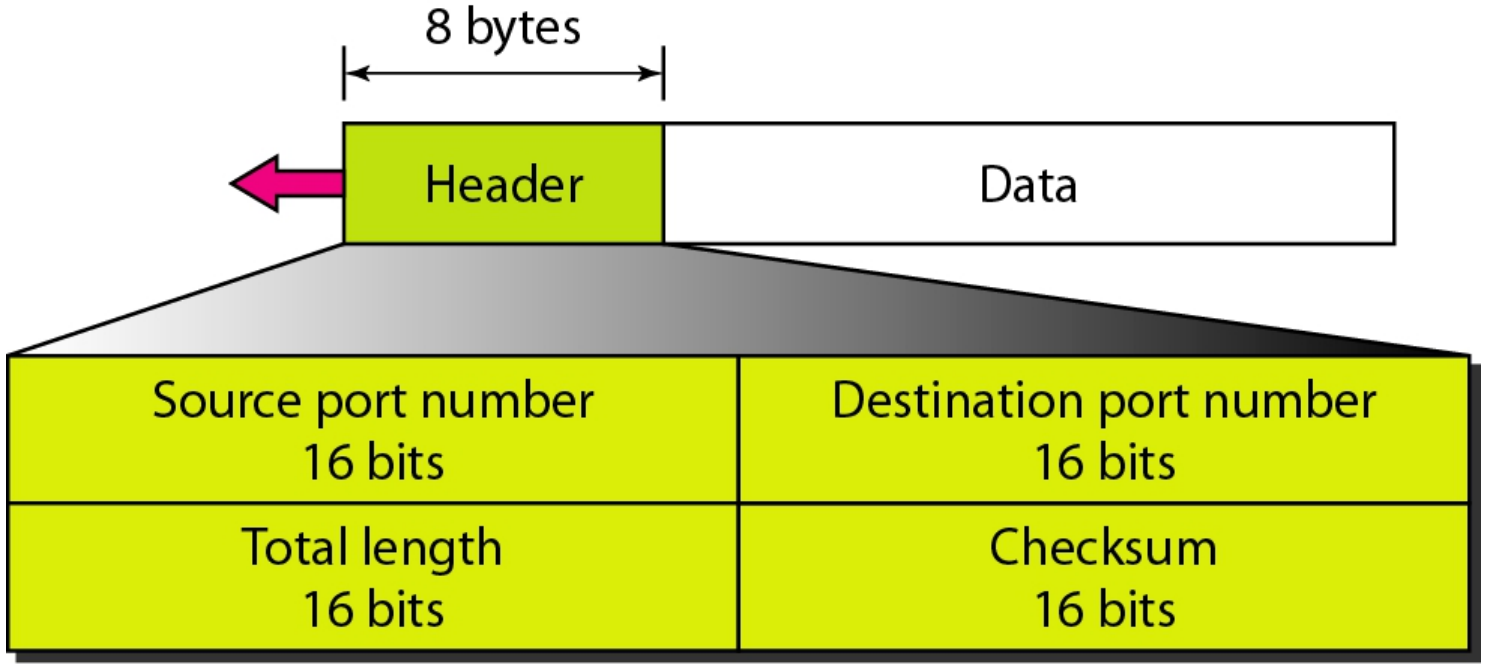


Figure 23.9 *User datagram format*

- **Source port number.**
  - This is the port number used by the process running on the source host.
  - It is 16 bits long, which means that the port number can range from 0 to 65,535.
  - If the source host is the client (a client sending a request), the port number, in most cases, is an ephemeral port number requested by the process and chosen by the UDP software running on the source host.
  - If the source host is the server (a server sending a response), the port number, in most cases, is a well-known port number.

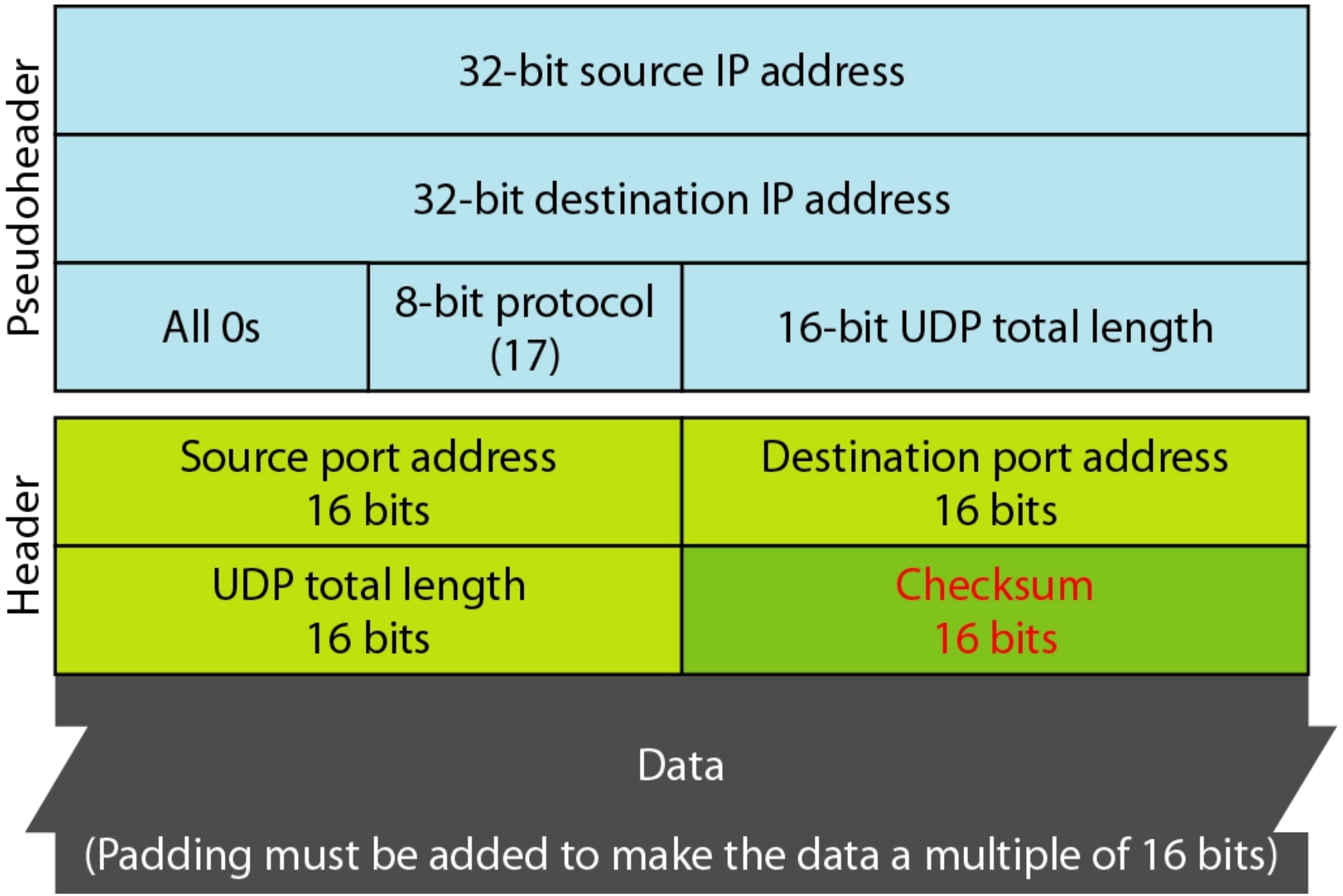
- **Destination port number.**
  - This is the port number used by the process running on the destination host. It is also 16 bits long.
  - If the destination host is the server (a client sending a request), the port number, in most cases, is a well-known port number.
  - If the destination host is the client (a server sending a response), the port number, in most cases, is an ephemeral port number.
  - In this case, the server copies the ephemeral port number it has received in the request packet.
- **Length.**
  - This is a 16-bit field that defines the total length of the user datagram, header plus data.
  - The 16 bits can define a total length of 0 to 65,535 bytes.
  - However, the total length needs to be much less because a UDP user datagram is stored in an IP datagram with a total length of 65,535 bytes.

- The length field in a UDP user datagram is actually not necessary.
- A user datagram is encapsulated in an IP datagram.
- There is a field in the IP datagram that defines the total length.
- There is another field in the IP datagram that defines the length of the header.
- So if we subtract the value of the second field from the first, we can deduce the length of a UDP datagram that is encapsulated in an IP datagram.
- **UDP length = IP length - IP header's length**

- **Checksum.**
  - This field is used to detect errors over the entire user datagram (header plus data).

- **Checksum**
  - The UDP checksum calculation is different from the one for IP and ICMP.
  - Here the checksum includes three sections: a pseudoheader, the UDP header, and the data coming from the application layer.
  - The pseudoheader is the part of the header of the IP packet in which the user datagram is to be encapsulated with some fields filled with Os (see Figure).

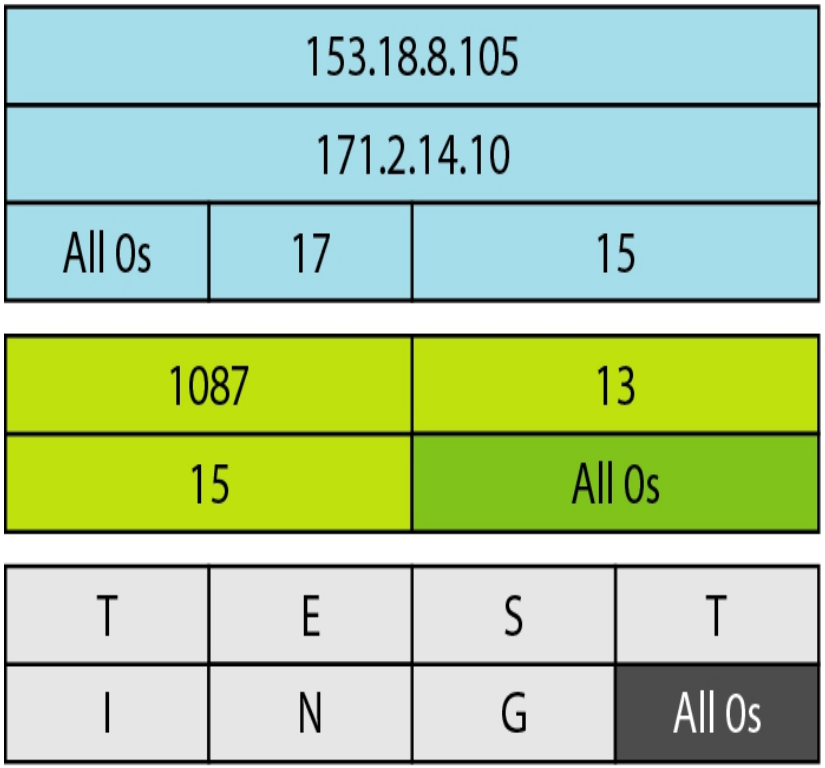
Figure 23.10 *Pseudoheader for checksum calculation*



- If the checksum does not include the pseudoheader, a user datagram may arrive safe and sound.
- However, if the IP header is corrupted, it may be delivered to the wrong host.
- The protocol field is added to ensure that the packet belongs to UDP, and not to other transport-layer protocols.
- The value of the protocol field for UDP is 17.
- If this value is changed during transmission, the checksum calculation at the receiver will detect it and UDP drops the packet.
- It is not delivered to the wrong protocol.
- Note the similarities between the pseudoheader fields and the last 12 bytes of the IP header.

- Figure 23.11 shows the checksum calculation for a very small user datagram with only 7 bytes of data.
- Because the number of bytes of data is odd, padding is added for checksum calculation.
- The pseudoheader as well as the padding will be dropped when the user datagram is delivered to IP.

Figure 23.11 ***Checksum calculation of a simple UDP user datagram***



10011001 00010010 → 153.18
00001000 01101001 → 8.105
10101011 00000010 → 171.2
00001110 00001010 → 14.10
00000000 00010001 → 0 and 17
00000000 00001111 → 15
00000100 00111111 → 1087
00000000 00001101 → 13
00000000 00001111 → 15
00000000 00000000 → 0 (checksum)
01010100 01000101 → T and E
01010011 01010100 → S and T
01001001 01001110 → I and N
01000111 00000000 → G and 0 (padding)

10010110 11101011 → Sum
01101001 00010100 → Checksum

- **Use of UDP**
  - The following lists some uses of the UDP protocol:
  - UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control.
  - It is not usually used for a process such as FTP that needs to send bulk data.
  - UDP is suitable for a process with internal flow and error control mechanisms.
  - UDP is a suitable transport protocol for multicasting.
  - Multicasting capability is embedded in the UDP software but not in the TCP software.
  - UDP is used for management processes such as SNMP.
  - UDP is used for some route updating protocols such as Routing Information Protocol(RIP).

- **Transmission Control Protocol (TCP):**
  - TCP, like UDP, is a process-to-process (program-to-program) protocol.
  - TCP, therefore, like UDP, uses port numbers.
  - Unlike UDP, TCP is a connection-oriented protocol; it creates a virtual connection between two TCPs to send data.
  - In addition, TCP uses flow and error control mechanisms at the transport level.
  - In brief, TCP is called a connection-oriented, reliable transport protocol.
  - It adds connection-oriented and reliability features to the services of IP.

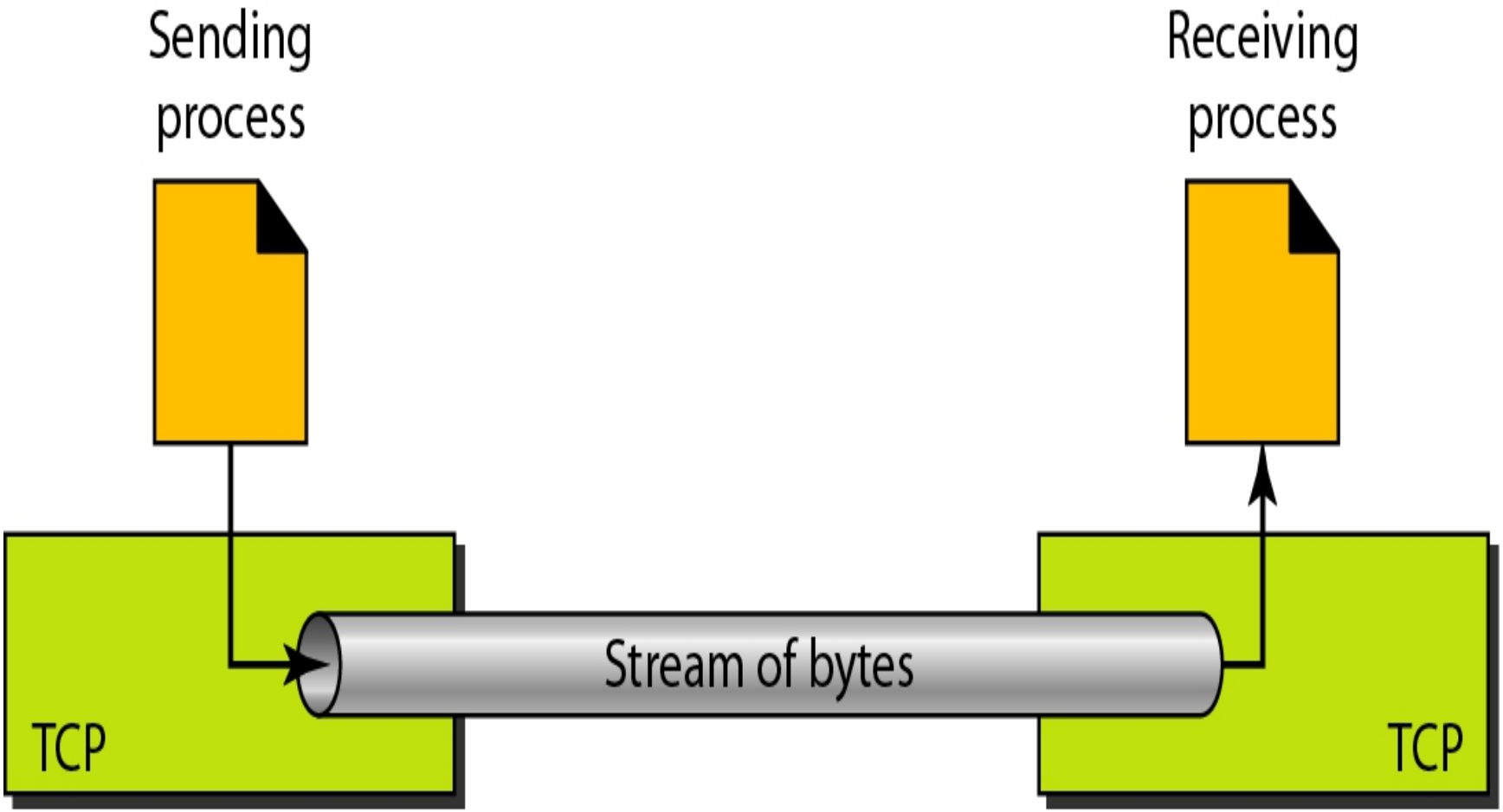
- **TCP Services**
  - **Process-to-Process Communication**
  - Like UDP, TCP provides process-to-process communication using port numbers.
  - Table lists some well-known port numbers used by TCP.

Table 23.2 *Well-known ports used by TCP*

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
20	FTP, Data	File Transfer Protocol (data connection)
21	FTP, Control	File Transfer Protocol (control connection)
23	TELNET	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol
111	RPC	Remote Procedure Call

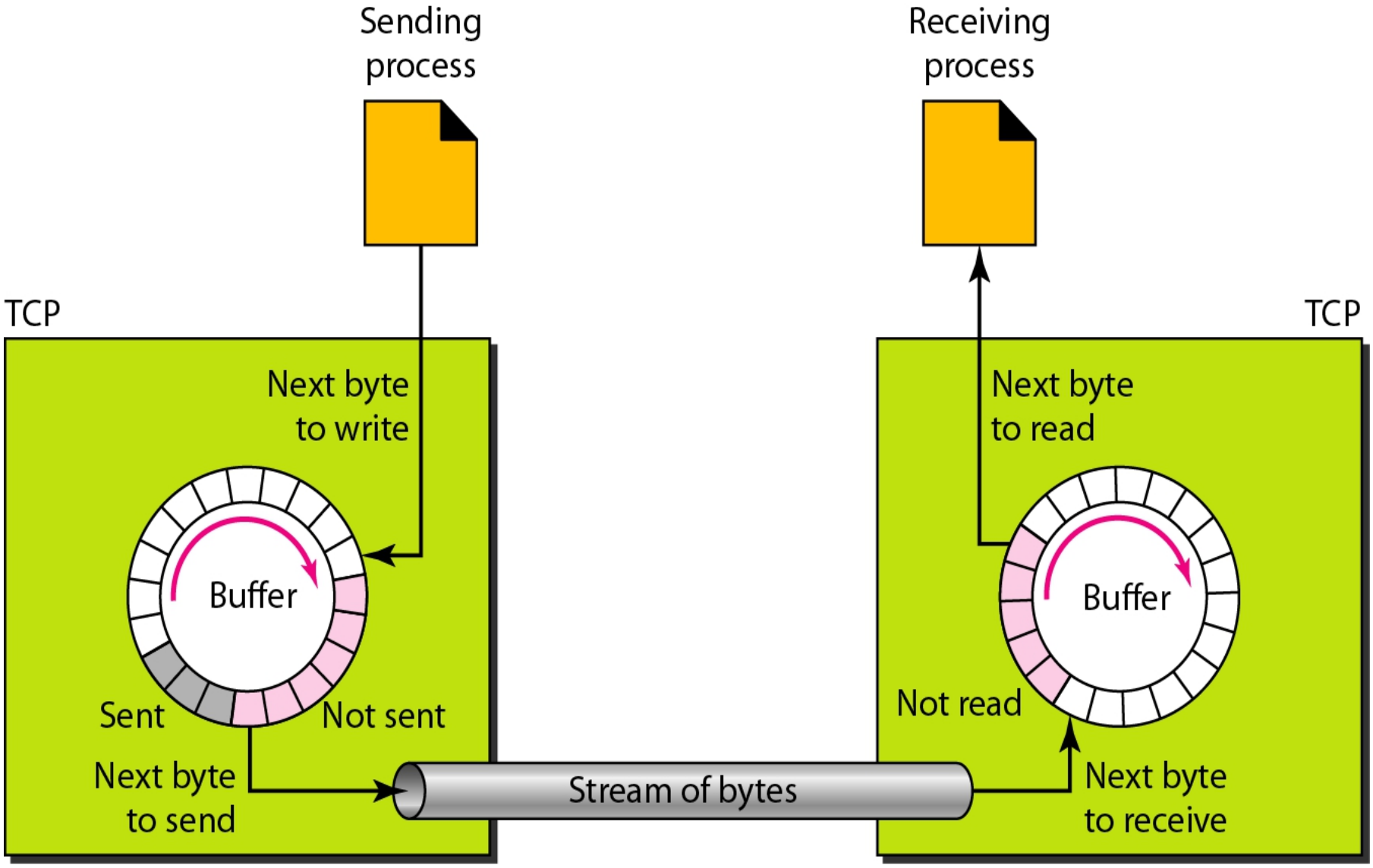
- **Stream Delivery Service**
  - TCP, unlike UDP, is a stream-oriented protocol.
  - In UDP, a process (an application program) sends messages, with predefined boundaries, to UDP for delivery.
  - TCP, on the other hand, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes.
  - TCP creates an environment in which the two processes seem to be connected by an imaginary "tube" that carries their data across the Internet.
  - This imaginary environment is depicted in Figure.
  - The sending process produces (writes to) the stream of bytes, and the receiving process consumes (reads from) them.

Figure 23.13 ***Stream delivery***



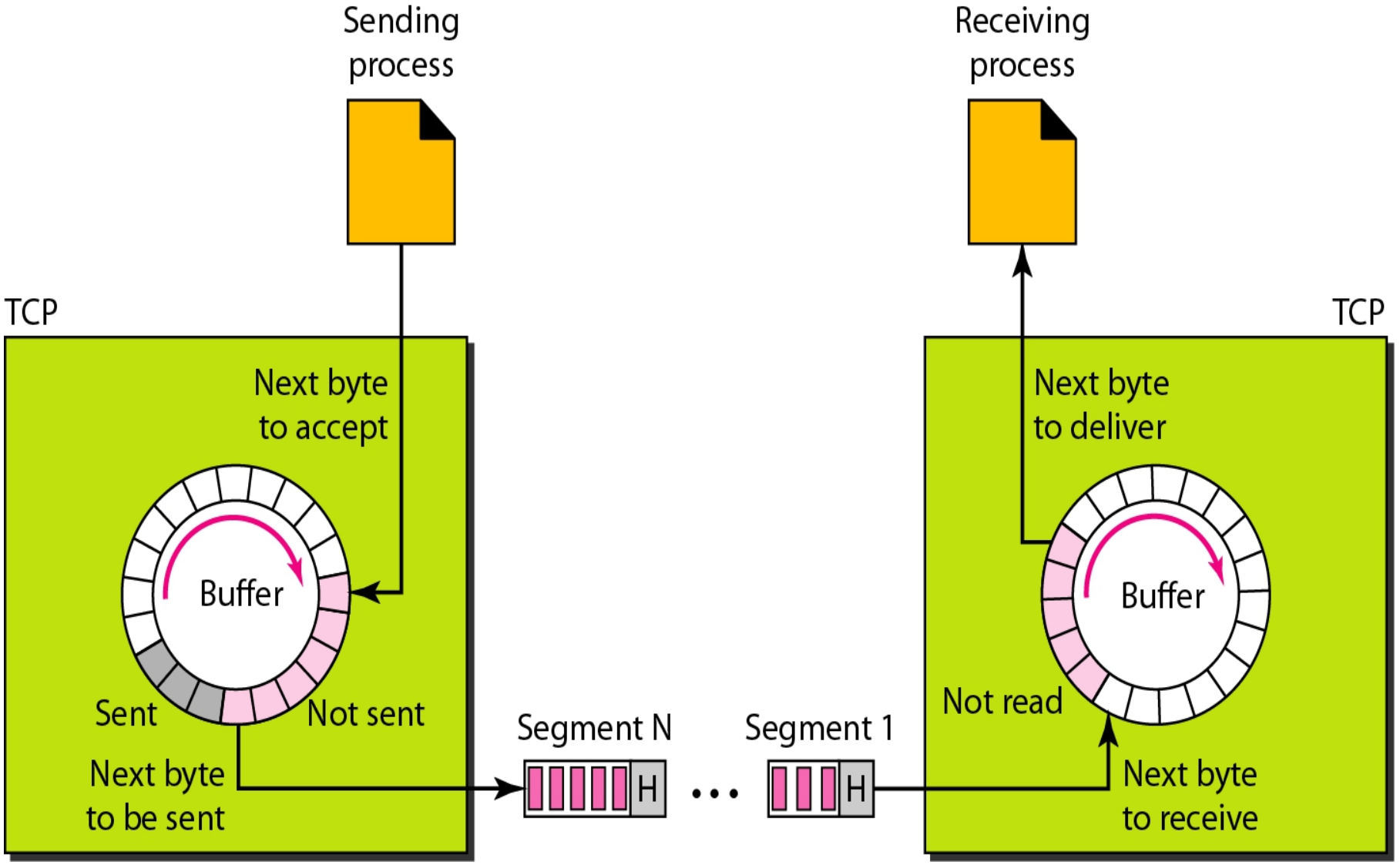
- **Sending and Receiving Buffers**
  - Because the sending and the receiving processes may not write or read data at the same speed, TCP needs buffers for storage.
  - There are two buffers, the sending buffer and the receiving buffer, one for each direction.
  - One way to implement a buffer is to use a circular array of 1-byte locations as shown in Figure.
  - For simplicity, we have shown two buffers of 20 bytes each; normally the buffers are hundreds or thousands of bytes, depending on the implementation.
  - We also show the buffers as the same size, which is not always the case.

Figure 23.14 *Sending and receiving buffers*



- **Segments**
  - Although buffering handles the disparity between the speed of the producing and consuming processes, we need one more step before we can send data.
  - The IP layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes.
  - At the transport layer, TCP groups a number of bytes together into a packet called a segment.
  - TCP adds a header to each segment (for control purposes) and delivers the segment to the IP layer for transmission.
  - The segments are encapsulated in IP datagrams and transmitted.
  - This entire operation is transparent to the receiving process.
  - Figure shows how segments are created from the bytes in the buffers.

Figure 23.15 *TCP segments*



- **Full-Duplex Communication**
  - TCP offers full-duplex service, in which data can flow in both directions at the same time.
  - Each TCP then has a sending and receiving buffer, and segments move in both directions.
- **Connection-Oriented Service**
  - TCP, unlike UDP, is a connection-oriented protocol.
  - When a process at site A wants to send and receive data from another process at site B, the following occurs:
  - 1. The two TCPs establish a connection between them.
  - 2. Data are exchanged in both directions.
  - 3. The connection is terminated.
- **Reliable Service**
  - TCP is a reliable transport protocol.
  - It uses an acknowledgment mechanism to check the safe and sound arrival of data.

- **TCP Features**
- **Numbering System**
  - Although the TCP software keeps track of the segments being transmitted or received, there is no field for a segment number value in the segment header.
  - Instead, there are two fields called the sequence number and the acknowledgment number.
  - These two fields refer to the byte number and not the segment number.
- **Byte Number**
  - TCP numbers all data bytes that are transmitted in a connection.
  - Numbering is independent in each direction.
  - When TCP receives bytes of data from a process, it stores them in the sending buffer and numbers them.
  - The numbering does not necessarily start from O.
  - Instead, TCP generates a random number between 0 and $2^{32} - 1$ for the number of the first byte.

- For example, if the random number happens to be 1057 and the total data to be sent are 6000 bytes, the bytes are numbered from 1057 to 7056.

- **Sequence Number**
  - After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent.
  - The sequence number for each segment is the number of the first byte carried in that segment.

- **Acknowledgment Number**
  - Communication in TCP is full duplex; when a connection is established, both parties can send and receive data at the same time.
  - Each party numbers the bytes, usually with a different starting byte number.
  - The sequence number in each direction shows the number of the first byte carried by the segment.
  - Each party also uses an acknowledgment number to confirm the bytes it has received.
  - However, the acknowledgment number defines the number of the next byte that the party expects to receive.
  - In addition, the acknowledgment number is cumulative, which means that the party takes the number of the last byte that it has received, safe and sound, adds 1 to it, and announces this sum as the acknowledgment number.
  - The term cumulative here means that if a party uses 5643 as an acknowledgment number, it has received all bytes from the beginning up to 5642.

- **Flow Control**
  - TCP, unlike UDP, provides flow control.
  - The receiver of the data controls the amount of data that are to be sent by the sender.
  - This is done to prevent the receiver from being over-whelmed with data.
  - The numbering system allows TCP to use a byte-oriented flow control.
- **Error Control**
  - To provide reliable service, TCP implements an error control mechanism.
  - Although error control considers a segment as the unit of data for error detection (loss or corrupted segments), error control is byte-oriented.
- **Congestion Control**
  - TCP, unlike UDP, takes into account congestion in the network.
  - The amount of data sent by a sender is not only controlled by the receiver (flow control), but is also detennined by the level of congestion in the network.

- **Segment**
  - A packet in TCP is called a segment.
  - **Format**
  - The format of a segment is shown in Figure.

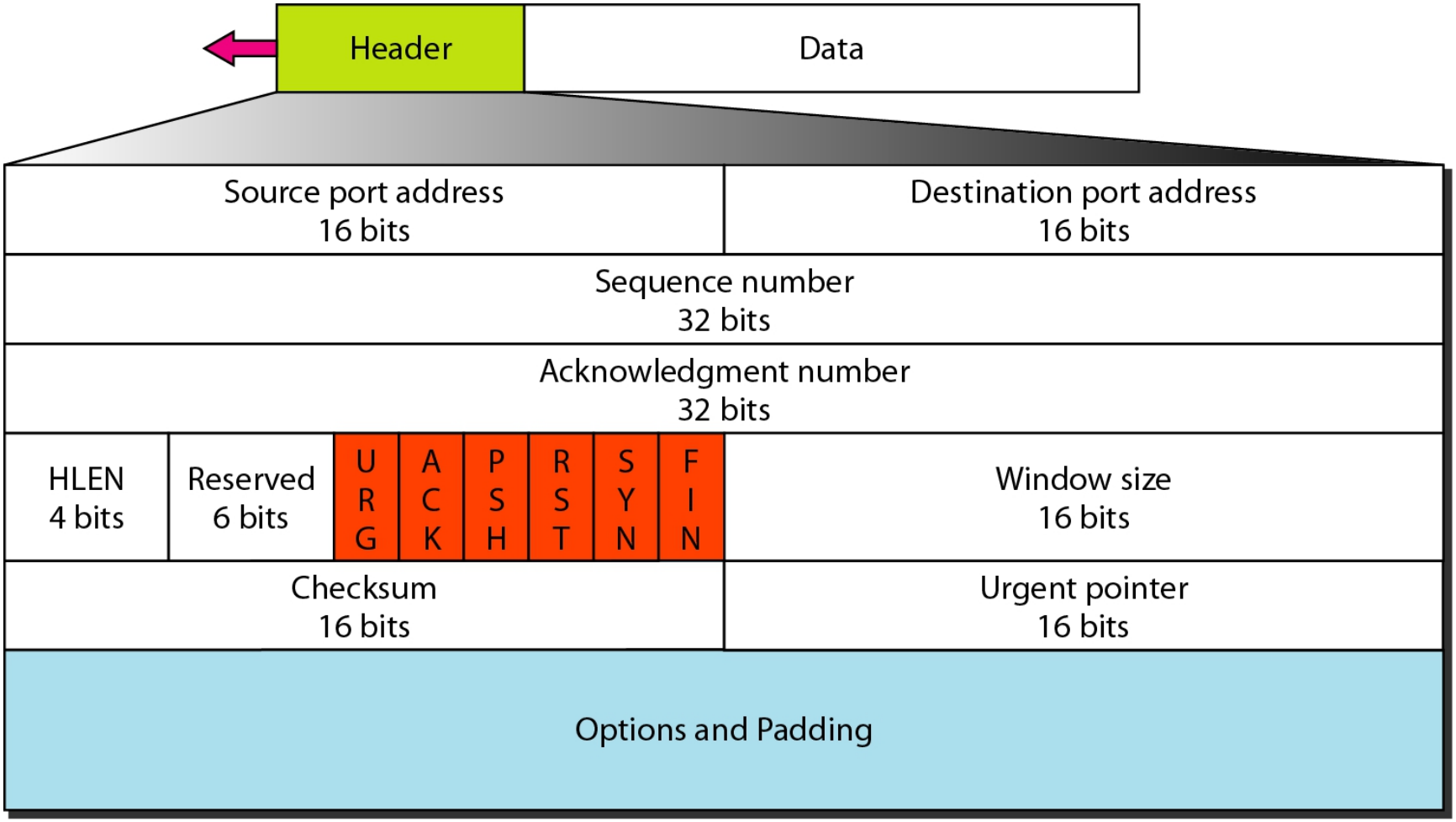


Figure 23.16 TCP segment format

- The segment consists of a 20- to 60-byte header, followed by data from the application program.
- The header is 20 bytes if there are no options and up to 60 bytes if it contains options.
- **Source port address.**
- This is a 16-bit field that defines the port number of the application program in the host that is sending the segment.
- This serves the same purpose as the source port address in the UDP header.
- **Destination port address**.
- This is a 16-bit field that defines the port number of the application program in the host that is receiving the segment.
- This serves the same purpose as the destination port address in the UDP header.

- **Sequence number.**
- This 32-bit field defines the number assigned to the first byte of data contained in this segment.
- TCP is a stream transport protocol.
- To ensure connectivity, each byte to be transmitted is numbered.
- The sequence number tells the destination which byte in this sequence comprises the first byte in the seg-ment.
- During connection establishment, each party uses a random number generator to create an initial sequence number (ISN), which is usually different in each direction.
- **Acknowledgment number.**
- This 32-bit field defines the byte number that the receiver of the segment is expecting to receive from the other party.
- If the receiver of the segment has successfully received byte number x from the other party, it defines x + I as the acknowledgment number.
- Acknowledgment and data can be piggybacked together.

- **Header length.**
- This 4-bit field indicates the number of 4-byte words in the TCP header.
- The length of the header can be between 20 and 60 bytes.
- Therefore, the value of this field can be between 5 (5 x 4 = 20) and 15 (15 x 4 = 60).
- **Reserved.**
- This is a 6-bit field reserved for future use.
- **Control.**
- This field defines 6 different control bits or flags as shown in Figure.
- One or more of these bits can be set at a time.
- These bits enable flow control, connection establishment and termination, connection abortion, and the mode of data transfer in TCP.
- A brief description of each bit is shown in Table.

Figure 23.17 ***Control field***

URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection

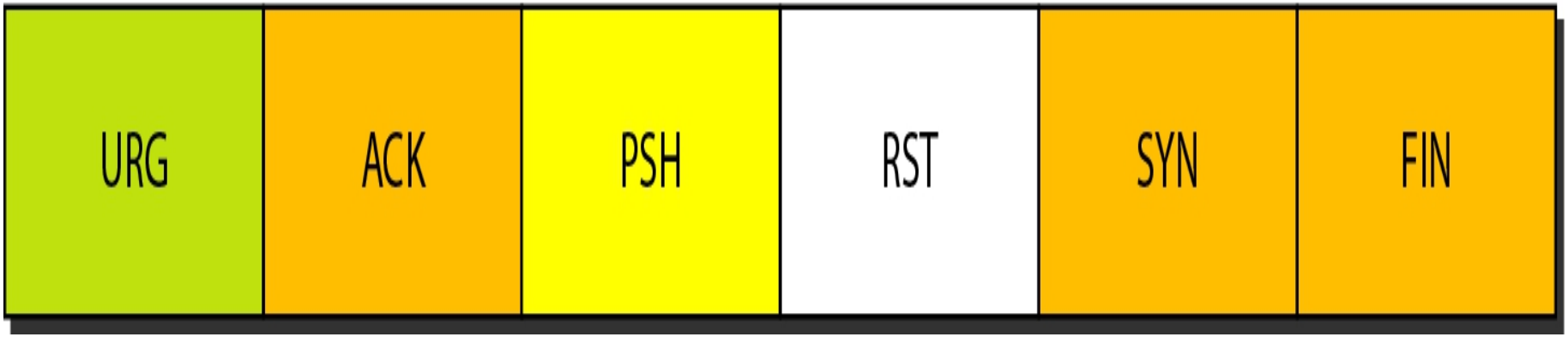


Table 23.3 ***Description of flags in the control field***

Flag	*Description*
URG	The value of the urgent pointer field is valid.
ACK	The value of the acknowledgment field is valid.
PSH	Push the data.
RST	Reset the connection.
SYN	Synchronize sequence numbers during connection.
FIN	Terminate the connection.

- **Window size.**
- This field defines the size of the window, in bytes, that the other party must maintain.
- Note that the length of this field is 16 bits, which means that the maximum size of the window is 65,535 bytes.
- This value is normally referred to as the receiving window (rwnd) and is determined by the receiver.
- The sender must obey the dictation of the receiver in this case.
- **Checksum.**
- This 16-bit field contains the checksum.
- The calculation of the checksum for TCP follows the same procedure as the one described for UDP.
- However, the inclusion of the checksum in the UDP datagram is optional, whereas the inclusion of the checksum for TCP is mandatory.

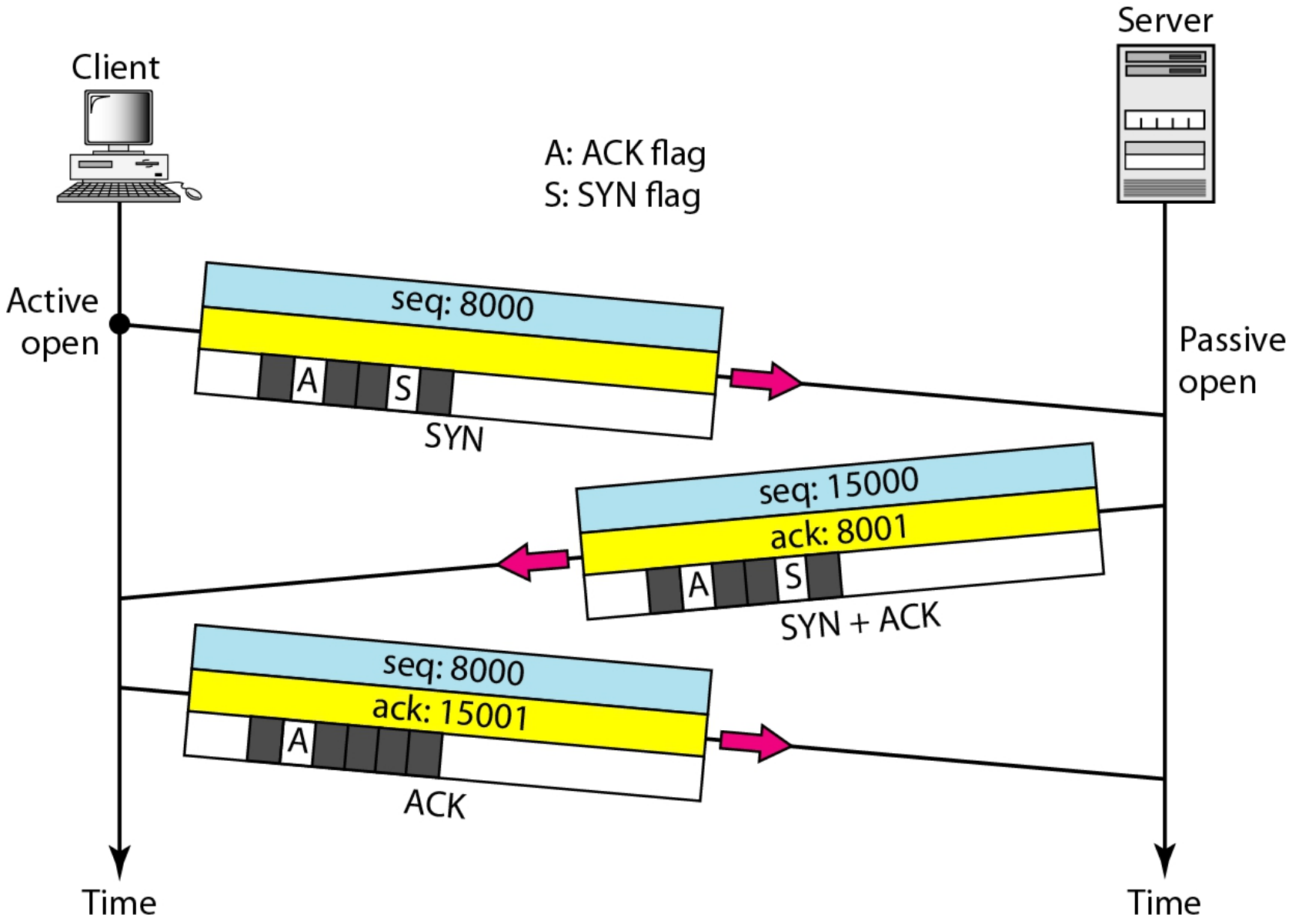
- **Urgent pointer**.
- This l6-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data.
- It defines the number that must be added to the sequence number to obtain the number of the last urgent byte in the data section of the segment.
- **Options.**
- There can be up to 40 bytes of optional information in the TCP header.

- **A TCP Connection**
  - TCP is connection-oriented.
  - A connection-oriented transport protocol establishes a virtual path between the source and destination.
  - All the segments belonging to a message are then sent over this virtual path.
  - Using a single virtual pathway for the entire message facilitates the acknowledgment process as well as retransmission of damaged or lost frames.
  - In TCP, connection-oriented transmission requires three phases:
    - connection establishment,
    - data transfer, and
    - connection termination.

- **Connection Establishment**
  - TCP transmits data in full-duplex mode.
  - When two TCPs in two machines are connected, they are able to send segments to each other simultaneously.
  - This implies that each party must initialize communication and get approval from the other party before any data are transferred.
  - **Three-Way Handshaking**
  - The connection establishment in TCP is called three- way handshaking.
  - In our example, an application program, called the client, wants to make a connection with another application program, called the server, using TCP as the transport layer protocol.
  - The process starts with the server.
  - The server program tells its TCP that it is ready to accept a connection.

- **This is called a request for a passive open.**
- Although the server TCP is ready to accept any connection from any machine in the world, it cannot make the connection itself.
- **The client program issues a request for an active open.**
- A client that wishes to connect to an open server tells its TCP that it needs to be connected to that particular server.
- TCP can now start the three-way handshaking process as shown in Figure.

Figure 23.18 ***Connection establishment using three-way handshaking***



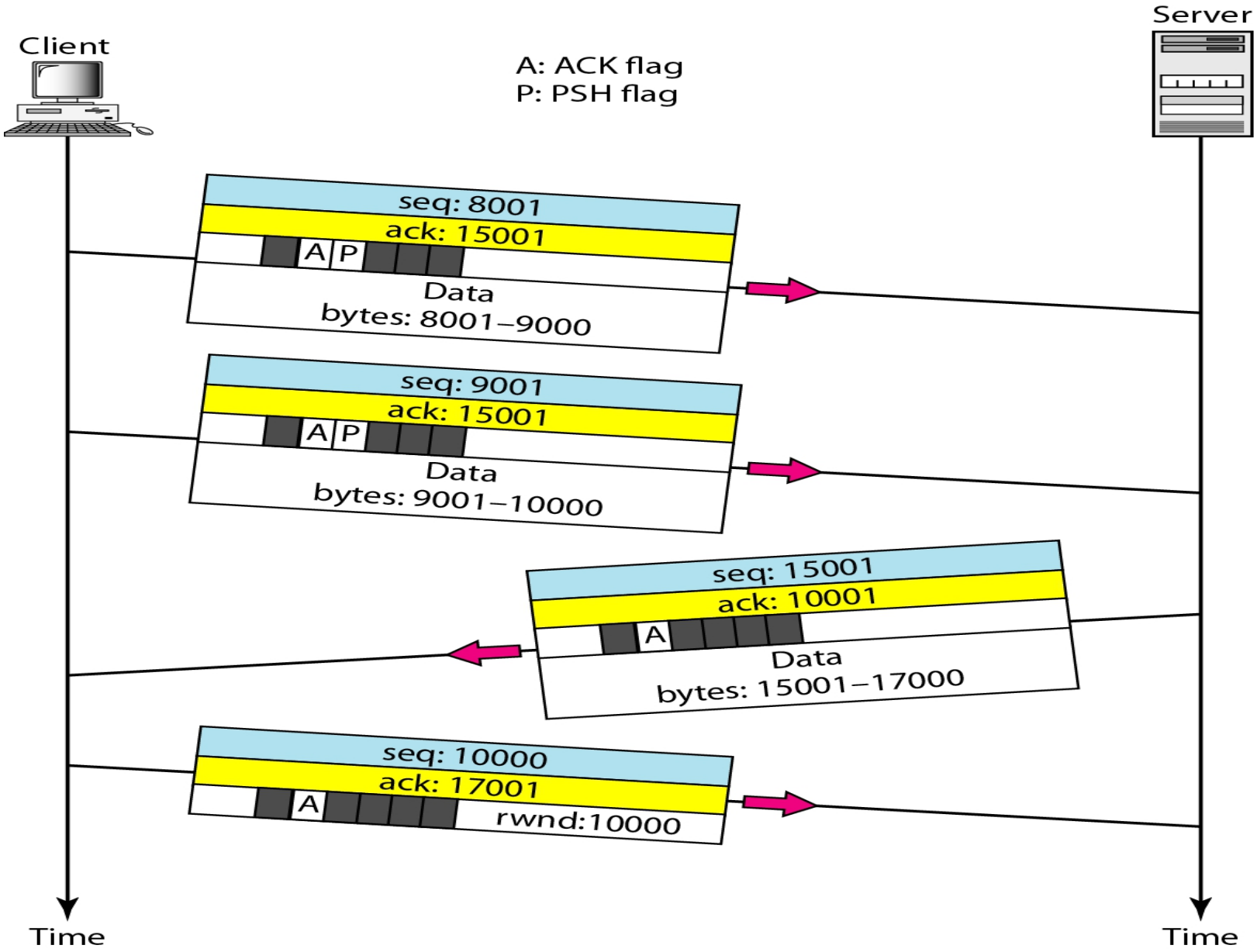
- The three steps in this phase are as follows.
- **1.** The client sends the first segment, a SYN segment, in which only the SYN flag is set.
- This segment is for synchronization of sequence numbers.
- It consumes one sequence number.
- When the data transfer starts, the sequence number is incremented by 1.
- We can say that the SYN segment carries no real data, but we can think of it as containing 1 imaginary byte.
- **2.** The server sends the second segment, a SYN + ACK segment, with 2 flag bits set: SYN and ACK.
- This segment has a dual purpose.
- It is a SYN segment for communication in the other direction and serves as the acknowledgment for the SYN segment.
- It consumes one sequence number.

- **3.** The client sends the third segment.
- This is just an ACK segment.
- It acknowledges the receipt of the second segment with the ACK flag and acknowledgment number field.
- Note that the sequence number in this segment is the same as the one in the SYN segment; the ACK segment does not consume any sequence numbers.

- **Simultaneous Open**
  - A rare situation, called a simultaneous open, may occur when both processes issue an active open.
  - In this case, both TCPs transmit a SYN + ACK segment to each other, and one single connection is established between them.

- **Data Transfer**
  - After connection is established, bidirectional data transfer can take place.
  - The client and server can both send data and acknowledgments.
  - Data traveling in the same direction as an acknowledgment are carried on the same segment.
  - The acknowledgment is piggybacked with the data.
  - Figure shows an example.
  - In this example, after connection is established (not shown in the figure), the client sends 2000 bytes of data in two segments.
  - The server then sends 2000 bytes in one segment.
  - The client sends one more segment.
  - The first three segments carry both data and acknowledgment, but the last segment carries only an acknowledgment because there are no more data to be sent.
  - Note the values of the sequence and acknowledgment numbers.
  - The data segments sent by the client have the PSH (push) flag set so that the server TCP knows to deliver data to the server process as soon as they are received.

Figure 23.19 *Data transfer*

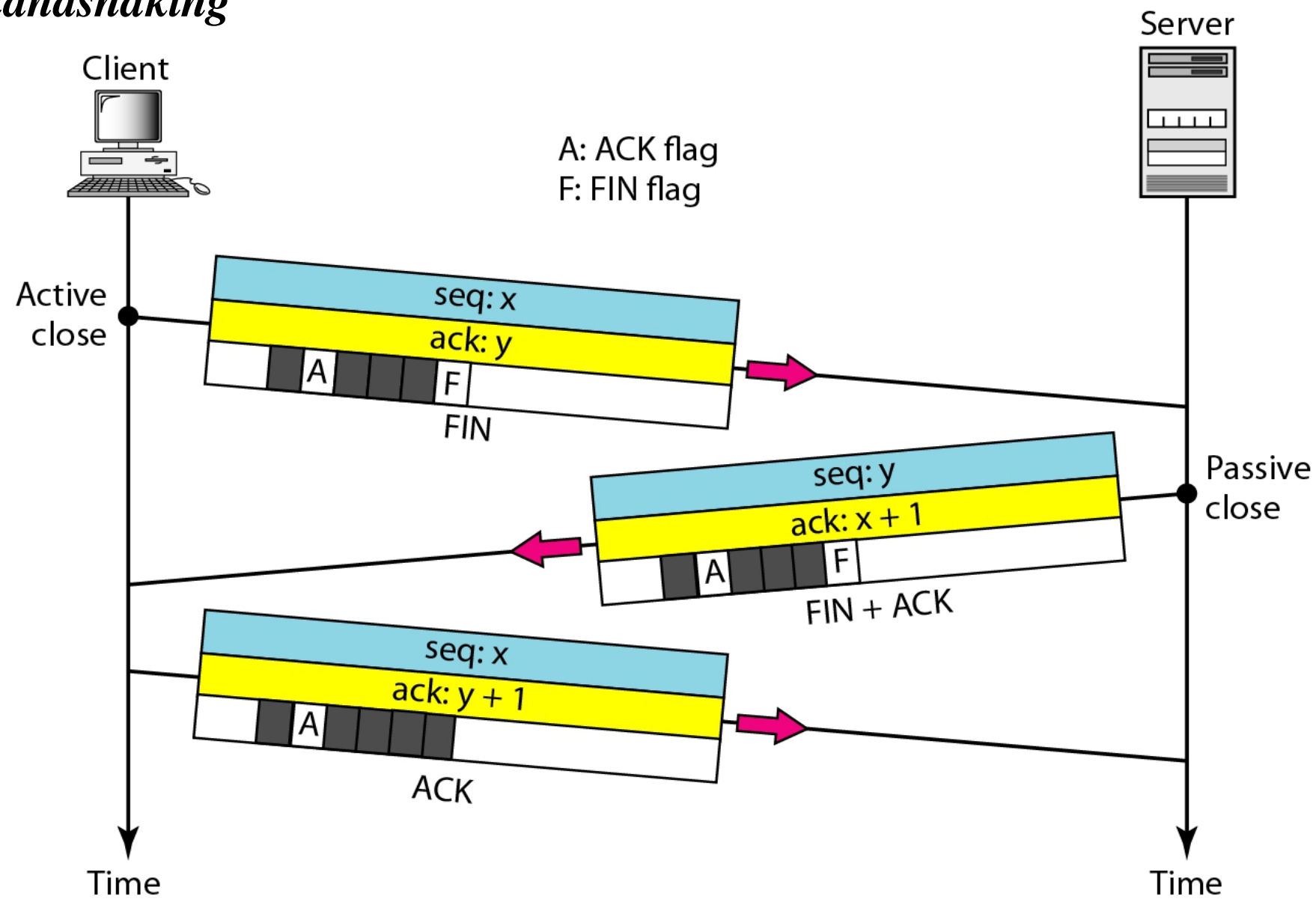


- **Pushing Data**
  - We saw that the sending TCP uses a buffer to store the stream of data coming from the sending application program.
  - The sending TCP can select the segment size.
  - The receiving TCP also buffers the data when they arrive and delivers them to the application program when the application program is ready or when it is convenient for the receiving TCP.
  - This type of flexibility increases the efficiency of TCP.
- **Urgent Data**
  - TCP is a stream-oriented protocol.
  - This means that the data are presented from the application program to TCP as a stream of bytes.
  - Each byte of data has a position in the stream.
  - However, on occasion an application program needs to send urgent bytes.

- The solution is to send a segment with the URG bit set.
- The sending application program tells the sending TCP that the piece of data is urgent.
- The sending TCP creates a segment and inserts the urgent data at the beginning of the segment.
- The rest of the segment can contain normal data from the buffer.
- The urgent pointer field in the header defines the end of the urgent data and the start of normal data.
- When the receiving TCP receives a segment with the URG bit set, it extracts the urgent data from the segment, using the value of the urgent pointer, and delivers them, out of order, to the receiving application program.

- **Connection Termination**
  - Any of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client.
  - Most implementations today allow two options for connection termination:
    - three-way handshaking and
    - four-way handshaking with a half-close option.
- **Three-Way Handshaking**
  - Most implementations today allow three-way handshaking for connection termination as shown in Figure .

Figure 23.20 ***Connection termination using three-way handshaking***

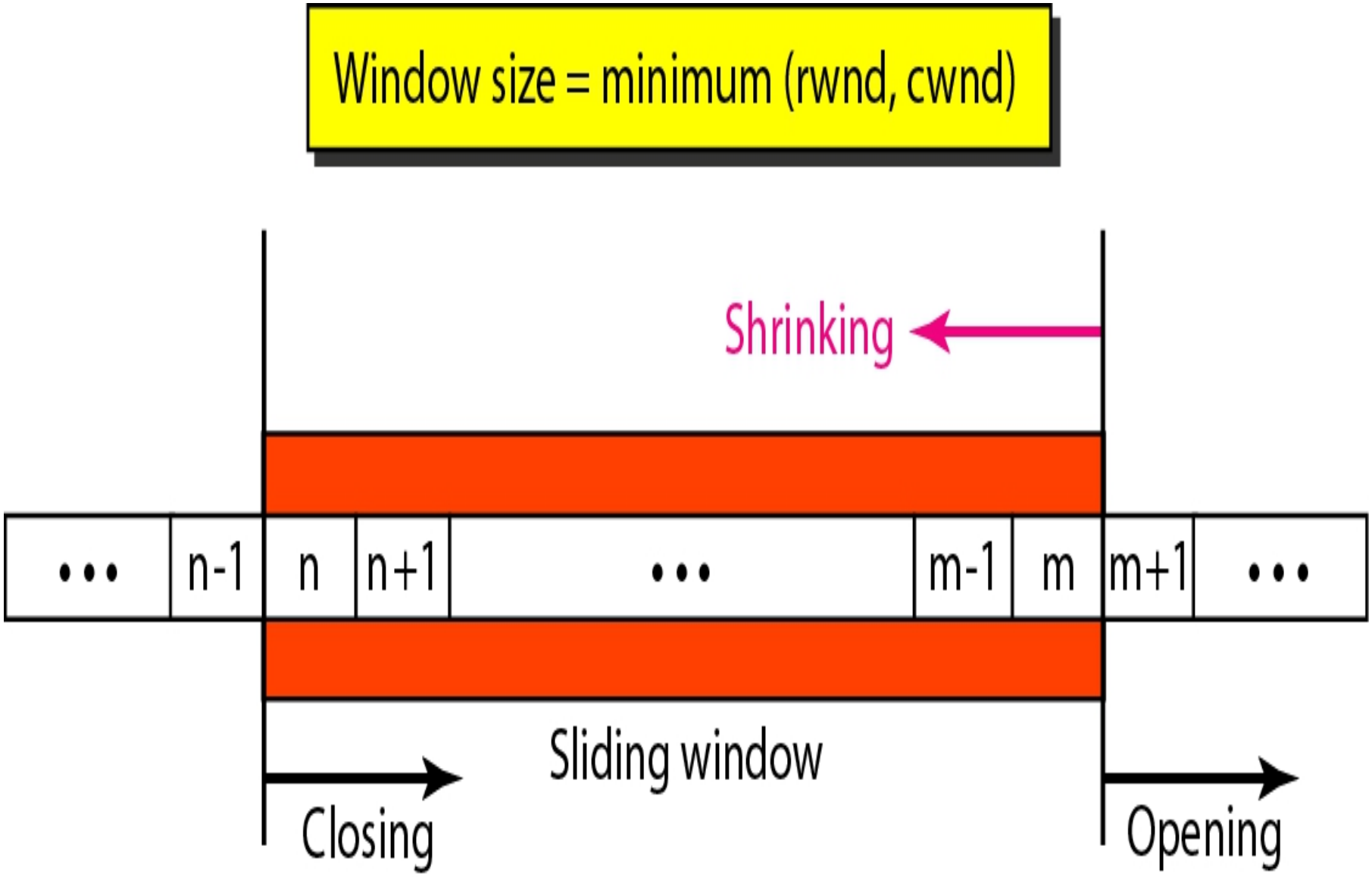


- **1.** In a normal situation, the client TCP, after receiving a close command from the client process, sends the first segment, a FIN segment in which the FIN flag is set.
- Note that a FIN segment can include the last chunk of data sent by the client, or it can be just a control segment as shown in Figure.
- If it is only a control segment, it consumes only one sequence number.
- **2.** The server TCP, after receiving the FIN segment, informs its process of the situation and sends the second segment, a FIN + ACK segment, to confirm the receipt of the FIN segment from the client and at the same time to announce the closing of the connection in the other direction.
- This segment can also contain the last chunk of data from the server.
- If it does not carry data, it consumes only one sequence number.

- **3.** The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server.
- This segment contains the acknowledgment number, which is 1 plus the sequence number received in the FIN segment from the server.
- This segment cannot carry data and consumes no sequence numbers.

- **TCP Flow Control**
  - TCP uses a sliding window, to handle flow control.
  - The sliding window protocol used by TCP, however, is something between the Go-Back-N and Selective Repeat sliding window.
  - The sliding window protocol in TCP looks like the Go-Back-N protocol because it does not use NAKs; it looks like Selective Repeat because the receiver holds the out-of-order segments until the missing ones arrive.
  - There are two big differences between this sliding window and the one we used at the data link layer.
    - First, the sliding window ofTCP is byte-oriented; the one we discussed in the data link layer is frame-oriented.
    - Second, the TCP's sliding window is of variable size; the one we discussed in the data link layer was of fixed size.

Figure 23.22 ***Sliding window***



- The imaginary window has two walls: one left and one right.
- The window is opened, closed, or shrunk.
- These three activities, as we will see, are in the control of the receiver (and depend on congestion in the network), not the sender.
- The sender must obey the commands of the receiver in this matter.
- **Opening a window** means moving the right wall to the right.
- This allows more new bytes in the buffer that are eligible for sending.
- **Closing the window means** moving the left wall to the right.
- This means that some bytes have been acknowledged and the sender need not worry about them anymore.

- **Shrinking the window** means moving the right wall to the left.
- This is strongly discouraged and not allowed in some implementations because it means revoking the eligibility of some bytes for sending.
- This is a problem if the sender has already sent these bytes.
- Note that the left wall cannot move to the left because this would revoke some of the previously sent acknowledgments.
- **The size of the window** at one end is determined by the lesser of two values: receiver window (rwnd) or congestion window (cwnd).
- The receiver window is the value advertised by the opposite end in a segment containing acknowledgment.
- It is the number of bytes the other end can accept before its buffer overflows and data are discarded.
- The congestion window is a value determined by the network to avoid congestion.

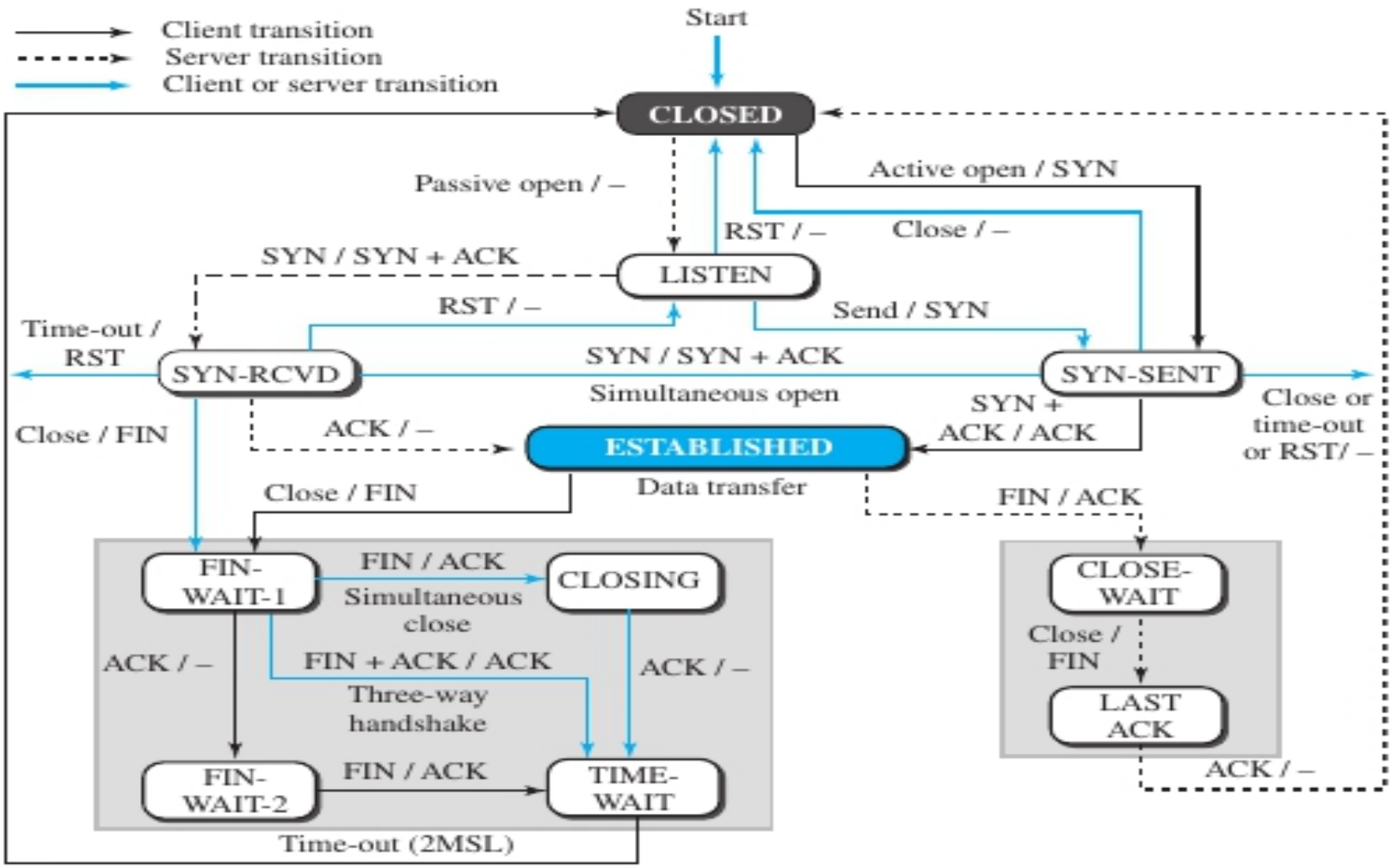
- **Example 23.4**
- **What is the value of the receiver window (rwnd) for host A if the receiver, host B, has a buffer size of 5000 bytes and 1000 bytes of received and unprocessed data?**
- **Solution**
- **The value of rwnd = 5000 – 1000 = 4000. Host B can receive only 4000 bytes of data before overflowing its buffer. Host B advertises this value in its next segment to A.**

- **Example 23.5**
- **What is the size of the window for host A if the value of rwnd is 3000 bytes and the value of cwnd is 3500 bytes?**
- **Solution**
- **The size of the window is the smaller of rwnd and cwnd, which is 3000 bytes.**

- **Some points about TCP sliding windows:**
- **The size of the window is the lesser of rwnd and cwnd.**
- **The source does not have to send a full window's worth of data.**
- **The window can be opened or closed by the receiver, but should not be shrunk.**
- **The destination can send an acknowledgment at any time as long as it does not result in a shrinking window.**
- **The receiver can temporarily shut down the window; the sender, however, can always send a segment of 1 byte after the window is shut down.**

- **TCP State Transition**
- To keep track of all the different events happening during connection establishment, connection termination, and data transfer, TCP is specified as the finite state machine (FSM) as shown in Figure 24.14.

Figure 24.14 *State transition diagram*



The state marked *ESTABLISHED* in the FSM is in fact two different sets of states that the client and server undergo to transfer data.

- **TCP State Transition**
- The figure shows the two FSMs used by the TCP client and server combined in one diagram.
- The rounded-corner rectangles represent the states.
- The transition from one state to another is shown using directed lines.
- Each line has two strings separated by a slash.
- The first string is the input, what TCP receives.
- The second is the output, what TCP sends.
- The dotted black lines in the figure represent the transition that a server normally goes through; the solid black lines show the transitions that a client normally goes through.

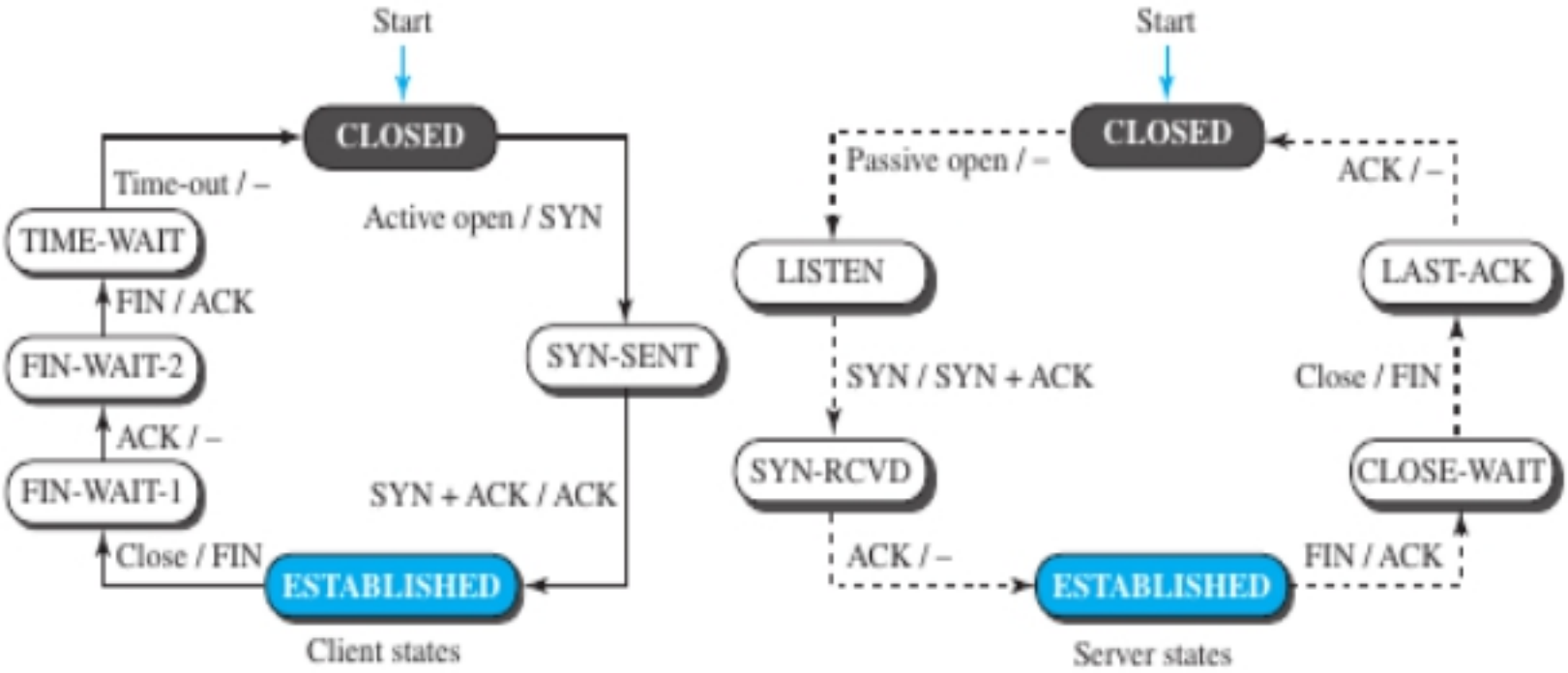
- **TCP State Transition**
- However, in some situations, a server transitions through a solid line or a client transitions through a dotted line.
- The colored lines show special situations.
- Note that the rounded-corner rectangle marked ESTABLISHED is in fact two sets of states, a set for the client and another for the server, that are used for flow and error control, as explained later in the chapter.
- We will discuss some timers mentioned in the figure, including the 2MSL timer, at the end of the chapter.
- We use several scenarios based on Figure 24.14 and show the part of the figure in each case.

Table 24.2 *States for TCP*

State	*Description*
CLOSED	No connection exists
LISTEN	Passive open received; waiting for SYN
SYN-SENT	SYN sent; waiting for ACK
SYN-RCVD	SYN + ACK sent; waiting for ACK
ESTABLISHED	Connection established; data transfer in progress
FIN-WAIT-1	First FIN sent; waiting for ACK
FIN-WAIT-2	ACK to first FIN received; waiting for second FIN
CLOSE-WAIT	First FIN received, ACK sent; waiting for application to close
TIME-WAIT	Second FIN received, ACK sent; waiting for 2MSL time-out
LAST-ACK	Second FIN sent; waiting for ACK
CLOSING	Both sides decided to close simultaneously

- **A Half-Close Scenario**
- Figure 24.15 shows the state transition diagram for this scenario.

Figure 24.15 *Transition diagram with half-close connection termination*



- **TCP State Transition**
- The client process issues an active open command to its TCP to request a connection to a specific socket address.
- TCP sends a SYN segment and moves to the SYN-SENT state.
- After receiving the SYN + ACK segment, TCP sends an ACK segment and goes to the ESTABLISHED state.
- Data are transferred, possibly in both directions, and acknowledged.
- When the client process has no more data to send, it issues a command called an active close.
- The TCP sends a FIN segment and goes to the FIN-WAIT-1 state.

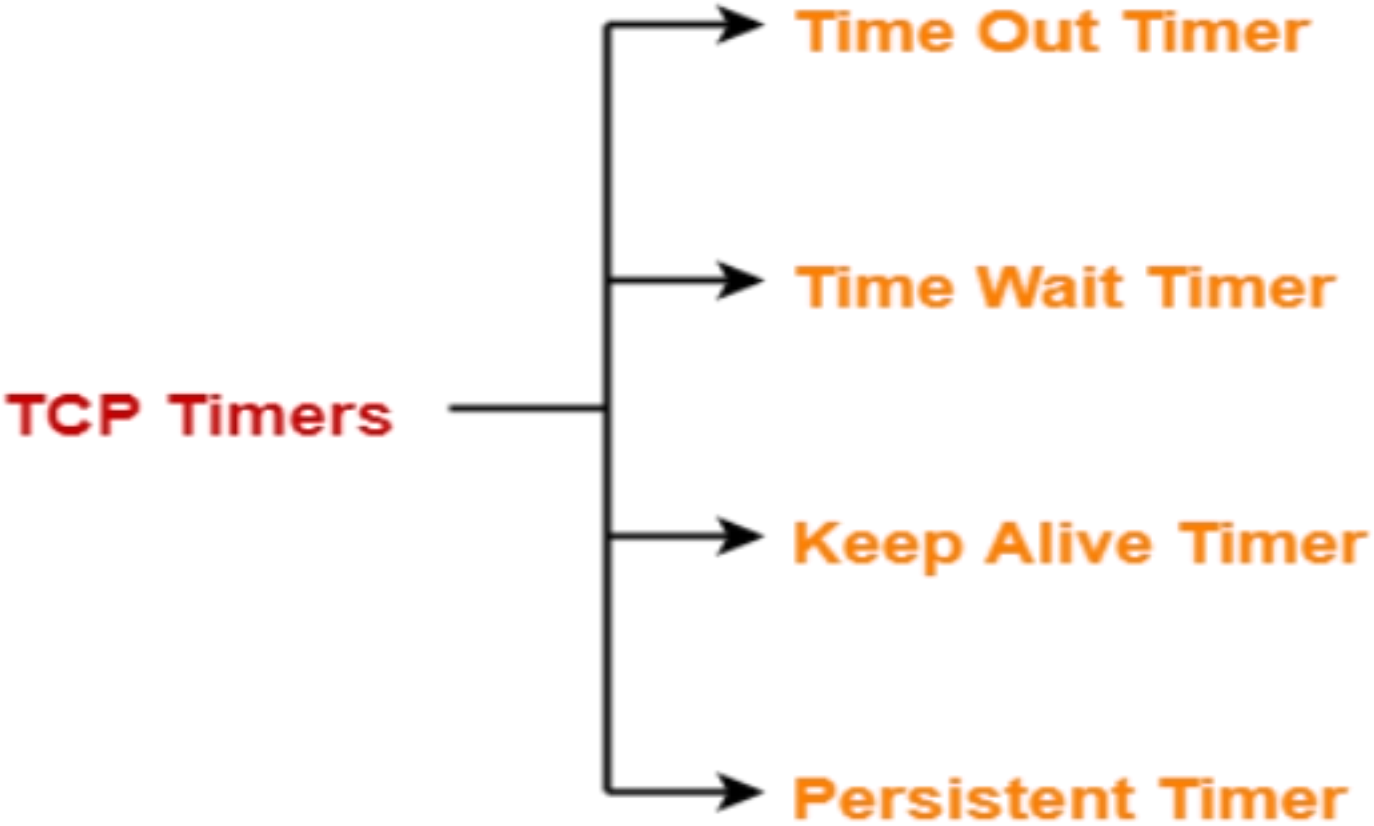
- **TCP State Transition**
- When it receives the ACK segment, it goes to the FIN-WAIT-2 state.
- When the client receives a FIN segment, it sends an ACK segment and goes to the TIME-WAIT state.
- The client remains in this state for 2 MSL seconds.
- When the corresponding timer expires, the client goes to the CLOSED state.

- **TCP State Transition**
- The server process issues a passive open command.
- The server TCP goes to the LISTEN state and remains there passively until it receives a SYN segment.
- The TCP then sends a SYN + ACK segment and goes to the SYN-RCVD state, waiting for the client to send an ACK segment.
- After receiving the ACK segment, TCP goes to the ESTABLISHED state, where data transfer can take place.
- TCP remains in this state until it receives a FIN segment from the client signifying that there are no more data to be exchanged and the connection can be closed.

- **TCP State Transition**
- The server, upon receiving the FIN segment, sends all queued data to the server with a virtual EOF marker, which means that the connection must be closed.
- It sends an ACK segment and goes to the CLOSE-WAIT state, but postpones acknowledging the FIN segment received from the client until it receives a passive close command from its process.
- After receiving the passive close command, the server sends a FIN segment to the client and goes to the LAST-ACK state, waiting for the final ACK.
- When the ACK segment is received from the client, the server goes to the CLOSE state.

- **TCP State Transition**
- The server, upon receiving the FIN segment, sends all queued data to the server with a virtual EOF marker, which means that the connection must be closed.
- It sends an ACK segment and goes to the CLOSE-WAIT state, but postpones acknowledging the FIN segment received from the client until it receives a passive close command from its process.
- After receiving the passive close command, the server sends a FIN segment to the client and goes to the LAST-ACK state, waiting for the final ACK.
- When the ACK segment is received from the client, the server goes to the CLOSE state.

- **TCP Timers**
- Timers used by TCP to avoid excessive delays during communication are called as TCP Timers.



- **Time Out Timer**
- TCP uses a time out timer for retransmission of lost segments.
- Sender starts a time out timer after transmitting a TCP segment to the receiver.
- If sender receives an acknowledgement before the timer goes off, it stops the timer.
- If sender does not receives any acknowledgement and the timer goes off, then TCP Retransmission occurs.
- Sender retransmits the same segment and resets the timer.
- The value of time out timer is dynamic and changes with the amount of traffic in the network.
- Time out timer is also called as Retransmission Timer.

- **Time Wait Timer**
- TCP uses a time wait timer during connection termination.
- Sender starts the time wait timer after sending the ACK for the second FIN segment.
- It allows to resend the final acknowledgement if it gets lost.
- It prevents the just closed port from reopening again quickly to some other application.
- It ensures that all the segments heading towards the just closed port are discarded.
- The value of time wait timer is usually set to twice the lifetime of a TCP segment.

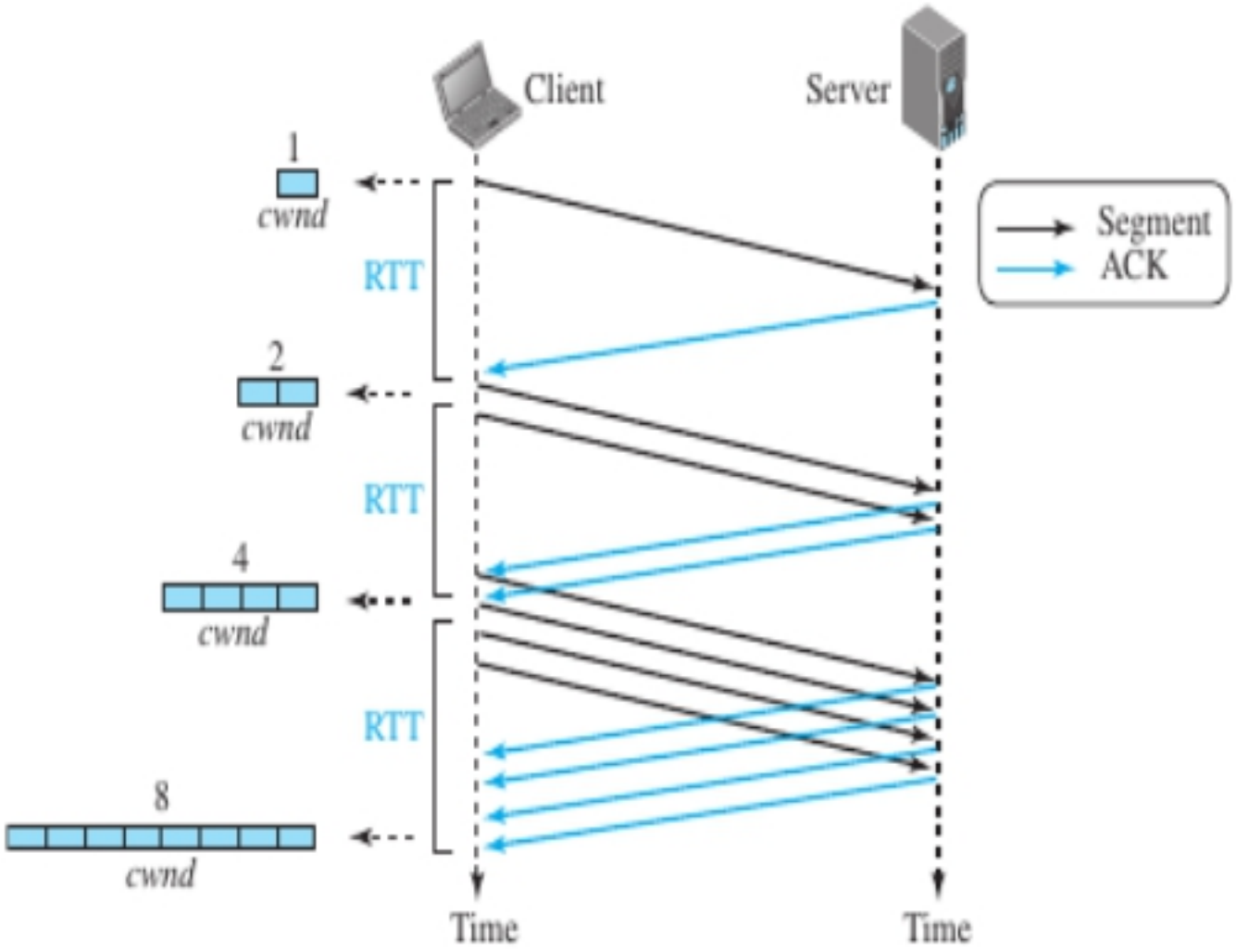
- **Keep Alive Timer**
- TCP uses a keep alive timer to prevent long idle TCP connections.
- Each time server hears from the client, it resets the keep alive timer to 2 hours.
- If server does not hear from the client for 2 hours, it sends 10 probe segments to the client.
- These probe segments are sent at a gap of 75 seconds.
- If server receives no response after sending 10 probe segments, it assumes that the client is down.
- Then, server terminates the connection automatically.

- **Persistent Timer**
- TCP uses a persistent timer to deal with a zero-widow-size deadlock situation.
- It keeps the window size information flowing even if the other end closes its receiver window.
- Consider the following situation-
- Sender receives an acknowledgment from the receiver with zero window size.
- This indicates the sender to wait.
- Later, receiver updates the window size and and sends the segment with the update to the sender.
- This segment gets lost.
- Now, both sender and receiver keeps waiting for each other to

- **Persistent Timer**
- Now, both sender and receiver keeps waiting for each other to do something.
- To deal with such a situation, TCP uses a persistent timer.

- **TCP Congestion Control: Slow Start.**
- **Slow Start: Exponential Increase**
- The slow-start algorithm is based on the idea that the size of the congestion window (cwnd) starts with one maximum segment size (MSS), but it increases one MSS each time an acknowledgment arrives.
- As we discussed before, the MSS is a value negotiated during the connection establishment, using an option of the same name.
- The name of this algorithm is misleading; the algorithm starts slowly, but grows exponentially.
- To show the idea, let us look at Figure 24.29.

Figure 24.29 *Slow start, exponential increase*



- We assume that rwnd is much larger than cwnd, so that the sender window size always equals cwnd.
- We also assume that each segment is of the same size and carries MSS bytes.
- For simplicity, we also ignore the delayed-ACK policy and assume that each segment is acknowledged individually.
- The sender starts with cwnd = 1.
- This means that the sender can send only one segment.
- After the first ACK arrives, the acknowledged segment is purged from the window, which means there is now one empty segment slot in the window.

- The size of the congestion window is also increased by 1 because the arrival of the acknowledgment is a good sign that there is no congestion in the network.
- The size of the window is now 2.
- After sending two segments and receiving two individual acknowledgments for them, the size of the congestion window now becomes 4, and so on.
- In other words, the size of the congestion window in this algorithm is a function of the number of ACKs arrived and can be determined as follows.

If an ACK arrives, $cwnd = cwnd + 1$.

If we look at the size of the *cwnd* in terms of round-trip times (RTTs), we find that the growth rate is exponential in terms of each round trip time, which is a very aggressive approach:

Start	→	$cwnd = 1 \rightarrow 2^0$
After 1 RTT	→	$cwnd = cwnd + 1 = 1 + 1 = 2 \rightarrow 2^1$
After 2 RTT	→	$cwnd = cwnd + 2 = 2 + 2 = 4 \rightarrow 2^2$
After 3 RTT	→	$cwnd = cwnd + 4 = 4 + 4 = 8 \rightarrow 2^3$

A slow start cannot continue indefinitely. There must be a threshold to stop this phase. The sender keeps track of a variable named *ssthresh* (slow-start threshold). When the size of the window in bytes reaches this threshold, slow start stops and the next phase starts.

In the slow-start algorithm, the size of the congestion window increases exponentially until it reaches a threshold.