Unit-4
The Transport Layer

The transport layer in the TCP/IP suite is located between the application layer and the network layer. It provides services to the application layer and receives services from the network layer. The transport layer acts as a connection between a client program and a server program, a process-to-process connection.
The transport layer is the heart of the TCP/IP protocol suite; it is the end-to-end logical vehicle for transferring data from one point to another in the Internet.

A transport-layer protocol provides for **logical communication** between application processes running on different hosts. By *logical communication*, it is as if the hosts running the processes were directly connected.
On the sending side, the transport layer converts the application-layer messages it receives from a sending application process into transport-layer packets, known as transport-layer **segments**.

1. Connectionless Transport: UDP

UDP is a connectionless protocol. No connection needs to be established between the source and destination before you transmit data. It is an **unreliable and connectionless protocol.**

As many applications are better suited for UDP for the following reasons:

1. Finer application level control over what data is sent and when
2. No connection establishment
3. No connection state
4. Small packet header overhead

Finer application level control over what data is sent and when: As soon as an application process passes data to UDP, UDP will package the data inside a UDP segment and immediately pass the segment to the network layer.
Real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss, TCP's service model is not particularly well matched to these applications' needs. These applications can use UDP.

No connection establishment: TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP – DNS would be much slower if it ran over TCP.

No connection state: UDP does not maintain connection state and does not track any of these parameters. For this reason, a server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP.

Scanned with OKEN Scanner

Small packet header overhead: The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead.

UDP segment structure

The Internet protocol suite supports a connectionless transport protocol called **UDP** ***(User Datagram Protocol)***. UDP provides a way for applications to send encapsulated IP datagrams without having to establish a connection.

UDP transmits **segments** consisting of an 8-byte header followed by the payload. The header is shown in Fig. The two ports serve to identify the endpoints within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port.

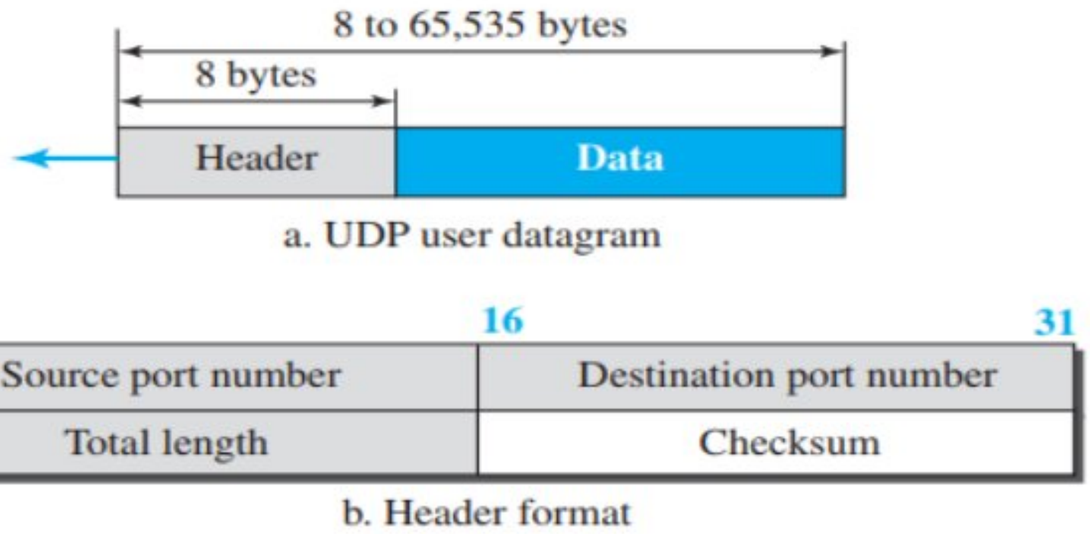


Fig: The UDP header

- ***Source port number:*** it is a 16-bits field. This field defines the source port number.
- ***Destination port number:*** it is a 16-bits field. This field defines the Destination port number.
- ***Total length:*** it is a 16-bits field. The total length field includes the 8-byte header and the data. The minimum length is 8 bytes, to cover the header. The maximum length is 65,535 bytes.
- ***Checksum:*** In checksum error detection scheme, the data is divided into k segments each of m bits.

 In the **source,** the segments are added using 1's complement arithmetic to get the sum. The sum is complemented to get the checksum.

 The checksum segment is sent along with the data segments.

 At the **destination**, all received segments are added using 1's complement arithmetic to get the sum. The sum is complemented.

 If the result is zero, the received data is accepted; otherwise discarded.

UDP checksum

The UDP checksum provides for **error detection**. That is, the checksum is used to determine whether bits within the UDP segment have been altered as it moved from source to destination. UDP at the sender side performs the 1s complement of the sum of all the 16-bit words in the segment, with any overflow encountered during the sum being wrapped around. This result is put in the checksum field of the UDP segment.

As an **example**, suppose that we have the following three 16-bit words:

Scanned with OKEN Scanner

0110011001100000
0101010101010101
1000111100001100

The sum of first two of these 16-bit words is

0110011001100000
0101010101010101
1011101110110101

Adding the third word to the above sum gives

1011101110110101
1000111100001100
0100101011000010

Note that this last addition had overflow, which was wrapped around. The 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s. Thus the 1s complement of the sum 0100101011000010 is 1011010100111101, which becomes the checksum.

At the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the packet, then clearly the sum at the receiver will be 1111111111111111. The sum is I's complemented If one of the bits is a 0, then we know that errors have been introduced into the packet.

UDP must provide error detection at the transport layer, on an end-end basis, if the end-end data transfer service is to provide error detection. This is an example of the celebrated **end-end principle** in system design.

2. The Internet Transport Protocols: TCP

TCP is a connection-oriented transport protocol. **TCP** ***(Transmission Control Protocol)*** was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork.

An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters. TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.

 The TCP Service Model: TCP service is obtained by both the sender and the receiver creating end points, called **sockets**.

Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a **port**.

For TCP service to be obtained, a connection must be explicitly established between a socket on one machine and a socket on another machine. The ***socket calls*** are listed in Fig.

A socket may be used for multiple connections at the same time. Connections are identified by the socket identifiers at both ends (socket1, socket2).

Scanned with OKEN Scanner

Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Fig: The socket primitives for TCP.

Port numbers : 1024 port numbers are reserved for standard services that can usually only be started by privileged users (e.g., root in UNIX systems). They are called well-known ports.

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Fig: Some assigned ports.

E.g: 4 * 512 bytes of data is to be transmitted.

For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as four 512-byte chunks, two 1024-byte chunks, one 2048-byte chunk, or some other way.

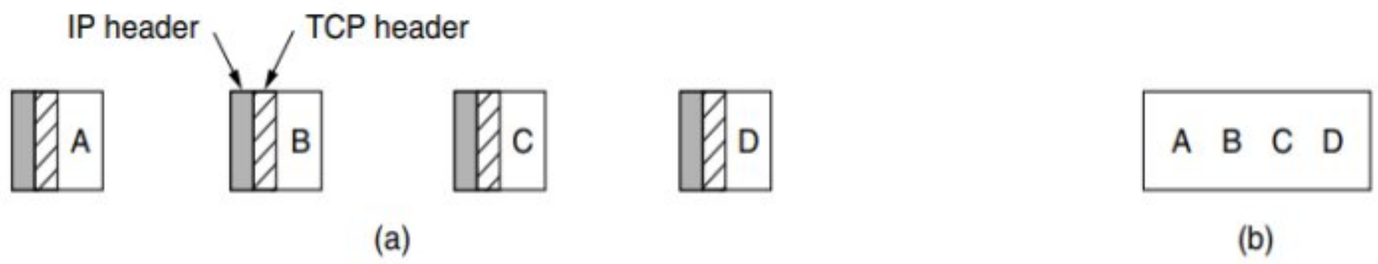


Fig: Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application

☒ **The TCP Protocol:** All TCP connections are full duplex and point to point i.e., multicasting or broadcasting is not supported. A TCP connection is a byte stream, not a message stream i.e., the data is delivered as chunks.

- A key feature of TCP is that every byte on a TCP connection has its own ***32-bit sequence number***.
- The basic protocol used by TCP entities is the ***sliding window protocol***. Separate 32-bit sequence numbers are carried on packets for the sliding window position in one direction and for acknowledgements in the reverse direction.
- When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment bearing an acknowledgement number equal to the next sequence number it expects to receive.
- If the sender's timer goes off before the acknowledgement is received, the sender

Scanned with OKEN Scanner

transmits the segment again.

- The sending and receiving TCP entities exchange data in the form of ***segments.***

A TCP segment consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes.

 The TCP Segment Header: The sending and receiving TCP entities exchange data in the form of ***segments.*** A TCP segment consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes.

Every segment begins with a fixed-format, 20-byte header. The fixed header may be followed by header options. Segments without any data are legal and are commonly used for acknowledgements and control messages.

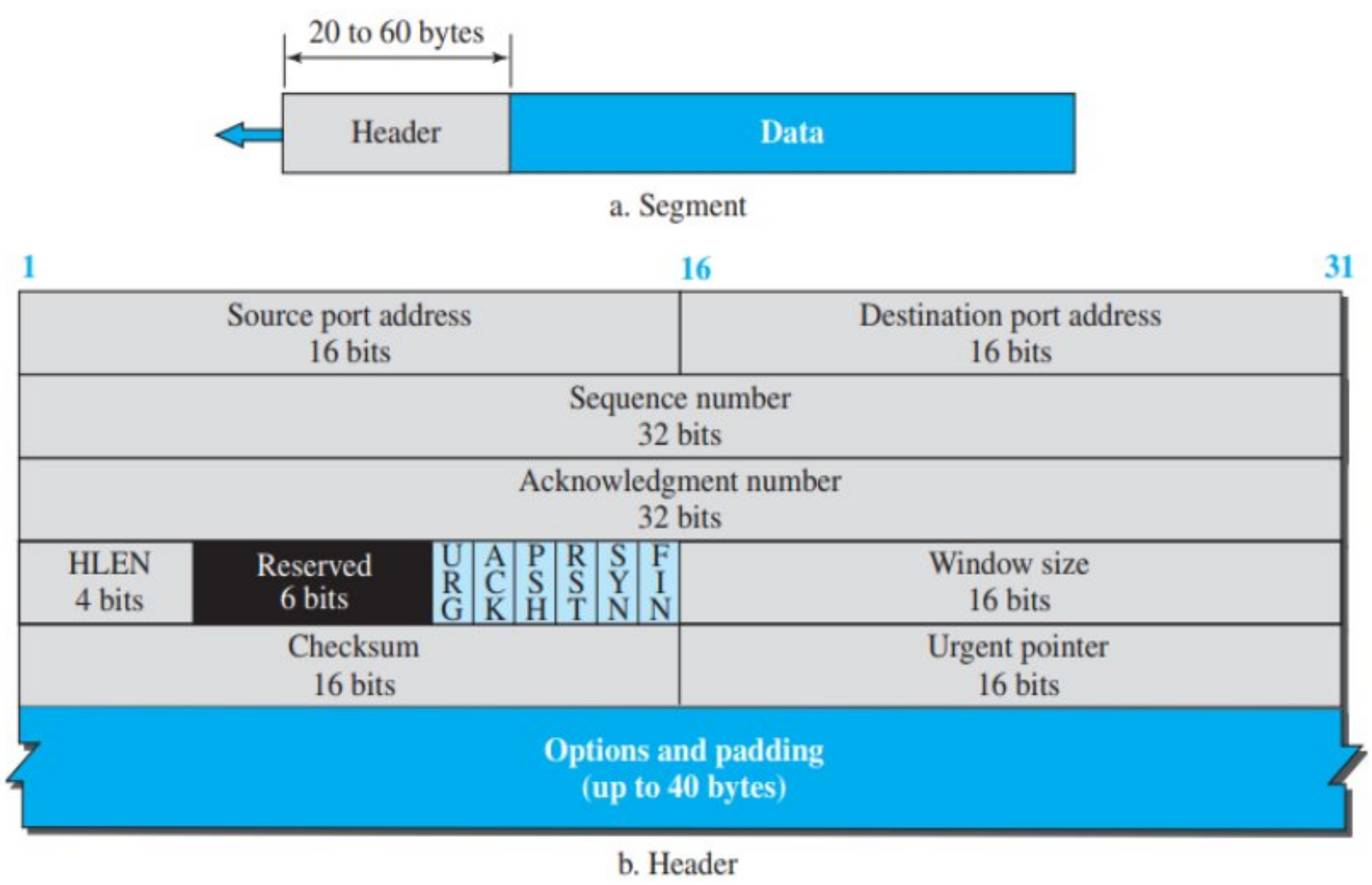


Fig: TCP segment format

- **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.
- **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.
- **Sequence number.** This 32-bit field defines the number assigned to the first byte of data contained in this segment. The sequence number tells the destination which byte in this sequence is the first byte in the segment.

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 returns x + 1 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 is always between 5 (5 × 4 = 20) and 15 (15 × 4 = 60).
- **Control.** This field defines 6 different control bits or flags, as shown in Fig. One or

Scanned with OKEN Scanner

more of these bits can be set at a time.

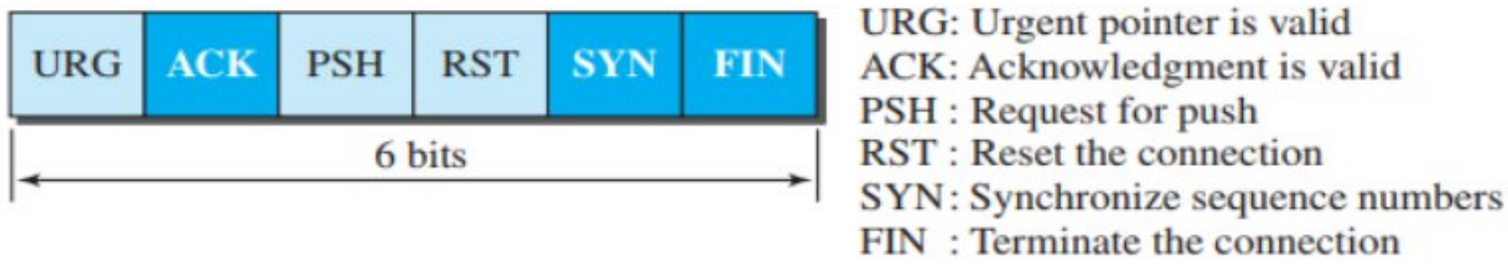


Fig: Control field

URG: It is set to 1 if URGENT pointer is in use, which indicates start of urgent data.
ACK: It is set to 1 to indicate that the acknowledgement number is valid.
PSH: Indicates pushed data
RST: It is used to reset a connection that has become confused due to reject an invalid segment or refuse an attempt to open a connection.
SYN: Used to establish connections.
FIN: Used to release a connection.

- **Window size.** This field defines the window size of the sending TCP in bytes.

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.

- **Checksum.** This 16-bit field contains the checksum.
  - In the **source,** the segments are added using 1's complement arithmetic to get the sum. The sum is complemented to get the checksum.
  - The checksum segment is sent along with the data segments.
  - At the **destination**, all received segments are added using 1's complement arithmetic to get the sum. The sum is complemented.
  - If the result is zero, the received data is accepted; otherwise discarded.
- **Urgent pointer.** This 16-bit field, which is valid only if the urgent flag is set, is used when the segment contains urgent data. It defines a value 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 logical path between the source and destination. All of the segments belonging to a message are then sent over this logical path.
In TCP, connection-oriented transmission requires ***three phases***: connection establishment, data transfer, and connection termination.

- **Connection Establishment:** TCP transmits data in full-duplex mode.

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 request is called a ***passive open***.
- The client program issues a request for an ***active open***. A client that wishes to connect to an open server tells its TCP to connect to a particular server.

Scanned with OKEN Scanner

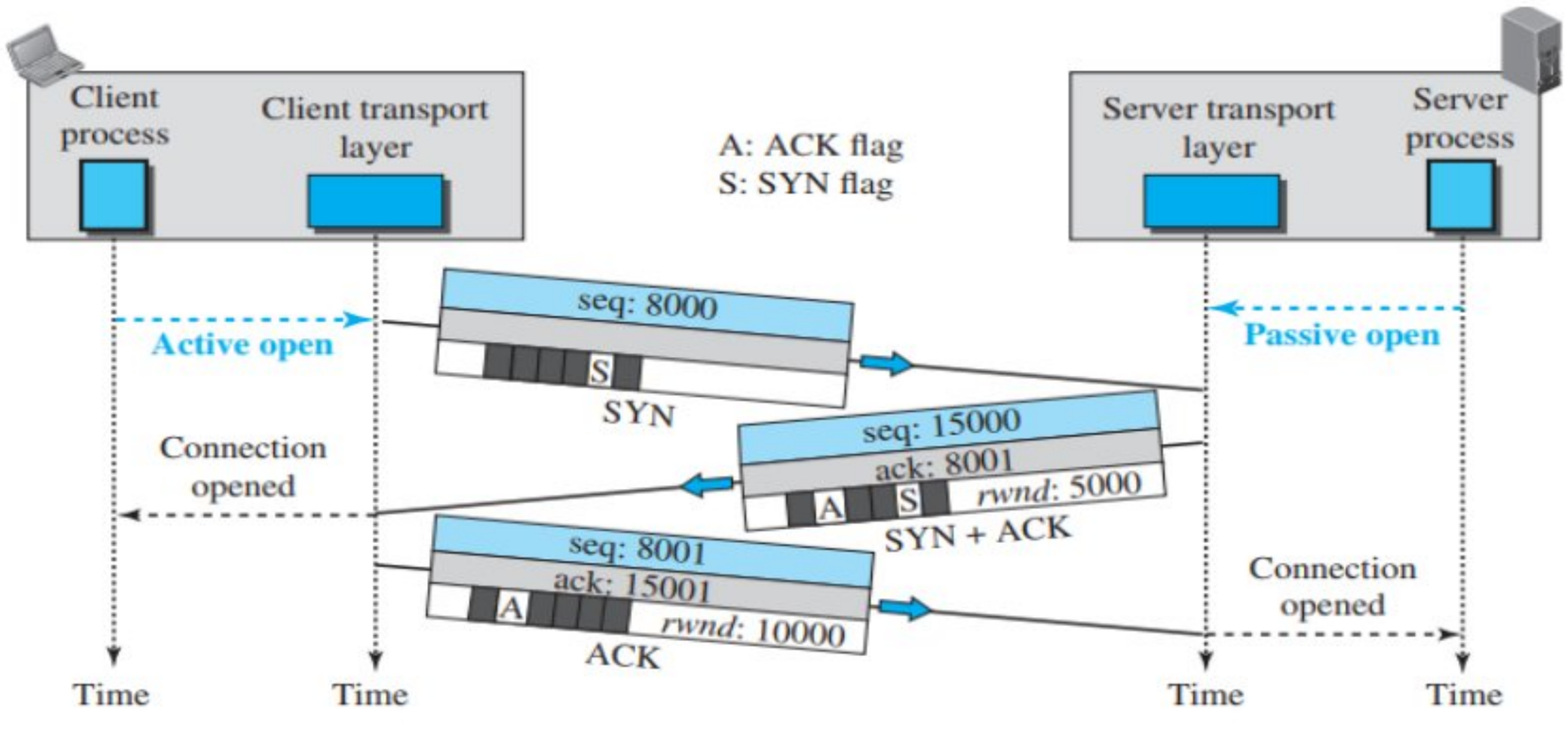


Fig: Connection establishment using three-way handshaking

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. A SYN segment cannot carry data, but it consumes one sequence number.

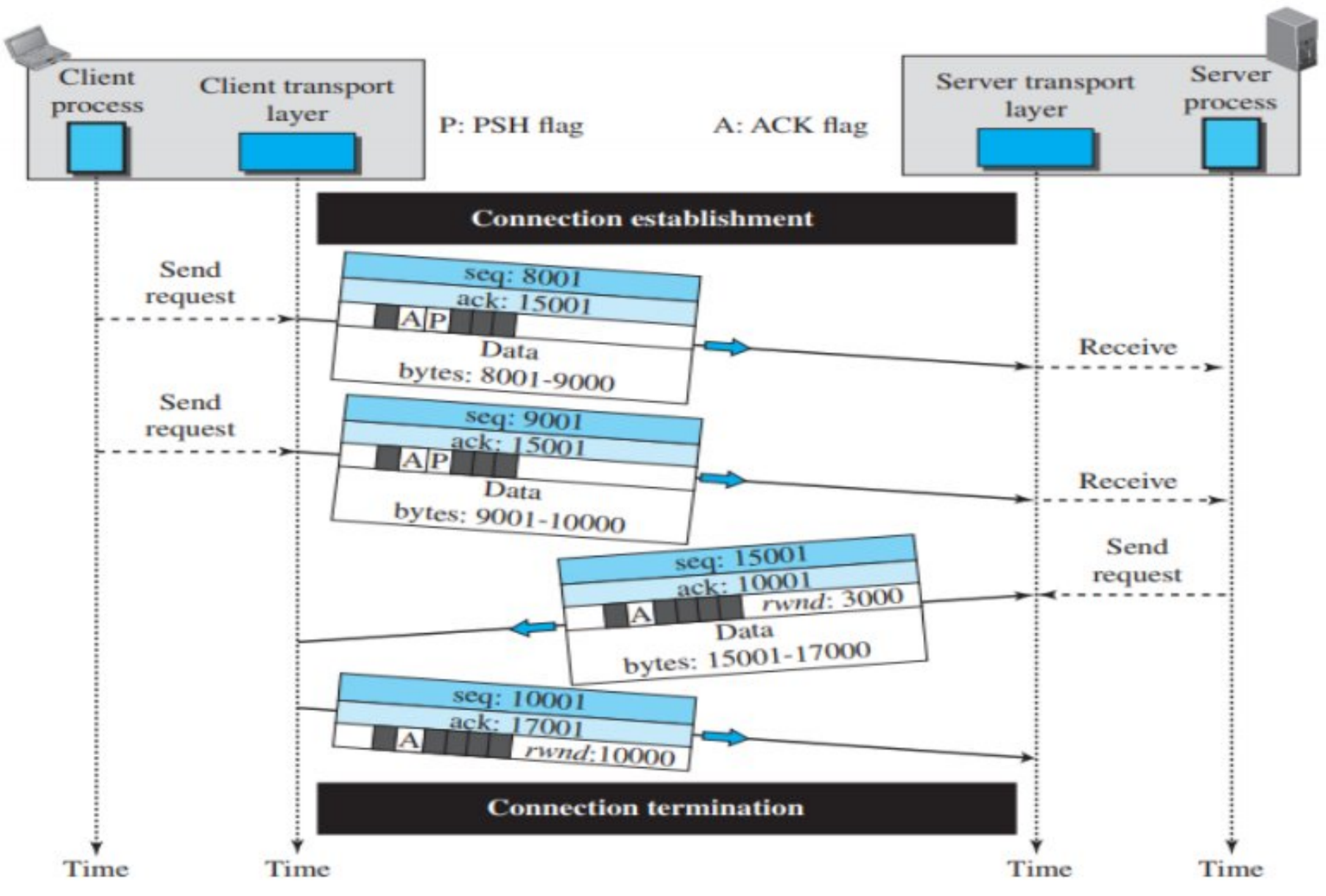
2. The server sends the second segment, a SYN + ACK segment with two flag bits set as: SYN and ACK. This segment has a dual purpose. A SYN + ACK segment cannot carry data, but it does consume one sequence number.

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.

Data Transfer: After connection is established, bidirectional data transfer can take place. The client and server can send data and acknowledgments in both directions.

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.

Pushing Data: 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.



Scanned with OKEN Scanner

Fig: Data transfer

- The application program at the sender can request a ***push*** operation. It must create a segment and send it immediately.
- The sending TCP must also set the ***push bit (PSH)*** to let the receiving TCP know that the segment includes data that must be delivered to the receiving application program.

Urgent Data: there are occasions in which an application program needs to send urgent bytes, some bytes that need to be treated in a special way by the application at the other end.

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.

Connection Termination: Either of the two parties involved in exchanging data (client or server) can close the connection, it is usually initiated by the client.

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:

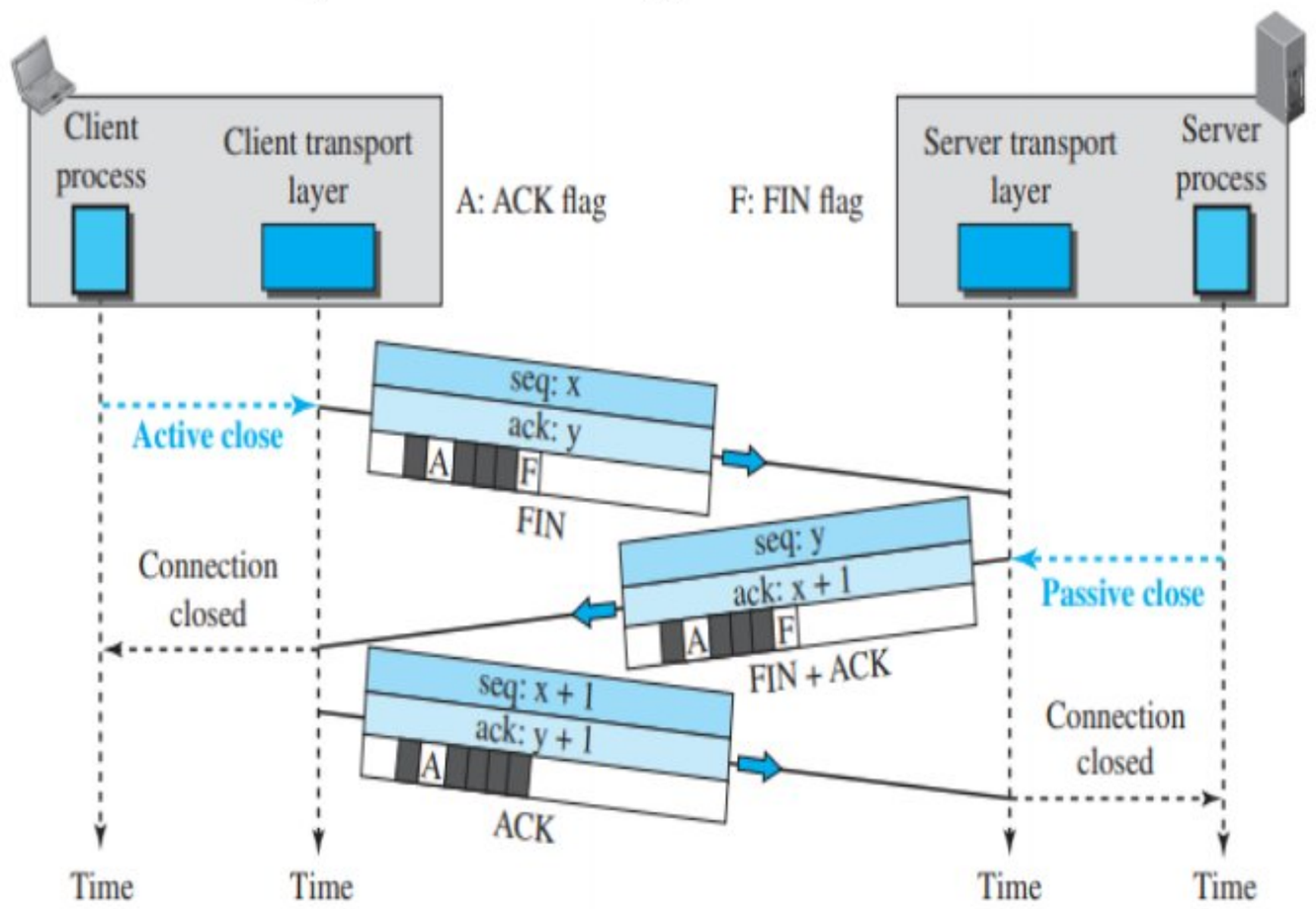


Fig: Connection termination using three-way handshaking

1. In this 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***. The FIN segment consumes one sequence number if it does not carry data.
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.
3. The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server.

Scanned with OKEN Scanner

Half-Close: In TCP, one end can stop sending data while still receiving data. This is called a ***halfclose***. Either the server or the client can issue a half-close request. It can occur when the server needs all the data before processing can begin.

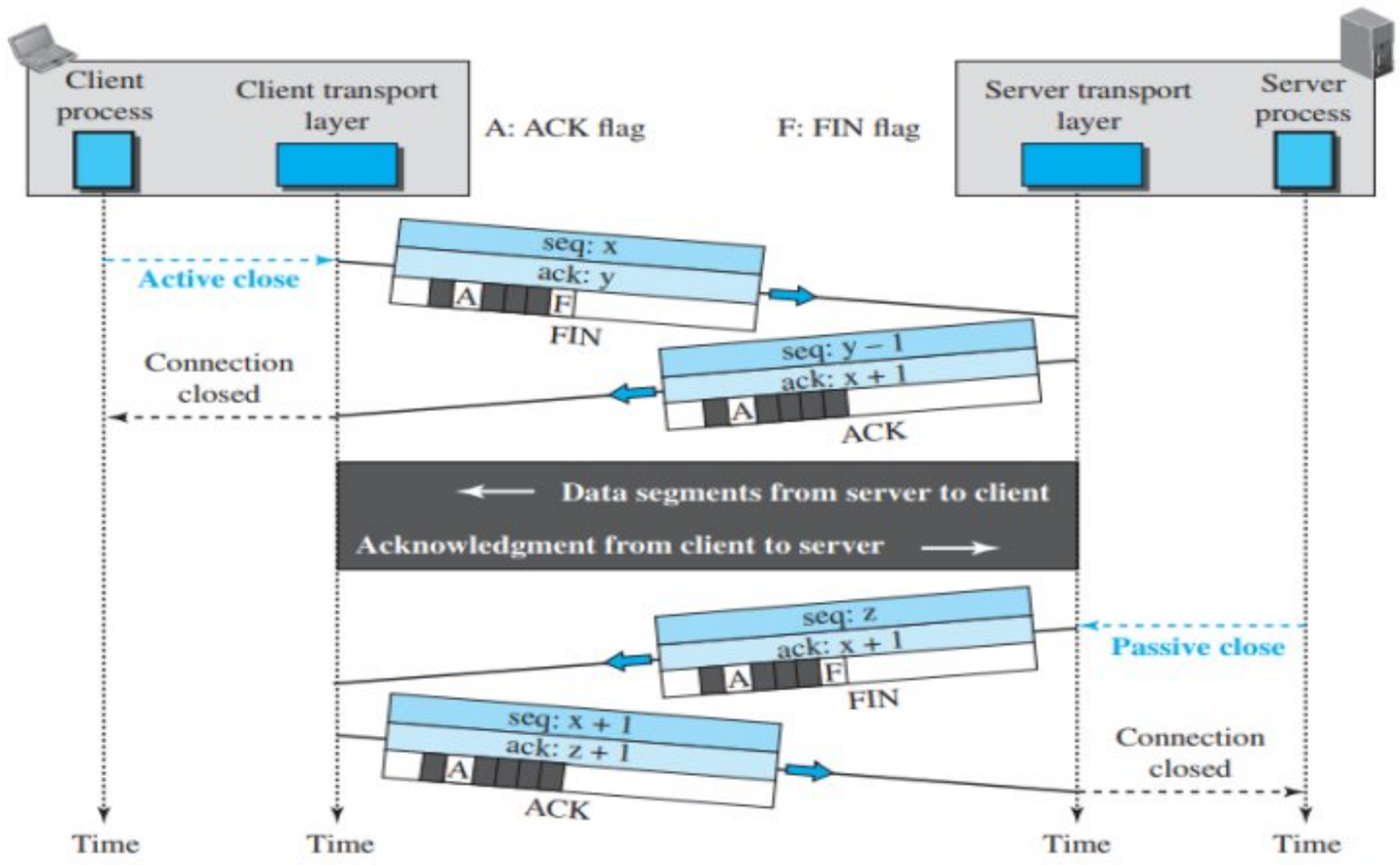


Fig: Half-close

- The data transfer from the client to the server stops. The client half-closes the connection by sending a FIN segment. The server accepts the half-close by sending the ACK segment.
- The server, however, can still send data. When the server has sent all of the processed data, it sends a FIN segment, which is acknowledged by an ACK from the client.
- After half-closing the connection, data can travel from the server to the client and acknowledgments can travel from the client to the server. The client cannot send any more data to the server.

Connection Reset: TCP at one end may deny a connection request, may abort an existing connection, or may terminate an idle connection. All of these are done with the ***RST (reset) flag.***

TCP Connection Management Modeling: The steps required establishing and release connections can be represented in a finite state machine with the 11 states listed in Fig.

Scanned with OKEN Scanner

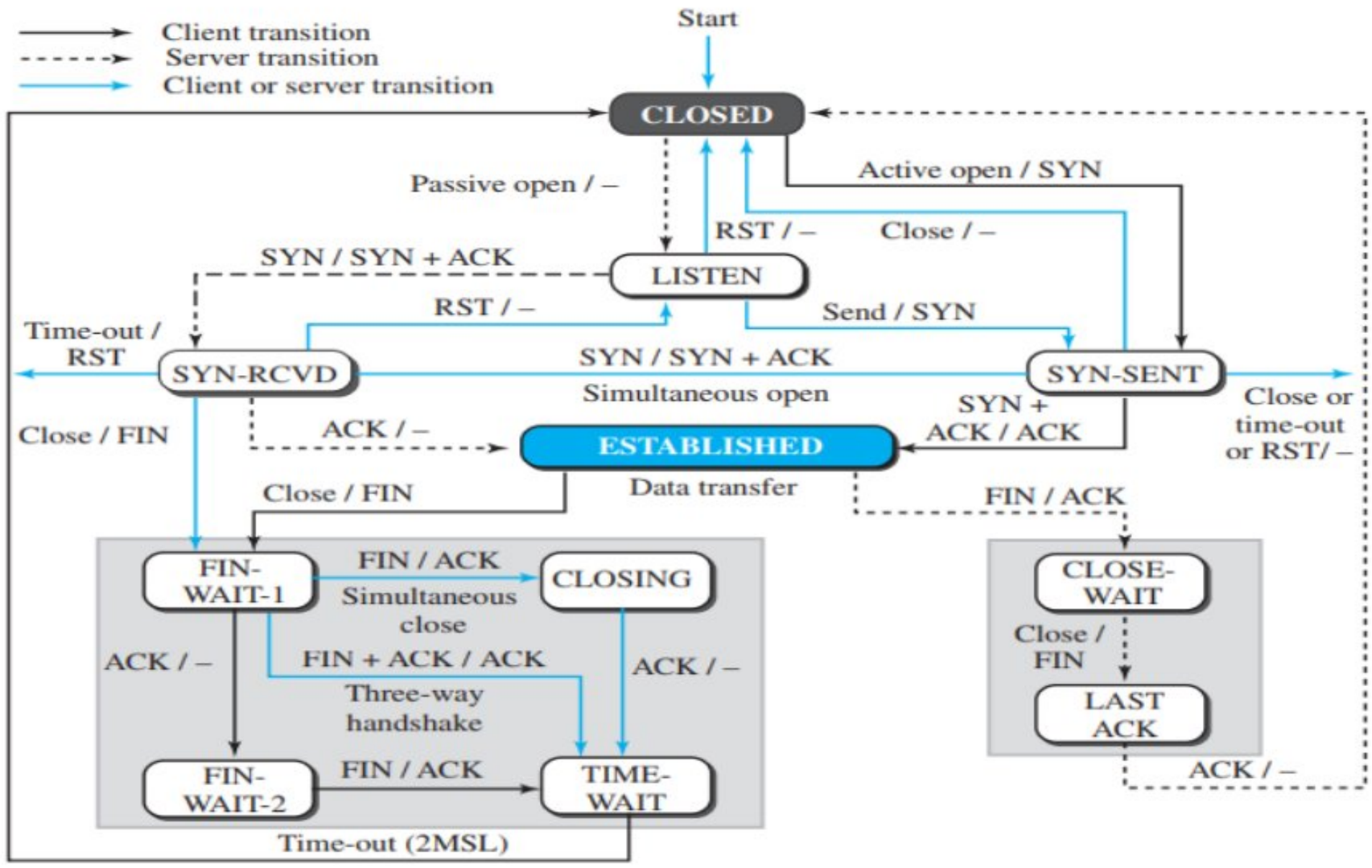


Fig: State transition diagram

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Fig: The states used in the TCP connection management finite state machine

TCP Sliding Window: window management in TCP decouples the issues of acknowledgement of the correct receipt of segments and receiver buffer allocation.
For example, suppose the receiver has a 4096-byte buffer, as shown in Fig. If the sender transmits a 2048-byte segment that is correctly received, the receiver will acknowledge the segment.

- However, since it now has only 2048 bytes of buffer space (until the application removes some data from the buffer), it will advertise a window of 2048 starting at the next byte expected.
- Now the sender transmits another 2048 bytes, which are acknowledged, but the advertised window is of size 0.
- The sender must stop until the application process on the receiving host has removed some data from the buffer, at which time TCP can advertise a larger window and more data can be sent.

Scanned with OKEN Scanner

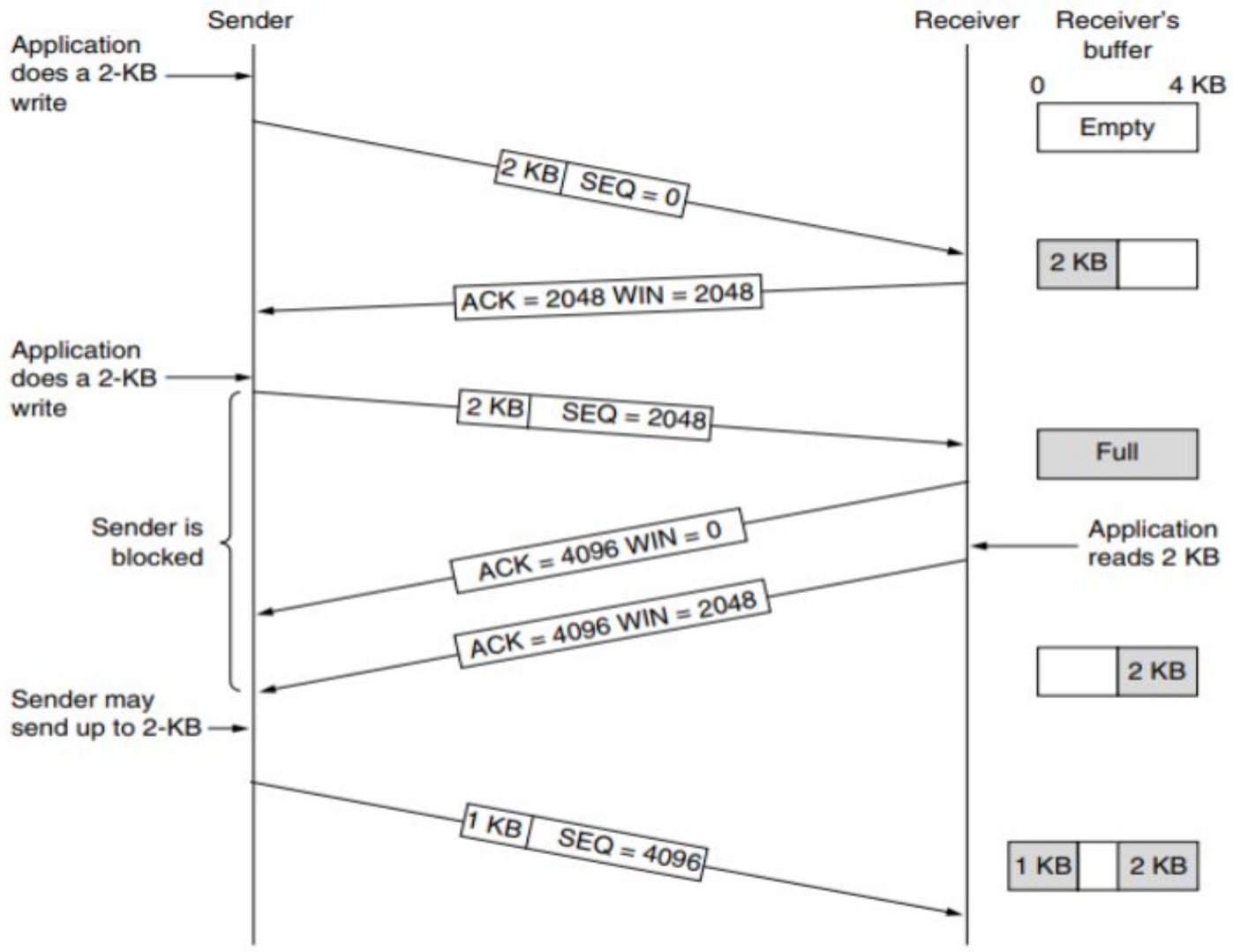


Fig: Window management in TCP

TCP Congestion Control: When the load offered to any network is more than it can handle, congestion builds up. The network layer detects congestion when queues grow large at routers and tries to manage it, if only by dropping packets. It is up to the transport layer to receive congestion feedback from the network layer and slow down the rate of traffic that it is sending into the network.

Congestion Policies: TCP's general policy for handling congestion is based on three algorithms: slow start, congestion avoidance, and fast recovery.

1. ***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. The algorithm starts slowly, but grows exponentially.

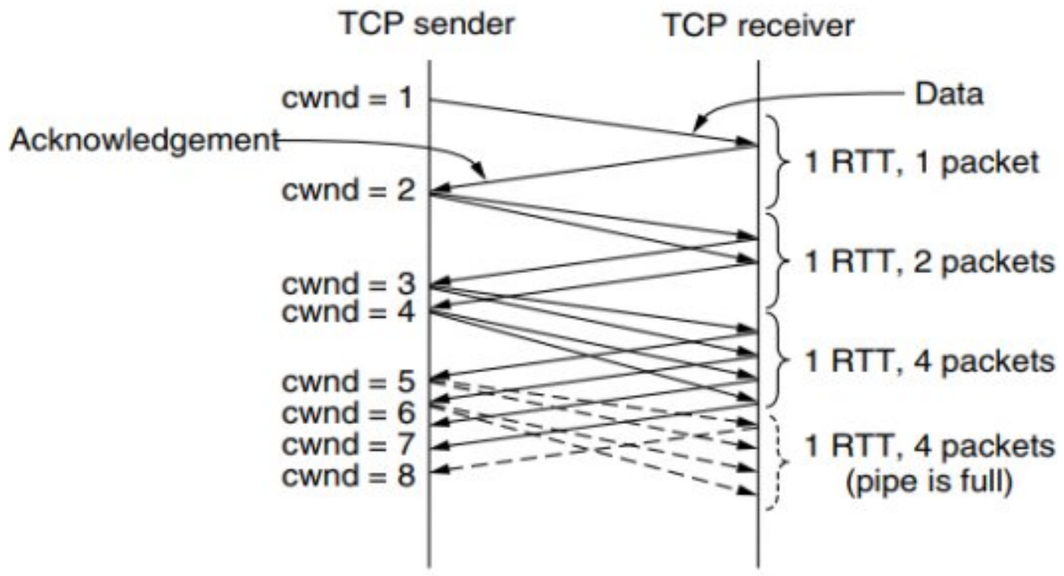


Fig: Slow start from an initial congestion window of one segment.

In the first round-trip time, the sender injects one packet into the network (and the receiver receives one packet). Two packets are sent in the next round-trip time, then four packets in the third round-trip time.

Scanned with OKEN Scanner

2. *Congestion Avoidance: Additive Increase:* If we continue with the slow-start algorithm, the size of the congestion window increases exponentially.

- To avoid congestion before it happens, we must slow down this exponential growth.
- TCP defines another algorithm called congestion avoidance, which increases the cwnd additively.
- In this algorithm, each time the whole "window" of segments is acknowledged, the size of the congestion window is increased by one.

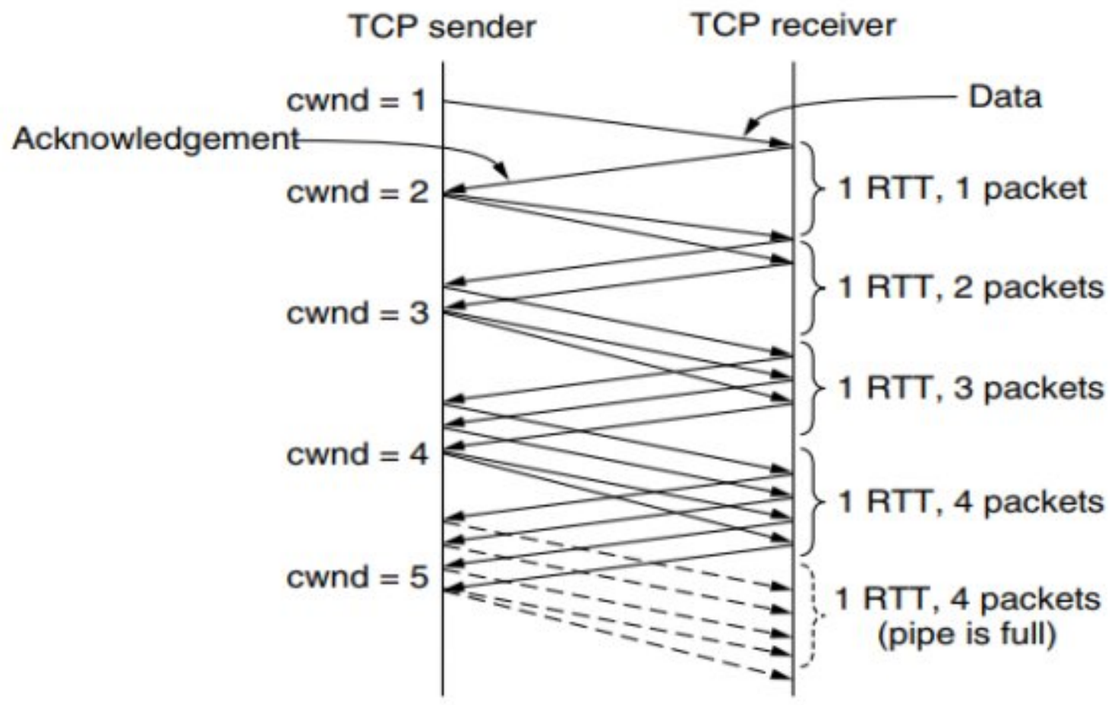


Fig: Additive increase from an initial congestion window of one segment

3. **Fast Recovery:** The fast-recovery algorithm is optional in TCP. It starts when three duplicate ACKs arrive, which is interpreted as light congestion in the network. Like congestion avoidance, this algorithm is also an additive increase, but it increases the size of the congestion window when a duplicate ACK arrives (after the three duplicate ACKs that trigger the use of this algorithm).

TCP Timer Management: TCP uses multiple timers to do its work. The most important of these is the **RTO (Retransmission TimeOut).**
When a segment is sent, a retransmission timer is started. If the segment is acknowledged before the timer expires, the timer is stopped.
If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted.

The question that arises is: how long should the timeout be? Determining the round-trip time to the destination is tricky. Even when it is known, deciding on the timeout interval is also difficult.

- If the timeout is set ***too short***, unnecessary retransmissions will occur, clogging the Internet with useless packets.
- If it is set ***too long***, performance will suffer due to the long retransmission delay whenever a packet is lost.
- The solution is to use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance.
- For each connection, TCP maintains a variable, SRTT (Smoothed Round-Trip Time) that is the best current estimate of the round-trip time to the destination in question.

Scanned with OKEN Scanner

- **SRTT** according to the formula : **SRTT = α SRTT + (1 − α) R** (Typically, α = 7/8)
- **RTTVAR** (RoundTrip Time VARiation) that is updated using the formula: **RTTVAR = β RTTVAR + (1 − β) | SRTT − R |** (typically β = 3/4)
- The retransmission timeout, **RTO**, is set to be: **RTO = SRTT + 4 × RTTVAR**

3. Congestion Control

If the transport entities on many machines send too many packets into the network too quickly, the network will become congested, with performance degraded as packets are delayed and lost.

- Controlling congestion to avoid this problem is the combined responsibility of the network and transport layers. Congestion occurs at routers, so it is detected at the network layer.
- However, congestion is ultimately caused by traffic sent into the network by the transport layer. The only effective way to control congestion is for the transport protocols to send packets into the network more slowly.

Desirable Bandwidth Allocation

The goal is to simply avoid congestion; it is to find a good allocation of bandwidth to the transport entities that are using the network.

A good allocation will deliver good performance because it uses all the available bandwidth but avoids congestion, it will be fair across competing transport entities, and it will quickly track changes in traffic demands.

Efficiency and Power: An efficient allocation of bandwidth across transport entities will use all of the network capacity that is available.

The **goodput** (or rate of useful packets arriving at the receiver) as a function of the offered load. This curve and a matching curve for the delay as a function of the offered load are given in Fig.

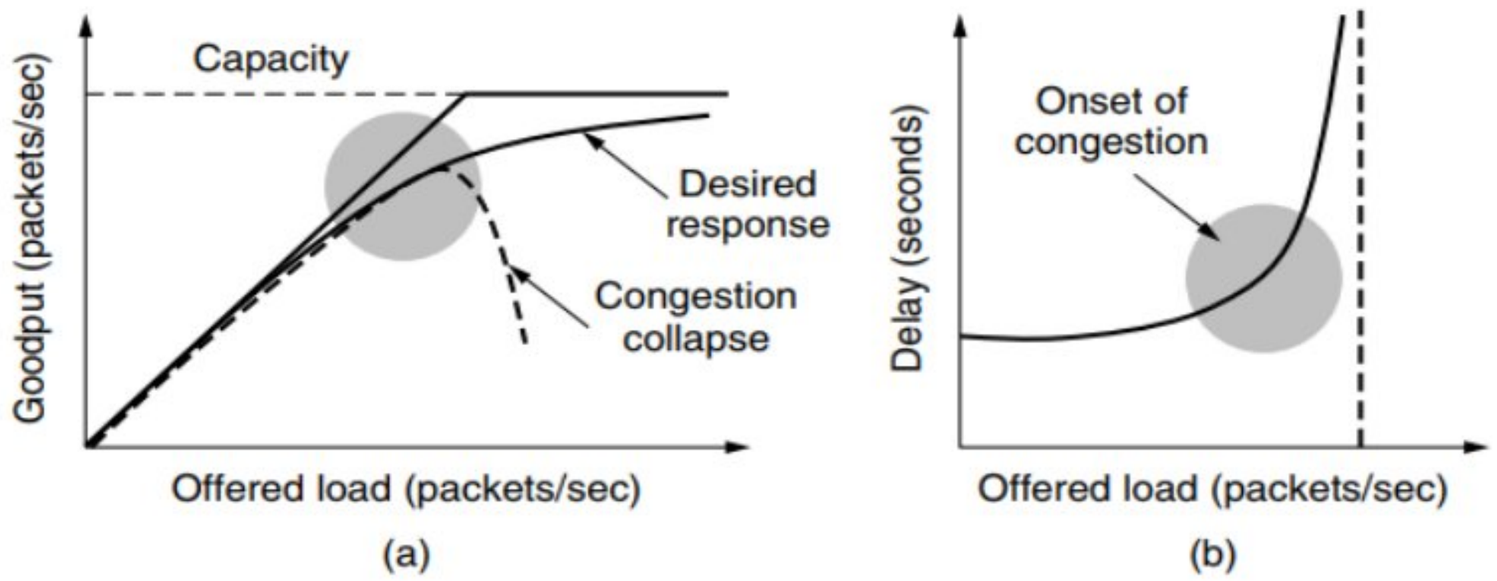


Fig: (a) Goodput and (b) delay as a function of offered load.

As the load increases in ***Fig (a)*** goodput initially increases at the same rate, but as the load approaches the capacity, goodput rises more gradually. This falloff is because bursts of traffic can occasionally mount up and cause some losses at buffers inside the network.

The corresponding delay is given in ***Fig (b)*** initially the delay is fixed. As the load approaches the capacity, the delay rises, slowly at first and then much more rapidly. This is again because of bursts of traffic that tend to mound up at high load.

Scanned with OKEN Scanner

Power will initially rise with offered load, as delay remains small and roughly constant, but will reach a maximum and fall as delay grows rapidly. The load with the highest power represents an efficient load for the transport entity to place on the network.

$$power = \frac{load}{delay}$$

☒ Regulating the Sending Rate

How do we regulate the sending rates to obtain a desirable bandwidth allocation? The sending rate may be limited by two factors.

The first is flow control, in the case that there is insufficient buffering at the receiver. The second is congestion, in the case that there is insufficient capacity in the network.

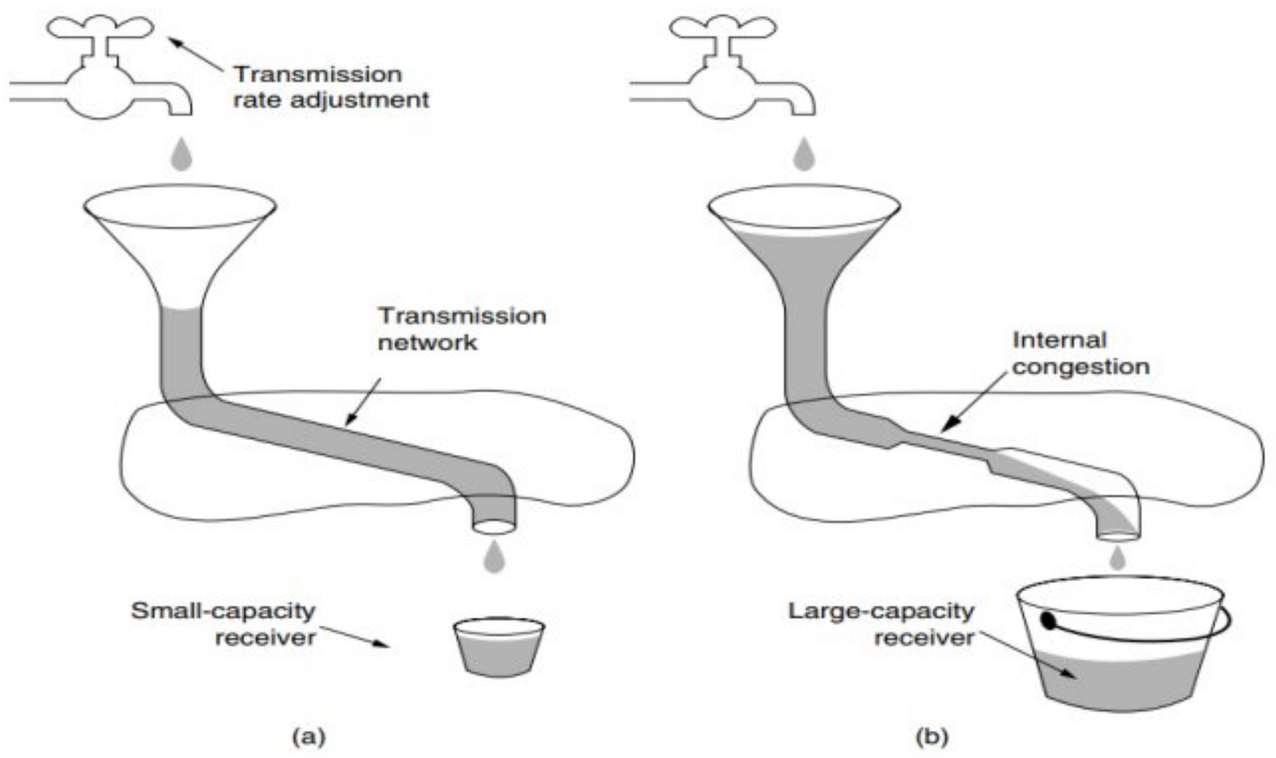


Fig: (a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.

In Fig (a), we see a thick pipe leading to a small-capacity receiver. This is a flow-control limited situation. As long as the sender does not send more water than the bucket can contain, no water will be lost. In Fig (b), the limiting factor is not the bucket capacity, but the internal carrying capacity of the network. If too much water comes in too fast, it will back up and some will be lost.

Scanned with OKEN Scanner