University of Southern California

Viterbi School of Engineering

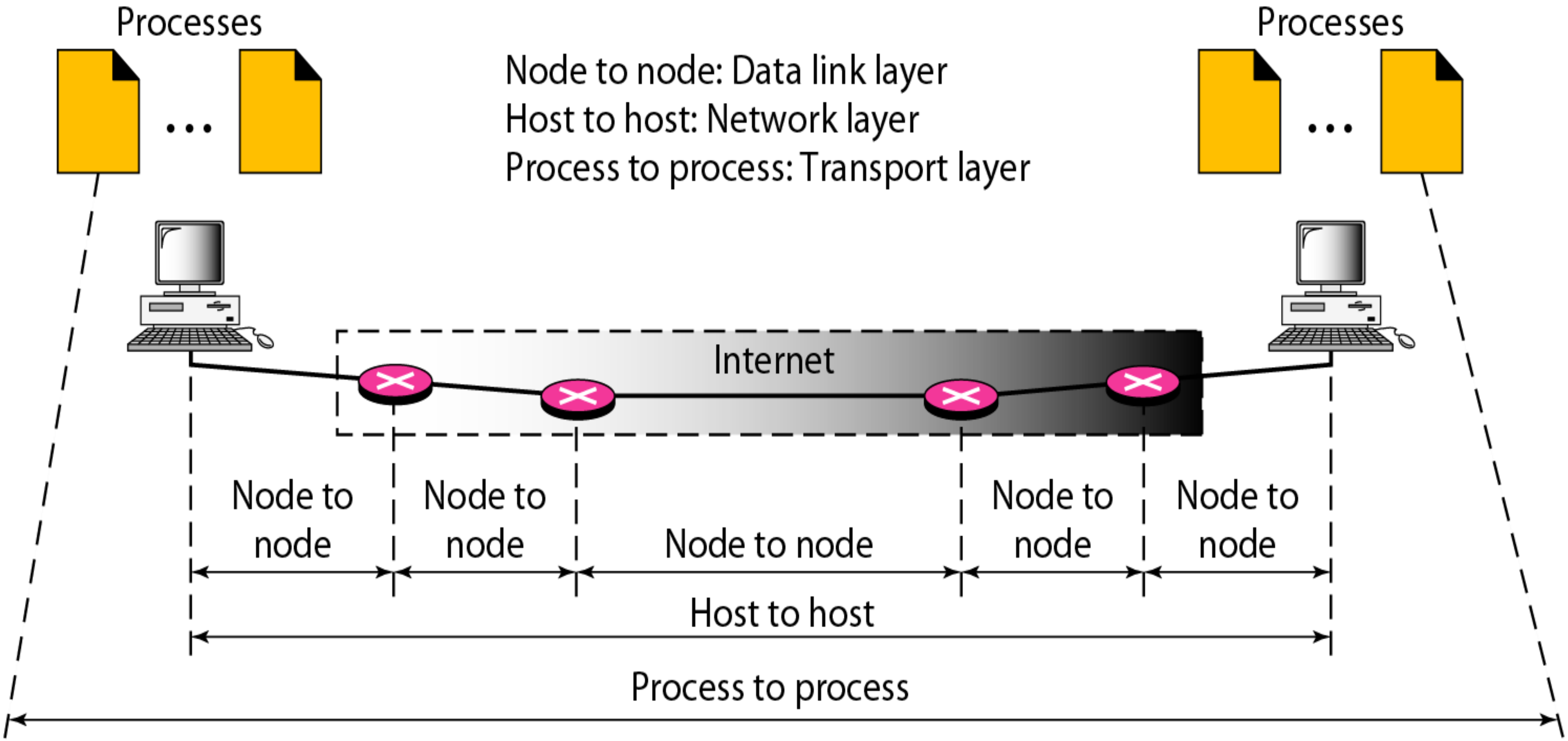
EE450
Computer Networks

Transport Layer

Shahin Nazarian

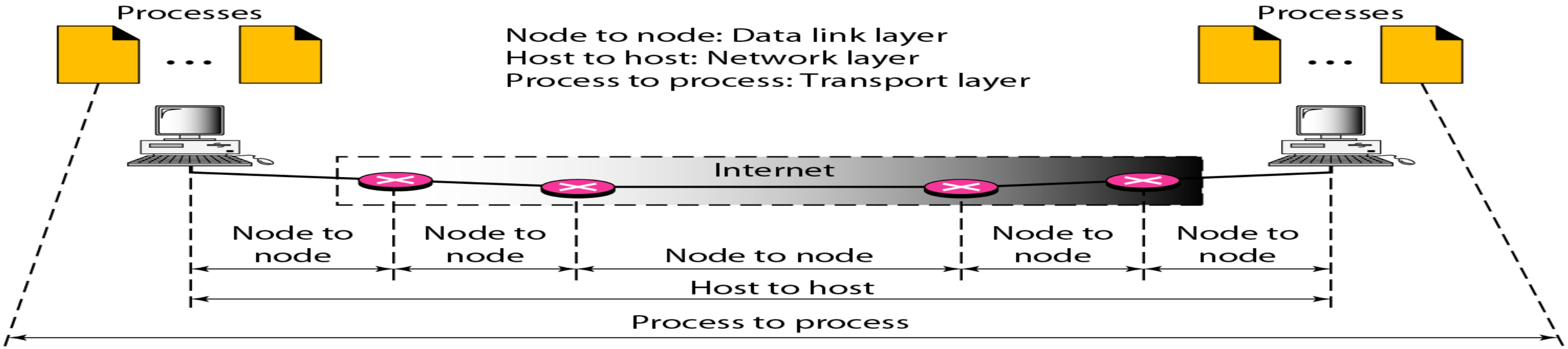
Spring 2013

Layer Relations



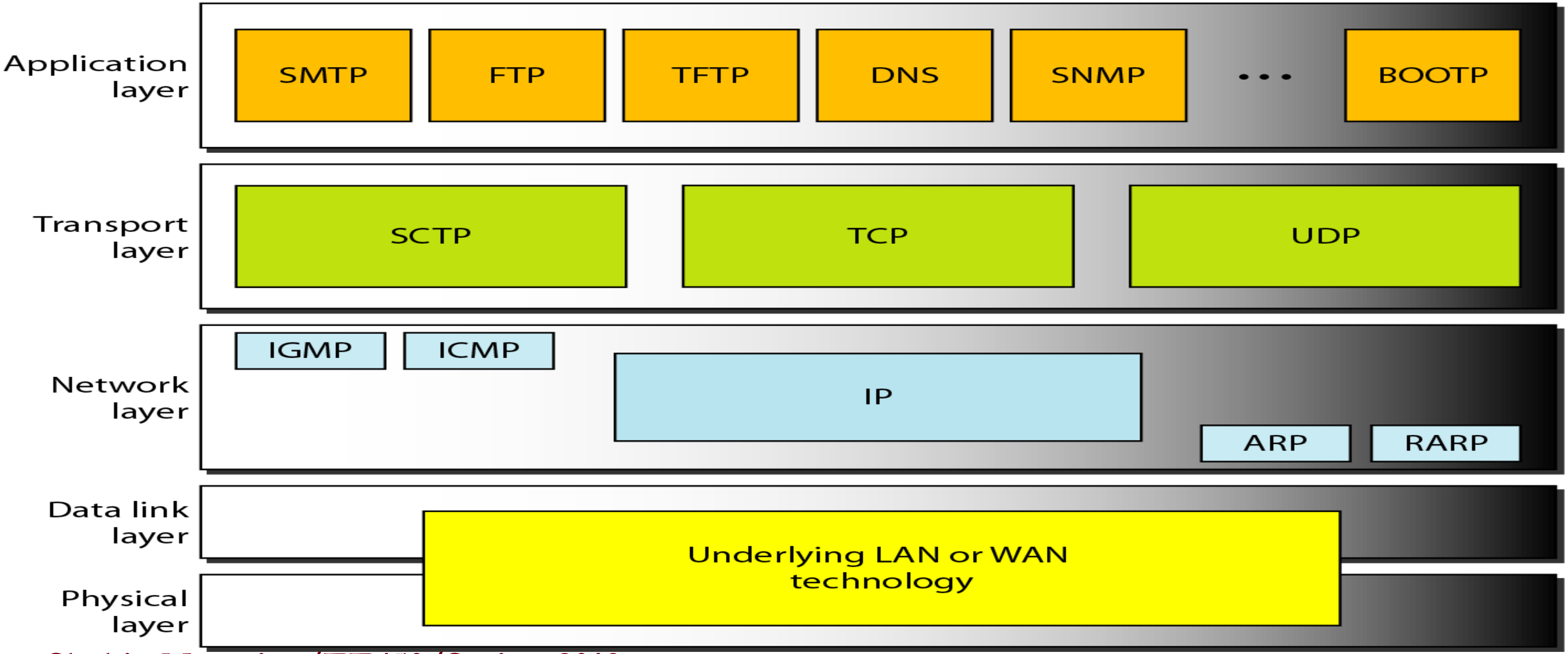
Transport Layer

- Data Link layer is about delivery of frames. Network layers about delivery of packets.
- Real communication takes place between two processes (application programs)
- At any moment several processes may be running on the source host and several on the destination host. To complete the delivery we need a mechanism to deliver data from one of these processes on source to the corresponding process on destination host



Transport Layer Protocols

- Two processes communicate in a client/server relationship
- Transport layer is only looked up on in the end hosts

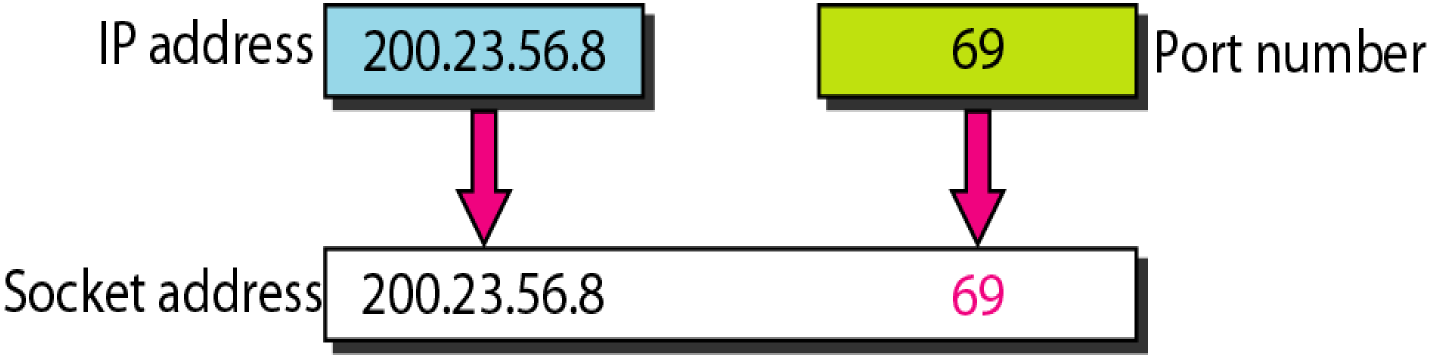


Transport Layer Protocols (Cont.)

- IP does not provide complete delivery. IP only delivers the packets to the host. However, the message which is encapsulated inside the packet needs to be delivered to the actual application; this delivery is done by the Transport layer protocols
- The following Transport layer protocols are defined by TCP/IP: TCP (Transmission Control Protocol), UDP (User Datagram Protocol) and SCTP (Stream Control Transport Protocol)
- TCP provides for end-to-end error checking, error control, flow control and congestion control
- UDP provides for end-to-end error detection only

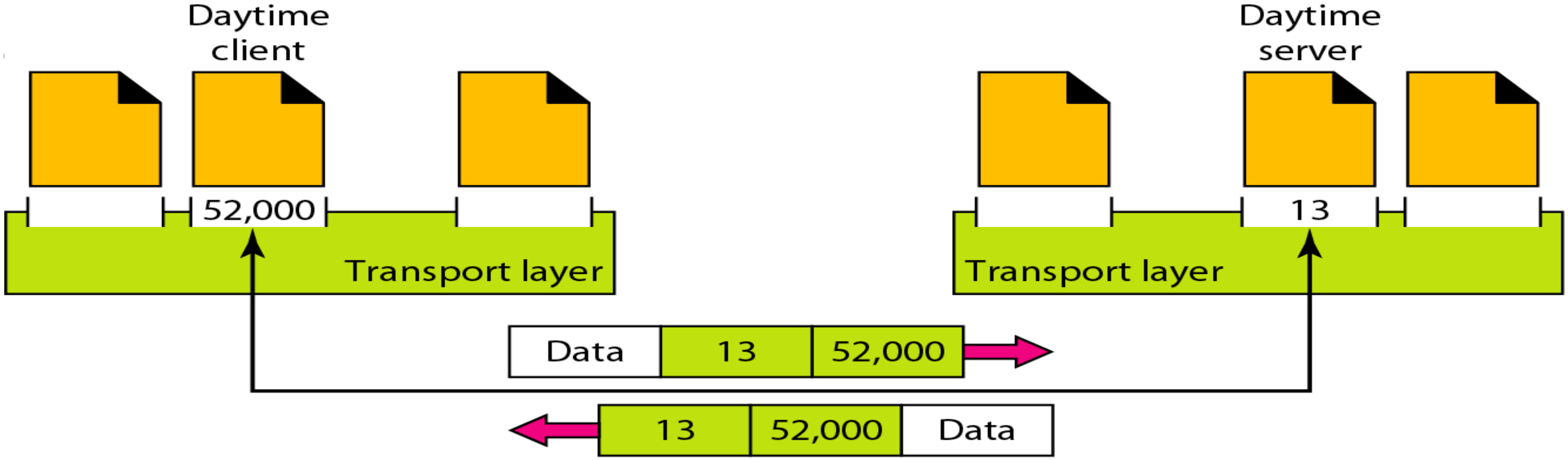
Transport Layer Protocols (Cont.)

- Error and flow control in Link layer were on a link-to-link basis, here the services are end-to-end, however the same ideas (ARQ Go-Back-N, Selective Repeat, …) are applied
- Local (remote) host is identified by its IP address, and local (remote) process by its port number
- IP address, port number and the transport protocol are together referred to as a socket



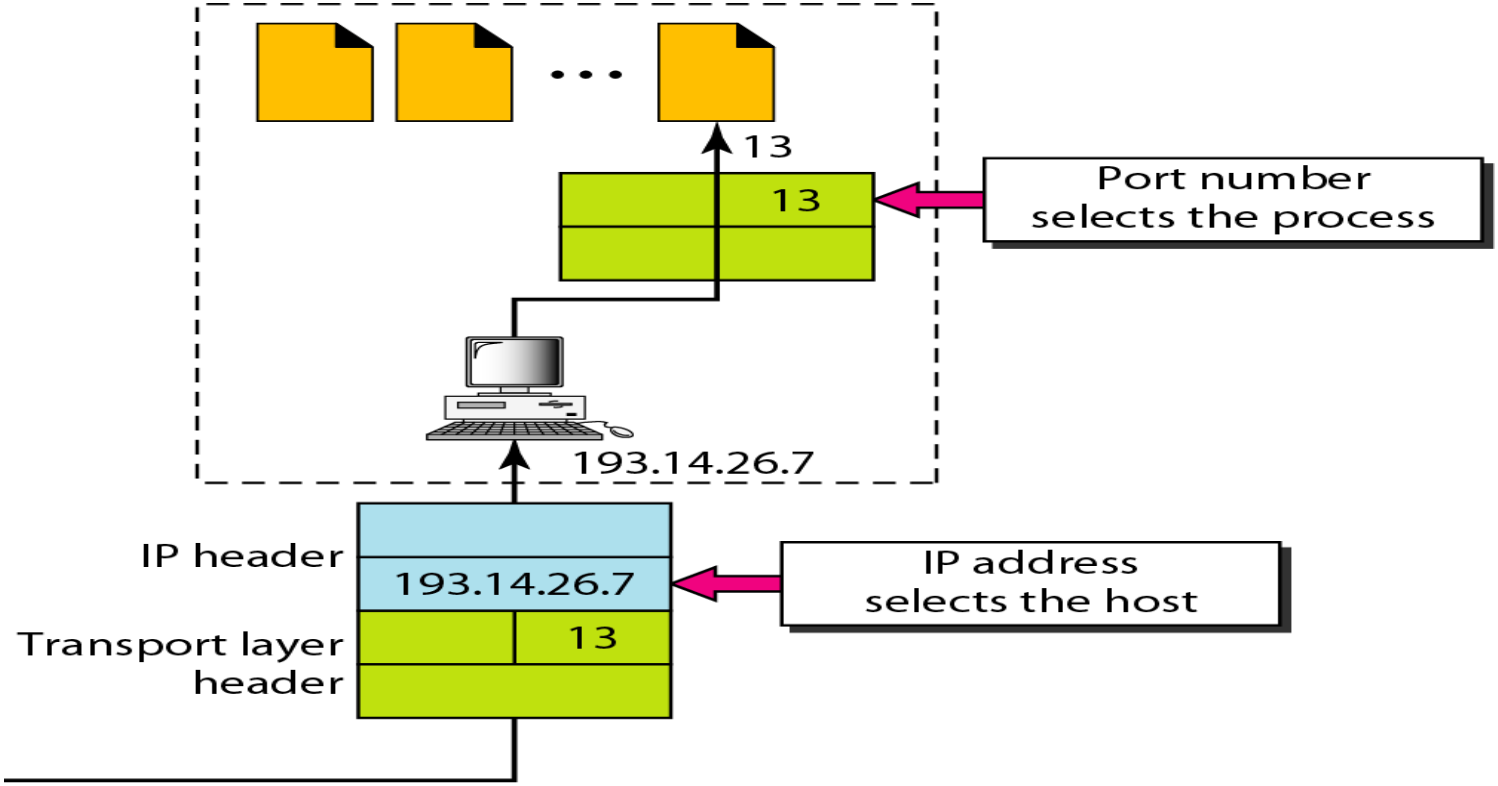
Port Numbers

- IP layer provides for header error checking and the data part may still be erroneous when it gets to Transport layer. UDP and TCP error checking provides for error checking on that
- When we write an application program we create a process
- Transport layer uses port numbers to distinguish btn processes



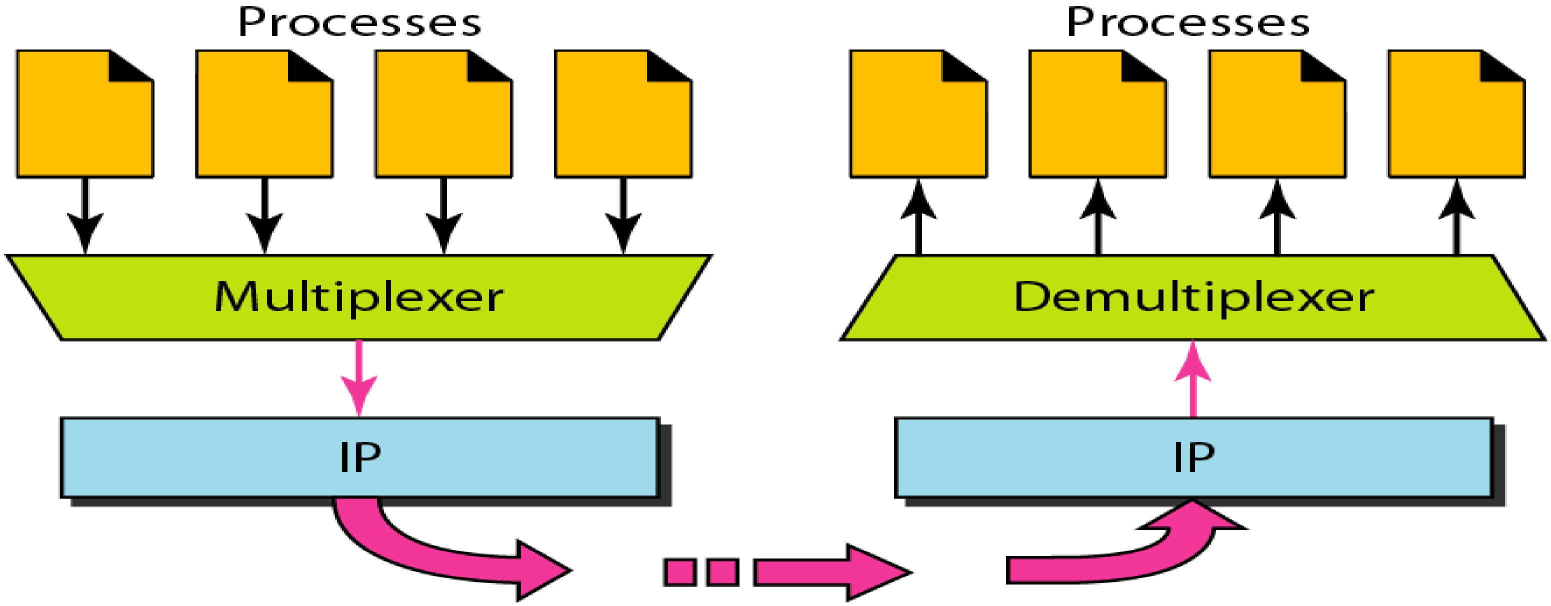
IP Addresses versus Port Numbers

- Well-known port numbers range between 0 and 1023
- Note that routers do not see the port number. To a router a port number is nothing but a part of the IP packet

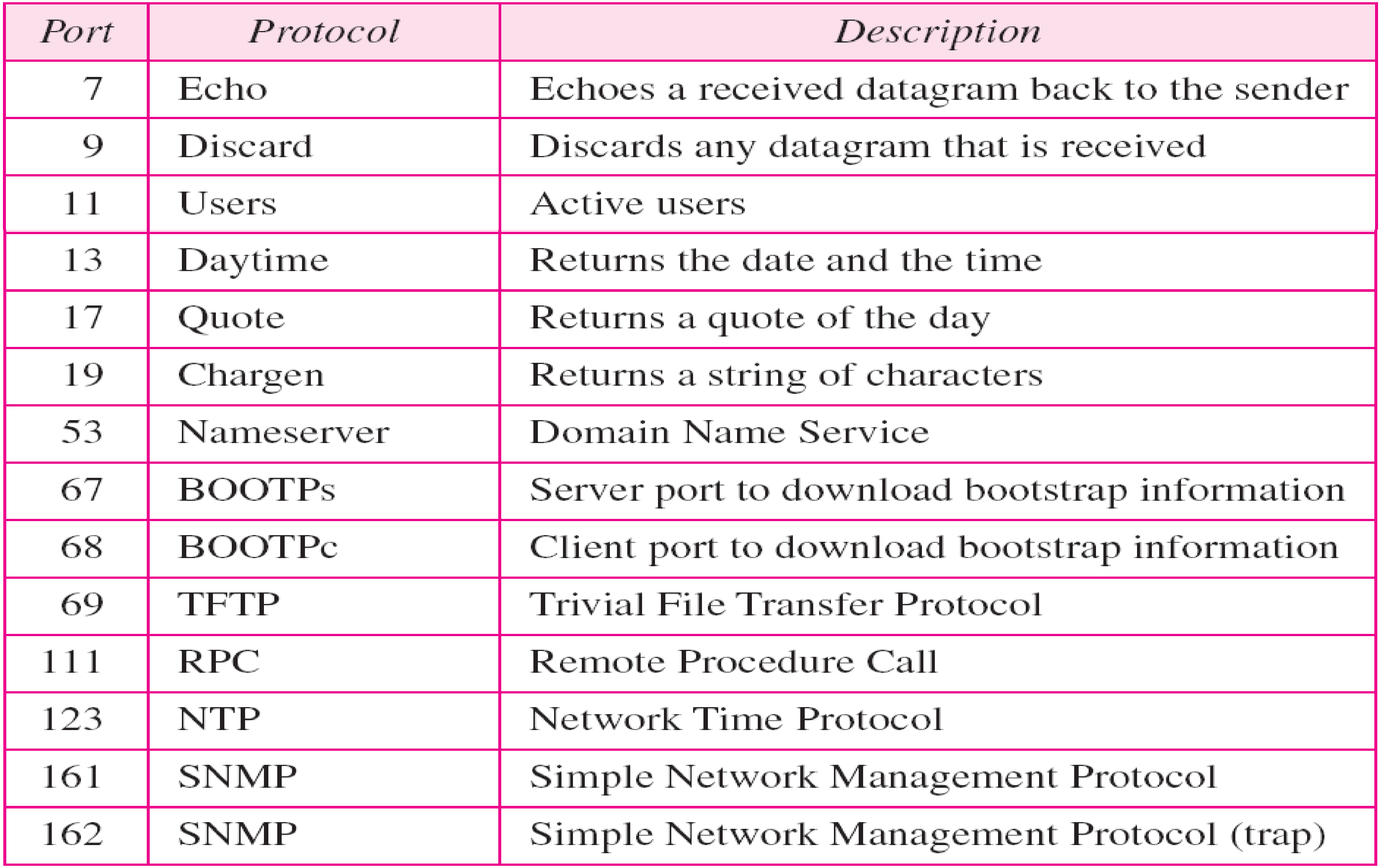


Process Aggregation

- We saw TDM and FDM multiplexing in Physical layer that aggregate traffic from several sources into a single link
- TCP or UDP do multiplexing by combining different applications into a single IP



Well-known Port Numbers Used by UDP



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)

A Note on DNS

- DNS primarily uses User Datagram Protocol (UDP) on port number 53 to serve requests

- DNS queries consist of a single UDP request from the client followed by a single UDP reply from the server

- However, DNS uses TCP when the response data size exceeds 512 bytes, or for some of its tasks

- Some operating systems, such as HP-UX (HP-Unix) are known to have resolver implementations that use TCP for all queries, even when UDP would suffice

User Datagram Protocol (UDP)

- UDP is a connectionless (i.e., **with no handshaking**,) **unreliable end-to-end** protocol
- Connectionless: if the local process has something to transmit to the remote process, it can go ahead and do so
- Unreliable: if the receiver detects an error, it will simply drop it and does not ask the sender to retransmit. Therefore it is possible the receiver receives incomplete message
- Applications that involve short request/response use UDP. This is because for them it does not make sense to go through the handshaking process (as is done in TCP.) Such applications, e.g., VoIP, cannot tolerate the delay of handshaking process and setup, however they can tolerate occasional errors

UDP (Cont.)

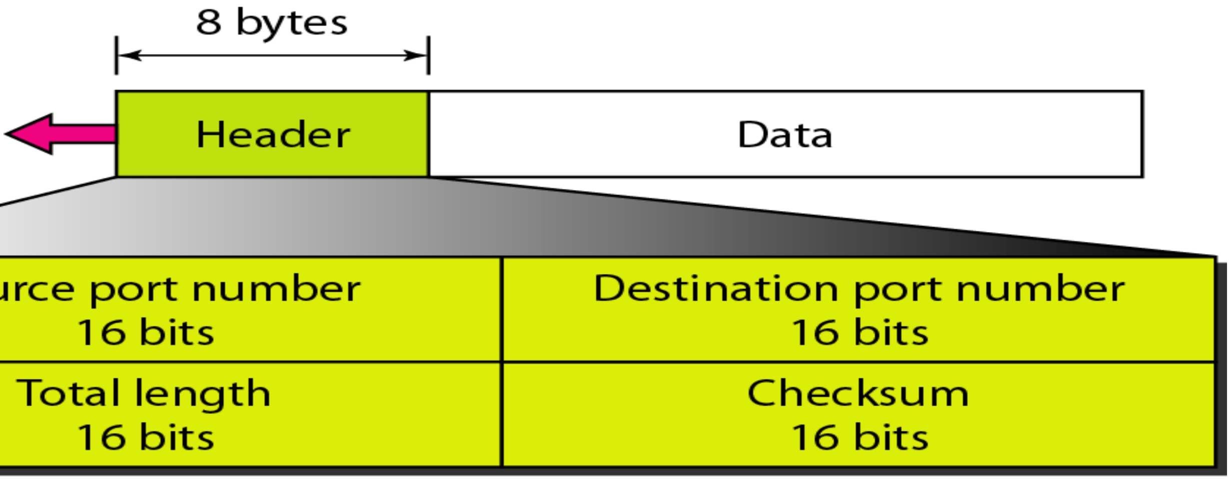
- Example: DNS – an end system has a host name and asks for the host's IP address. The messages are short here, therefore UDP is mainly used

- "Packet" and "datagram" may be used interchangeably as the unit of data in layers 3 and 4, however we would like to be consistent and use packet for layer 3 and datagram for layer 4 UDP

- Comparing **UDP datagram** and **TCP segment**, datagram is very simple. More functionality in TCP makes the header of the segment more complicated

UDP Datagram Format

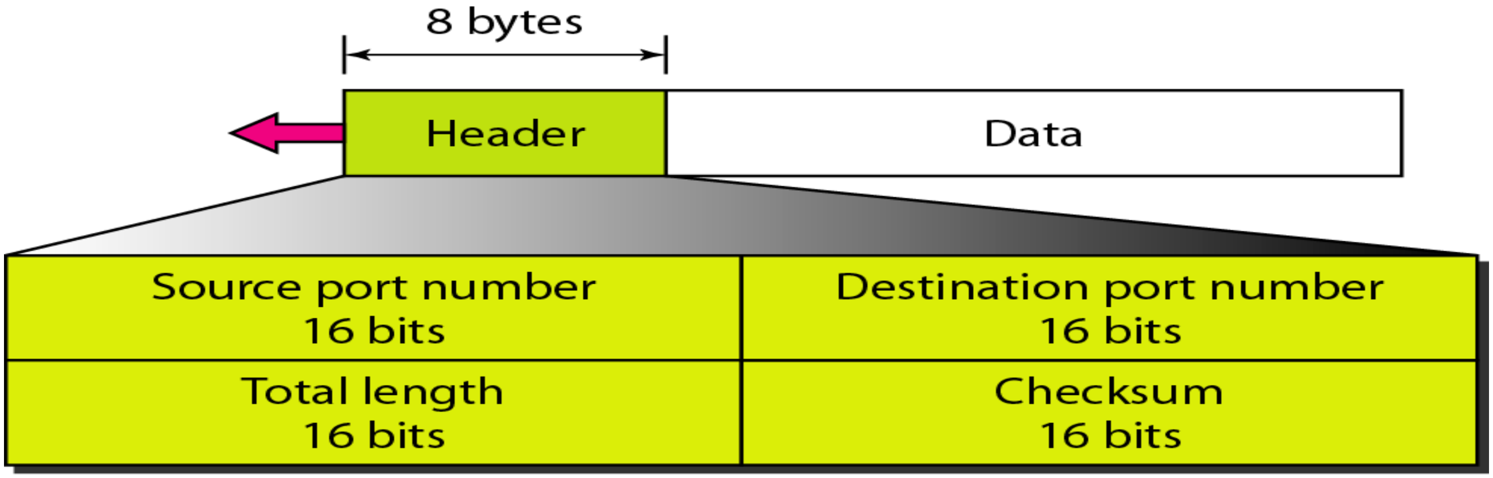
- Question: Is the source port number the well-known port number?
  - It is, if the sender is the server
- Checksum is the field for the end-to-end checking. The check is for the entire datagram (remember that in layer 3, IP header checksum was for the header part only)



Source port number 16 bits	Destination port number 16 bits
Total length 16 bits	Checksum 16 bits

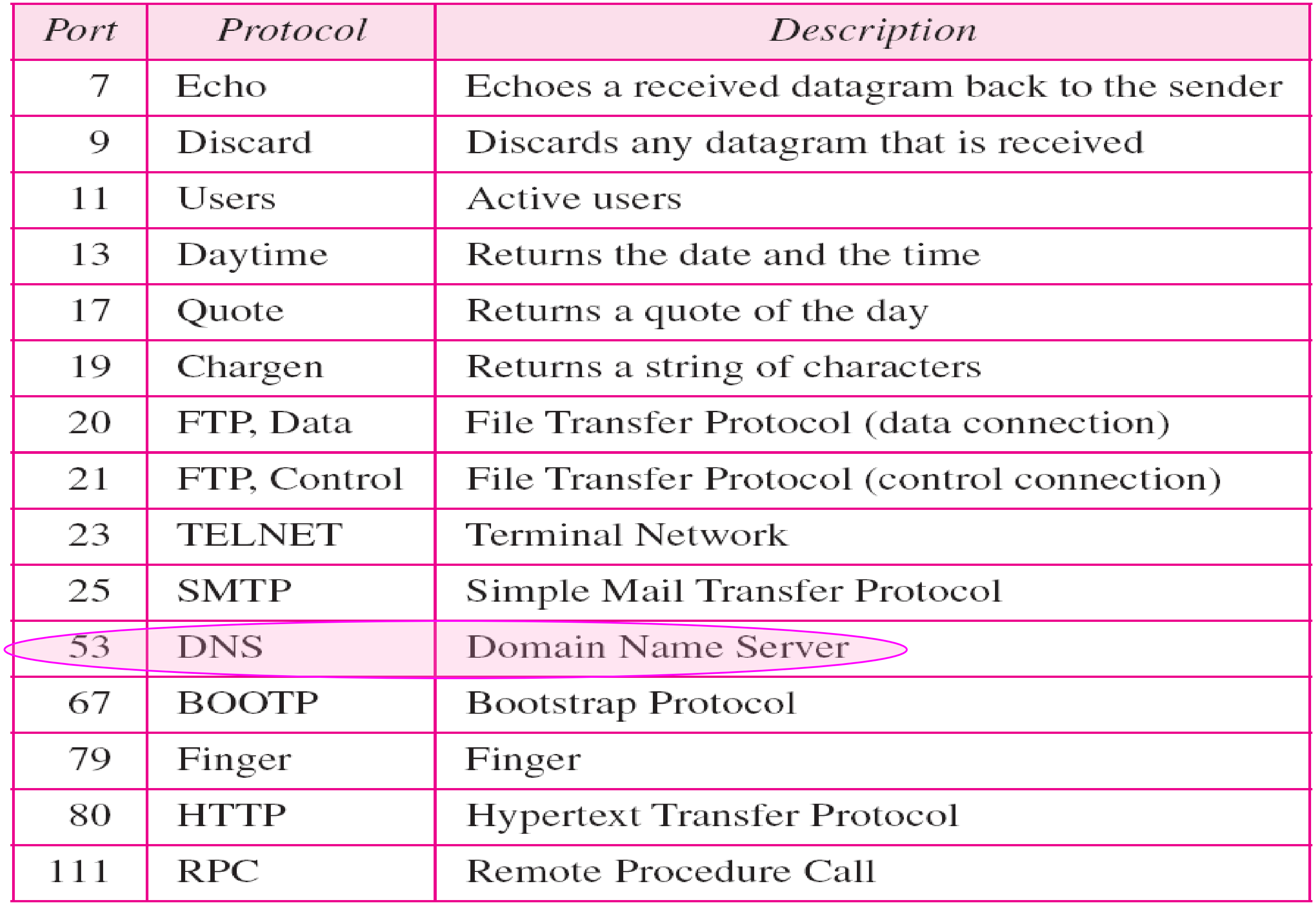
UDP Datagram Format (Cont.)

- Also keep in mind that the IP is inside the network, and packet is going through routers in the network and we do not want the routers to be slowed down to check the entire packet, alternatively, we would like the routers to check only for the header because the destination address is in the header and we leave error checking to layer 4
- We can have up to 2^{16} port addresses



Source port number 16 bits	Destination port number 16 bits
Total length 16 bits	Checksum 16 bits

Well-known Port Numbers 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

- **Read the notes on DNS (the page with UPD-based well-known port numbers**

Transport Control Protocol (TCP)

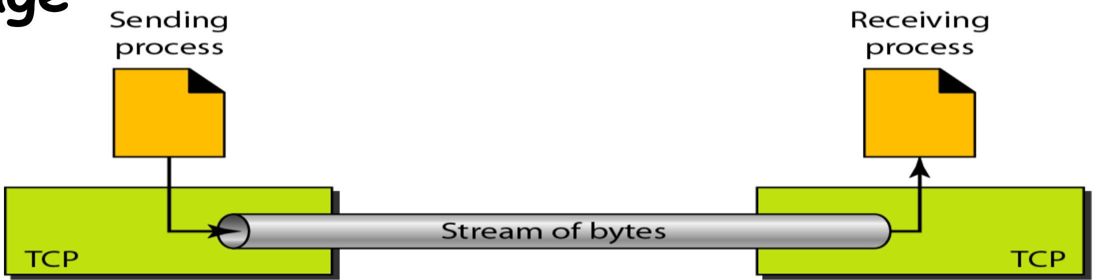
- TCP is a connection-oriented end-to-end protocol; it creates a virtual connection between two TCPs to send data
- Again we should note that connection-oriented characteristic of TCP does not have anything to do with the network
- TCP is point-to-point, therefore it does not support multicasting or broadcasting, because it is connection-oriented and needs handshaking process, i.e., TCP at the client site and TCP at the server site need to shake hands before client or the server can exchange data and handshaking in multicasting and broadcasting is not defined and supported in TCP

TCP (Cont.)

- Review our slides on socket programming. On the client site TCP, the socket is created, the connect command is issued (i.e., the connection-oriented relationship that is established between the client and the server)
- The client initiates the connection. On server site, the socket is created and bound and then it waits and listens for incoming requests from the client site
- TCP is a reliable service on top of IP which is an unreliable service, so TCP will target the issues created by IP's unreliability

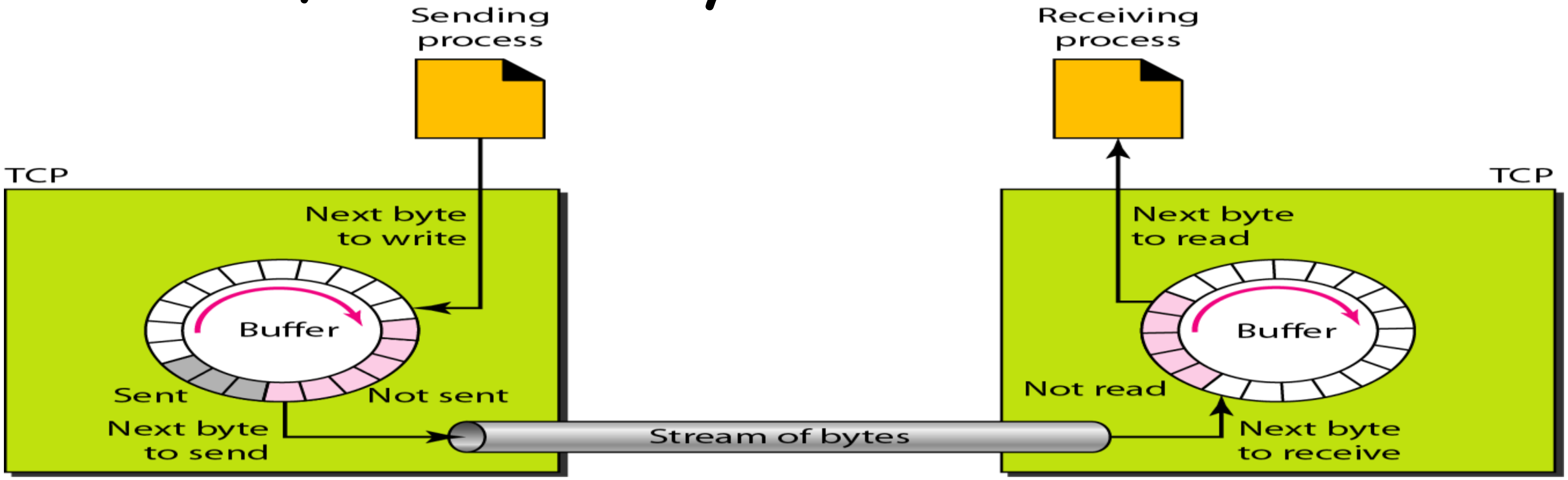
TCP (Cont.)

- TCP provides for FDX (full-duplex) service, i.e., both sides can transmit at the same time. Note that in half duplex the transactions are in the form of request, response which is not the case for TCP
- TCP is appropriate for applications such as HTTP, SMPT, Telnet, FTP, etc. which cannot tolerate data losses, however they can tolerate excessive delay required to setting up the logical connection using handshaking (this is not a physical connection)
- TCP is a connection-oriented protocol. This means TCP does not care where the start and the end of the message are. When the application, generates the data, it splits it into bytes. TCP does not care about size of the message and boundary of the message



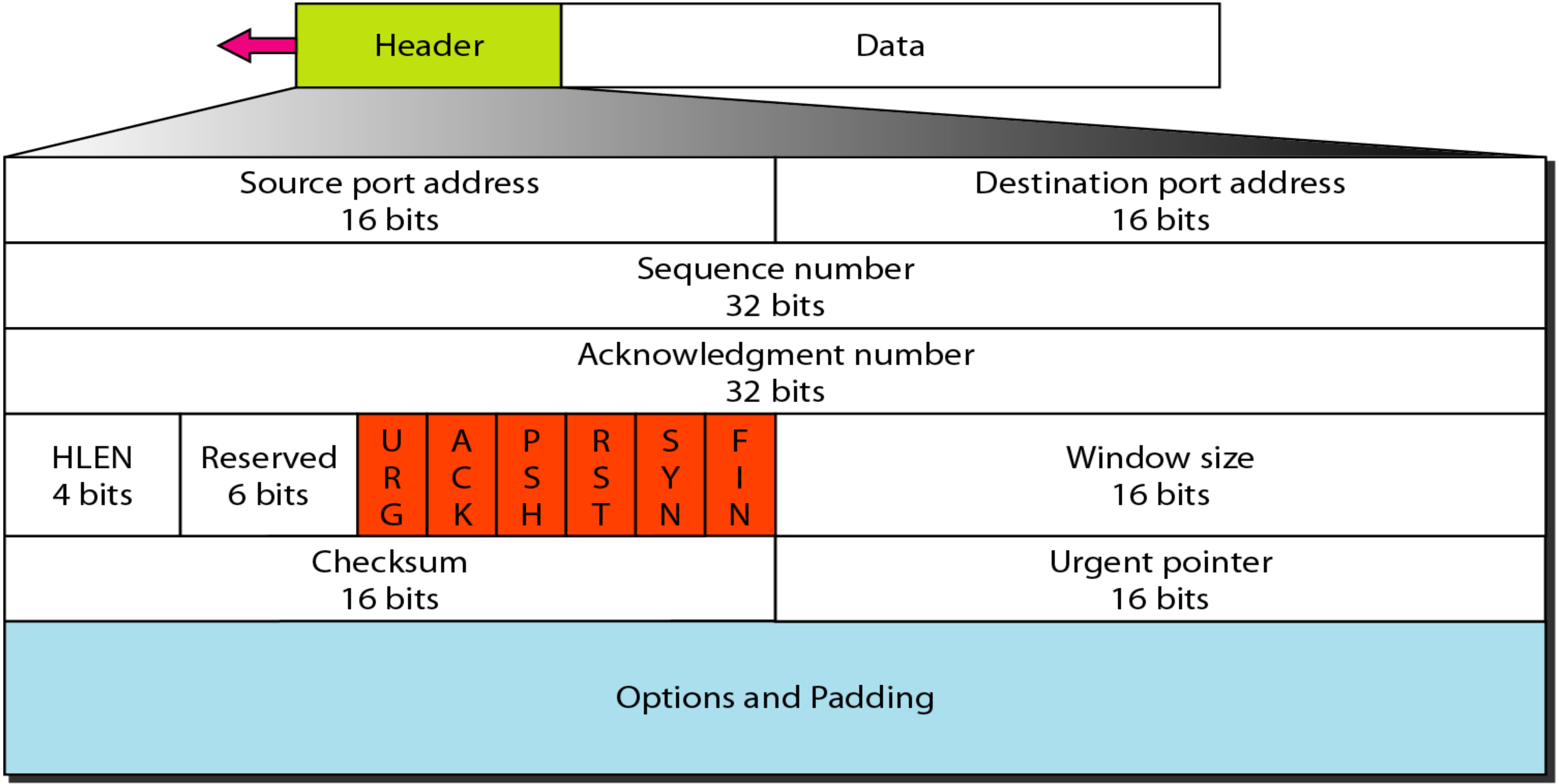
Sending and Receiving Buffers

- Bytes are inserted inside the segment. The window will specify how many bytes can still be transmitted
- The window-based mechanism is similar to what we've seen in Link layer (i.e., the frames that were sent.) Those that can still be sent, and those outside of the window and hence cannot be sent yet
- Also, every time sender transmits bytes, its window will shrink. Every time it gets ACK the window will slide. We saw this before, in the 2nd layer



TCP Segment Format

- Port numbers are exactly the same as those in UDP datagram



Header

Data

Source port address
16 bits

Destination port address
16 bits

Sequence number
32 bits

Acknowledgment number
32 bits

HLEN
4 bits

Reserved
6 bits

URG

ACK

PSH

RST

SYN

FIN

Window size
16 bits

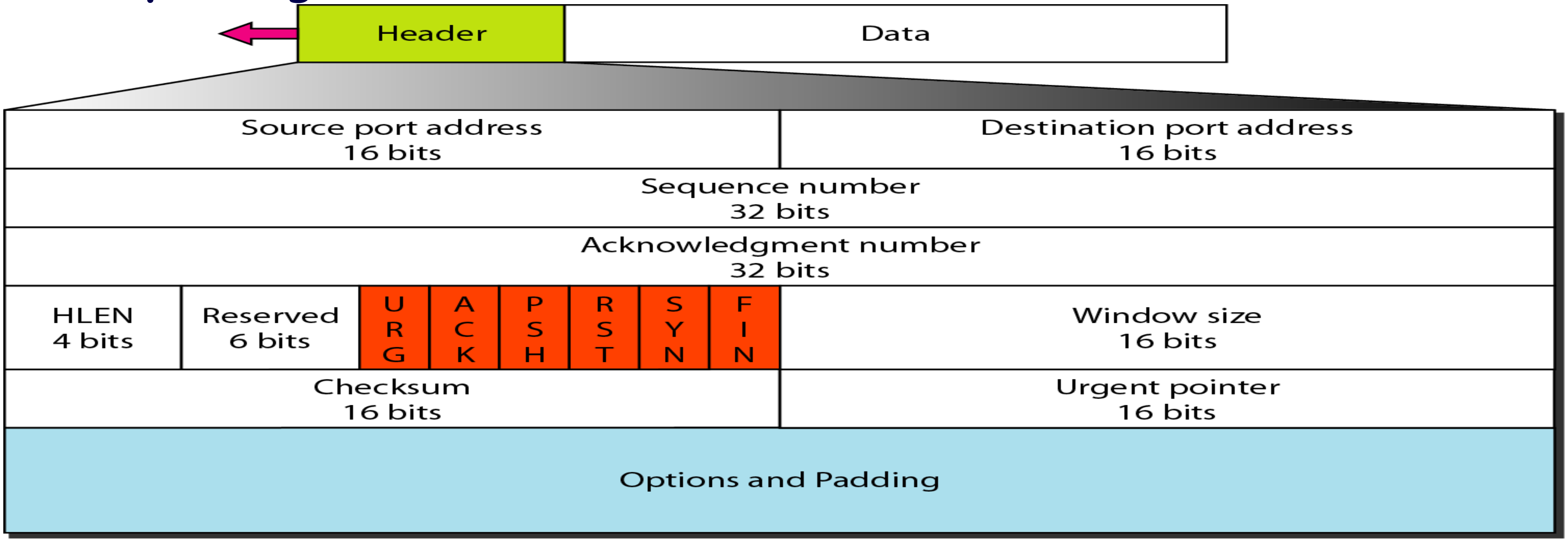
Checksum
16 bits

Urgent pointer
16 bits

Options and Padding

TCP Segment Format (Cont.)

- IP (layer 3) and UDP (layer 4) are connectionless and unreliable, so we did not need to have sequence numbers in an IP packet or UDP datagram as we do here for a TCP segment (we had sequence numbers also in point-to-point protocol in Link layer, but not the multi-point Ethernet.) Whenever you need to do error control and flow control, you need to sequence your messages, otherwise there is no way of requesting for retransmission



Source port address
16 bits | Destination port address
16 bits

Sequence number 32 bits

Acknowledgment number 32 bits

HLEN 4 bits	Reserved 6 bits	URG	ACK	PSH	RST	SYN	FIN	Window size 16 bits

Checksum 16 bits | Urgent pointer 16 bits

Options and Padding

TCP Segment Format (Cont.)

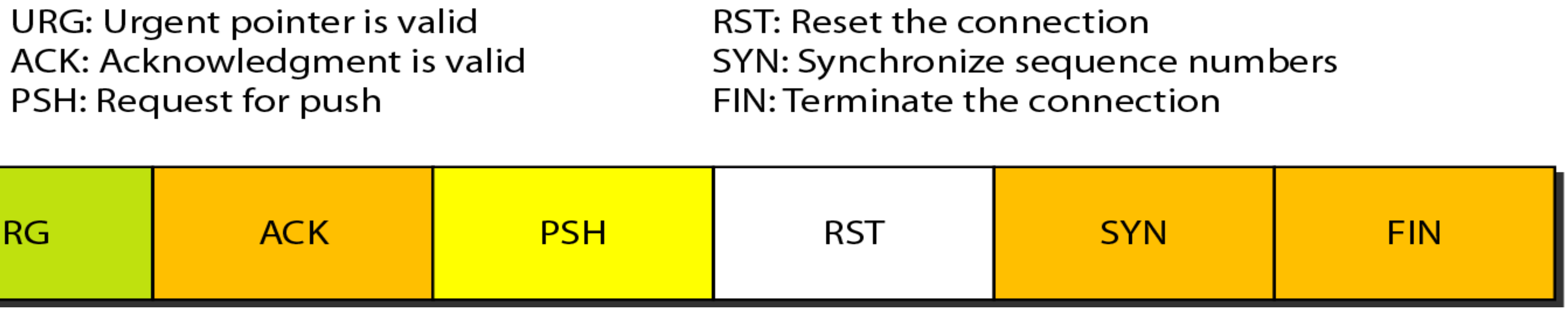
- Application layer messages can be huge, so even if we break them into segments, the number of bytes in each segment can be huge, so we have to wrap around and reuse the sequence number (only if the old sequence number is already acknowledged.) This concept is also similar to what we saw in Link layer
- Acknowledgement number, sent by receiver, is the number of the first byte expected to be received by the receiver
- HLEN shows the header length (20 to 60 bytes) similarly to what we saw for HLEN of IP packet header

TCP Segment Format (Cont.)

- Control segment has no data field, i.e., the destination has no data to send but it wants to acknowledge some received bytes
- However the sender can piggyback the acknowledgment with data bytes it is sending (instead of separate control and data segments)
- ACKs are cumulative, e.g., ACK500, means receiver has received all the bytes prior to 500, next if sender receives a segment with an ACK1400 it means receiver received all bytes prior to 1400
- Reserved is for future use (when we design a protocol, we better leave some bits for future, because later on we would be in trouble if we need extra bits, but have no free space

TCP Segment Format – Flags

- Several flags are used:
  - **Urgent pointer**: Specifies the last byte of urgent data in the segment. Urgent data starts from the first byte of the segment. The receiving program processes the urgent data immediately, then TCP informs the application and resumes back to stream queue
    - Urgent data can be sent with normal data
  - **URG** points out that the urgent pointer is valid
  - *ACK* means the ACK # is valid number
  - **RST** when the connection has some trouble, it is used to reset it



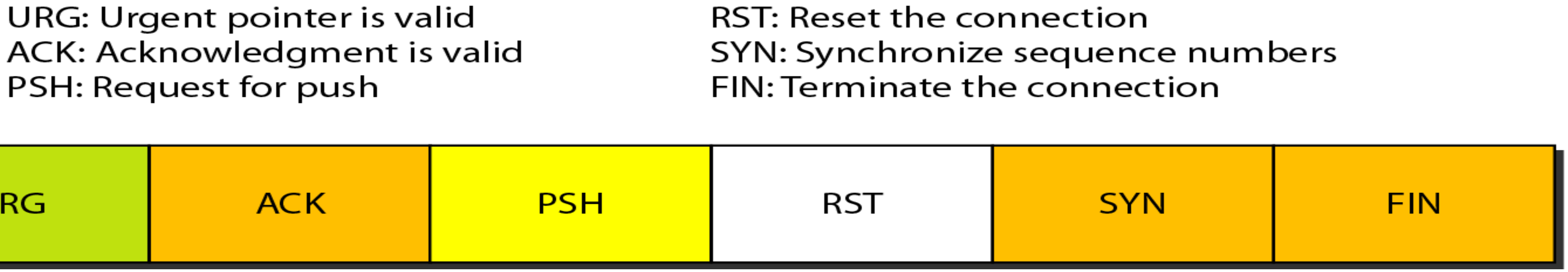
URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push

RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection

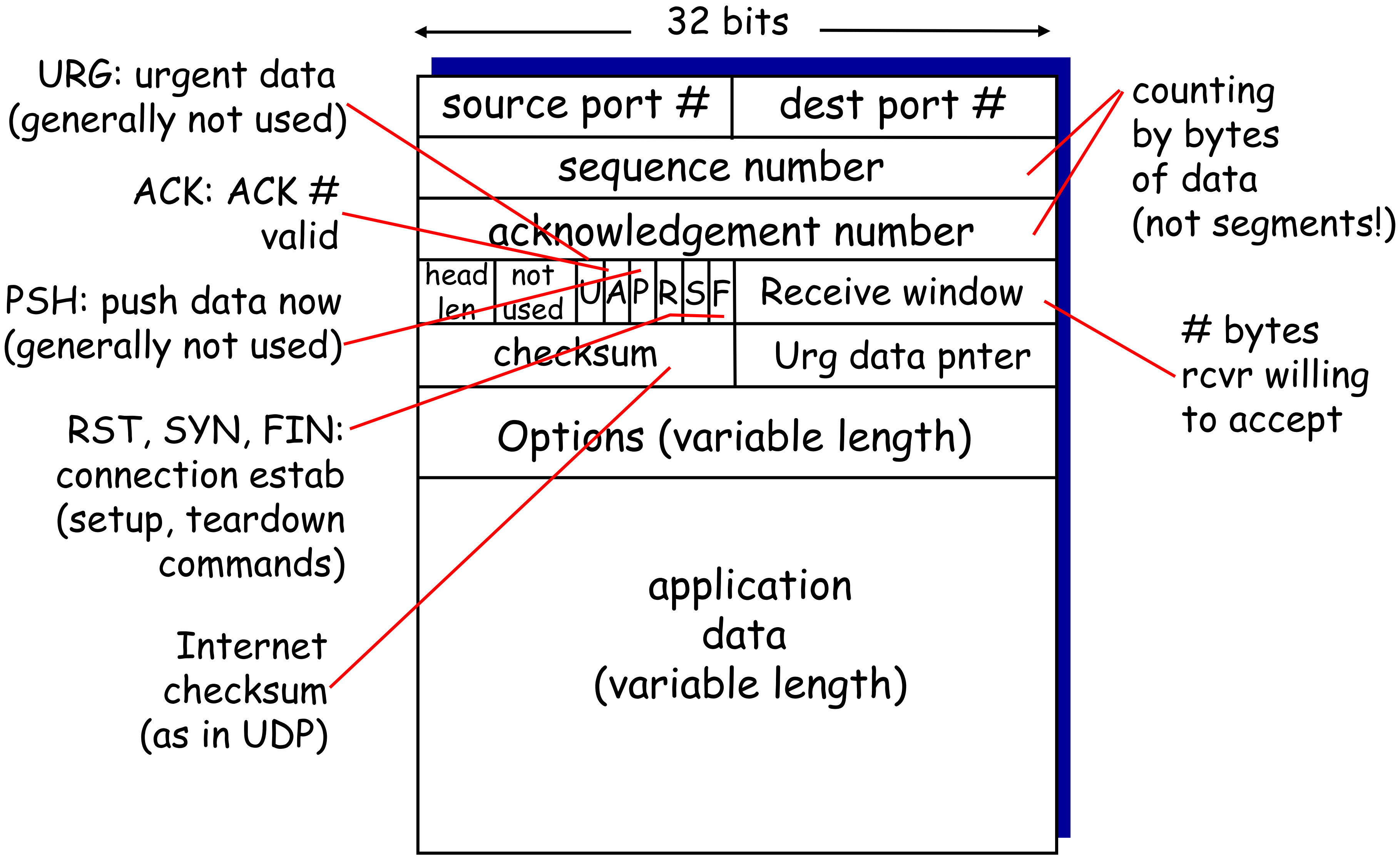
URG	ACK	PSH	RST	SYN	FIN

TCP Segment Format – Flags (Cont.)

- **PSH**: When an application has data that it needs to have sent across the internetwork immediately, it sends the data to TCP, and then uses the TCP *push function.* The destination device's TCP sw, seeing this bit sent, will know that it should not just take the data in the segment it received and buffer it, but rather push it through directly to the application layer
- Each site has two windows, send and receive both, because TCP supports FDX


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

Summary of TCP Segment Structure



32 bits

source port # | dest port #

sequence number

acknowledgement number

head len | not used | U | A | P | R | S | F | Receive window

checksum | Urg data pnter

Options (variable length)

application data (variable length)

URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection estab (setup, teardown commands)

Internet checksum (as in UDP)

counting by bytes of data (not segments!)

# bytes rcvr willing to accept

TCP Efficiency

- **Window size**: Unit in window size is byte. Maximum size is 2^{16} bytes (at the beginning it was thought this size was large enough)
- Suppose the bandwidth is 1Gbps and assume RTT of 256msec, then BW × Delay product will be 32MBytes. However maximum window size is 2^{16} = 65536 Bytes, therefore TCP efficiency is 65536Bytes/32MBytes =0.2%
- The solution is to use an optional field called the window scaling factor (in the option field) which is of power of 2; e.g., for a scaling factor of 2^2 the window size is $2^2 \times 2^{16} = 2^{18}$ Bytes

TCP Efficiency

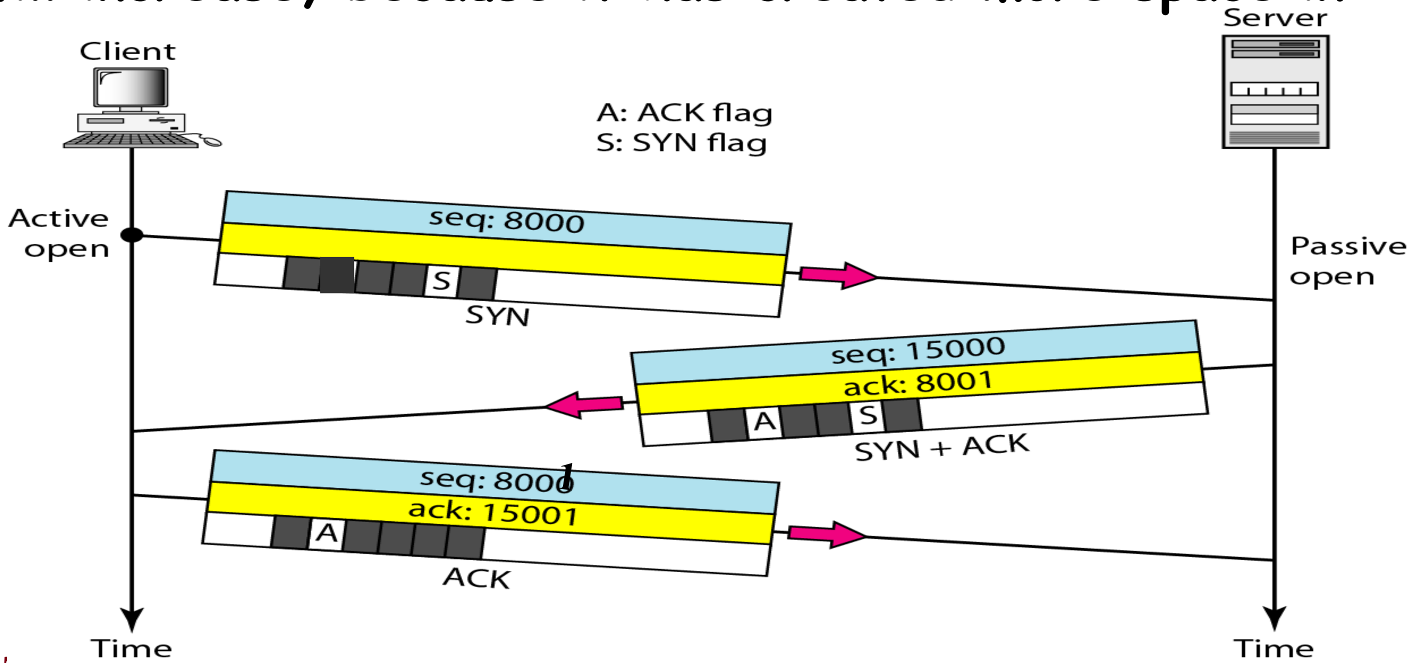
- Scaling factor needs to be agreed upon during the call setup and is going to stay the same during the connection

- Note: client is actively open, meaning you open it when you want to setup a connection. Server is passively open and listening

- Each site randomly chooses a starting point (i.e., a sequence number starting point) and the other site needs to know about it

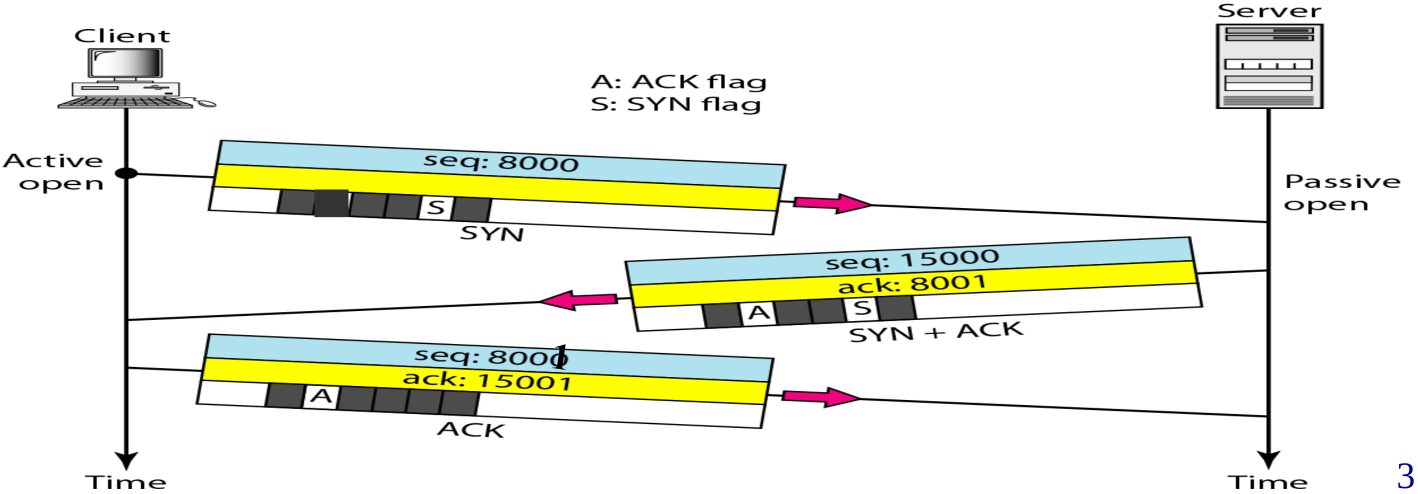
Window Size

- The receiver uses a procedure to prevent the sender from overwhelming it with bytes (more than it can handle.) this procedure is called **flow control** (not congestion control). If the receiver buffer is full, the receive window size will be zero. The sender has to stop
- Every time the receiver delivers the bytes (that it has received with no error and acknowledged) to the application layer, the window size will increase, because it has created more space in the buffer



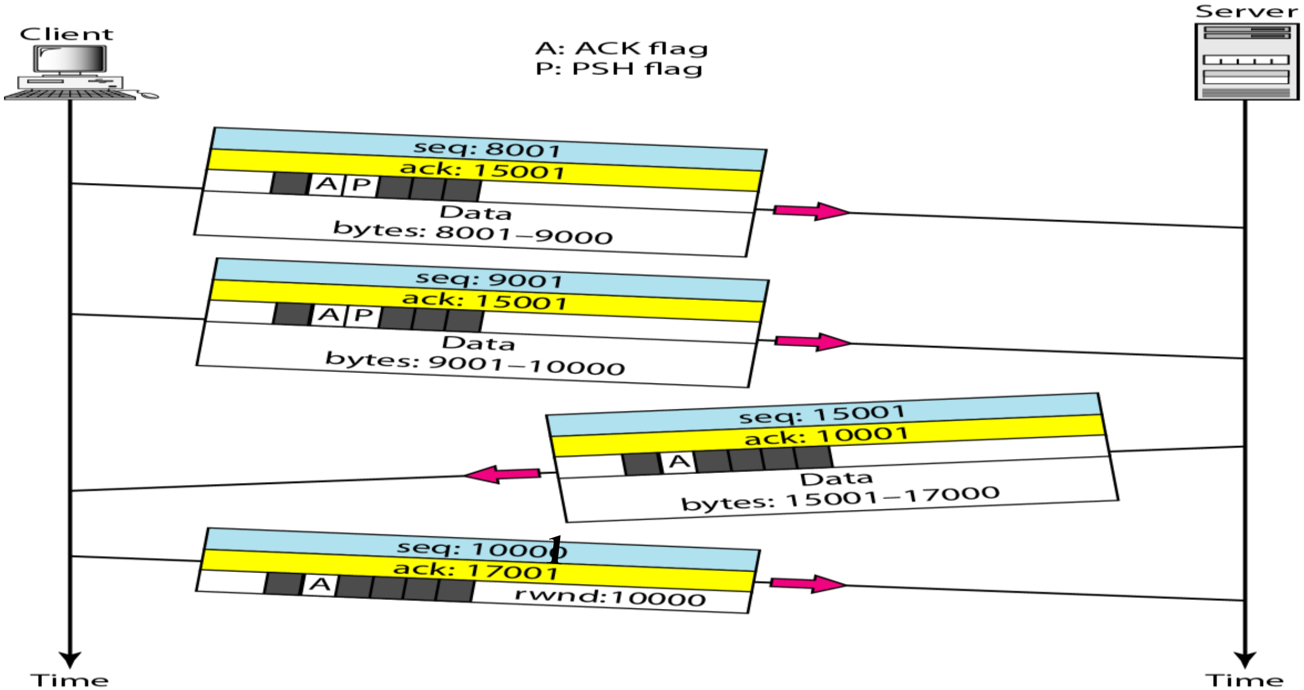
Connection Establishment Using Three-Way Handshaking

- Client sends a **SYN** segment. **S** flag is set to say it is a synch segment
- A **SYN + ACK** segment cannot carry data, but does consume one sequence number. 15000 is the **ISN** (**Initial Sequence Number**) of the server
- At this point in time, sequence number synchronization is accomplished, however server does not know about that, therefore client has to send an **ACK** segment. ACK can carry a few bytes of data



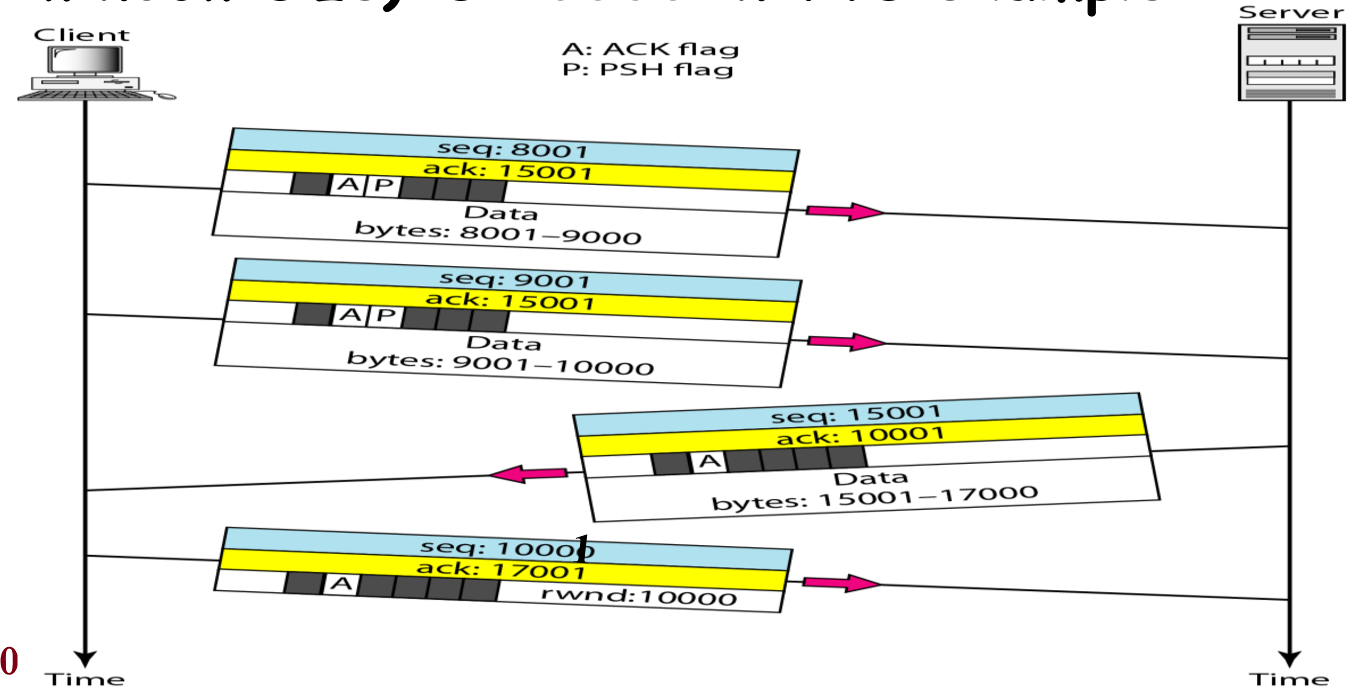
Data Transfer

- Here there are 1000 bytes of data with sequence number starting with 8001. ACK is still 15001
- Here server sends 2000 bytes of data coming from its application layer. ack is cumulative, therefore 10001 means all the bytes prior to 10001 are received with no error detected and the next one expected is 10001



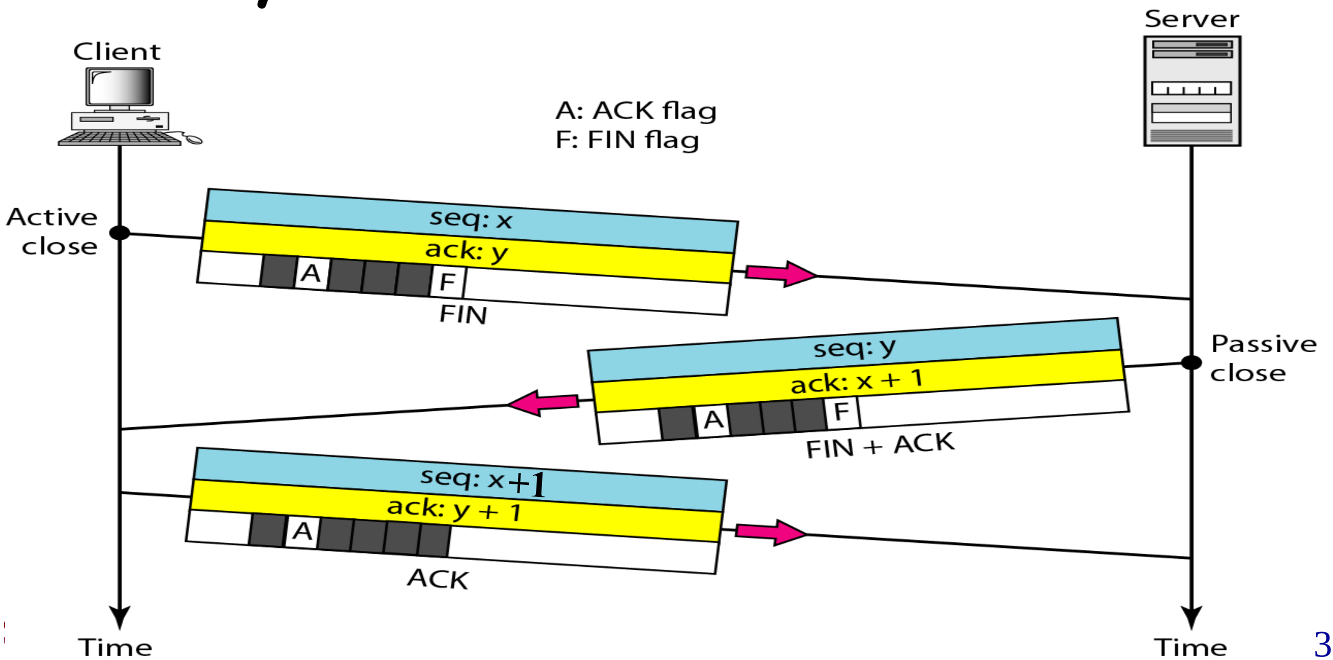
Data Transfer (Cont.)

- Note that the received (with no error) bytes may not be acknowledged individually, but they have to be acknowledged
- The last segment acknowledges the receipt of bytes prior to 17001 with no errors detected and next to expect is 17001
- **rwnd** (receive window size) is 10000 in this example



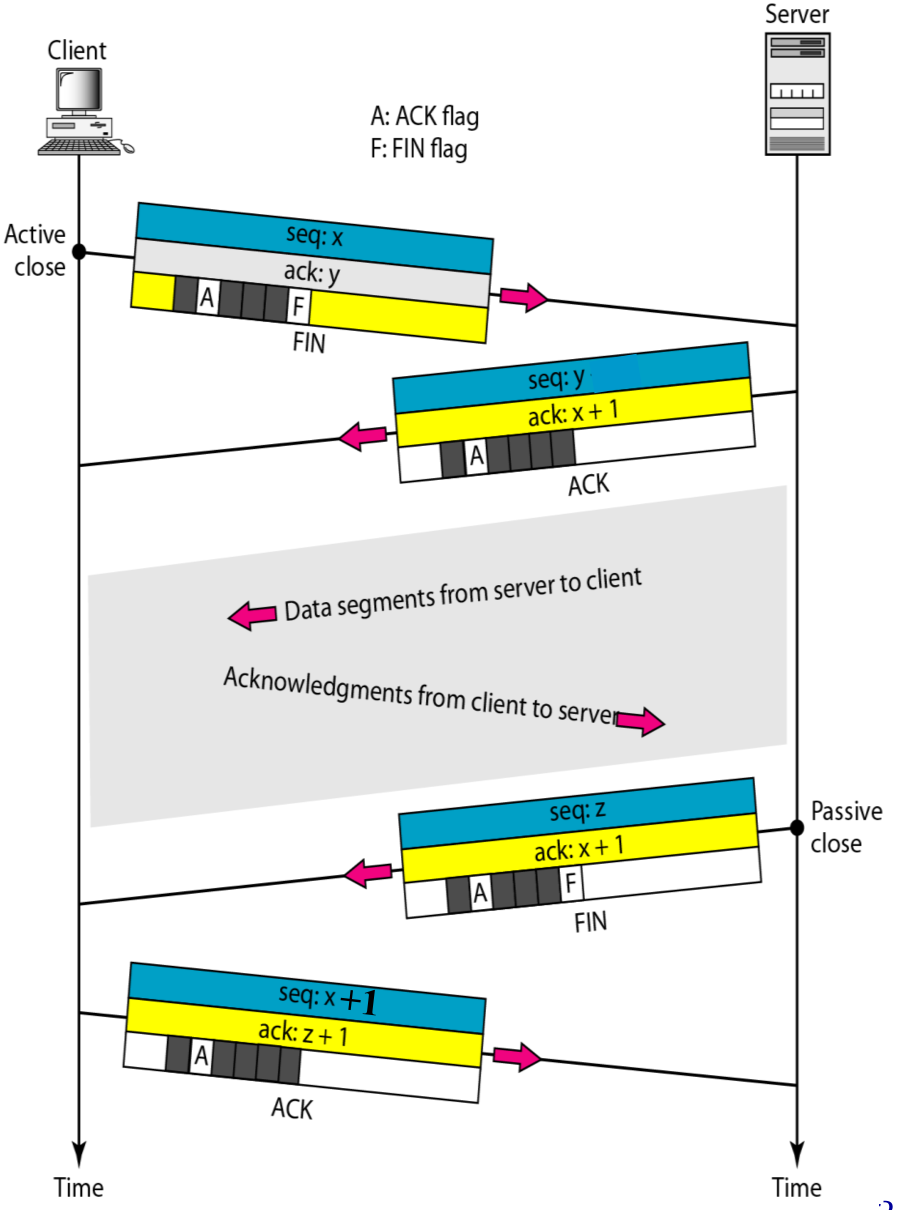
Data Transfer (Cont.)

- The **FIN** segment consumes one sequence number if it does not carry data
- By terminating the connection we mean terminating the handshaking process, i.e., the relationship between client and server
- The client socket and child socket will close, but parent socket will stay



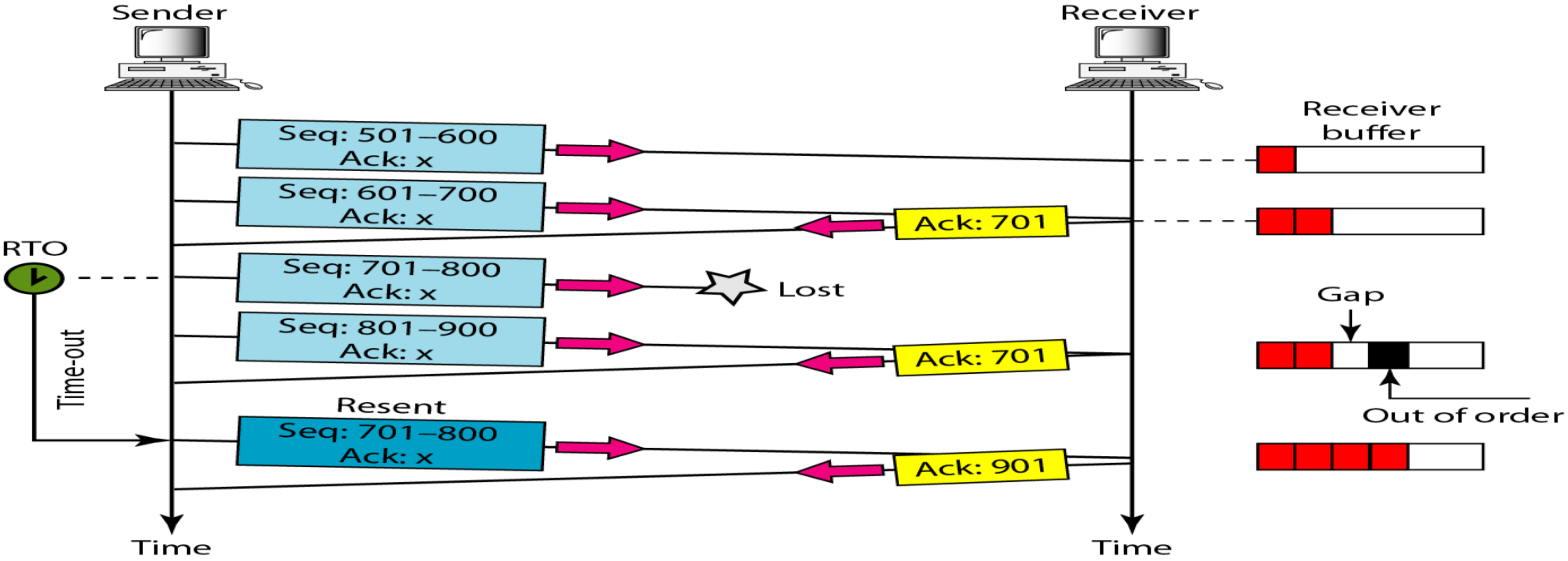
Four-Way Termination (Half Close)

- Just because client is terminating the connection, does not mean that the server does not have more bytes to send to the client. When client is issuing a termination, it means client does not have more bytes to send. But server may have more bytes to send. Then the client should ACK those ones, but not send more data bytes



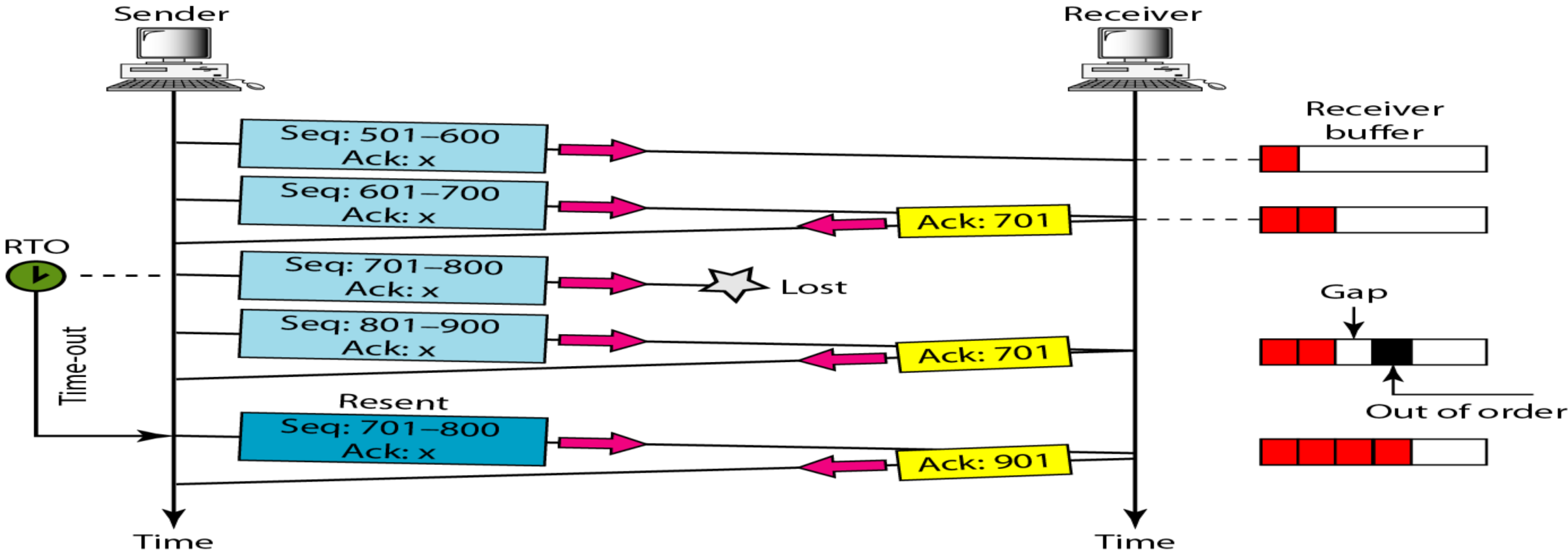
Lost Segment Scenario – Retransmission after Time Out (RTO)

- TCP retransmission after timeout (RTO) occurs when ACK does not arrives on time, or does not arrive at all
- RTO is often not a fixed value, but changes to improve the network. RTO value is updated based on the RTT (Round Trip Time) of segments. It is dynamically adjusted based on RTT; however the problem is that RTT is a random variable



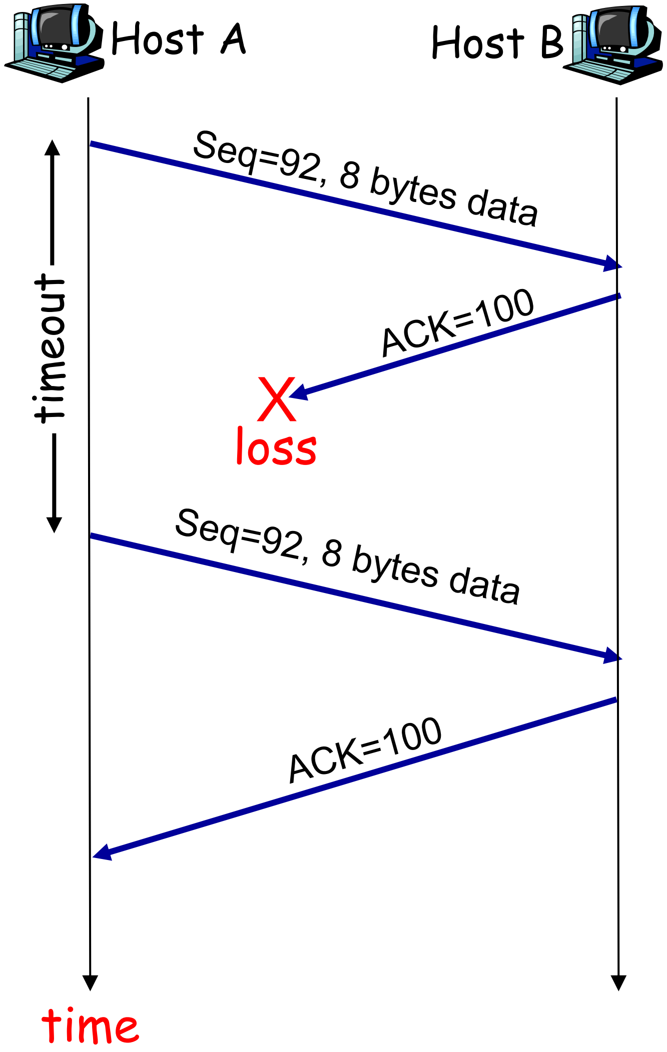
Lost Segment Scenario – RTO (Cont.)

- Complicated algorithms are typically required to calculate RTO based on RTT, e.g., moving average based algorithms, where a certain amount of data is averaged, then it is moved to include new samples of data
- This chart is based on Selective Repeat, because receiver would've dropped, out of order segments if it had used Go-Back-N



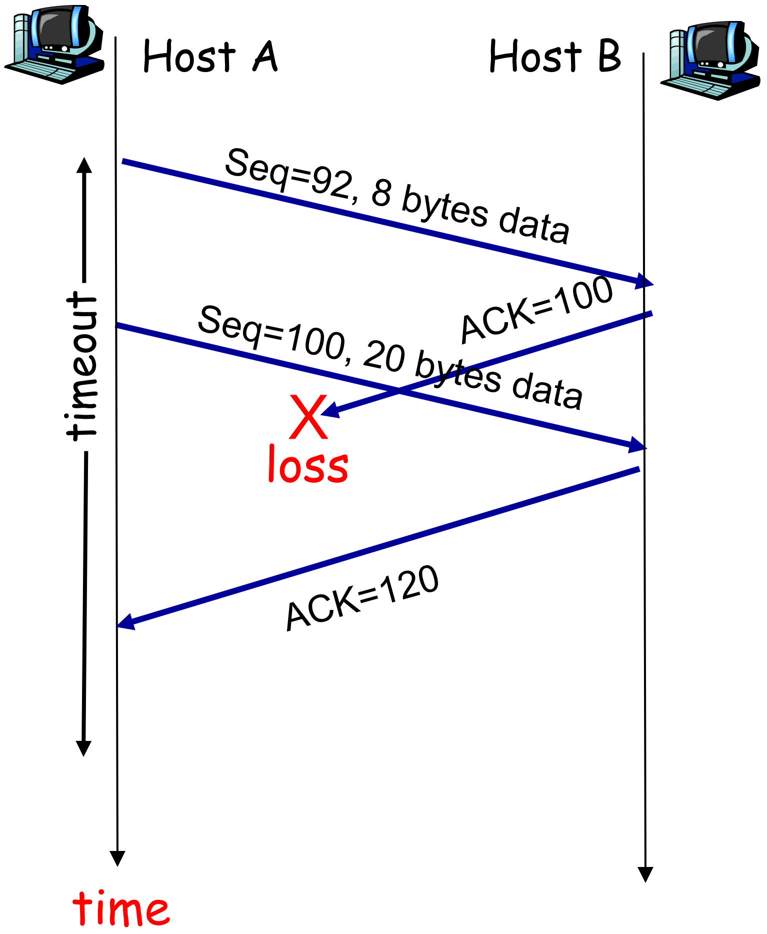
Example I - Lost ACK Scenario

- In this example RTT increases, because of the RTO



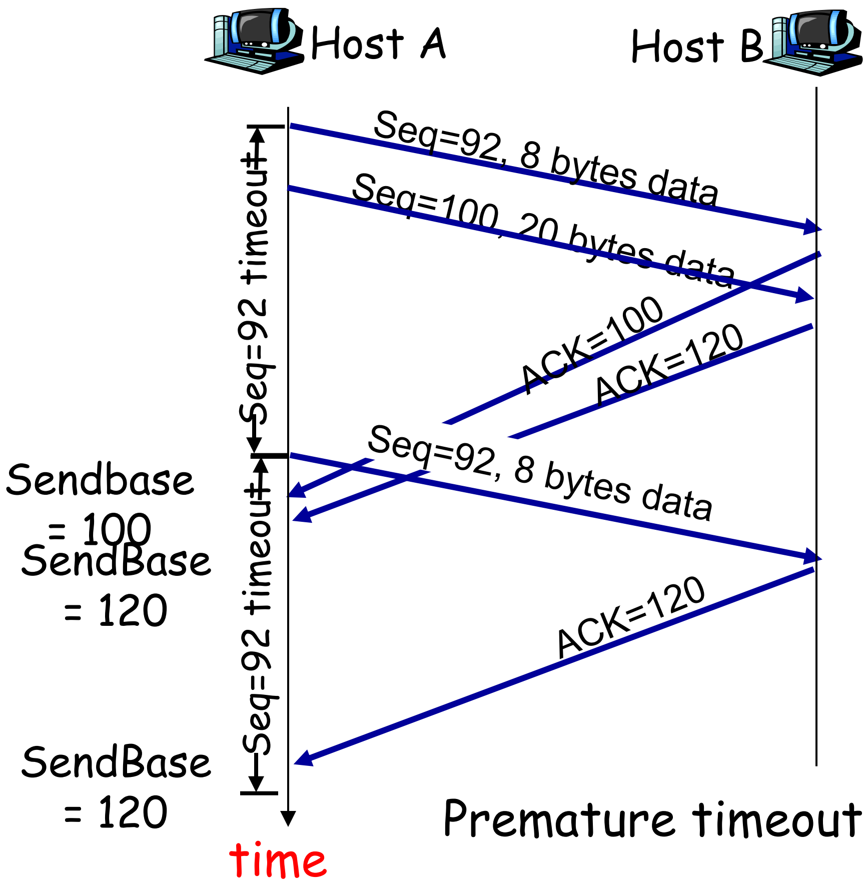
Example II – Cumulative ACK Scenario

- Seq=92 and 8 bytes means the first one is 92 and the last one is 99
- ACK100 got lost, but sender does not know, so it sends seq=100 and 20 bytes
- The sender sends ACK120; therefore lost ACK did not result in any harm in this example
- Here, neither the sender, nor the receiver knows that ACK was lost



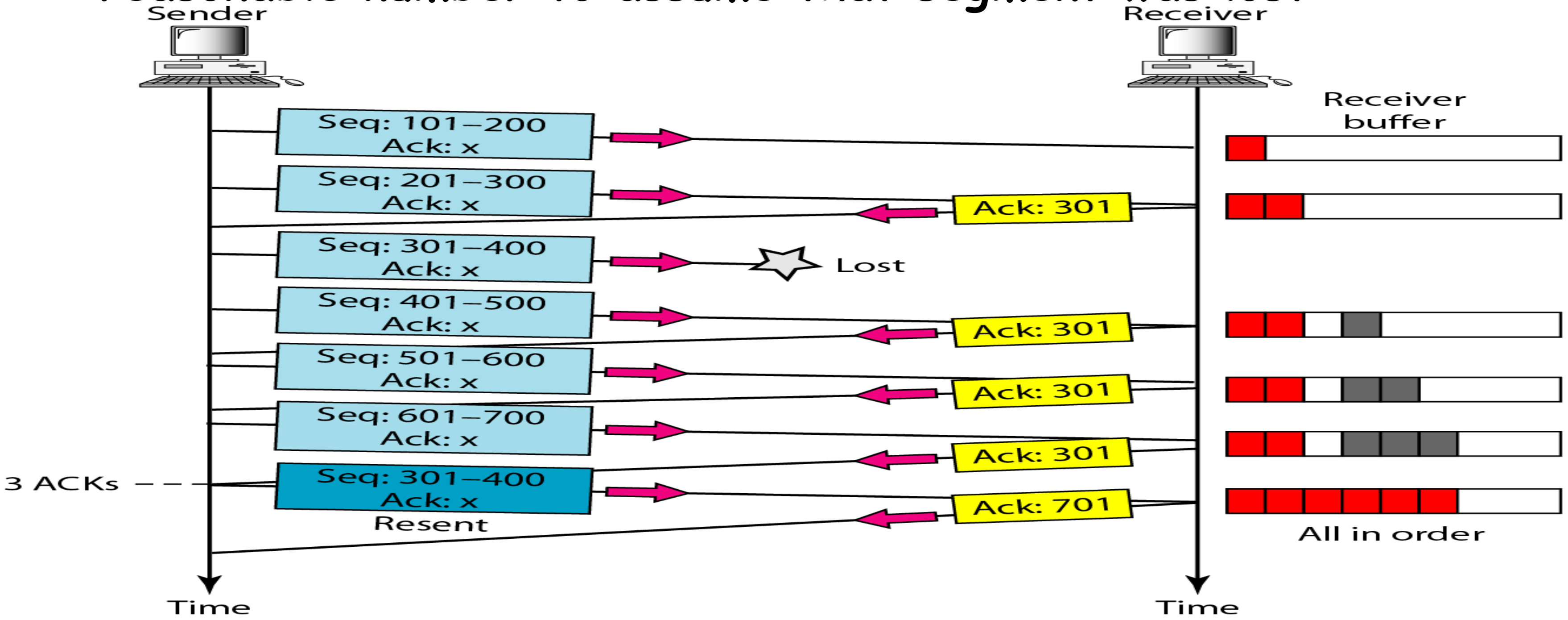
Example III – Premature Timeout

- ACK100 was delayed due to congested network so time out expired and sender resent 92, 8bytes, and receiver sent ACK120
- Now the RTT can be considered from resent of seq92 to ACK100, which is very small, but in reality RTT was from first sent seq92 to ACK100, so TCP recommends if you need to transmit a segment due to timeout expiration, do not consider that for RTT measurements in moving average algorithms because it would be misleading



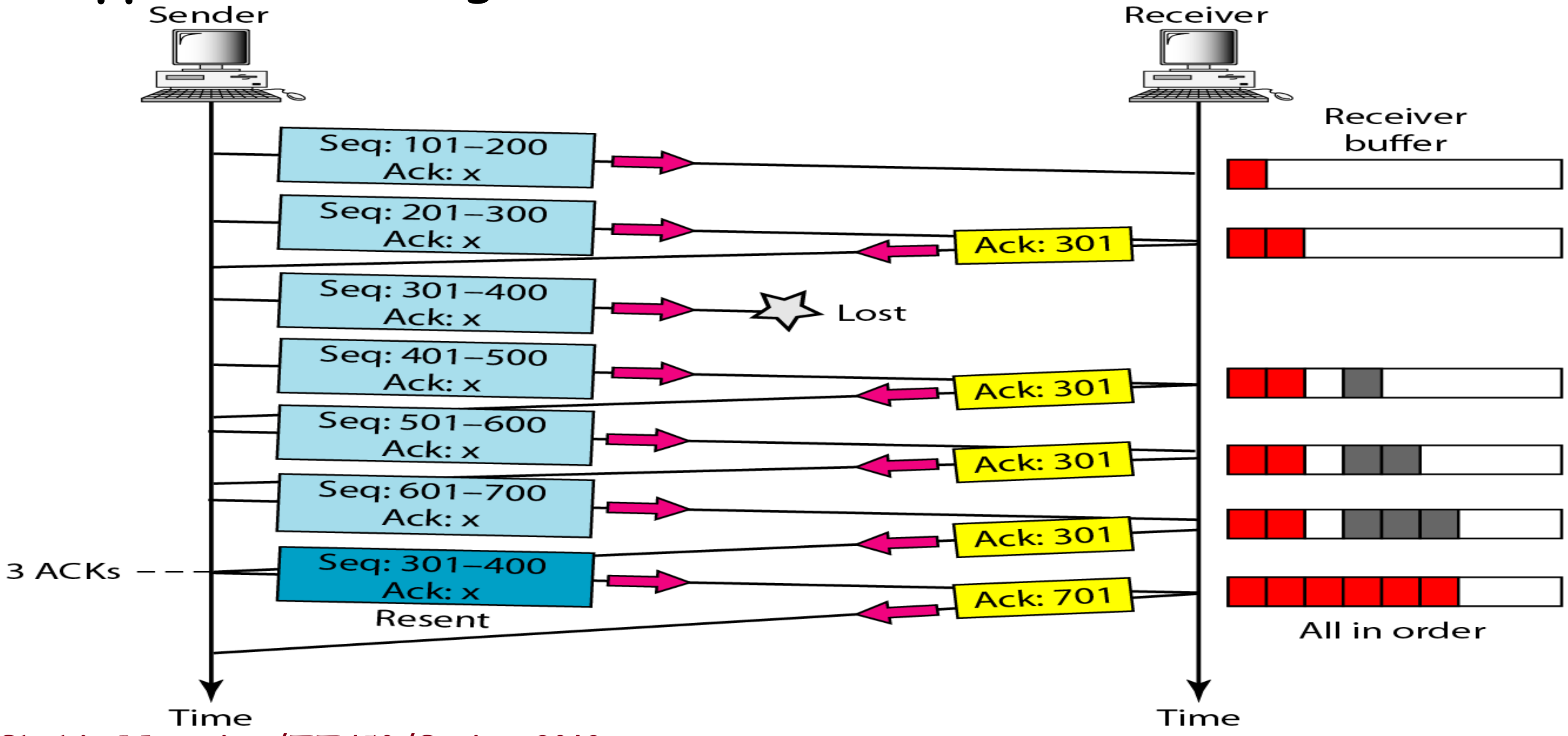
Retransmission after 3 ACKs (Fast Retransmission)

- Time-out period often relatively long:
  - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs
- Based on the experiments, it was concluded that 3 ACKS is a reasonable number to assume that segment was lost



Fast Retransmission (Cont.)

- Note: When sender receives ACK301 the 2nd time, still it does not resend 301-400, because the timeout has not expired yet
- However, if sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost



TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- Initialize TCP variables:
  - Seq. #s
  - Buffers, flow control info (e.g. `RcvWindow`)
- *Client:* connection initiator

```
Socket clientSocket = new
  Socket("hostname","port

  number");
```

- *Server:* contacted by client

```
Socket connectionSocket =
  welcomeSocket.accept();
```

Three way handshake:

Step 1: Client host sends TCP SYN segment to server

- Specifies initial seq #
- No data

Step 2: Server host receives SYN, replies with SYNACK segment

- Server allocates buffers
- Specifies server initial seq. #

Step 3: Client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management (Cont.)

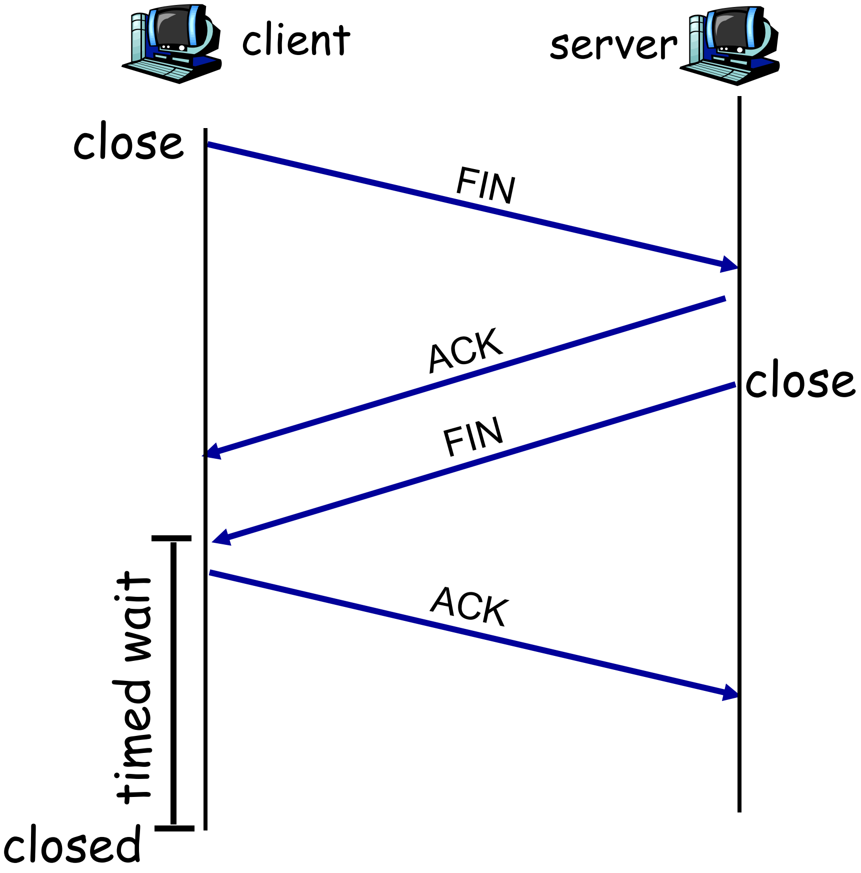
Closing a connection:

Client closes socket:

```
    clientSocket.close();
```

Step 1: Client end system sends TCP FIN control segment to server

Step 2: Server receives FIN, replies with ACK. Closes connection, sends FIN



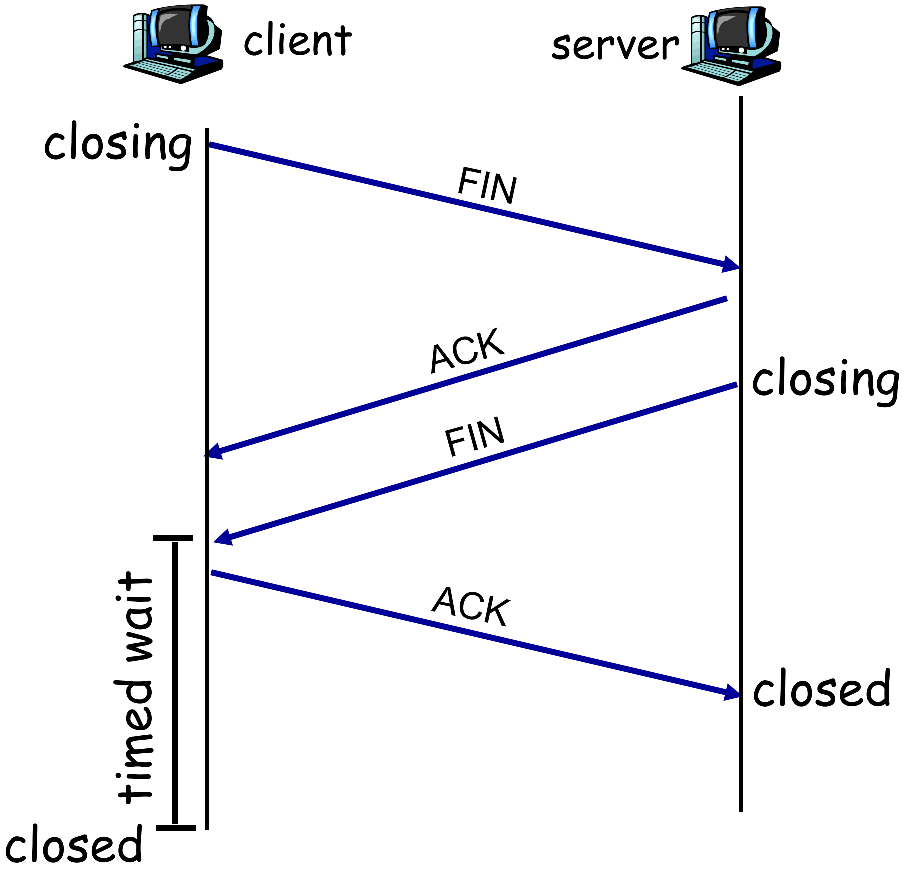
TCP Connection Management (Cont.)

Step 3: Client receives FIN, replies with ACK

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: Server, receives ACK. Connection closed

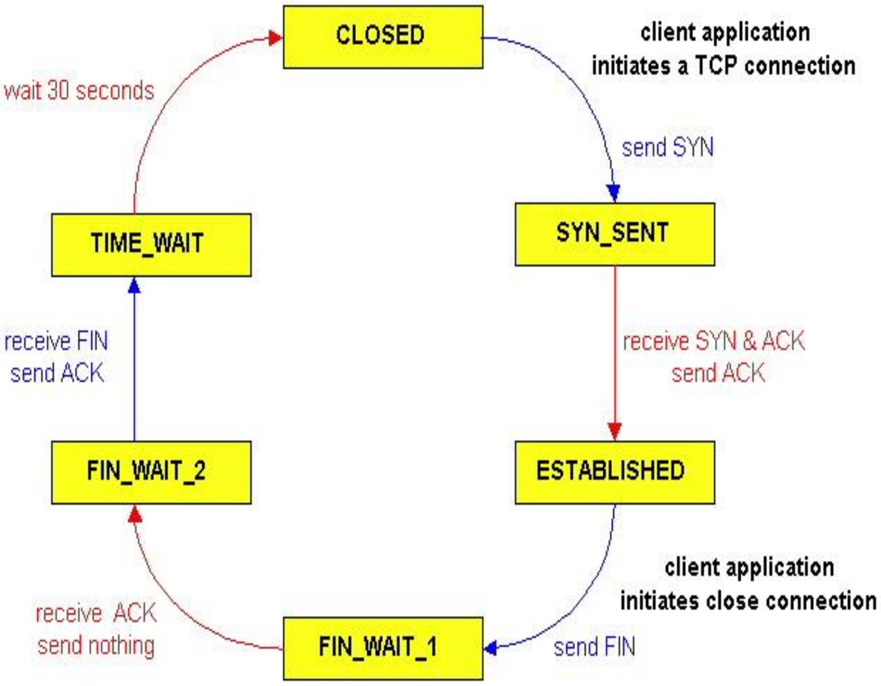
Note: With small modification, can handle simultaneous FINs



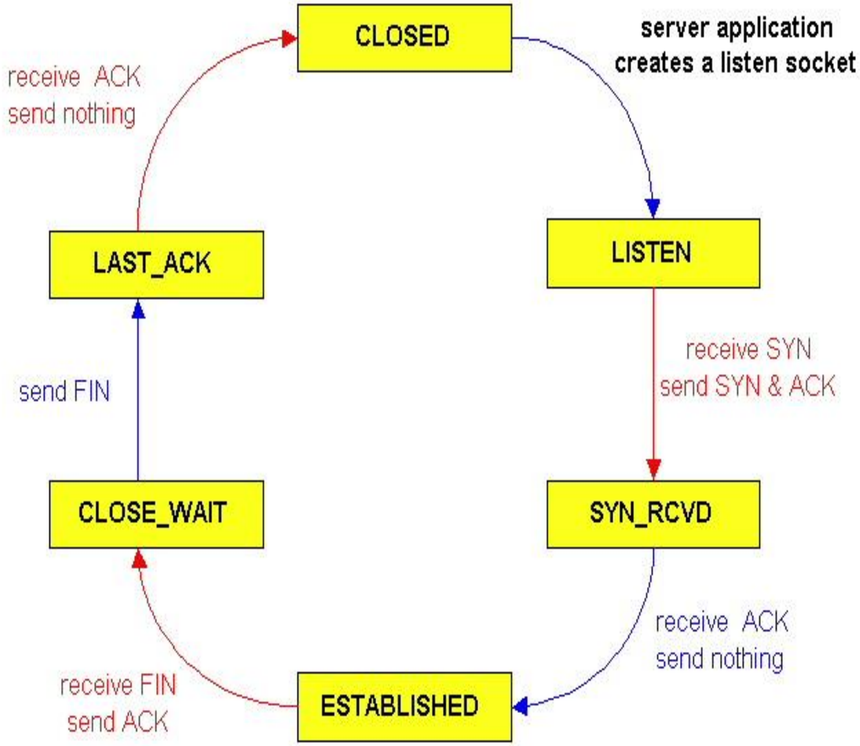
(Optional to Study) TCP Connection Management (Cont.)

- Segments may be received out of order, because they are inside the packets and packets may be received out of order
- Therefore it make sense to wait, giving the chance that there are still some segments in the network that have not yet arrived. Therefore when FIN is received still client waits a couple of seconds and then it closes

TCP client lifecycle

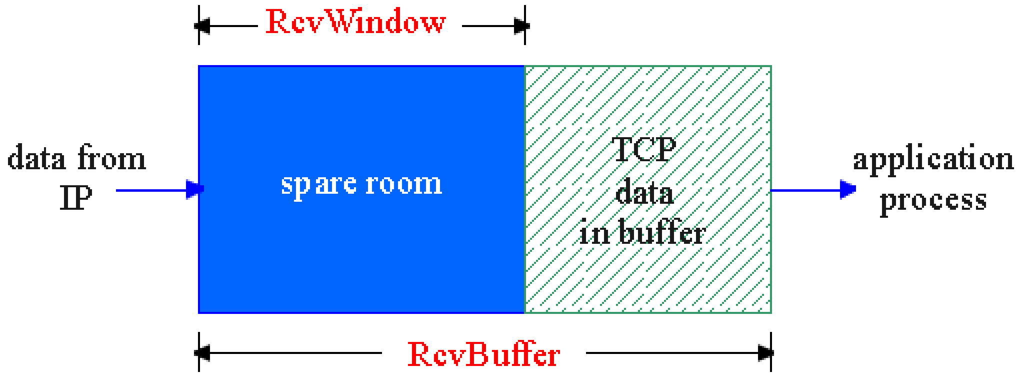


TCP server lifecycle



TCP Flow Control

- One mechanism is ARQ (i.e., based on positive ACK and timeout)
- Another mechanism is referred to as the 3-duplicate ACKs
- If receiver receives too many bytes, it means the sender is getting greedy and eventually the blue will be gone and buffer will fill up completely
- When bytes are delivered to application, then blue moves to the right
- Even if the bytes are received in buffer, checked for error and no error was found, still does not necessarily mean that the bytes will be delivered to application right away, because application may be slow at reading from buffer

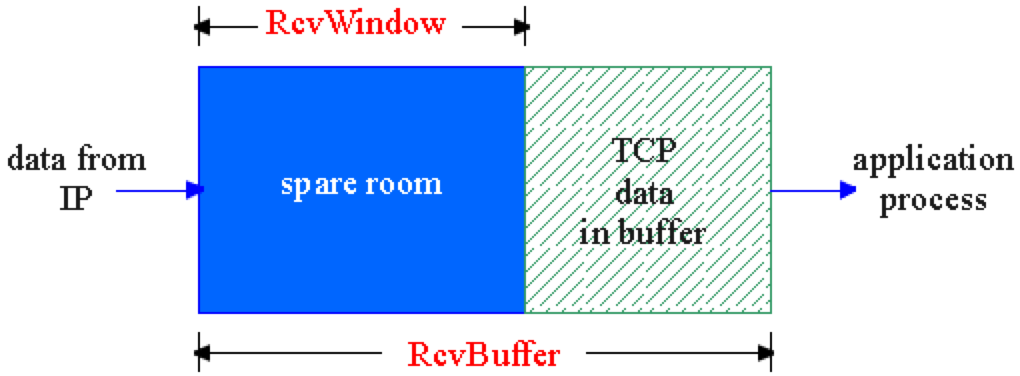


TCP Flow Control (Cont.)

- Receive side of TCP connection has a receive buffer:

flow control

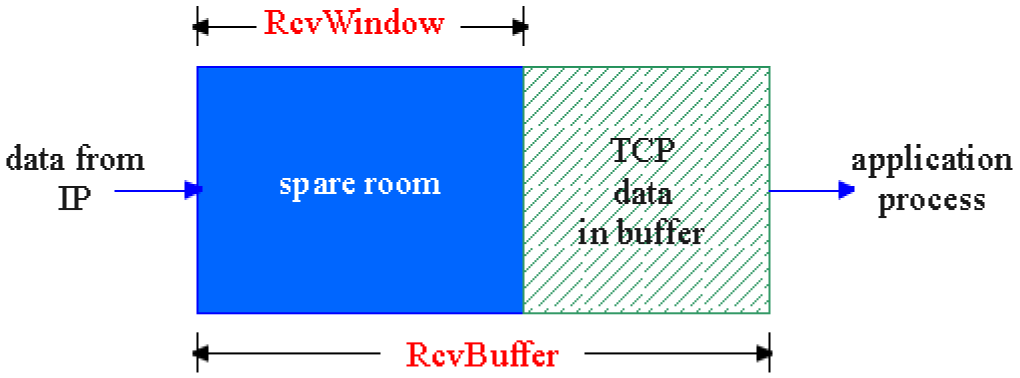
Sender won't overflow receiver's buffer by transmitting too much, too fast



- Speed-matching service: matching the send rate to the receiving app's drain rate

- App process may be slow at reading from buffer

TCP Flow Control: How it Works



- Rcvr advertises spare room by including value of `RcvWindow` in segments
- Sender limits unACKed data to `RcvWindow`
  - Guarantees receive buffer doesn't overflow

(Suppose TCP receiver discards out-of-order segments
Spare room in buffer

= **RcvWindow**

= `RcvBuffer-[LastByteRcvd - LastByteRead]`

Here the assumption is that Go-Back-N ARQ is used. Note that Selective Repeat ARQ is another option

Principles of Congestion Control

- Congestion informally means “too many sources sending too much data too fast for network to handle”
- Congestion in a network may occur if the load on the network (the number of packets sent to the network) is greater than the capacity of the network (the number of packets a network can handle)
- Congestion control refers to the mechanisms and techniques to control the congestion and keep the load below the capacity
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queuing in router buffers)
- A top-10 research problem in computer networks!

Congestion vs Flow Control

- Congestion Control is different from flow control! The flow is when receiver gets filled up. Congestion is when network gets filled up. The receiver tells the sender how much space it has left with. However if network is congested, network does not tell the hosts that it is congested (not feasible, due to high number of hosts)
- The sender needs to determine whether the network is congested or not. Flow control (explicit) is easier that congestion control (implicit)
- Question: How does the sender know the network is congested? One indication is when the timeout expires, however timeout expiration does not mean that the network is necessarily congested. What if the receiver received the packet in error and dropped them and did nothing (no NAK was considered in TCP)

Congestion Control (Cont.)

- The sender does not have a way of knowing the actual reason behind timeout, therefore sender always assumes that timeout expired because of congestion not because of the receiver
- If sender receives 3 ACKs for a segment that has already been acknowledged, the sender concludes that the segment was lost, because the network was congested or the packet was lost, however it knows that the network was not badly congested
- Note that flow control helps easing the task of the congestion control, because when the receiver informs the sender that it does not have space in its buffer, the sender needs stop or slow down, and hence the network will in turn be relieved of its congestion a little bit

Congestion Control (Cont.)

- **Advertised window** (**awnd** or **W_a**) The receiver tells the sender, how much space it has on its buffer. awnd is a dynamic parameter (is not constant)
- **Congestion window** (**cwnd**, or **W_c**) The network does not tell the sender how much space it has, so the sender needs to guess; this guess may be denoted by cwnd or W_c
- Note: both awnd and cwnd are measured in bytes. Both are dynamic parameters

Congestion Control (Cont.)

- Example I: awnd=50 and the sender's guess (cwnd) is 60

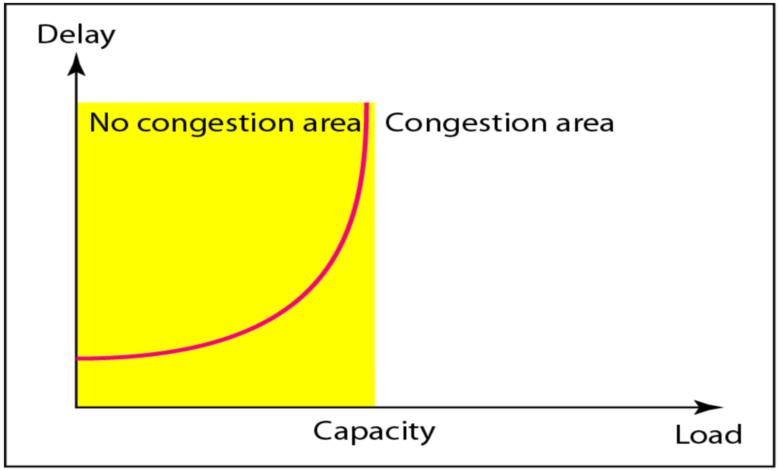
- Example II: awnd=60 and the sender's guess is 50

- Sender assumes the maximum number of bytes it can send is Min (awnd, cwnd)

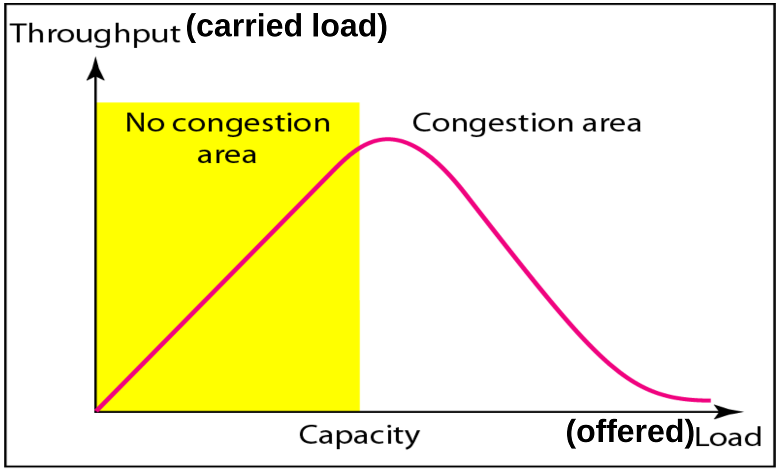
- In the following slides we will assume that awnd = ∞ to be able to focus on congestion control problem rather than combined flow and congestion control problems

Principles of Congestion Control

- We know the RTO depends on RTT, however RTT varies (a random variable.) TCP needs an algorithm to estimate RTO
- A reminder: **Offered load** is what is dumped into the network (considering all the users, not just one user) and **throughput** (**carried load**) is what is delivered both are in **bits per second** or **packets per second**



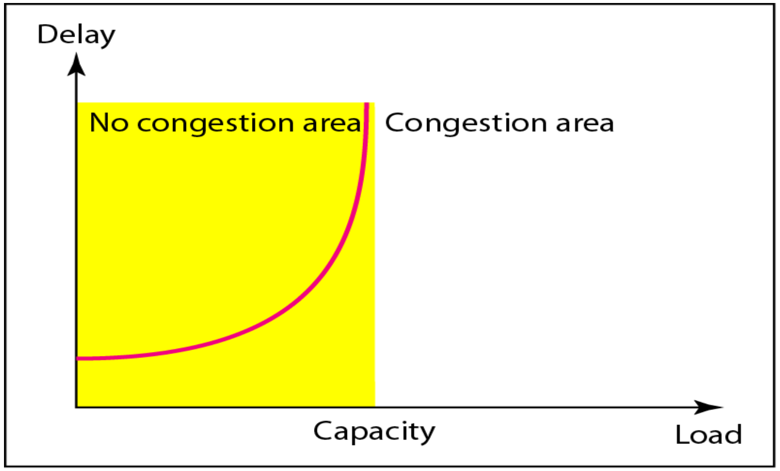
a. Delay as a function of load



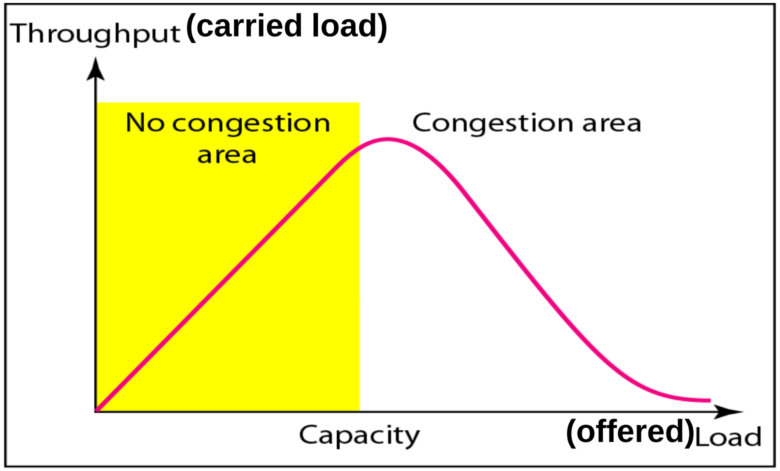
b. Throughput as a function of load

Principles of Congestion Control (Cont.)

- At the beginning the resources are not utilized, so whatever the sender dumps to the network, is going to be delivered, then it reaches the capacity, i.e., the maximum throughput
- Then the network gets congested and starts dropping the packets. However the users keep retransmitting; therefore on the right side of the curve overload is not only the new packets, but also the retransmitted packets. Therefore we can say, the retransmission made congestion and throughput even worse



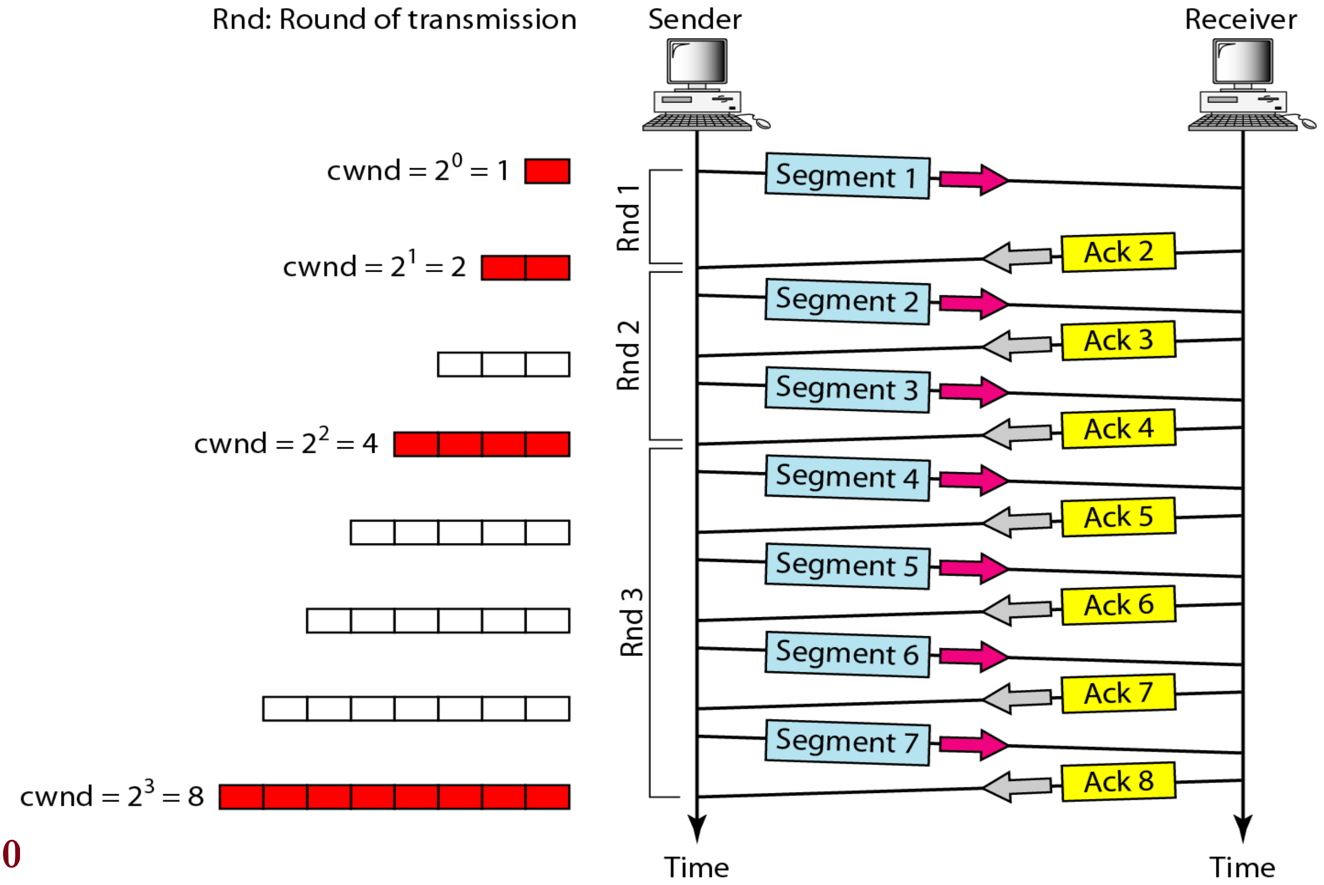
a. Delay as a function of load



b. Throughput as a function of load

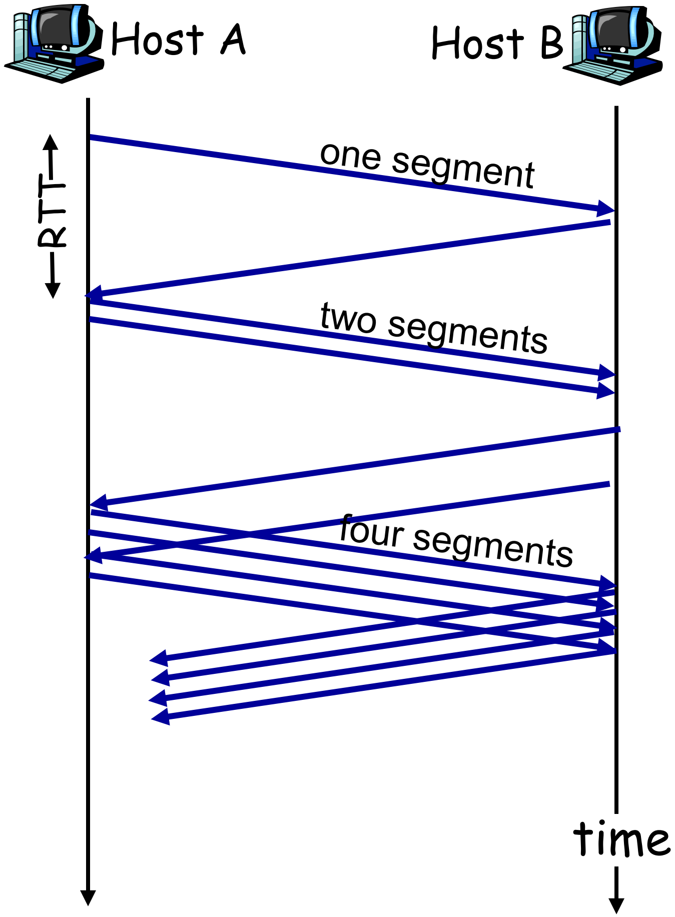
TCP Slow Start Phase

- In the Slow Start (SS) algorithm, cwnd is initially set to 1 and then increased by 1, i.e., MSS or Max Segment Size for every ACK received
- cwnd will eventually reach a threshold, this will result the process to go to another phase referred to as congestion avoidance



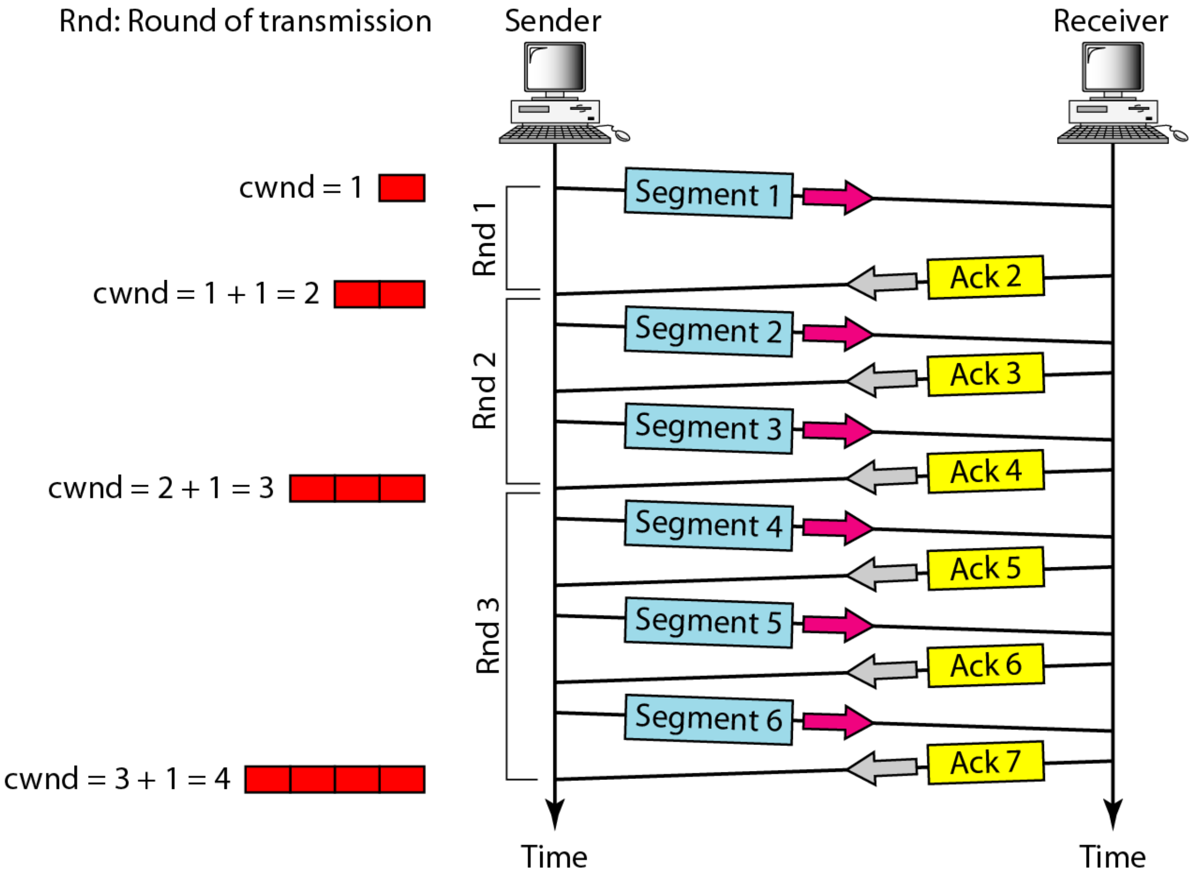
Slow Start (Cont.)

- When connection begins, increase rate exponentially until first loss event:
  - Double `cwnd` every RTT
  - Done by incrementing `cwnd` for every ACK received
- Summary: Initial rate is slow but ramps up exponentially fast



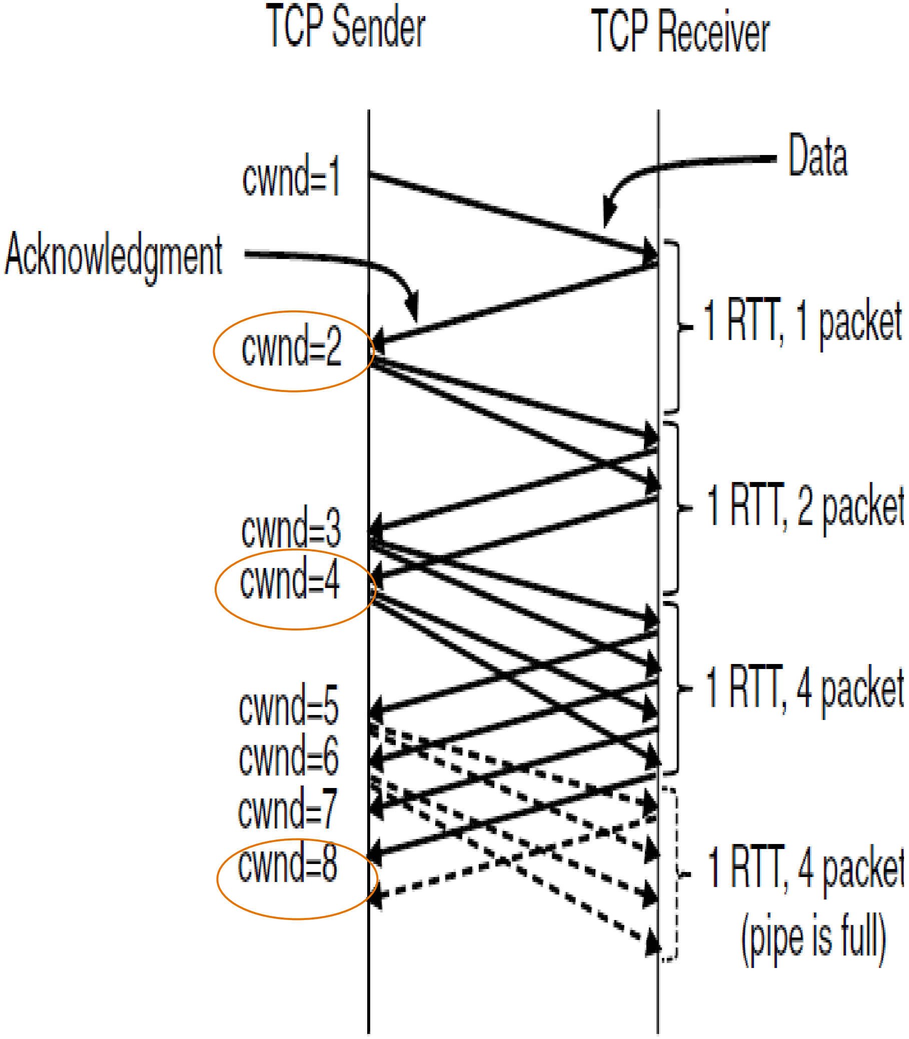
Congestion Avoidance, Additive Increase

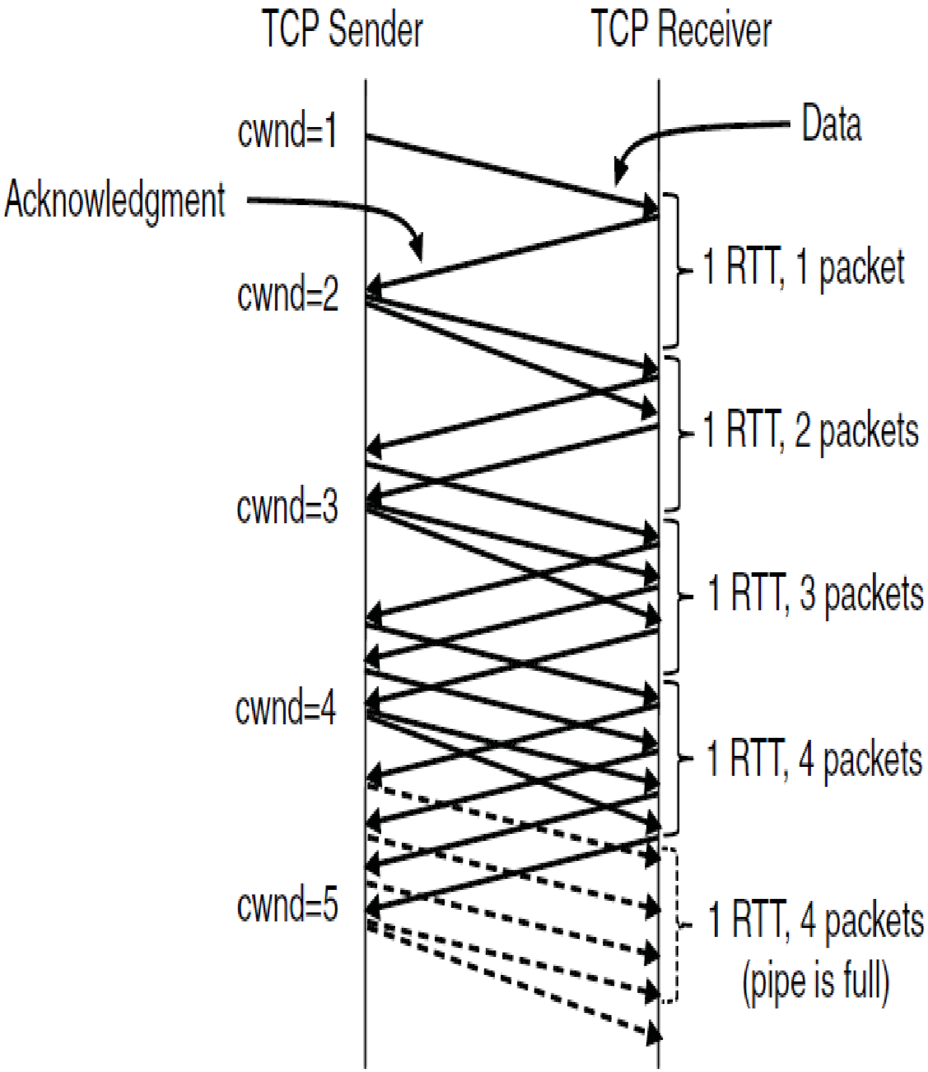
- In the congestion avoidance algorithm, the size of the congestion window increases additively (therefore, this phase is denoted by **AI** or **Additive Increase**) until **congestion is detected**



SS vs AI

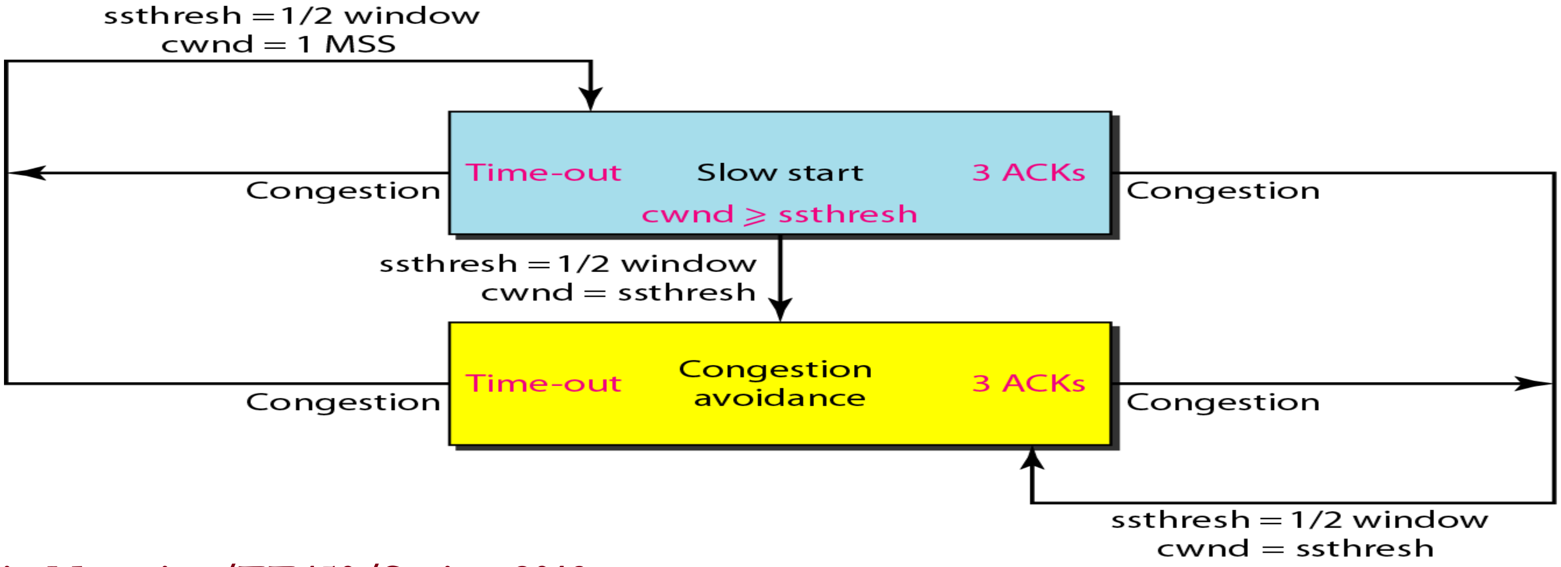
- Here both start with cwnd of 1 segment





TCP Congestion Policy Summary

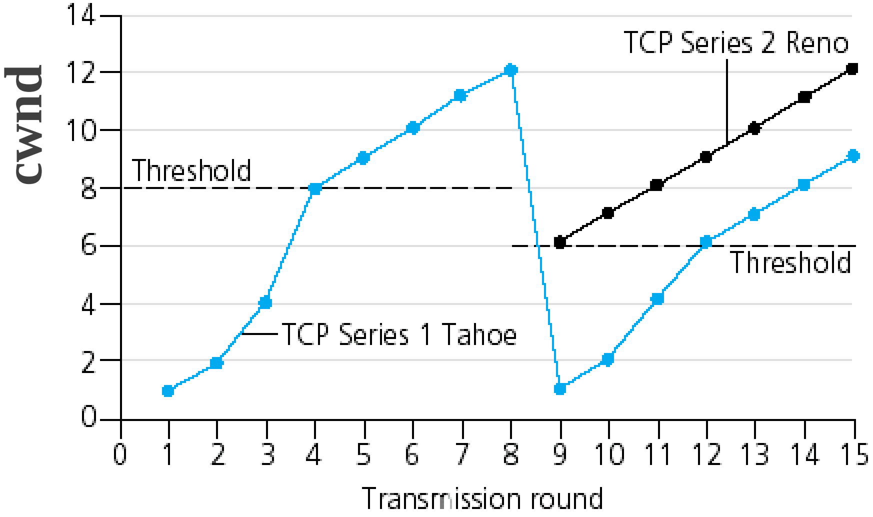
- An implementation reacts to congestion detection in one of the following ways:
  - Tahoe Implementation: If detection is by timeout, a new slow start (SS) phase starts
  - Reno Implementation or Fast Recovery: If detection is by 3 ACKs, a new congestion avoidance phase starts



Fast Recovery (Reno Implementation)

Q: When should the exponential increase switch t linear?

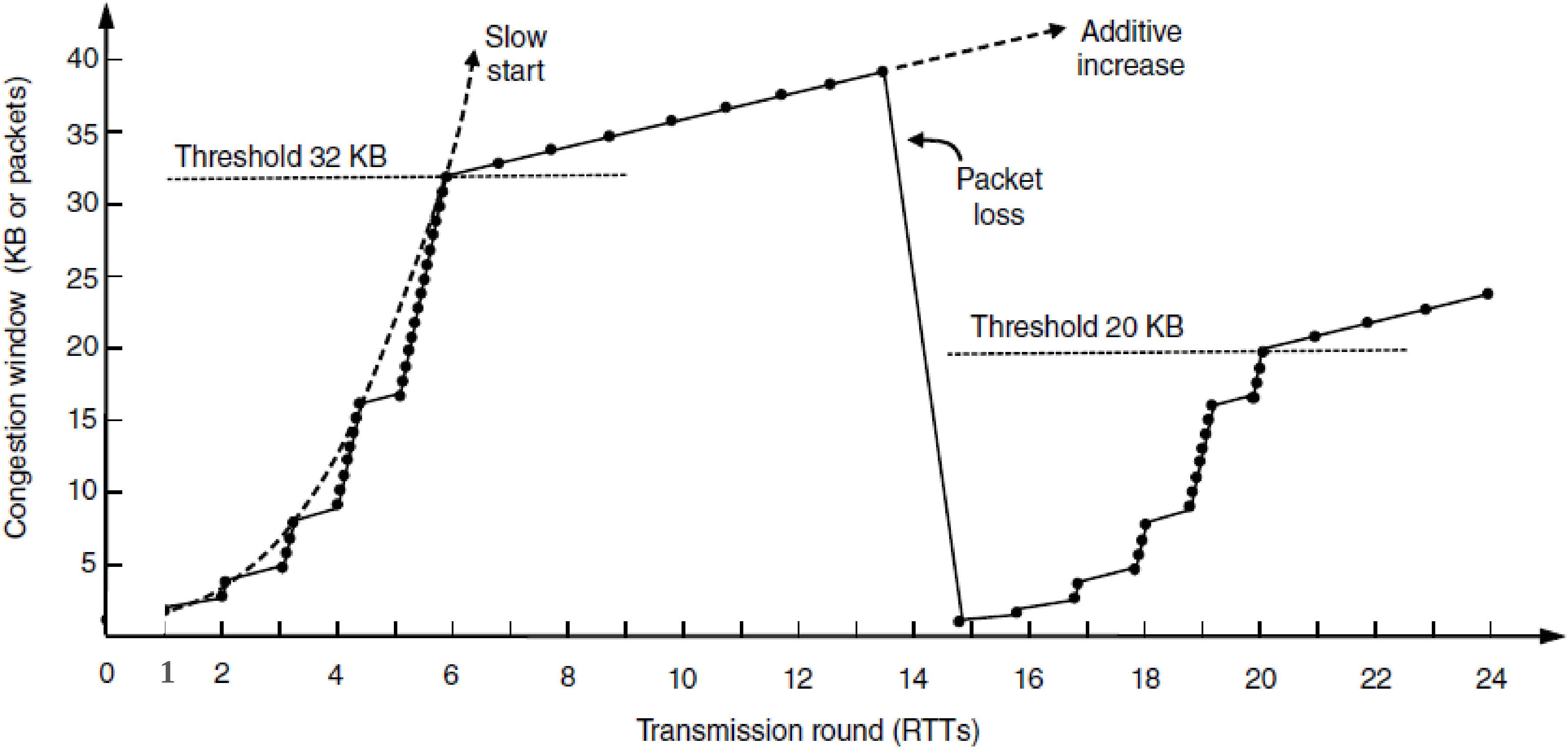
A: When *cwnd* gets to 1/2 of its value before timeout



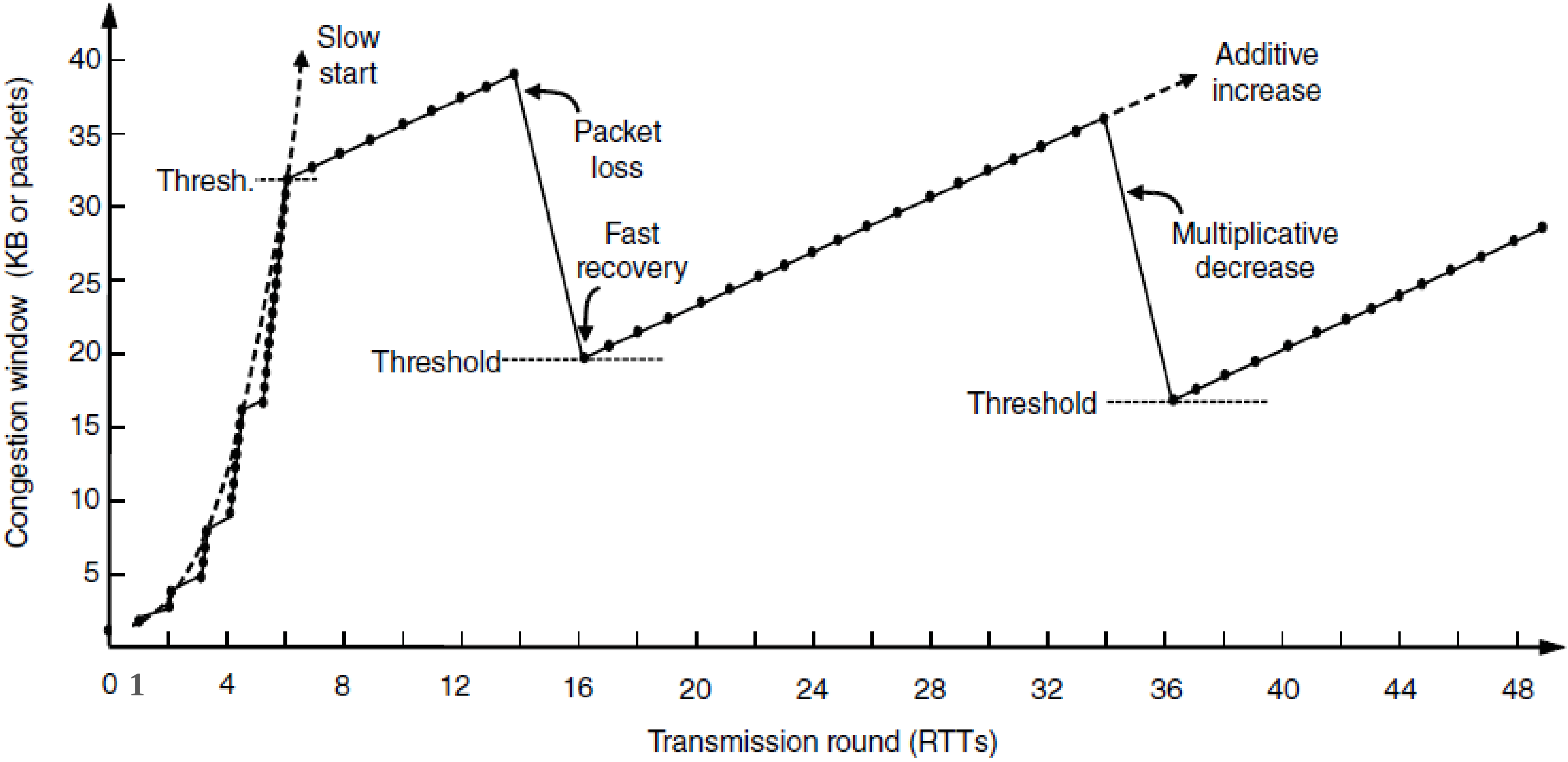
Implementation:

- Variable Threshold. At loss event, Threshold is set to 1/2 of *cwnd* just before loss event

Tahoe Implementation Example



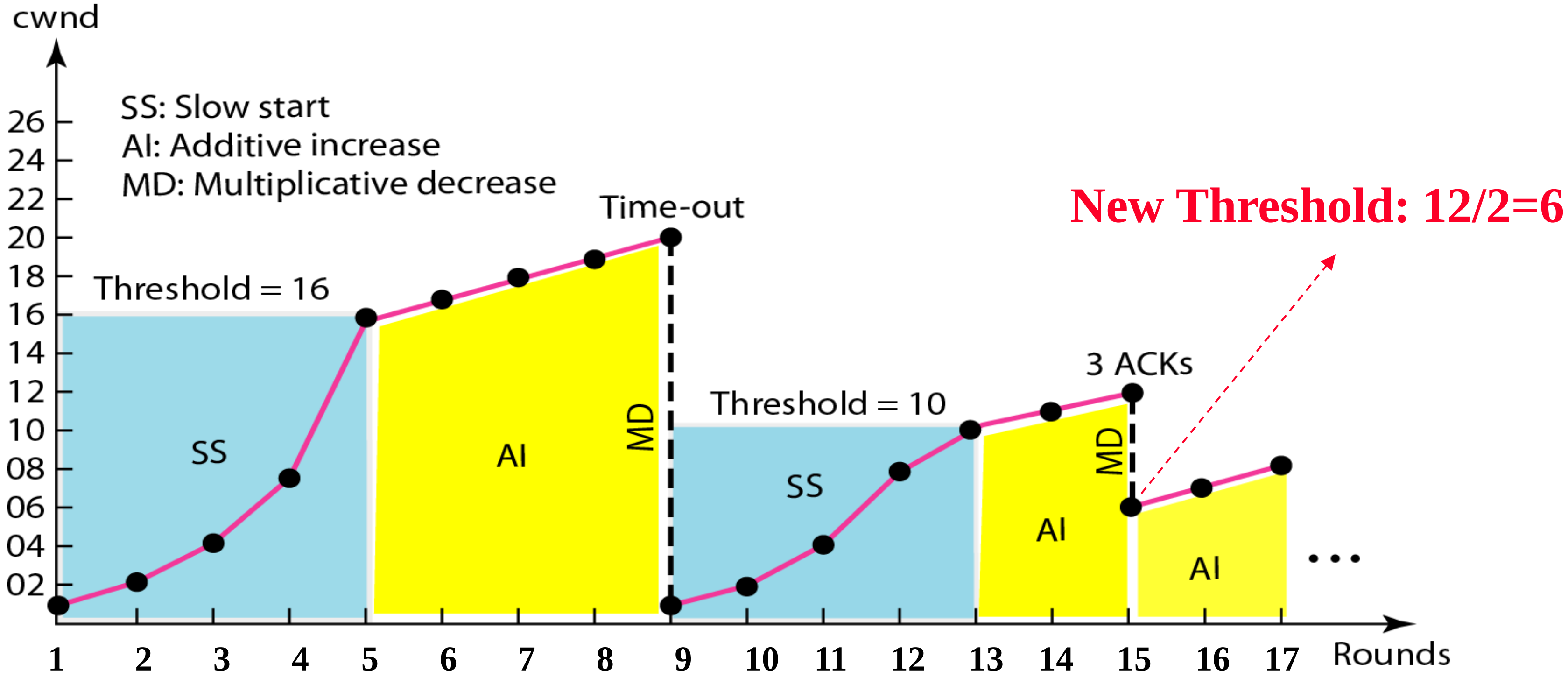
Reno Implementation Example



Summary: TCP Congestion Control

- When cwnd is < Threshold, sender is in **slow start** phase, window grows exponentially
- When cwnd is > Threshold, sender is in **congestion avoidance** phase, window grows linearly
- When a **triple duplicate ACK** occurs, Threshold is set to cwnd/2 and cwnd is set to threshold
- When **timeout** occurs, Threshold is set to cwnd/2 and cwnd is set to 1 MSS

TCP Congestion Policy Summary



New Threshold: 12/2=6