CSE 3231
Computer Networks

Chapter 6
The Transport Layer
part 2

William Allen, PhD

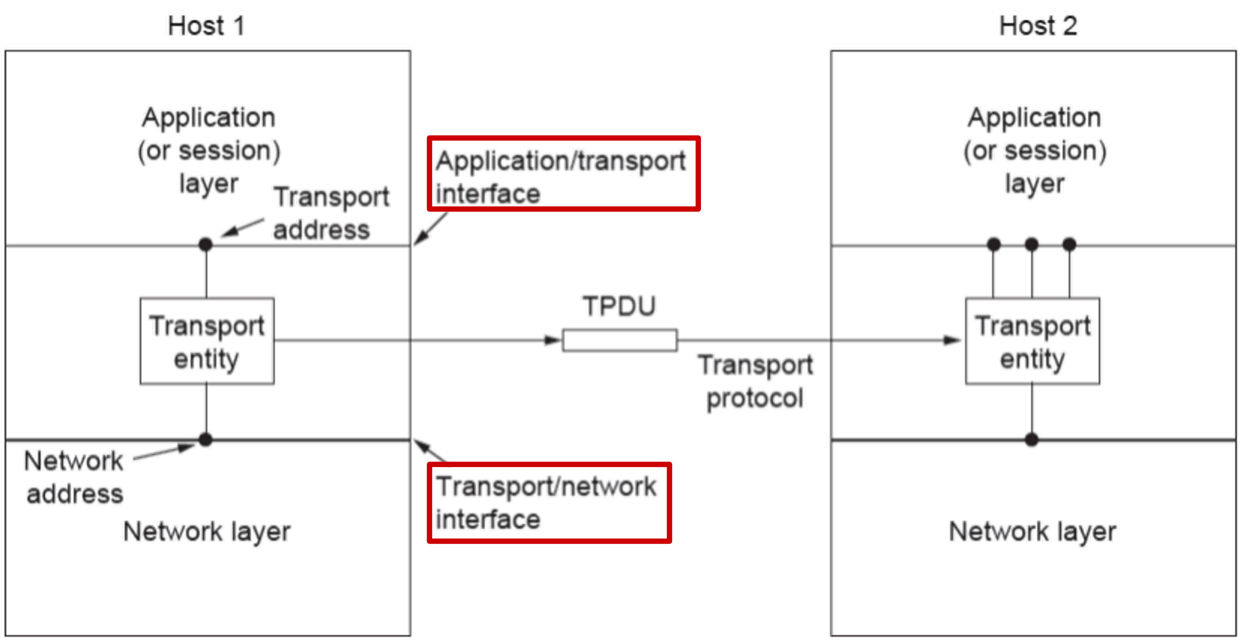
Spring 2022

Services Provided by Transport Layer Protocols

The transport layer provides the following:

– can add *reliability* to packets sent by the network layer

– offers *connectionless* (e.g., UDP) and *connection-oriented* (e.g, TCP) services to applications



Berkeley Sockets

Very widely used primitives developed at UC Berkeley to support TCP networking on UNIX

- "*sockets*" are used as transport layer endpoints
- Originally accessed in C/C++, now available in all major programming languages

Primitive	Meaning
SOCKET	Create a new communication end point
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

Transport Service Primitives

Functions that applications can call to transport data over a simple connection-oriented service:

- Client calls `CONNECT`, `SEND`, `RECEIVE`, `DISCONNECT`

Primitive	Segment sent	Meaning
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

Transport Service Primitives

The following Transport Service primitives are used at the client end of the connection

- *connect* - client requests creation of a connection with a specific server
- *send* - client transmits segments to a server over an established connection
- *receive* - client receives segments from a server over an established connection
- *disconnect* - client notifies the server that the client is finished sending data and is *ready to disconnect* (server has to confirm disconnection because it may not have received all segments yet)

Transport Service Primitives

Functions that applications can call to transport data over a simple connection-oriented service:

- Client calls `CONNECT`, `SEND`, `RECEIVE`, `DISCONNECT`
- Server calls `LISTEN`, `RECEIVE`, `SEND`, `DISCONNECT`

Primitive	Segment sent	Meaning
LISTEN	(none)	Block until some process tries to connect
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

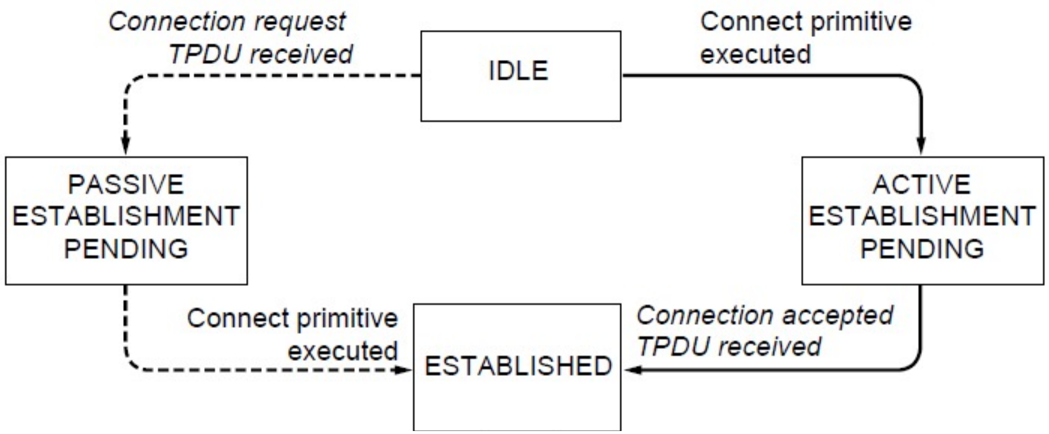
Transport Service Primitives

The following Transport Service primitives are used at the server end of the connection

- *listen* - server is waiting for a request from a client to create a connection (i.e., a CONNECT request)
- *send* - server transmits segments to a client over an established connection
- *receive* - server receives segments from a client over an established connection
- *disconnect* - after a server has received a disconnect request from a specific client *and* has received all expected data segments from that client, it will drop the connection with that client

Transport Service Primitives

State diagram for a connection-oriented service



server accepts client's request and connection is established

client and server can send and receive data segments

Solid lines (right) show client state sequence

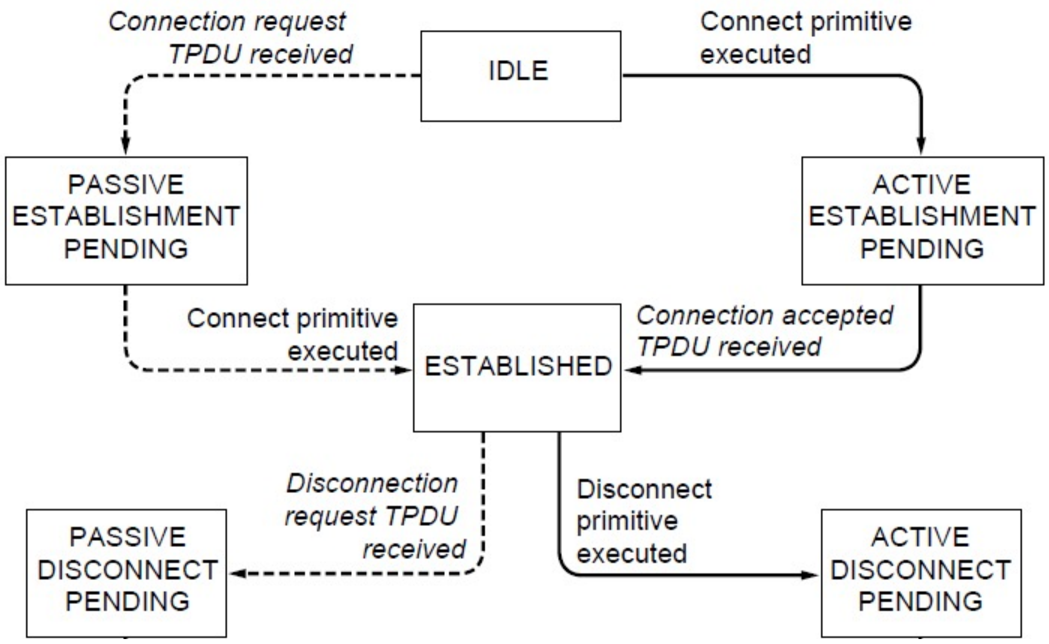
Dashed lines (left) show server state sequence

Transitions in italics are due to segment arrivals.

TPDU - *Transport Protocol Data Unit* (packet that contains a transport layer header and data segment)

Transport Service Primitives

State diagram for a connection-oriented service



client has requested disconnection,
waits on server to complete

Solid lines (right) show client state sequence

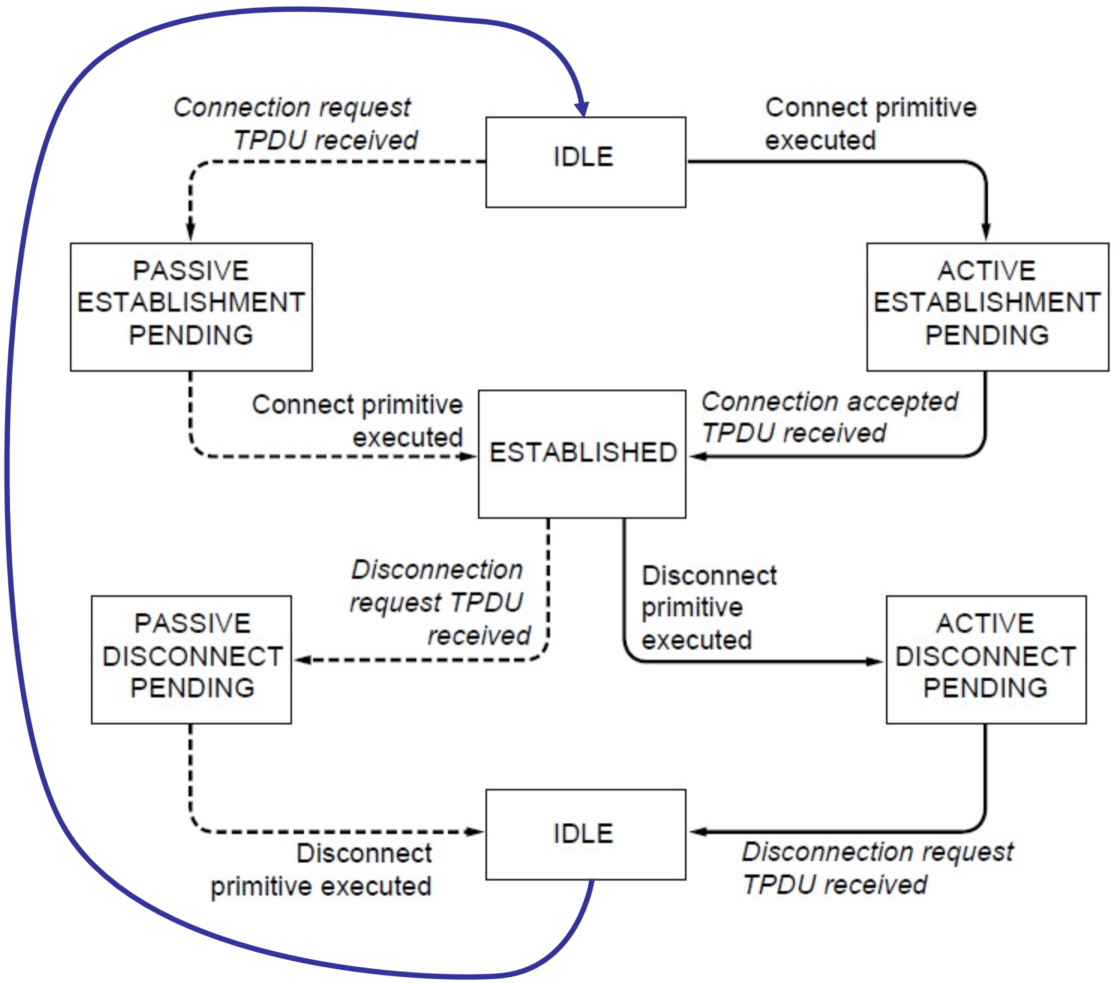
Dashed lines (left) show server state sequence

Transitions in italics are due to segment arrivals.

TPDU - *Transport Protocol Data Unit* (packet that contains a transport layer header and data segment)

Transport Service Primitives

State diagram for a connection-oriented service



Solid lines (right) show client state sequence

Dashed lines (left) show server state sequence

Transitions in italics are due to segment arrivals.

TPDU - *Transport Protocol Data Unit* (packet that contains a transport layer header and data segment)

Connection-Oriented Transport (TCP)

- TCP also uses port numbers to multiplex & demultiplex connections between processes
- In contrast to UDP, the Transmission Control Protocol (TCP) offers the following services
  - Connection oriented
    - Sender must establish connection before transmission
    - Sender notified of delivery or of error
  - Byte-stream service
    - Data transmission and reception are similar to file I/O
  - Reliable delivery
    - Guarantee that packets will be assembled in the order at the destination before delivery to the application

TCP Segment

- TCP is a byte-oriented protocol, which means that the sender writes bytes into a TCP connection and the receiver reads bytes out of the TCP connection.
  - Although "*byte stream*" describes the service TCP offers to application processes, TCP does not transmit individual bytes.
  - As with UDP, data is exchanged between TCP endpoints in *segments*.

TCP Segment Size

- The overall size of a message is often larger than the Data Link MTU (Ethernet: 1500 bytes) and may exceed the IP packet size (64k bytes)
- TCP divides larger messages into segments so that each segment fits into one IP packet before it passes them to the network layer for delivery
- To avoid the additional delays caused by fragmentation, it is common for TCP to create segments that fit into the much smaller Data Link MTU rather than fitting to the IP packet size
  - assembling fragments takes time, losses delay more

TCP Segment Size

- As an example, the TCP maximum segment size (*MSS*) for an Ethernet LAN would be:

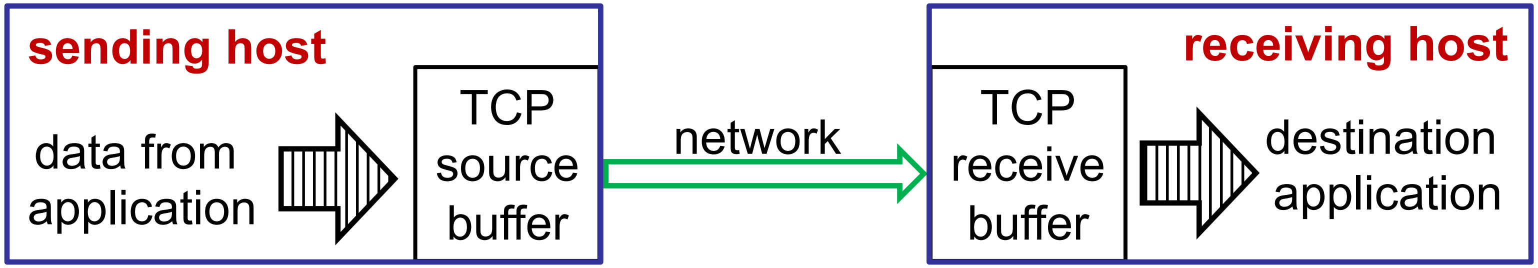
  (Ethernet payload size) - (IP header size) - (TCP header size)

  = (1500) - (20 without options) - (20 without options) = 1460

- However, as we have seen, the MTU across a network path may be less than 1500 bytes
  - RFC 1191 proposes the MTU discovery process (discussed earlier) to find the maximum MTU size that will avoid fragmentation along the path
  - Modern TCP implementations will use this technique to reduce the likelihood of fragmentation occurring

TCP Segment

- TCP buffers enough bytes from the sending process to fill a reasonably sized segment
- Then the sender transmits this segment to its peer on the destination host.

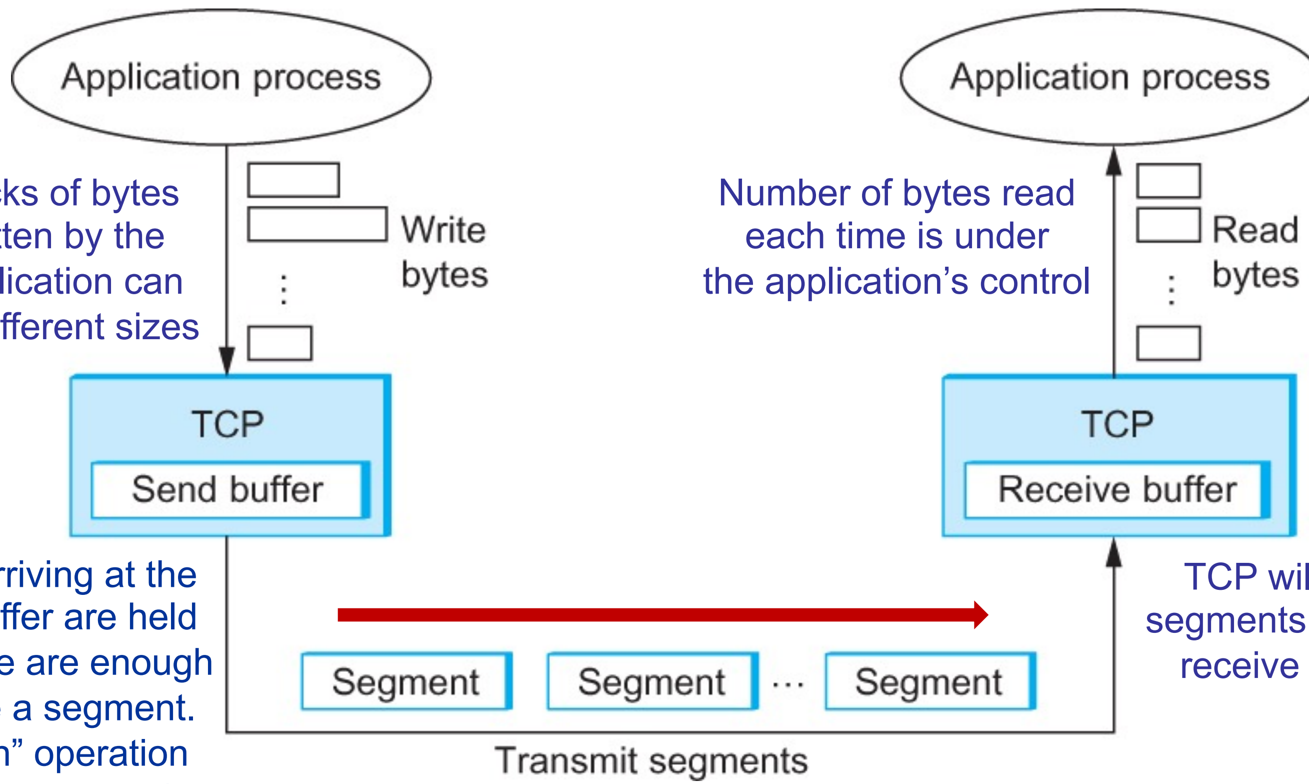


- TCP at the destination end copies the contents of the segment into a receive buffer
- The receiving process can then read the data from the buffer at its own pace.

TCP Segment

Blocks of bytes written by the application can be different sizes

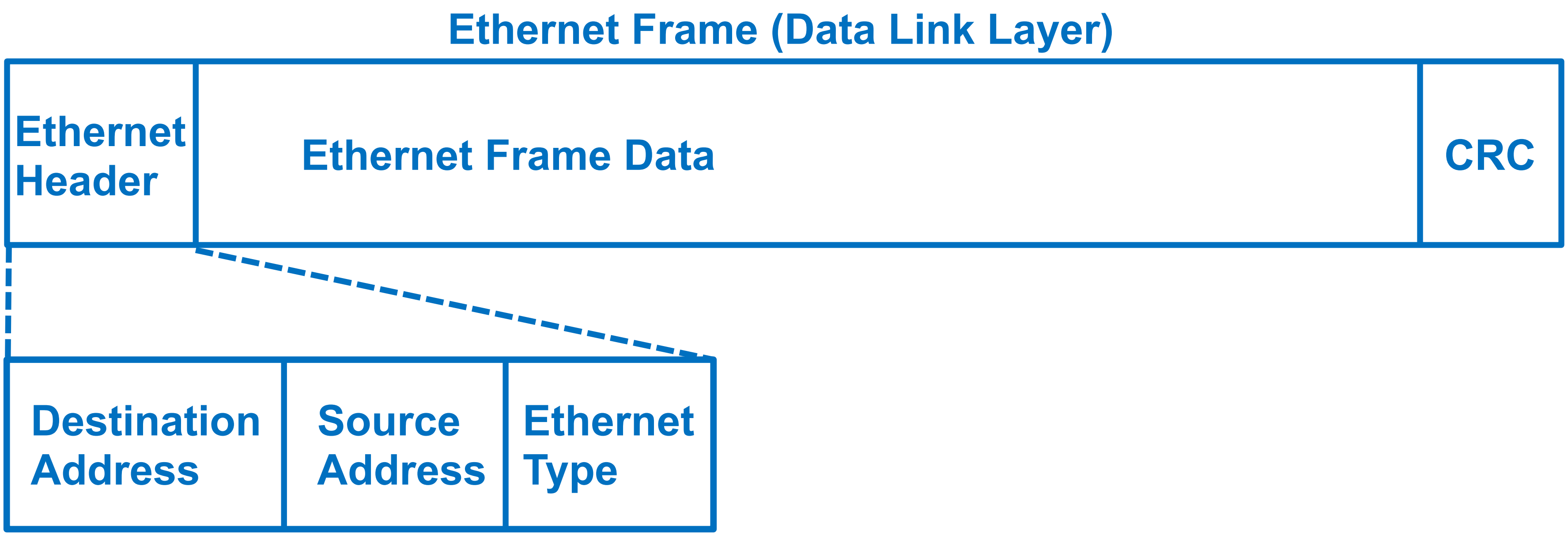
Number of bytes read each time is under the application's control

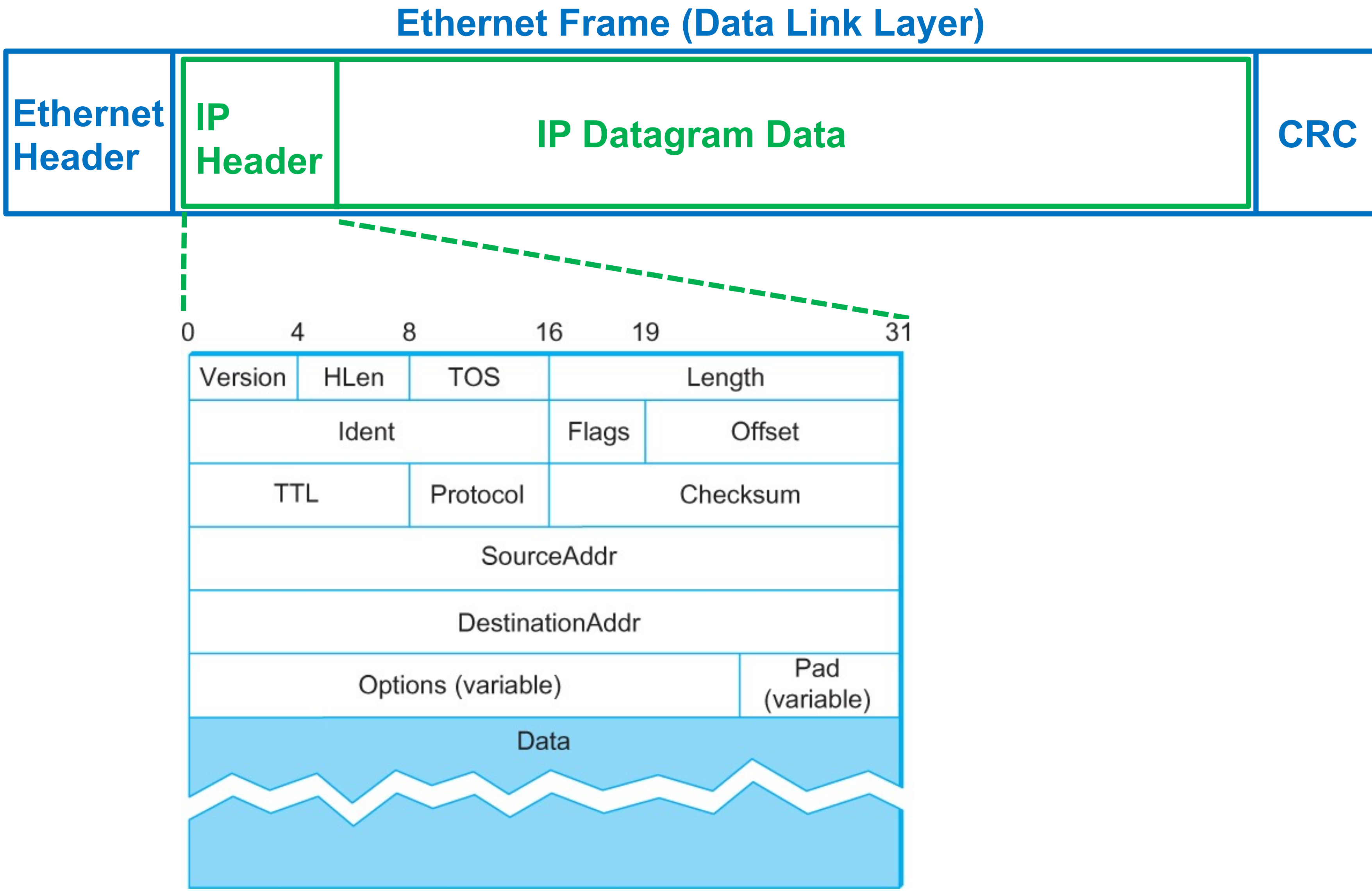


Bytes arriving at the send buffer are held until there are enough to make a segment. (a "flush" operation can force a send)

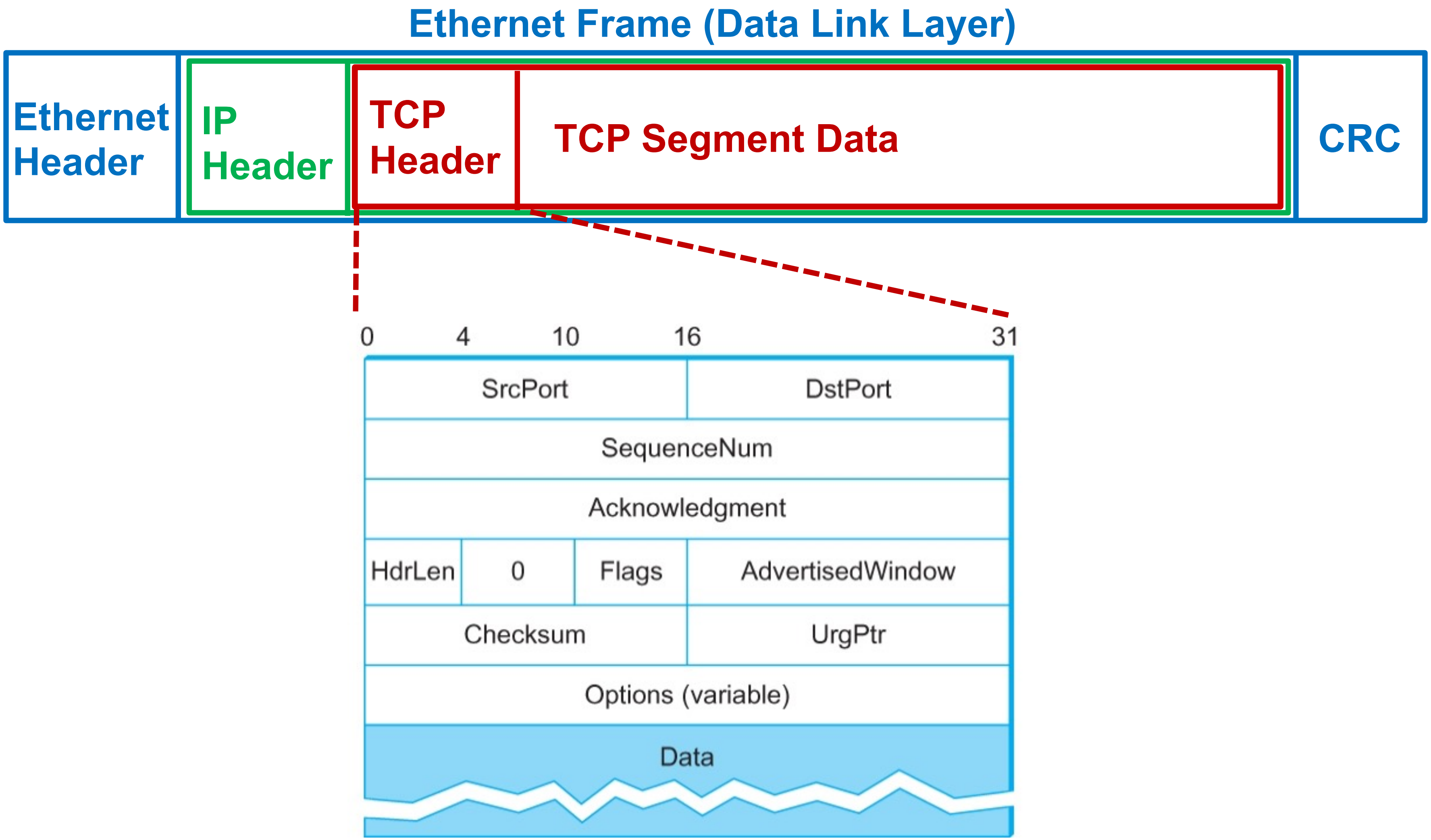
TCP will load segments into the receive buffer

How TCP manages a byte stream.

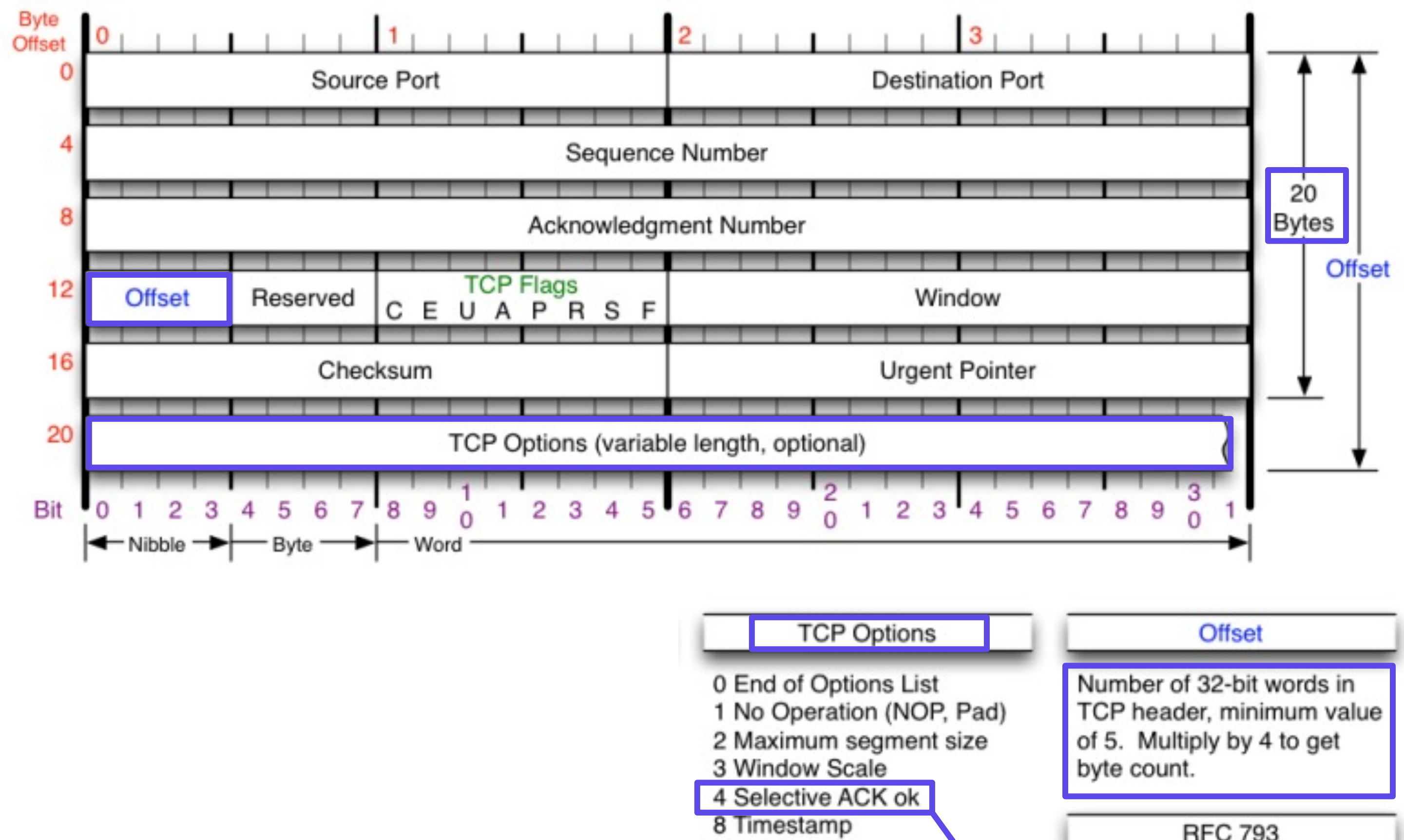

Ethernet Frame (Data Link Layer)
Ethernet Header
Ethernet Frame Data
CRC
Destination Address
Source Address
Ethernet Type



Ethernet Frame (Data Link Layer)
Ethernet Header
IP Header
IP Datagram Data
CRC
0
4
8
16
19
31
Version
HLen
TOS
Length
Ident
Flags
Offset
TTL
Protocol
Checksum
SourceAddr
DestinationAddr
Options (variable)
Pad (variable)
Data

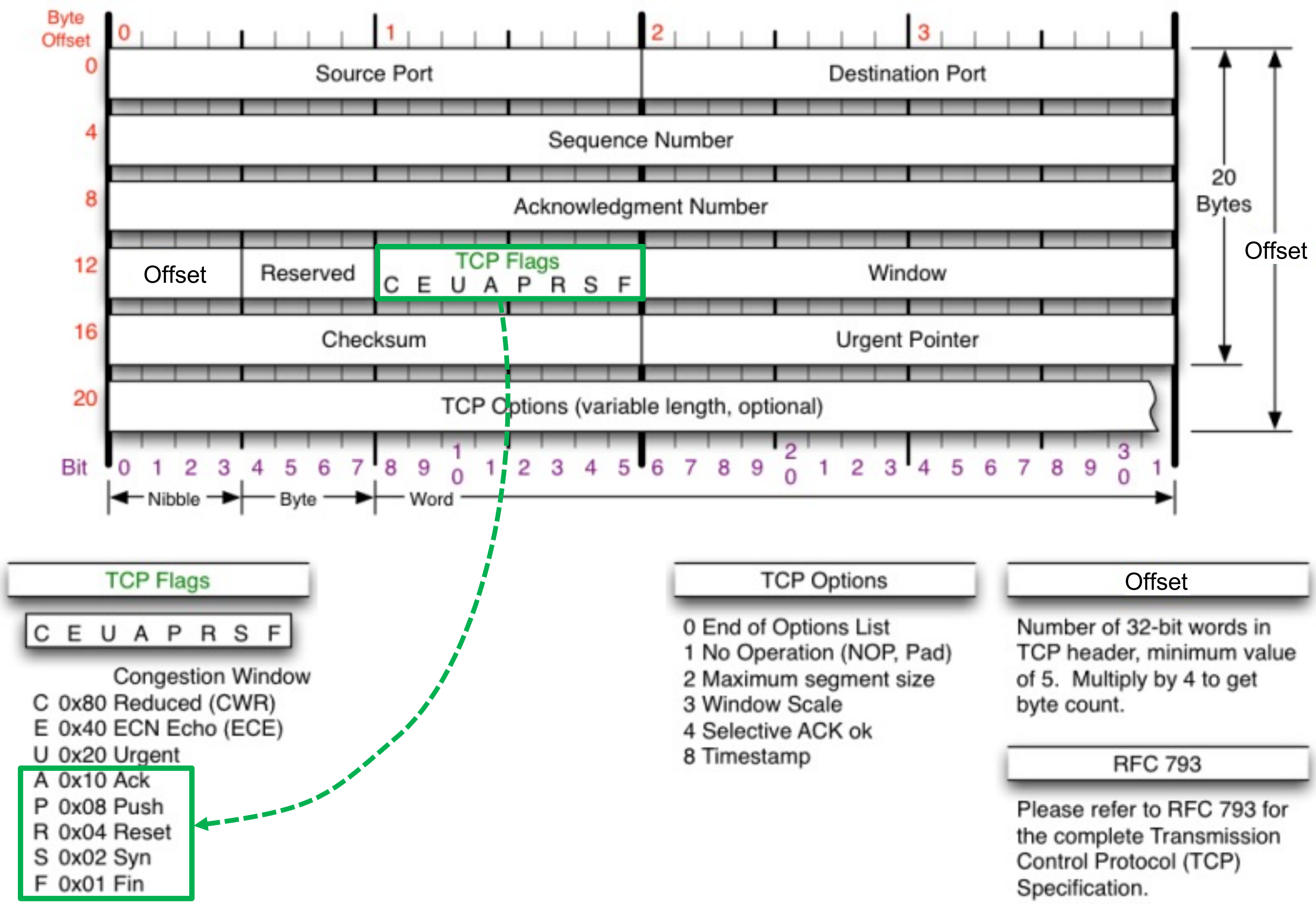

Ethernet Frame (Data Link Layer)
Ethernet Header
IP Header
TCP Header
TCP Segment Data
CRC
0
4
10
16
31
SrcPort
DstPort
SequenceNum
Acknowledgment
HdrLen
0
Flags
AdvertisedWindow
Checksum
UrgPtr
Options (variable)
Data

TCP Header



Selective acknowledgement (can send ACKs out of order)

TCP Header

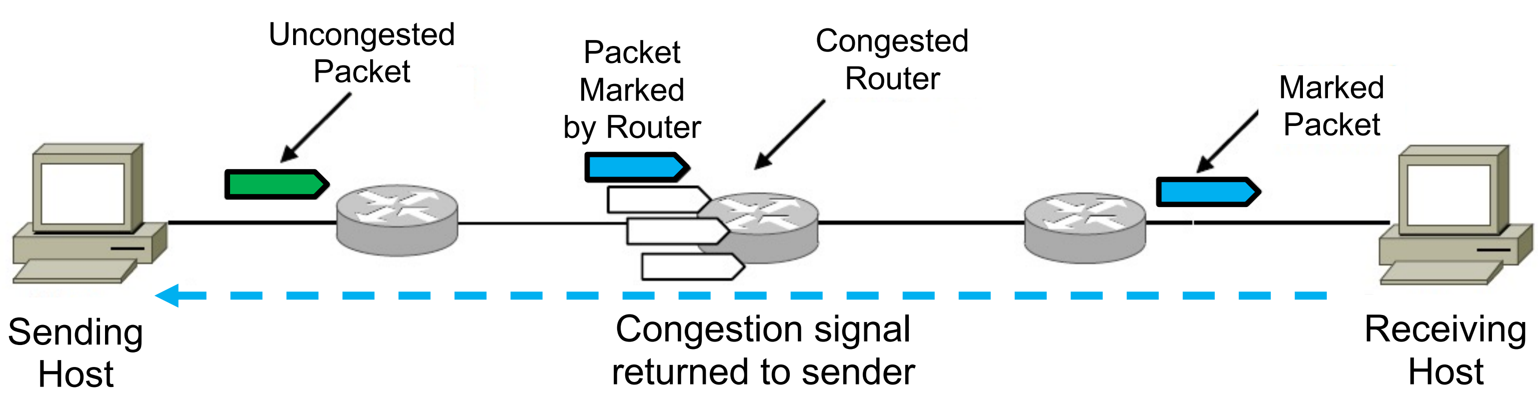


Selective acknowledgement (can send ACKs out of order)

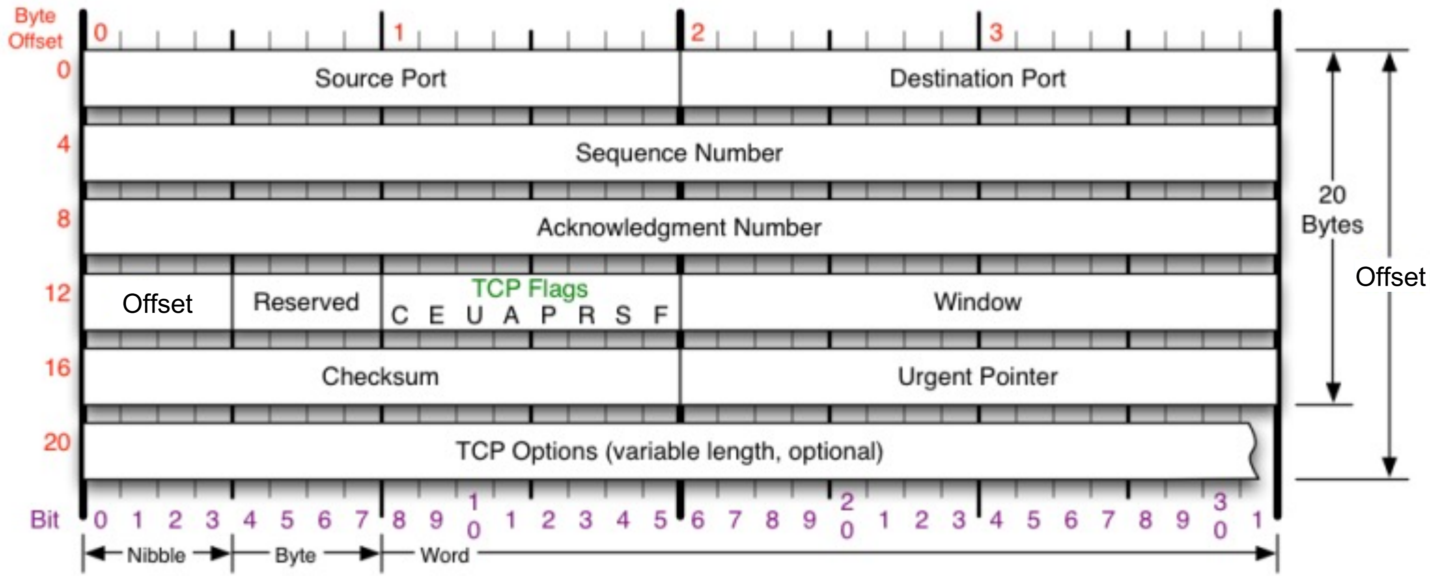
From Chapter 5

Congested routers can signal hosts to slow the rate of transmission of traffic

- ECN (Explicit Congestion Notification) marks packets suffering from congestion
  - the receiving host notifies the sender to slow the rate of transmission



TCP Header



TCP Flags

C E U A P R S F

Congestion Window

C 0x80 Reduced (CWR)
E 0x40 ECN Echo (ECE)
U 0x20 Urgent
A 0x10 Ack
P 0x08 Push
R 0x04 Reset
S 0x02 Syn
F 0x01 Fin

Congestion Notification

ECN (Explicit Congestion Notification). See RFC 3168 for full details, valid states below.

Packet State	DSB	ECN bits
Syn	0 0	1 1
Syn-Ack	0 0	0 1
Ack	0 1	0 0
No Congestion	0 1	0 0
No Congestion	1 0	0 0
Congestion	1 1	0 0
Receiver Response	1 1	0 1
Sender Response	1 1	1 1

TCP Options

0 End of Options List
1 No Operation (NOP, Pad)
2 Maximum segment size
3 Window Scale
4 Selective ACK ok
8 Timestamp

Offset

Number of 32-bit words in TCP header, minimum value of 5. Multiply by 4 to get byte count.

RFC 793

Please refer to RFC 793 for the complete Transmission Control Protocol (TCP) Specification.

Selective acknowledgement (can send ACKs out of order)

TCP Header

Selective acknowledgement (can send ACKs out of order)

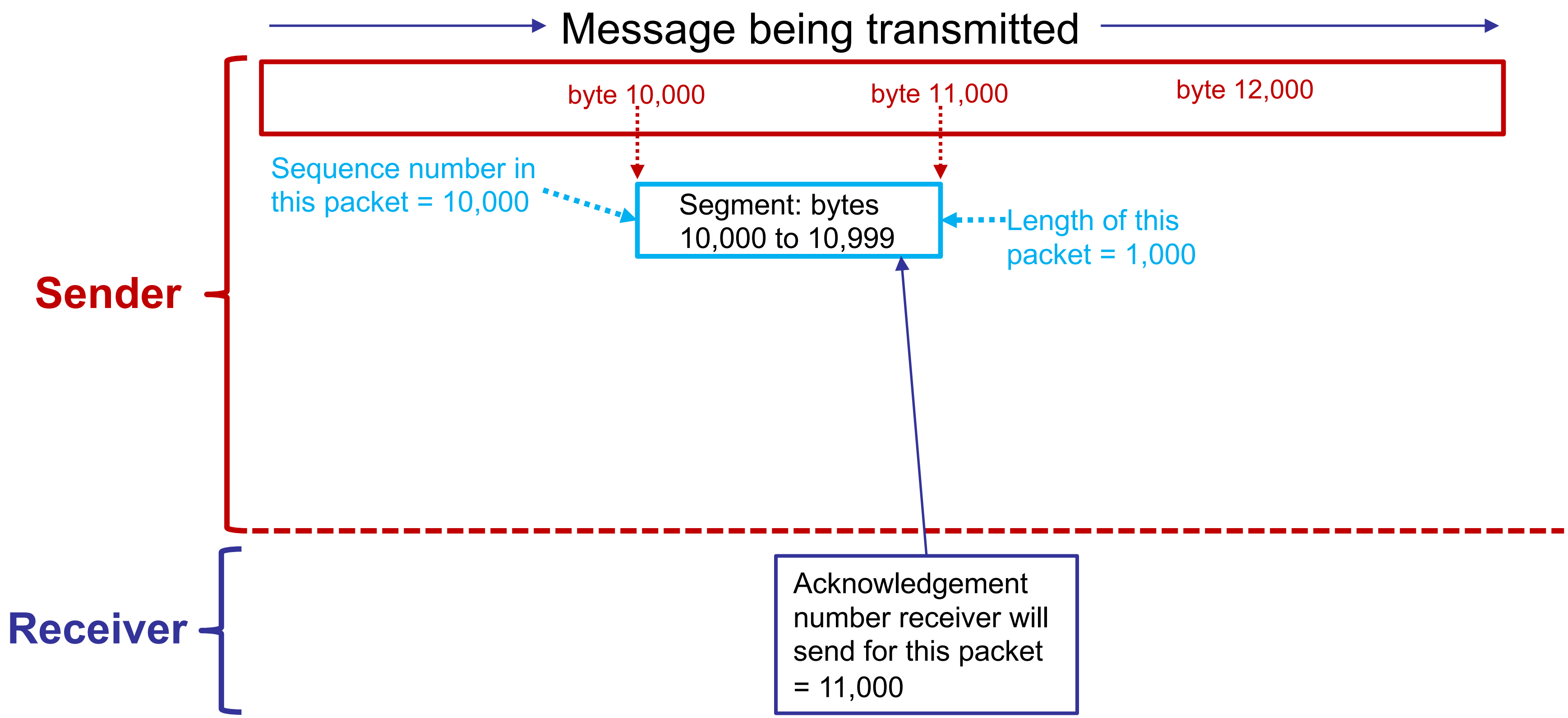
TCP Header

- The *SrcPort* and *DstPort* fields identify the source and destination ports, respectively.
- The *Acknowledgment*, *SequenceNum*, and *AdvertisedWindow* fields are all involved in TCP's sliding window algorithm.
  - Because TCP is a byte-oriented protocol, each byte of data has a sequence number; the *SequenceNum* field contains the sequence number for the first byte of data carried *in that segment*.
  - The *Acknowledgment* and *AdvertisedWindow* fields carry information back to the sender about the flow of data across the connection.

TCP Header

- The *acknowledgement number* is calculated from the sequence numbers and the lengths of the segments that are being acknowledged
- In other words, this indicates the *next sequence number* the receiver is *expecting* to receive
  - this supports *cumulative acknowledgements* since it covers all bytes received since the last ACK was sent
  - if a packet is received that contains sequence number 10,000 and has 1,000 bytes of data, byte 11,000 is the next byte expected at the receiver and that segment's acknowledgement number would 11,000

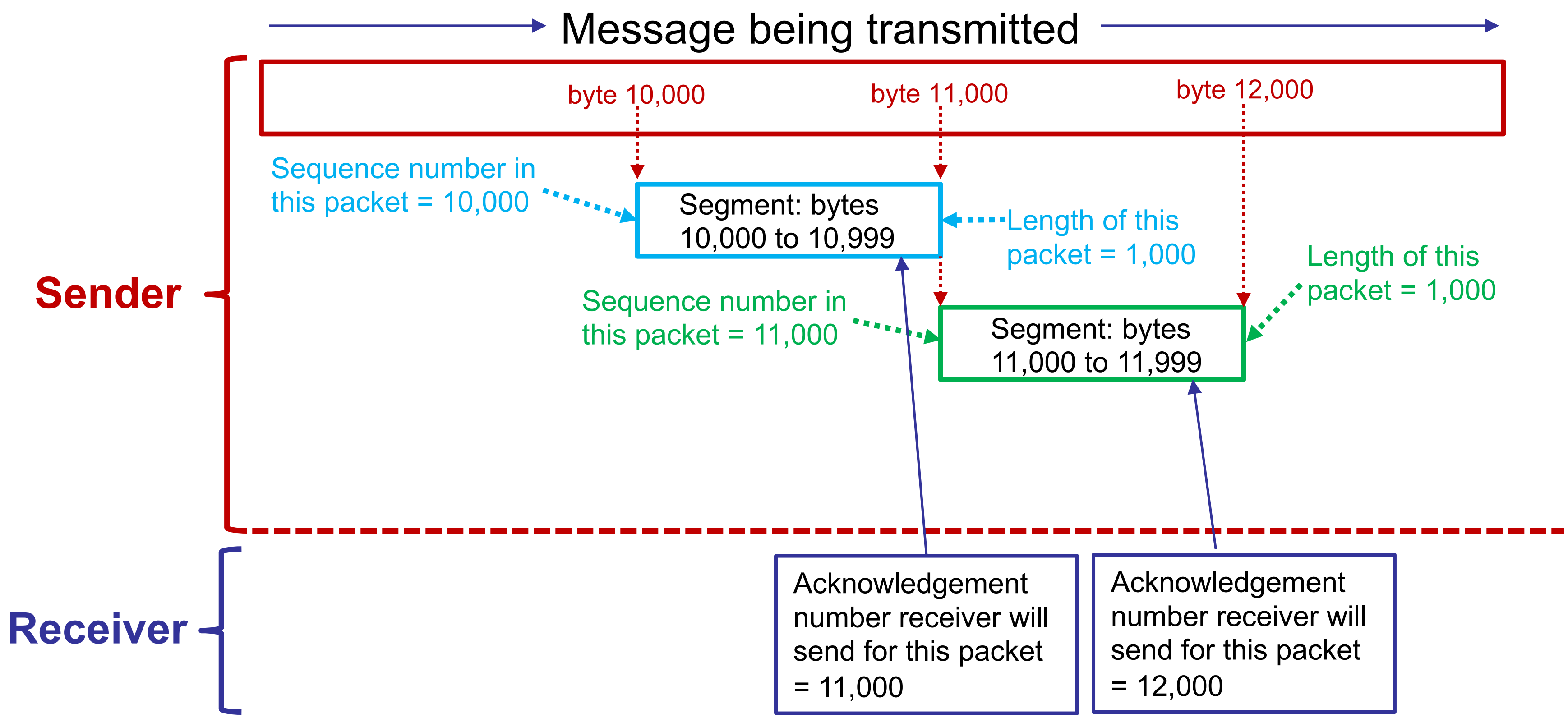
TCP Header



Exchange of packets in a TCP connection

- sequence numbers in packets from sender
- acknowledgement numbers in replies from receiver

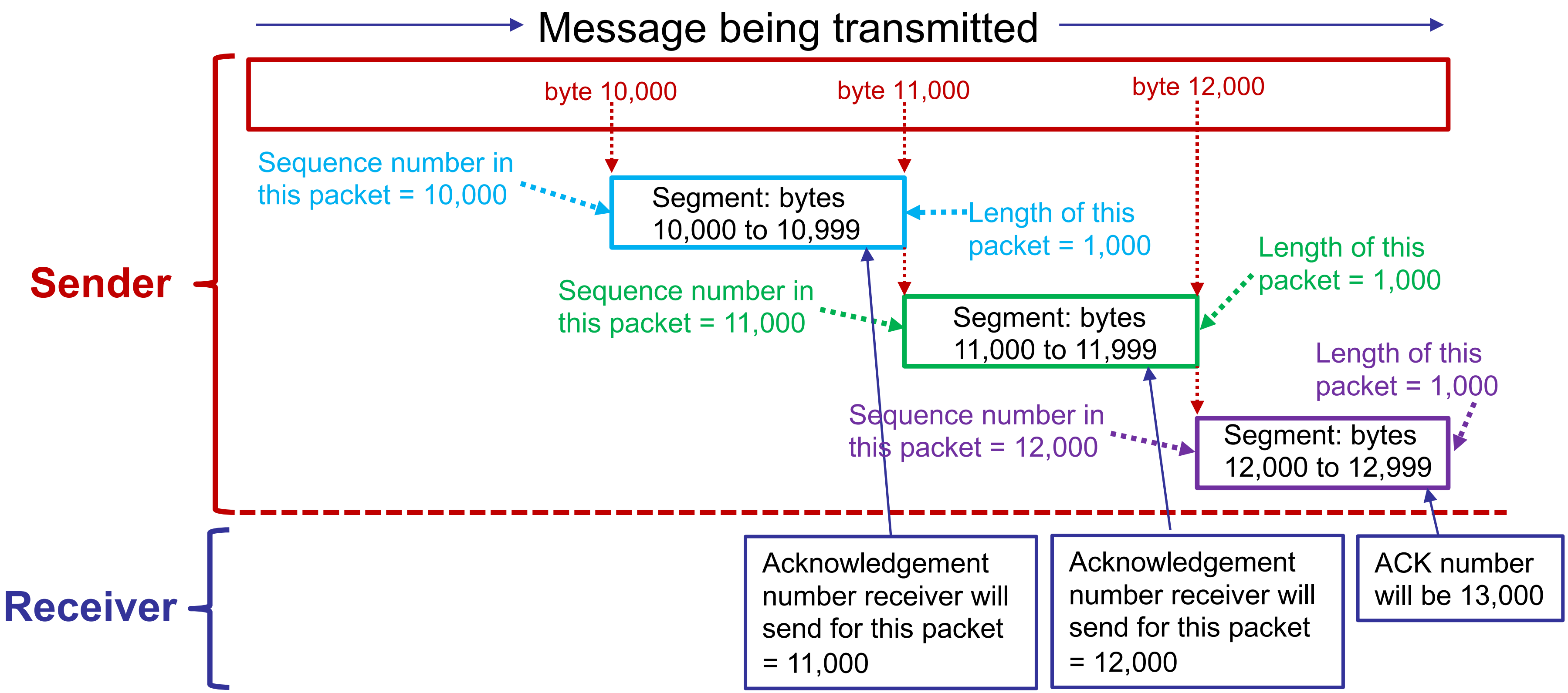
TCP Header



Exchange of packets in a TCP connection

- sequence numbers in packets from sender
- acknowledgement numbers in replies from receiver

TCP Header



Exchange of packets in a TCP connection

- sequence numbers in packets from sender
- acknowledgement numbers in replies from receiver

TCP Header

- The 6-bit *Flags* field is used to relay control information between TCP peers.
  - The possible flags include *SYN*, *FIN*, *RESET*, *PUSH*, *URG*, and *ACK*.
  - The SYN and FIN flags are used when establishing and terminating a TCP connection, respectively.
  - The ACK flag is set any time the Acknowledgment field is valid, notifying the receiver that this packet acknowledges reception of an earlier packet.
  - This allows a return packet to both carry data and to signal Acknowledgement of a previous packet

U	0x20	Urgent
A	0x10	Ack
P	0x08	Push
R	0x04	Reset
S	0x02	Syn
F	0x01	Fin

TCP Header

- The *URG* flag signifies that this segment contains urgent data. When this flag is set, the UrgPtr field indicates where the non-urgent data in this segment begins.
  - The urgent data is contained at the front of the segment body, up to and including a value of UrgPtr bytes into the segment.
- The *PUSH* flag is used to notify the receiver that it should send all of the data in the input buffer to the application (most applications ignore this feature)
- Finally, the *RESET* flag signifies that the receiver has become confused and wants to end the connection

TCP Connection Management

- Connection establishment
  - *3-way handshake*
- Flow Control
  - *Sliding Window* flow control protocol
  - *Feedback* from receiver for window adjustment
- Congestion Control
  - *Retransmission timeout* (RTO)
  - *Round Trip Time* (RTT)
- Connection termination
  - *4-way handshake*

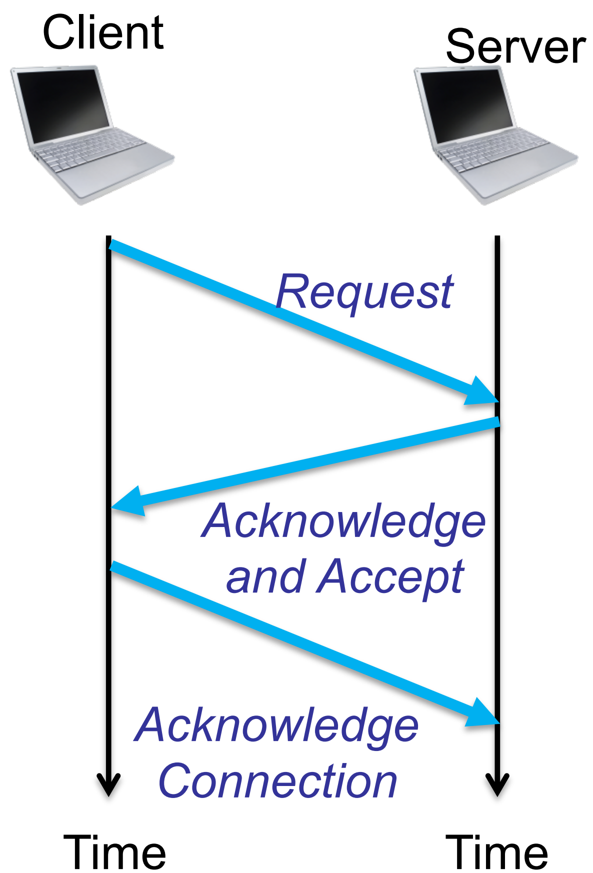
TCP Connection Management

The purpose of the *three-way connection process* is to increase the probability that both endpoints "know" that the connection request was accepted

- without the final ACK, the server cannot be sure that the client node received its notification that the request was accepted
  - the client's ACK is sometimes piggybacked on the first data packet from the requesting node, saving time
- when the server accepts the request, it makes an entry in a queue and starts a timer to wait for the client's ACK of the acceptance
  - if the timer runs out before the ACK is received, the entry is deleted and the acceptance is dropped

TCP Connection Establishment

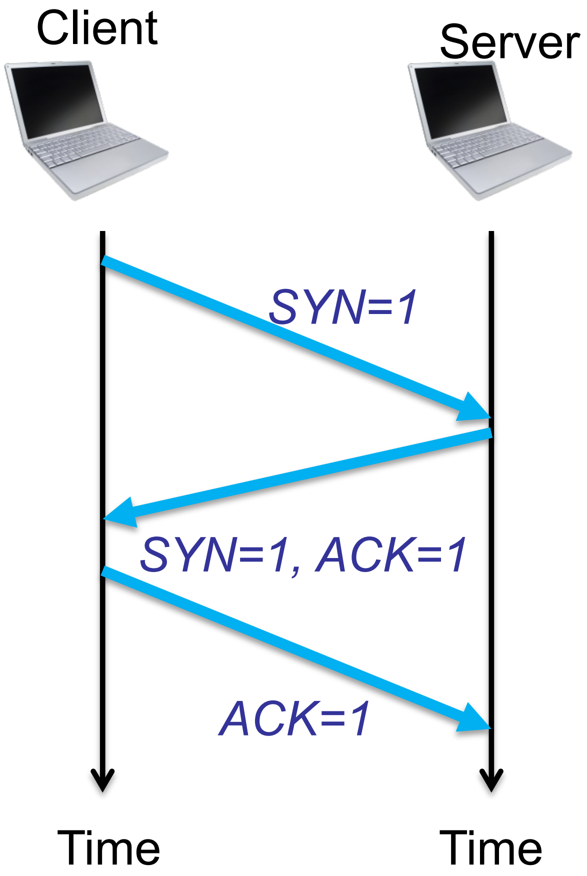
- TCP uses a *3-way handshake* for connection establishment between sender and receiver, before data can be transmitted
- The handshake process involves three steps:
  1. The client sends a request to create a connection to the server
  2. The server acknowledges that it received the request and agrees to accept the connection
  3. The client acknowledges that it received the server's acceptance and the connection is fully established



TCP Connection Establishment

- The *SYN*, and *ACK* flags in the TCP header are used to signal those steps in the connection process by setting bits to 0 or 1
- The *three-way handshake* involves the setting of each of these bits in the following order:
  1. SYN=1, ACK=0 (*SYN*) request connection
  2. SYN=1, ACK=1 (*SYN+ACK*) ACK request
  3. SYN=0, ACK=1 (*ACK*) confirm ACK
- The handshake process happens for each individual connection, as defined by the combination of:

  [srcIP, srcPort, destIP, destPort]



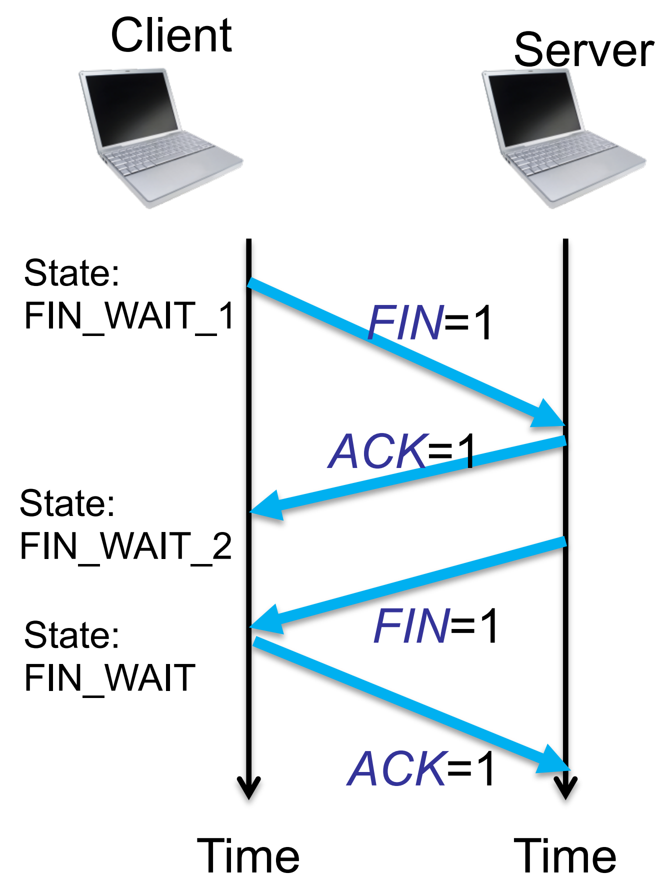
TCP Connection Establishment

Time	Source	Destination	Protocol	Length	Info
0.017558	172.16.50.21	172.16.31.3	TCP	74	20 → 34308 [SYN] Seq=0 Win=28200 Len=0 MSS=1410
0.017589	172.16.31.3	172.16.50.21	TCP	74	34308 → 20 [SYN, ACK] Seq=0 Ack=1 Win=27960 Len=0
0.017731	172.16.50.21	172.16.31.3	TCP	66	20 → 34308 [ACK] Seq=1 Ack=1 Win=28288 Len=0

1. the node at 172.16.50.21 *requests* a TCP connection with the node at 172.16.31.3
2. the node at 172.16.50.3 *acknowledges* the request from the node at 172.16.31.21 and confirms that it *accepts* (with the SYN bit)
3. the node at 172.16.50.21 *acknowledges* the acceptance from the node at 172.16.31.3 and the TCP connection is established

TCP Connection Termination

- The termination of a TCP connection also requires a multi-stage negotiation between end-points
- After sending a FIN request, the client waits on:
  - *FIN_WAIT_1* (waiting for ACK)
  - *FIN_WAIT_2* (waiting for FIN)
  - *FIN_WAIT* (if ACK is lost, allows a retransmission of the FIN)
- Requires two FIN and two ACK



TCP Connection Termination

```
0.017820   172.16.50.21    172.16.31.3     FTP    120 Response: 150 Opening ASCII mode data connection
0.017979   172.16.31.3     172.16.50.21    TCP     66 34308 → 20 [FIN, ACK] Seq=1 Ack=1 Win=28032 Len=0
0.018152   172.16.50.21    172.16.31.3     TCP     66 20 → 34308 [FIN, ACK] Seq=1 Ack=2 Win=28288 Len=0
0.018164   172.16.31.3     172.16.50.21    TCP     66 34308 → 20 [ACK] Seq=2 Ack=2 Win=28032 Len=0
```

The last data item is transferred from 172.16.50.21

1. the node at 172.16.31.3 *acknowledges* that data and *requests disconnection* (FIN)
2. the node at 172.16.50.21 *acknowledges* that request and *confirms disconnection* (FIN+ACK)
3. the node at 172.16.31.3 *acknowledges* that *disconnection request* (FIN+ACK) and the connection is terminated

Note: piggybacking ACK allowed fewer packets, but each endpoint sent a FIN and at least one ACK

TCP Connection State Modeling

The TCP connection finite-state machine has more states than the UDP example

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

TCP Connection State Modeling

Solid line is the normal path for a client.

Dashed line is the normal path for a server.

Light lines are unusual events.

Transitions are labeled by the cause and action, separated by a slash.