

Transport Layer Protocols:

TCP (Transmission Control Protocol)

Transport Layer Functions

- Multiplex/Demultiplex traffic for different applications
- Other possible function a protocol may or may not implement
 - Establish and maintain end-to-end connections
 - Guarantee reliable, inorder, end-to-end transfer
 - Provide end-to-end flow control
 - Congestion control
 - TCP implements all of the above, UDP implements none
- Lowest level end-to-end protocol
 - Header generated by sender is interpreted only by the destination, not by routers
 - Routers view transport header as part of the payload/data

Transmission Control Protocol (TCP)

- End-to-end, reliable, in-order transfer
- Connection-oriented protocol
 - Explicit connection establishment and termination
- Stream oriented
 - Data to be sent is interpreted as a stream of bytes, with no boundaries
 - Actual transfer breaks this stream into packets of arbitrary size
- Full duplex
- Flow control, Error detection and control
- Congestion Control
- Specified originally in RFC 793, many other related RFCs are there

Port, Endpoint, and Connection

- Port
 - A 16 bit integer used to identify an application using
- Endpoint
 - A 2-tuple <host, port>
 - **host** is an IP address
 - Commonly called a *socket*
- Connection
 - Defined by two endpoints
 - Two connections will have at least one endpoint different (but can have one endpoint same)
 - **Messages are demultiplexed based on connections, not ports**

Classification of Ports

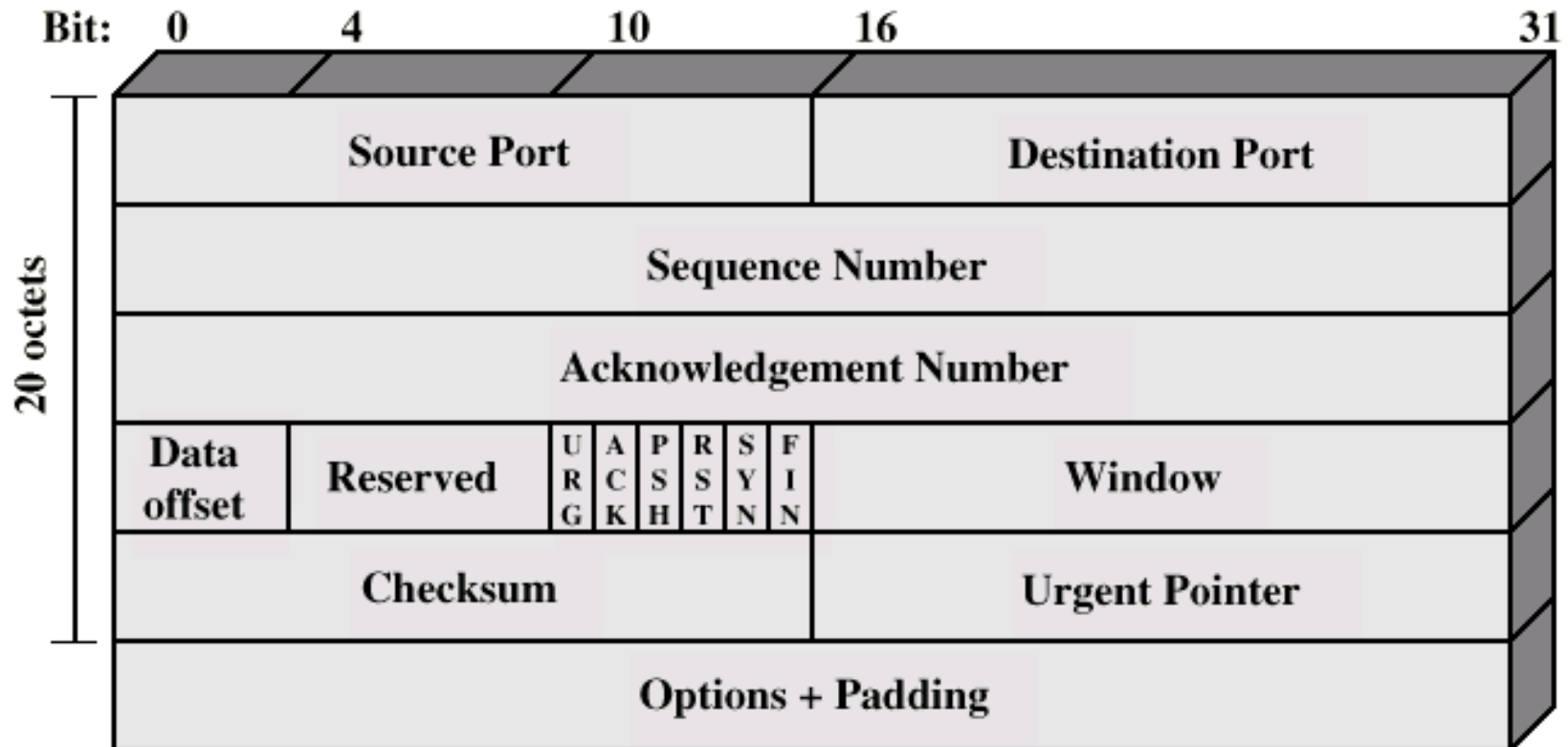
- Well-known/reserved ports
 - Ports upto 1023
 - Normally used for protocols with wide applicability
 - Assigned by IANA
 - Examples
 - ftp – 21,20, telnet – 23 etc.
- Registered ports
 - Ports 1024 – 49151
 - Used to avoid port collisions between user-level applications developed independently when installed on the same machine
 - Registration in ICANN registries is voluntary, though recommended
- Dynamic or private ports (also called ephemeral ports)
 - Ports 49152 – 65535
 - Can be used by anyone for anything

Stream and Segment

- Data viewed as a **byte stream**, i.e., a sequence of bytes
- **Segment** – the unit of transfer between TCP s/w on two machines
 - The stream of bytes is divided into segments, each segment is given a TCP header, and transmitted
 - Usually 1 segment is encapsulated in 1 IP datagram
 - Segments may not contain any data
 - Ex – segments used to establish/terminate connections, send acks etc.
- **Sequence Number** – used to specify position within the stream
 - Each TCP segment will contain a 32-bit sequence no. to identify its position in the stream

- Maximum Segment Size (MSS)
 - Maximum size of a segment (excluding header)
 - Ideally, should be (Minimum MTU of any link from the source to the destination – TCP header size – IP header size)
 - Avoids fragmentation and reassembly at IP layer, which is costly
 - Hard to know minimum MTU of links end-to-end, though can be known in some cases, for ex, if all links are Ethernet
 - Default MSS = 536
 - All IP based networks must support a MTU of 576
 - Can be changed during connection establishment time using TCP options (will see shortly)
 - Cannot be changed once connection is established

TCP Header



- Before looking at the fields, note that TCP is full-duplex
 - If A establishes a connection to B, data can flow from either A to B or B to A
- For each-way transfer, say from A to B, some header fields are used for the forward direction (for example, data from A to B) and some for the reverse direction (for example, ack from B to A for the data sent from A to B)
- Piggybacking is used to reduce number of messages
 - Example: if data is transferred from A to B, the ack for it from B to A can be piggybacked (as value in a header field) when data is transferred from B to A

Header Fields

- Source Port and Destination Port: identifies sending and receiving applications
- Sequence Number: byte number of the first byte in this segment in the data stream sent by the application
 - Note that this is an actual byte number and not offset from the start, as first byte's sequence number can be random and need not be 0 or 1
- Acknowledgement Number: byte number that the sender of this message expects to receive next (for the data stream being sent in the opposite direction)

- Data offset (also called HLEN): header length in multiples of 4 bytes
 - Specified where does the data start in the segment
 - Needed because Options filed in the header can have variable length
- Window: How much data (in bytes) is the sender of the message (i.e., the receiver of the stream being sent in the opposite direction) willing to accept
 - Basically, size of free space in the buffer for the sender of the message
 - This field is advertised on all segments, carrying data or ack

- TCP Options
 - Window Scaling: provides multiple by which the window size advertised is to be multiplied
 - Maximum segment size: allows a receiver to specify the maximum size of a segment it is willing to accept
 - Selective Acknowledgments (SACK)
 - Some others, we will not do
- Checksum
 - For error detection
 - 16-bit word size, one's complement of the one's complement sum of words
 - However, uses a pseudo-header (will see shortly)

- TCP Flags

U	A	P	R	S	F
R	C	S	S	Y	I
G	K	H	T	N	N

- URG - urgent pointer is valid
- ACK - the acknowledgment number is valid
- PSH - The receiver should pass this data to the application as soon as possible (“push”)
- RST - reset the connection
- SYN - synchronize the sequence numbers to initiate a connection
- FIN - sender is finished sending data

- Note that IP address is not part of TCP header
 - This is as it should be, as IP address is a network layer information
- Then how does TCP software use the IP addresses to identify connections?
 - IP header is stripped off before the data is passed up to TCP layer
- IP software passes the source and destination addresses separately with each segment sent up

Computing TCP Checksum

- Prepends a **psuedo-header** before the TCP segment
- Pads octets of zeros to make (psuedo-header + TCP segment + pad length) multiple of 16
- Checksum computed on this entire thing
- Psuedo-header and pad octets are not transmitted, just used for calculating the checksum (*then why have it?*)
- Receiver will do the same and compare checksum

TCP psuedo-header

0	16	31
Source IP address		
Destination IP address		
Zero	Protocol	TCP Length

- Protocol = 6 (value for TCP in protocol field of IP datagram)
- TCP Length = length of TCP segment incl. TCP header (but not incl. psuedo-header and pads, it is computed (*how?*))

Basic Data Transfer

- Connection established
- All transmission involves TCP segments
- Application generates data
- TCP software puts application data in send buffer
- TCP segments are formed from the data in the send buffer
 - Can be of different sizes, formed at different times (we will see more details)
 - TCP header added
- Sender sends the TCP segments one by one as they are formed subject to send window size
- Timer set for each segment set, sender waits for acknowledgments
- If timeout, retransmit. If ack received, change window and transmit more segments if possible

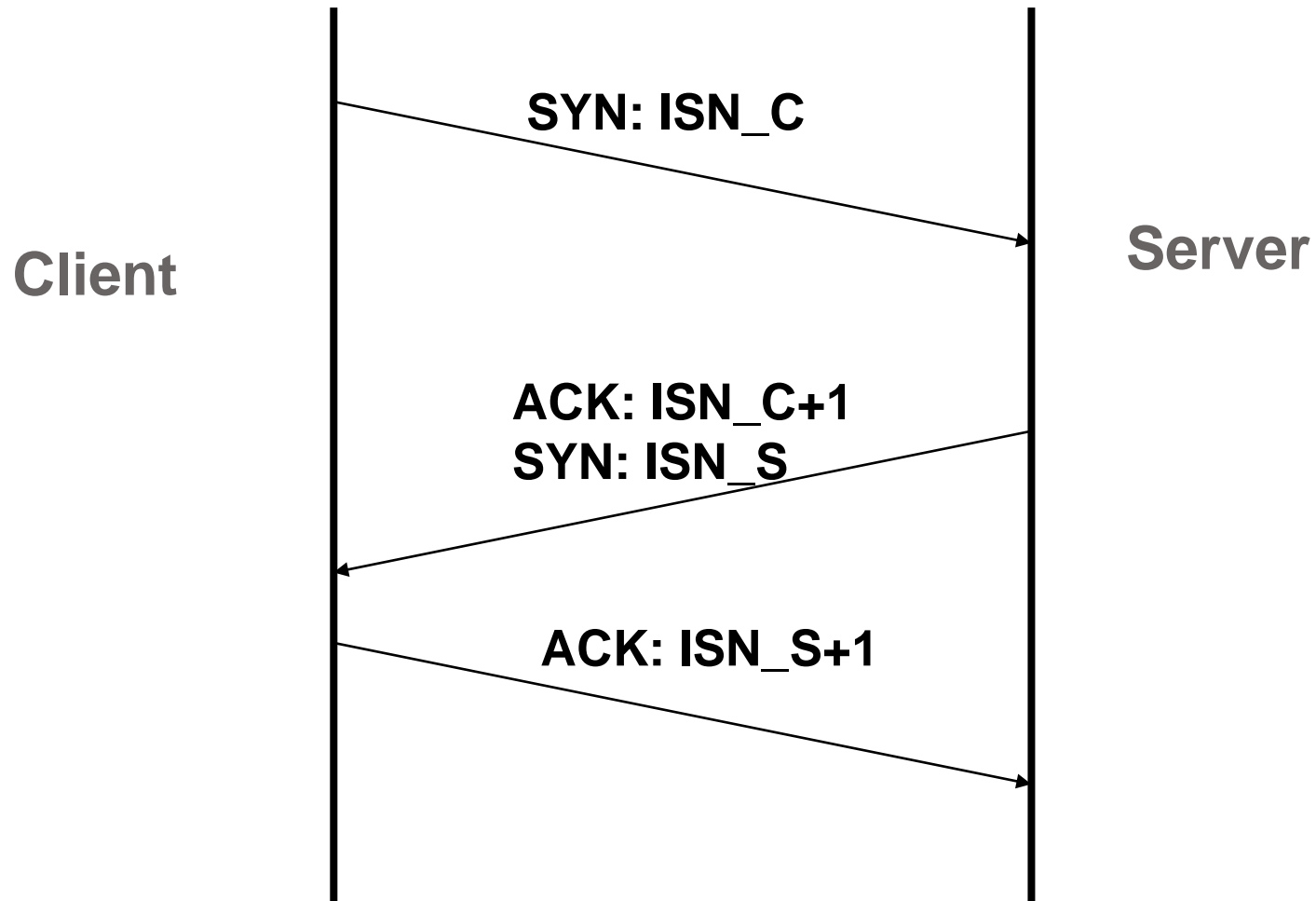
- Receiver sends acknowledgments by sending another TCP segment
 - Can be an acknowledgements segment (ACK flag set and a valid acknowledgement no. field) with no data (how to know this?)
 - Acknowledgements can be piggybacked on TCP segments carrying data in the other direction
 - Acknowledgement specifies next sequence no. the receiver expects
 - Cumulative positive acknowledgement, no NAK
- The above repeats until connection is terminated

Connection Establishment

- Purpose
 - Both sides should know that both sides are ready for data transfer
 - Each side should know the other side's **Initial Sequence Number (ISN)** – the starting sequence number of the first byte in the stream that will be sent
 - First byte address cannot always start at 0 or 1, there are problems of confusion between old and new connections
 - ISN's are usually chosen randomly
 - For full-duplex communication between A and B, A should know B's ISN and B should know A's ISN
 - Can negotiate certain TCP options also
- Done through **Three-Way Handshake**

Three-Way Handshake

- The client sends a SYN segment (SYN flag set) specifying the port number of the server and the client's ISN
- The server responds with a SYN + ACK segment (SYN and ACK flags set)
 - Sequence No. field contains server's ISN
 - Server acknowledges the client SYN using client ISN+1 in the acknowledgement no. field.
- The client acknowledges the SYN from the server using an ACK segment (ACK flag set) with the server's ISN+1 in the acknowledgement no. field
- The side sending the first SYN is said to perform an **active open**. The other side performs a **passive open**.
- However, after a connection is established, it is full-duplex communication, with no master/slave



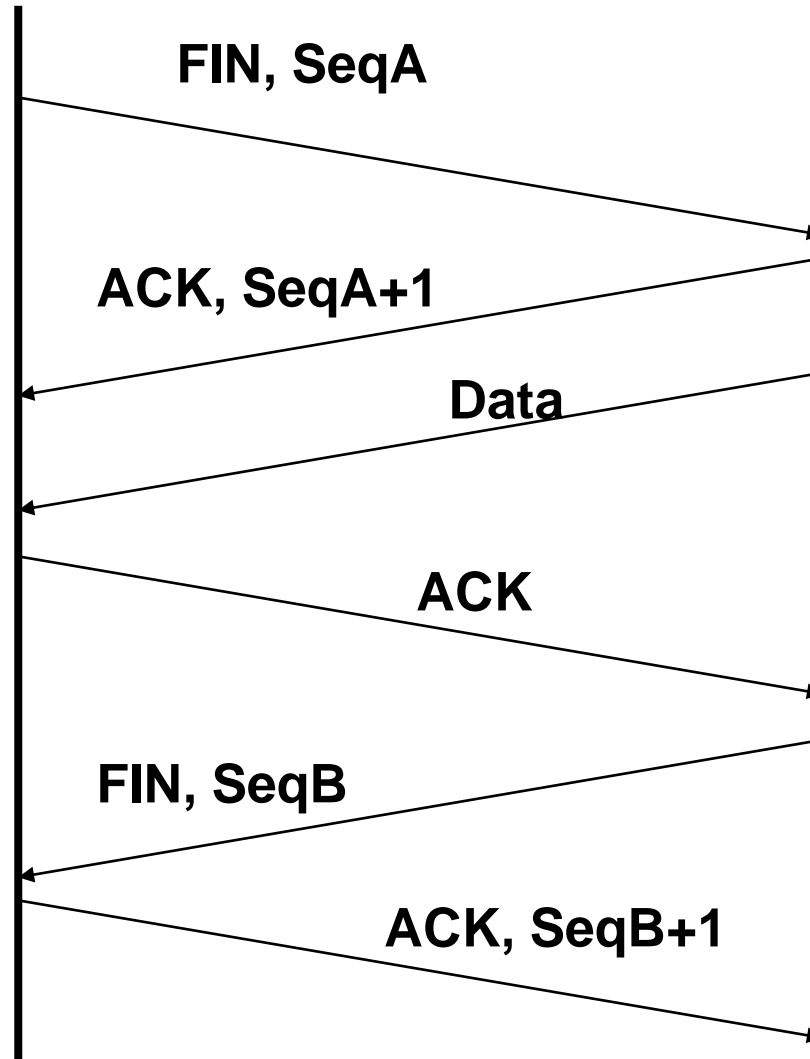
- Full duplex communication can now happen

Normal Connection Termination

- First note that a TCP connection opened is full-duplex. For termination, it is viewed as 2 separate connections, one in each direction
 - Each can be terminated separately
- Either side can initiate termination
 - Send FIN segment (FIN flag set)
 - Indicates that FIN sender is not going to send any more data
- Acknowledging FIN
 - FIN receiver must acknowledge the FIN segment by sending an ACK segment (ACK flag set) with acknowledgement no. = FIN segment sequence number + 1
- FIN receiver can continue sending data
 - Half open connection
 - FIN sender must continue to acknowledge, just cannot send any more data
- The above repeats when the other side wants to close connection

A

B



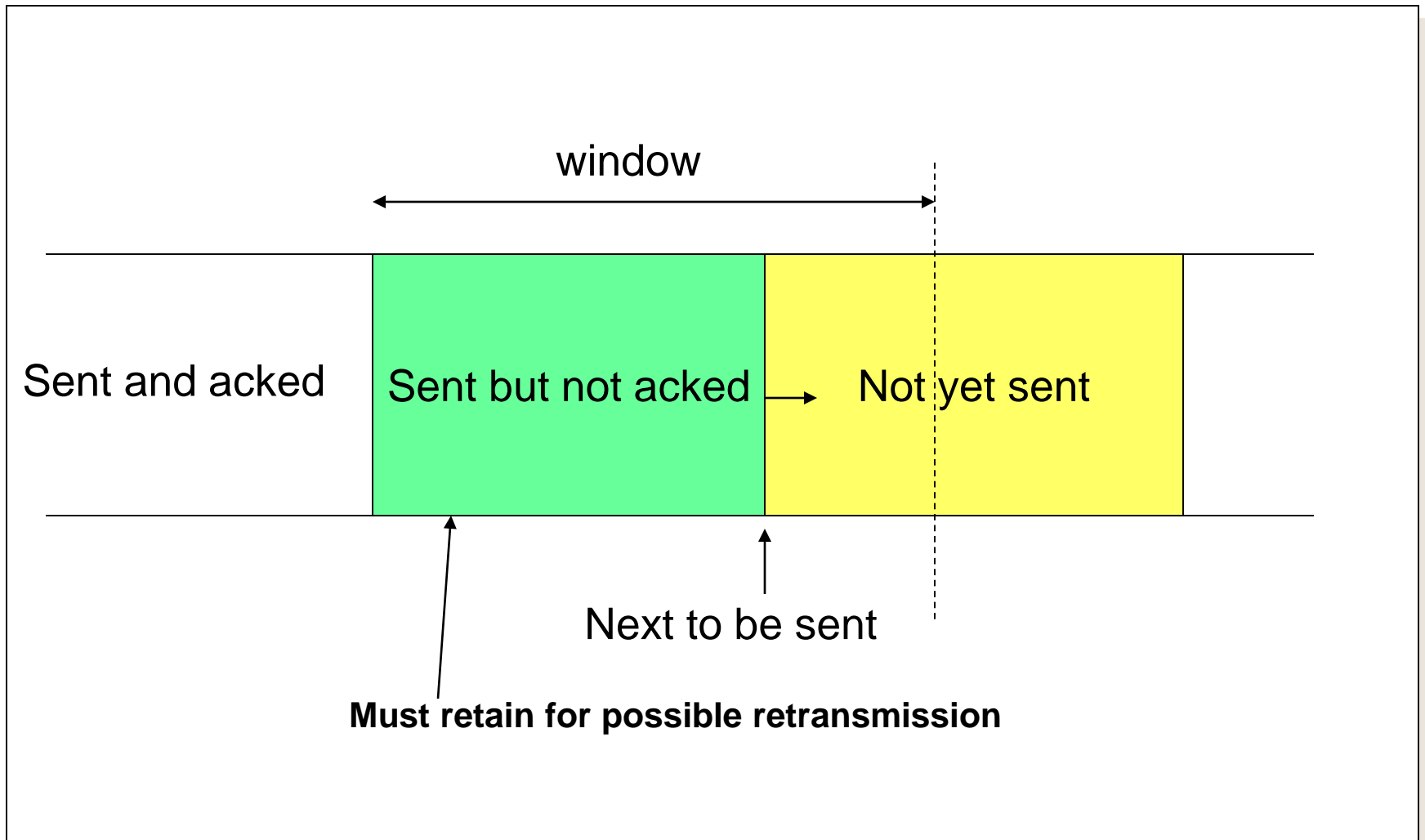
TCP Flow Control

- TCP uses a sliding window protocol without selective or negative acknowledgments.
 - Selective acknowledgments would let the protocol say it's missing a range of bytes. TCP can only say that it has received "up to byte N".
 - The protocol has no way to specify a negative acknowledgment. It can only say what has been received
- Concepts same as discussed earlier, with some differences

Sliding Window Based Flow Control

- TCP sliding window works at byte level
 - So we talk about how many bytes are sent and ack'ed, upto what byte can be sent etc.
 - NOT how many segments are sent etc.
- Sender maintains a window of size n and start of window X
- Sender can send up to n bytes starting from byte X without receiving an acknowledgement
- When the first p bytes of data are acknowledged then the window slides forward by p bytes to $X+p$. Sender can now send n bytes starting from $X+p$
- Window size determines how much unacknowledged data can the sender send as usual

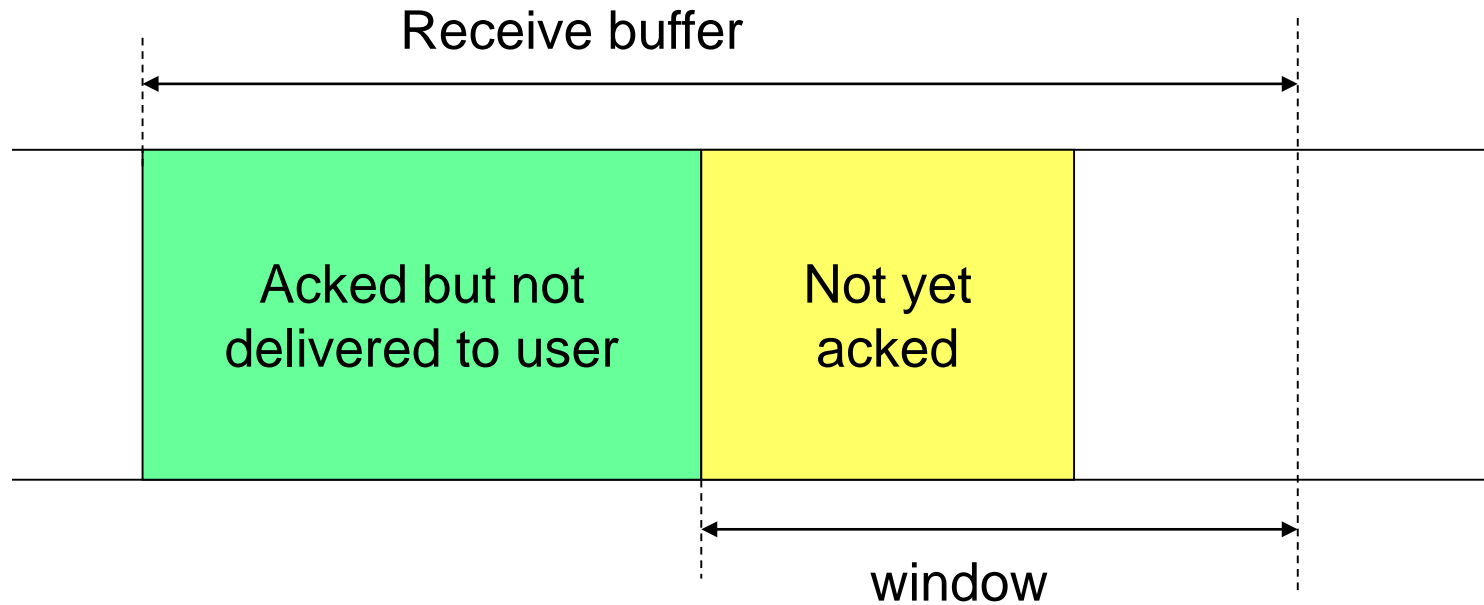
Sender Side Window



Problem

- Acknowledgment may be sent immediately by receiver, but receiver can delete acknowledged data from its buffer only after the data has been delivered to the application
- Application may read the data at different speeds and at different times
- So, depending on when and how fast the application reads the data, the receiver's window size may change
 - So even if the receiver current window size is W bytes and the receiver receives and acknowledges p bytes, the window size may not be reset to W , may still be only $W - p$ until the application picks up the data

Receiver Side Window



Solution

- Receiver tells sender what is the current window size in every segment it transmits to the sender (in **Window** field of header)
 - This can be data sent from receiver to sender in the other direction, or ack's for the data received from the sender
- Sender uses this advertised window size instead of a fixed value
- Window size (also called **advertised window**) is continuously changing, = current free buffer space at receiver
- Can go to zero; sender not allowed to send anything!
- Naïve implementations can cause **silly window syndrome**

Silly Window Syndrome

- The problem of TCP sending very small segments
 - Small segments are bad
 - Too much overhead given headers have to go with each segment
- Can be caused by both sender side and receiver side applications
 - Sender side application sending data very slowly
 - Receiver side application reading data from the receive buffer very slowly

Receiver Side Silly Window Syndrome

- Suppose that sender has lot of data to send, and sends one window full of data
- The application on the receiver side does not read the data yet, so receiver's buffer is full, so receiver advertises a window size of 0, sender blocks (but has more data to send)
- The application now reads very slowly, say 1 byte at a time
- For each 1 byte read, the receiver advertises a window of size 1
- Sender sends 1 more byte of data
- This repeats, causing a lot of 1-byte segments to be sent

- Solutions

- Do not advertise small sized windows

- Acknowledge data frames that arrive, but keep advertising a 0-sized window until either (i) half of the receive buffer is free, or (ii) receiver buffer has at least MSS amount of free space

- Delay the acknowledgements for data frames that arrive

- Sender side window cannot move, so cannot transmit more data than current window

Sender Side Silly Window Syndrome

- Caused when the application generates data slowly
 - Say 1 byte every time
- If data is sent immediately, lots of small segments are sent
- If it is to be delayed, how long to delay? No clue when the Sender side application will generate data again
- **Nagle's Algorithm:**
 - If there is previous data that is sent but not acknowledged, place any further data to be sent in the send buffer but do not send until:
 - Either an acknowledgement is received, or
 - MSS sized data is available for sending
- Nagle's algorithm decides when are segments sent by a TCP sender, subject to window restrictions

When are ACKs sent?

- Suppose A is sending data to B
- TCP acks are cumulative, acknowledges the longest contiguous sequence from the start that is received
 - If ISN of A = 1000, A has sent 4 segments with sequence numbers 1001 (why not 1000?), 1600, 2800, and 3100, and B has received the 1st, 2nd and 4th segments only, then
 - Longest contiguous sequence from start received is byte numbers 1001 to 2799
 - TCP will ack with ack no. = 2800 (next byte expected)
 - Segment 1 and 2 are received in-order, Segment 3 is a missing segment, Segment 4 is a segment received out-of order
 - Note that receiver does not know there is one missing segment, it just knows there is at least one

- B sends an acknowledgement if/when
 - If B has data to send to A, always piggyback the ack for the data received from A (ACK flag set, byte no. of next byte to expect put in Acknowledgement Number field)
 - If B has no data to send, receives a segment from A in-order (sequence number = next sequence number expected)
 - If all previous in-order segments are acknowledged, delay sending the acknowledgement until one more segment arrives or a time elapses (typically 500 milliseconds), then send
 - If B gets an out-of-order segment having a higher than expected sequence number, send an ack with next sequence number expected
 - If the receiver gets a missing segment which extends the longest contiguous byte stream from the start that it has received, send an ack with the next sequence number to expect
 - If a duplicate segment arrives, discard the segment but send an ack with the next sequence number expected

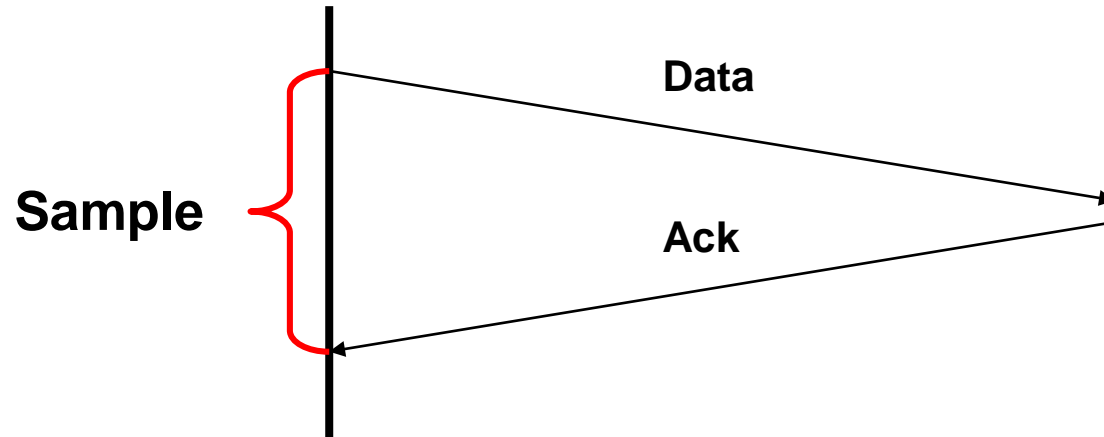
Ensuring Reliable, Inorder Transfer

- Checksum (mostly) guarantees end-to-end data integrity
- Sequence numbers detect packet sequencing problems:
 - Duplicate: ignore
 - Reordered: reorder or drop
 - Lost: retransmit
- Lost segments detected by sender
 - Use timeout to detect lack of acknowledgment
- Retransmission requires that sender keep copy of the data.
 - Copy is discarded when ack is received

Ensuring Reliability

- Keeps separate timer for each unacknowledged segment
- Uses retransmissions of segments whose timers expire
- Retransmission time-out depends on round-trip delay
 - Round-trip delay varies based on path followed, network condition etc. So how to set?
 - Solution - estimate RTT dynamically

Estimating Round-trip Delay



- Every Data/Ack pair gives new RTT estimate
- Can get lots of short-term fluctuations

TCP Round-trip Estimator

- Original
 - Round trip times estimated as a moving average:
 - $\text{New RTT} = \alpha (\text{old RTT}) + (1 - \alpha) (\text{new sample})$
 - Set timeout to $\beta \times \text{RTT}$
 - Typically, $\alpha = 0.8-0.9$, $\beta = 2$ originally
- However, a fixed β does not adapt well to high variance in RTT
 - Ideally, $\text{timeout} > \text{real RTT}$
 - $= \text{estimated RTT} + X$
 - If there is a high variance in the RTT, need a higher X to ensure timeout is always greater than RTT
 - RTT variance is high at high loads

TCP Round-trip Estimator

- Modified:
 - Set timeout = Estimated RTT + $\delta \times$ deviation
 - $\delta = 4$ typically
- How to compute the deviation in RTT?
 - Deviation = $(1 - \rho) \times \text{deviation} + \rho \times (\text{sample RTT} - \text{estimated RTT})$
 - Typically, $\rho = 0.25$

Acknowledgement Ambiguity

- If a segment is retransmitted, and an ack for it is received, is it an ack for the retransmitted frame or the original frame? What RTT sample value to take?
- Solution: [Karn's algorithm](#)
 - Do not accept samples for segments that are retransmitted
 - Use a timer backoff scheme to increase timeout to account for scenarios when round trip delay increases suddenly

Karn's Algorithm

- Compute timeout using the estimated round trip time as before
 - Use only samples for segments that are not retransmitted
- If timer expires and retransmission occurs
 - For every retransmission, set timeout = $\gamma \times$ timeout, subject to an upper limit
 - Typically, $\gamma = 2$

Ensuring In-order Delivery

- TCP uses sequence number field in segment headers to reconstruct the data stream at the receiver side
- What happens to segments received out-of-order?
 - TCP specification does not restrict, up to implementations

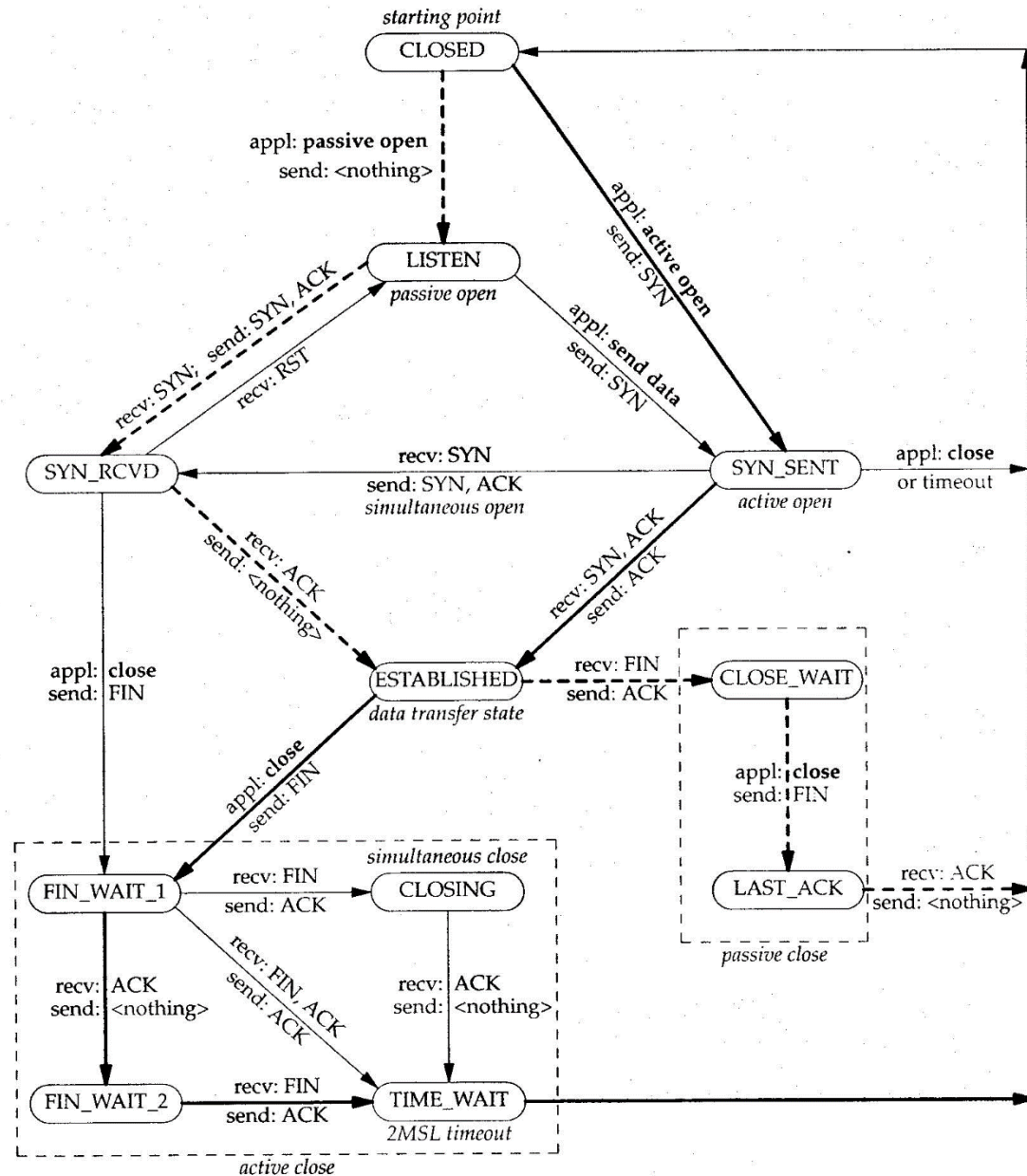
Out of Band Data

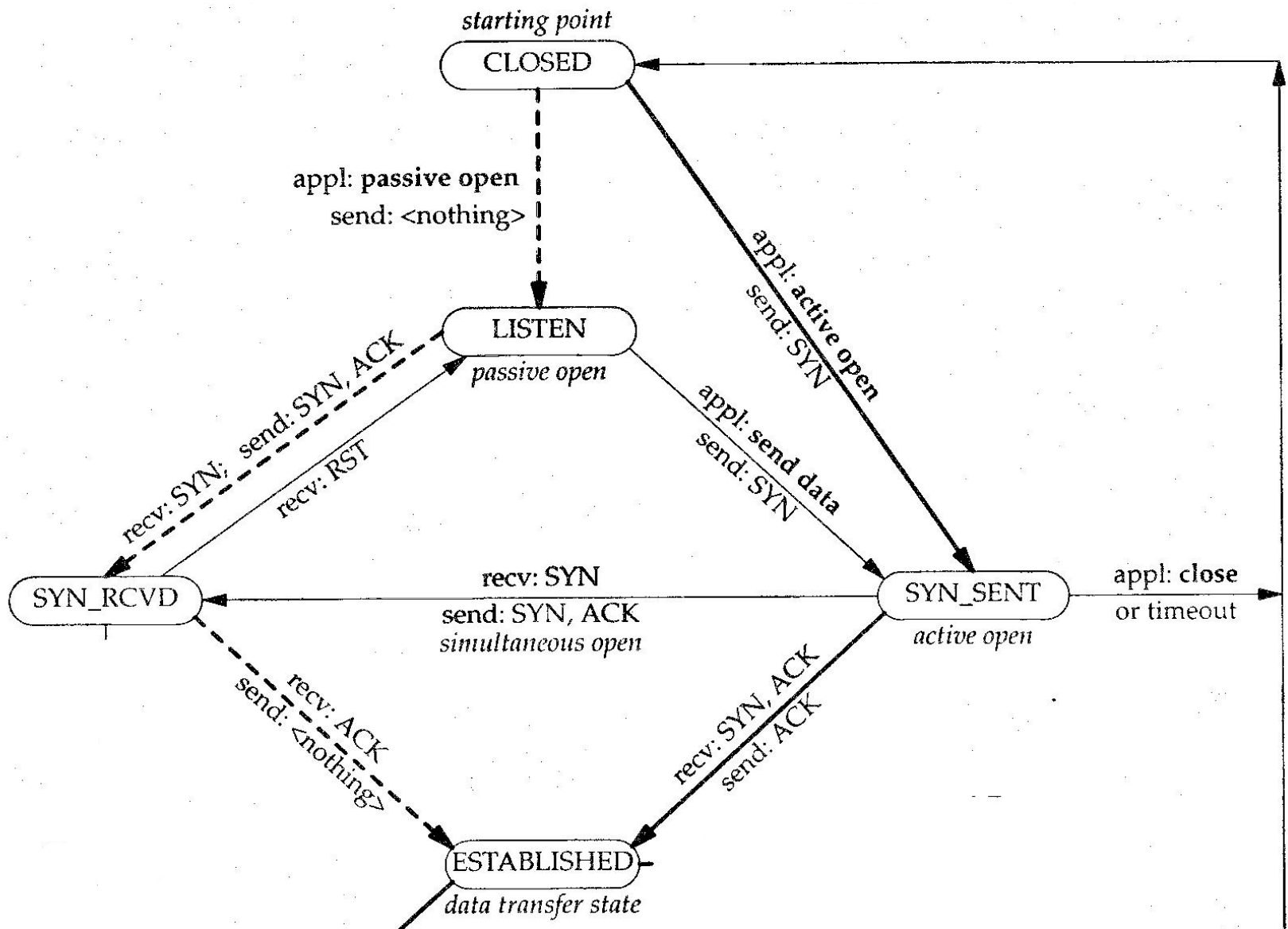
- Sometimes there is need to send urgent data that needs to be delivered to the application out of turn
- Set URG flag bit to indicate presence of out-of-band data in segment
- Set URGENT pointer to position in segment where urgent data ends

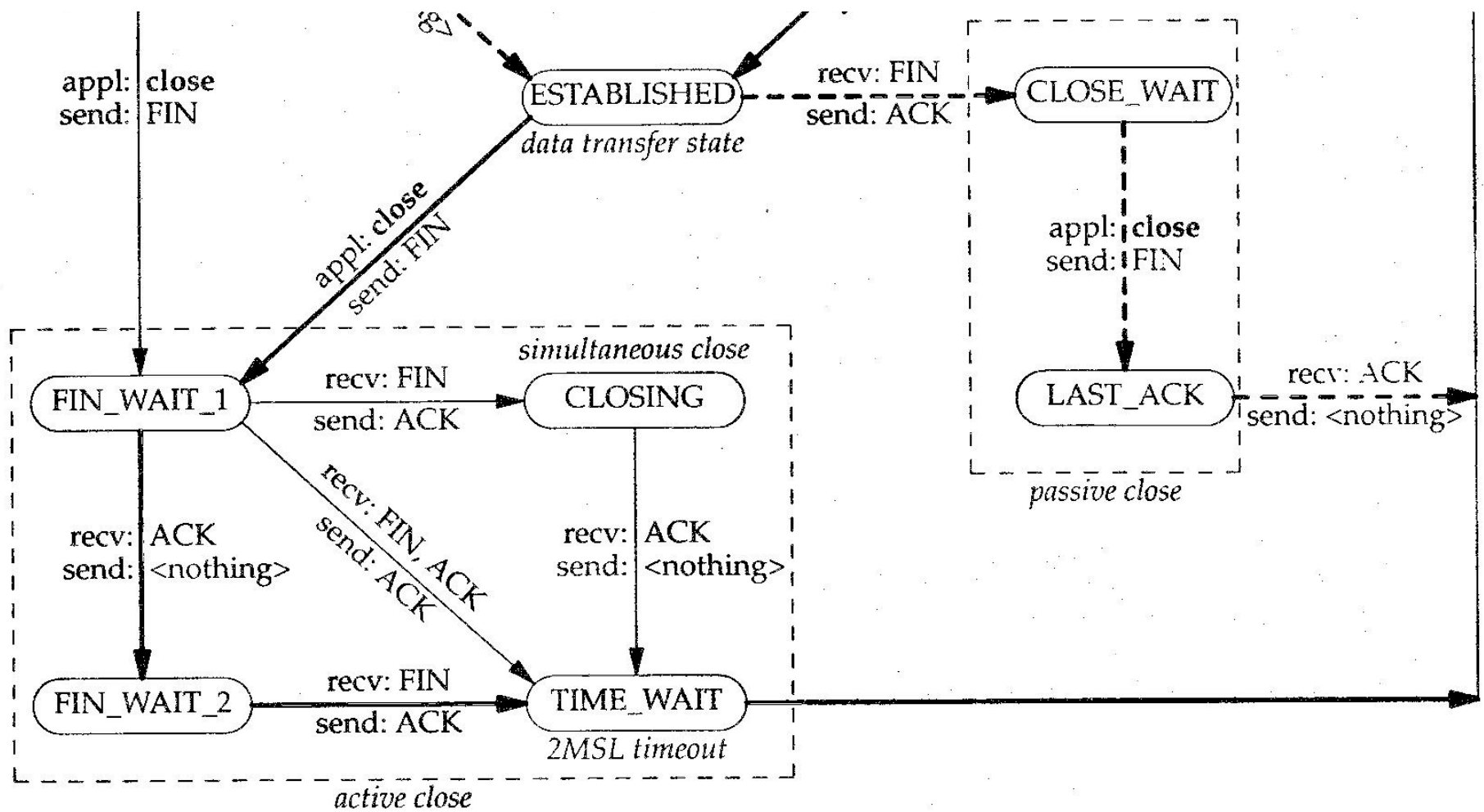
Resetting a TCP Connection

- Can be used to abort a TCP connection in case of any abnormal condition
- Initiated by one side, sends a TCP segment with the RST flag set in header
- The other side responds with a TCP segment with the RST flag set in header
- Immediately releases all resources and closes the connection in both directions

TCP State Diagram





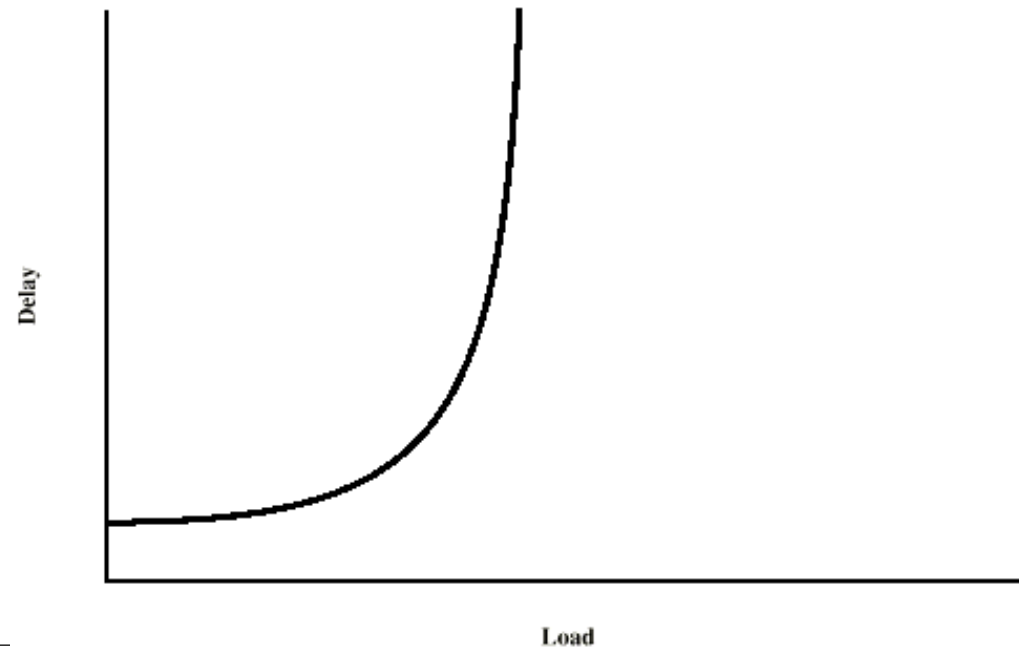
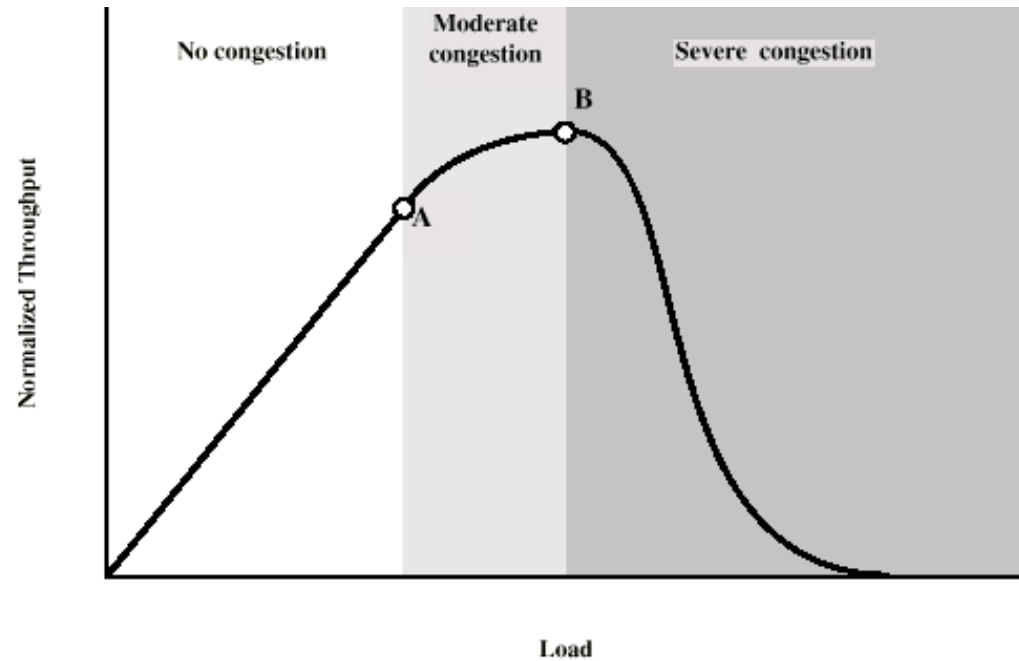


TCP Congestion Control

What is Congestion

- The number of packets transmitted on the network is greater than the capacity of the network
- More specifically
 - Packets come to some router at a rate higher than the rate at which the router can process packets and send them out
 - Causes packets to get queued at router buffers
 - Eventually the buffer fills up, causes packets to start getting dropped
- Why is it bad?
 - Retransmissions cause bandwidth wastage
 - Delay is increased
 - **Congestion Collapse** - retransmissions due to drop due to congestion can further increase congestion!!

Effect of Congestion



Congestion Control

- Congestion control aims to keep number of packets below level at which performance falls off dramatically
- End-to-end flow control is not enough!
 - Independent senders can each have flow control with their receivers, but together can still inject large number of packets in the network

How to detect congestion?

- Implicit congestion Signaling
 - Infers congestion indirectly from other events such as delay, drop
 - Transmission delay may increase with congestion
 - Packet may be discarded
 - Source can detect these as implicit indications of congestion and dynamically adjust packet sending rate
 - Basis for TCP congestion control

- Explicit congestion signaling
 - Congested router may send special packets back to the source
 - Ex. ICMP source quench packets
 - Source can cut back on sending rate until no more ICMP source quench packets received
 - ECN (Explicit Congestion Notification)
 - Router sets flag in header of IP packet from sender to receiver to indicate congestion
 - Part of TOS bits in IP header used
 - Receiver echoes back the flag in TCP header back to sender so that sender can adjust rate
 - Additional flags defined

TCP Congestion Control

- Sender estimates level of congestion from packet delay/drop
- Send more when no congestion detected
- Slow down when congestion detected
- Two issues to address:
 - Detecting congestion
 - Adjusting sending rate

Detecting Congestion

- Detected by detecting
 - Increasing round trip delay
 - TCP Vegas
 - We will not do
 - Packet drops
 - Most commonly used
 - TCP Tahoe, Reno, New Reno,...
- How to detect potential packet drops?
 - Timeout
 - Too many duplicate acknowledgements
 - Indicates packets are reaching (since ack's are sent) but some earlier packet has not reached yet
 - Probably lost because if out of order, only a few duplicate acks will be there

Adjusting Sending Rate

- Based on TCP Congestion Window (**cwnd**)
- Limits how much data can be in transit (similar to receiver window advertisement, but purpose is different)
- Max. window size at sender at any time =
$$\min(\text{cwnd}, \text{recvAdvertisedWindow})$$
- **cwnd** is varied to control sending rate to address congestion
 - Varied by sender, congestion control is sender-side task
- Receiver advertised window varied to address end-to-end flow control
 - Varied by receiver, as discussed earlier

- Basic principle
 - On detecting packet drop, decrease `cwnd`
 - On receiving ack, increase `cwnd`
- Two phases of TCP congestion control
 - slow start
 - congestion avoidance
- Which phase to do?
 - Based on a variable `ss_thresh` (slow start threshold)
 - `cwnd` initially set to maximum segment size (MSS)
 - slow start : $cwnd \leq ss_thresh$
 - congestion avoidance : $cwnd > ss_thresh$

Before we go any further

Lets clear up some confusion that may arise
in interpreting `cwnd`

- `cwnd` is implemented as number of bytes (as it should be as TCP is byte-oriented)
- However, most descriptions talks about `cwnd` in terms of number of segments
- A segment in the context of `cwnd` means a full-size segment (size = maximum segment size, MSS), so easy to convert
- RFC 2581 talks in terms of both
- We will also talk about `cwnd` in term of no. of segments (easier to follow from text for students)

- Various implementations of TCP congestion control exist, differing mostly in how and when **cwnd** is computed
- We will do TCP Tahoe (the original TCP) and TCP Reno (the next extension, which has the basic things still used)

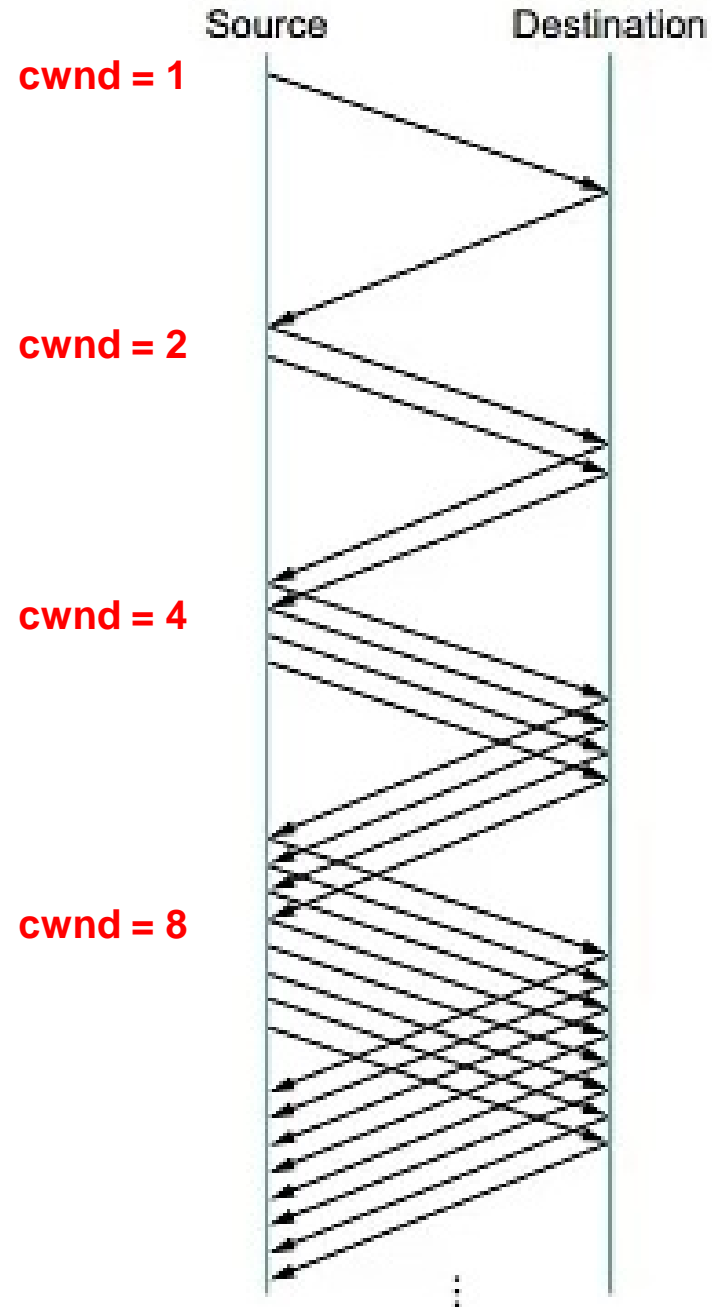
TCP Tahoe

- ARPANET switches to TCP in 1963
- Internet suffers congestion collapse in 1986
- Van Jacobson fixes TCP, TCP Tahoe introduced in 1987-88
 - Slow start, congestion avoidance, fast retransmit
- Extended with Fast Recovery in TCP Reno in 1990

Slow Start

- Used to find a good sending rate initially at startup, or while recovering after congestion
- Whenever starting traffic on a new connection, or whenever increasing traffic after congestion was detected:
 - Initialize $cwnd = 1$
 - Each time a segment is acknowledged, increment $cwnd$ by 1
- Continue until
 - ss_thresh is reached, or
 - Timeout, or
 - 3 duplicate ACKs received

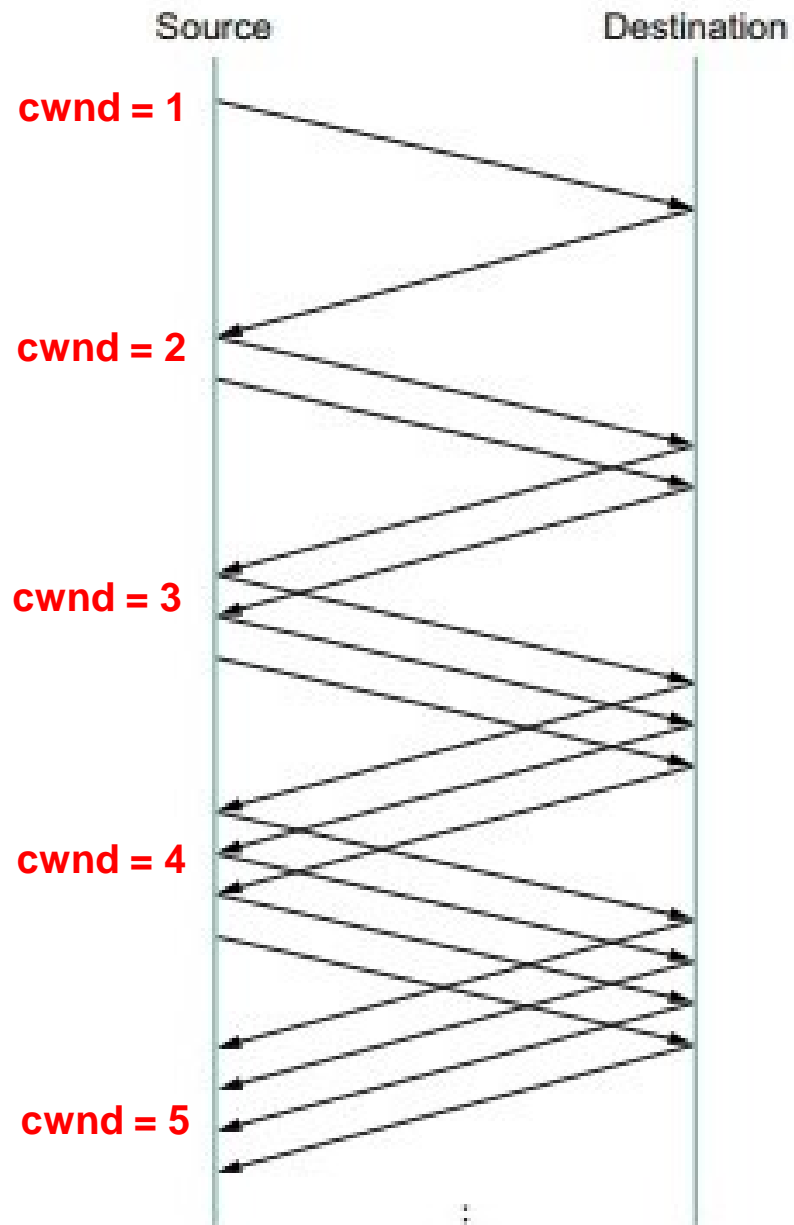
- Slow start is not so slow!!



- The change in congestion window is per ack
 - So a cumulative ack will still increase the window by 1 only
- Given a $ss_thres = 8$, work out how many RTTs it will take for slow start to reach the threshold if there is one ack per 2 segments

Congestion Avoidance

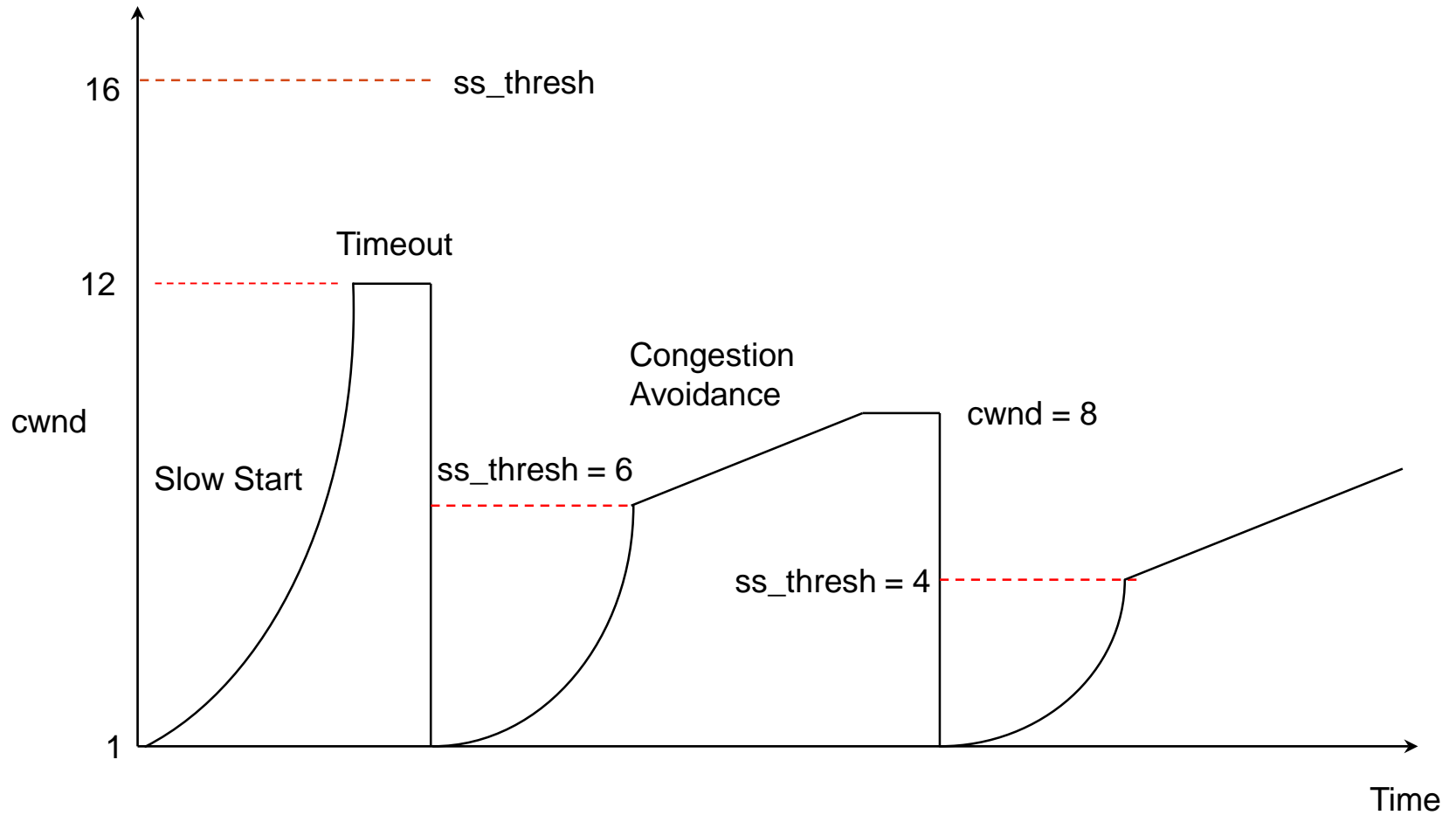
- Slow start sets a “good” congestion window fast
- Congestion avoidance slows down the increase in *cwnd*
 - Why do we need to slow down even if there is no timeout?
- If $cwnd > ss_thresh$ then
 - each time a segment is acknowledged
 - increment *cwnd* by $1 / cwnd$ ($cwnd += 1 / cwnd$)
- *cwnd* is increased by one only if all segments have been acknowledged
 - Increases by 1 per RTT, vs. doubling per RTT in slow start
 - Additive Increase



On Detecting Congestion

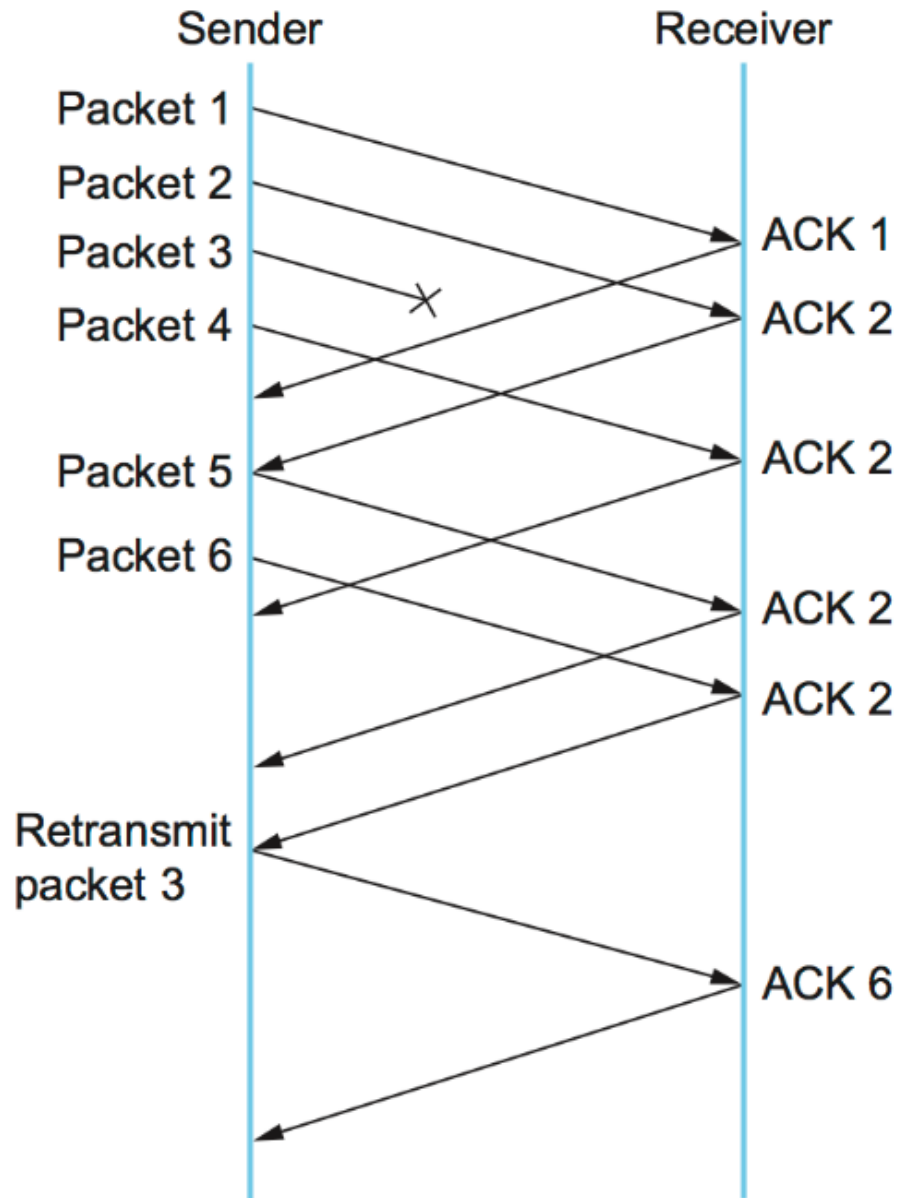
- On a timeout
 - ss_thresh is set to half the current size of the congestion window:
$$ss_thresh = cwnd / 2$$
 - $cwnd$ is reset to one
$$cwnd = 1$$
 - Slow-start phase is entered
 - This is called **multiplicative decrease**

Example



Fast Retransmit

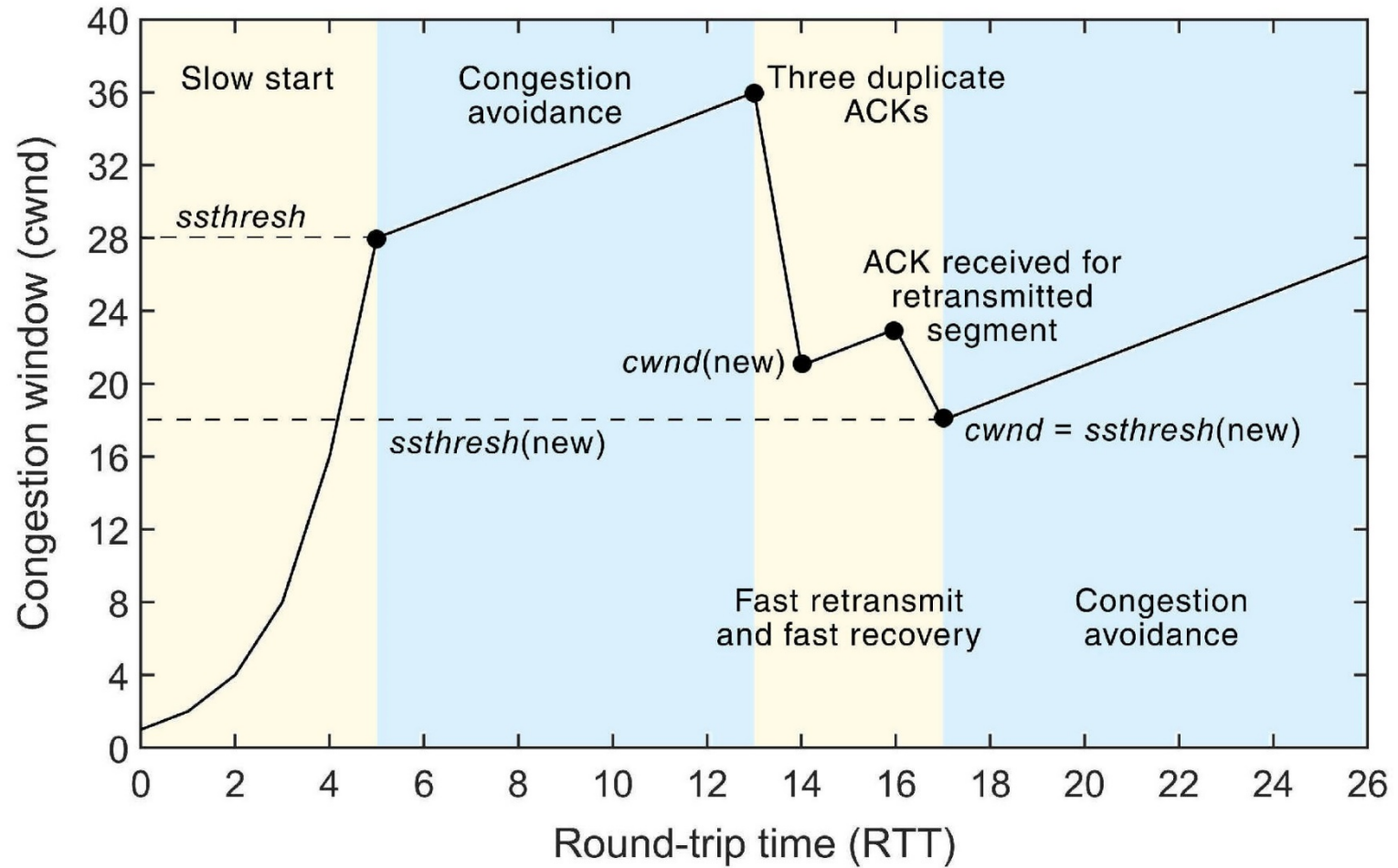
- On receiving 3 duplicate ACKs (total 4) for the same segment,
 - Retransmit the segment without waiting for timeout
 - Set $ss_thresh = cwnd / 2$
 - Set $cwnd = cwnd / 2 + 3$
 - Why the +3?



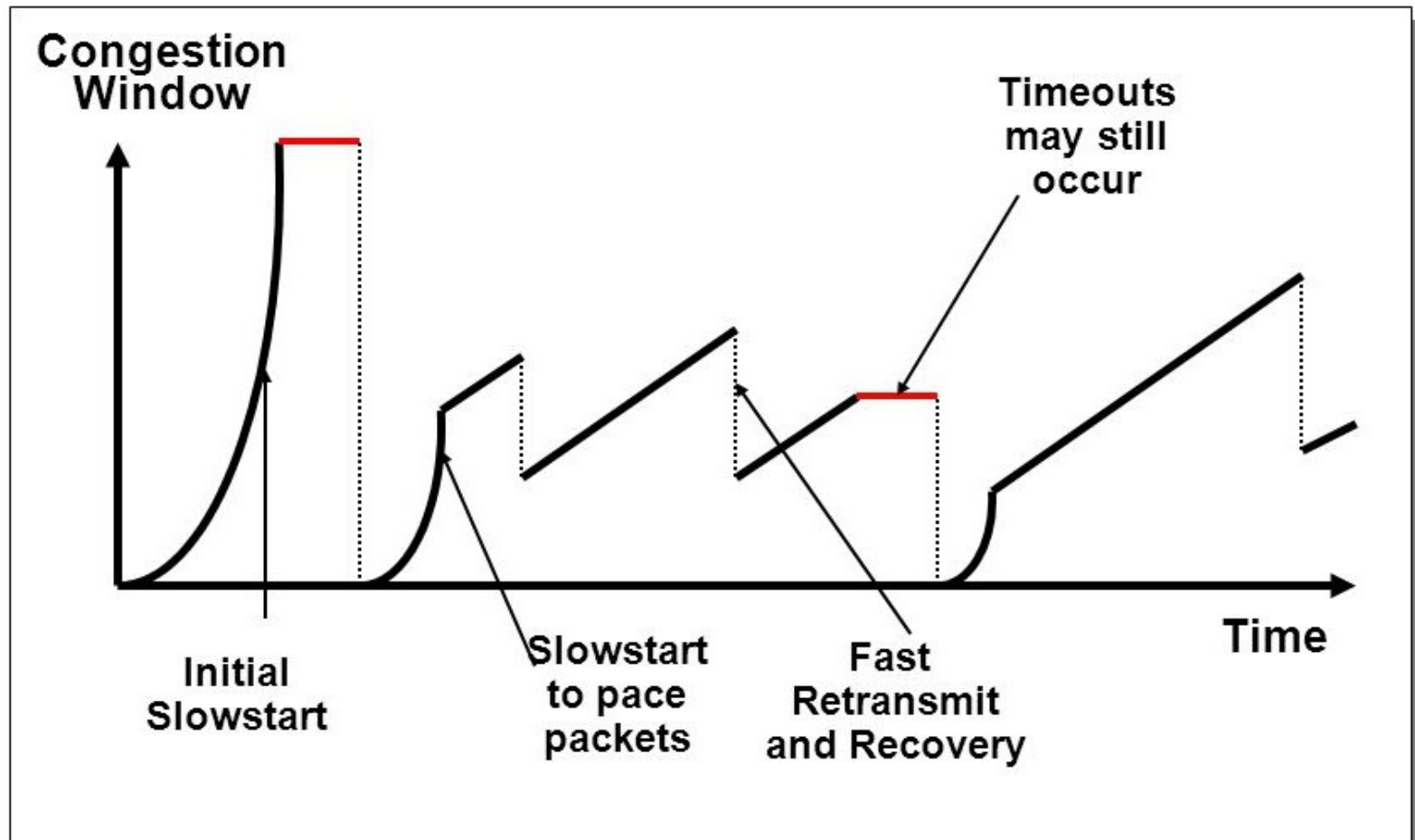
TCP Reno

- Adds **Fast Recovery**
 - On 3 duplicate ACKs, go back to congestion avoidance
- Fast Retransmit and Fast Recovery are implemented together
- On receiving 3 duplicate ACKs for the same segment,
 - Retransmit the segment without waiting for timeout
 - Set $ss_thresh = cwnd / 2$
 - Set $cwnd = cwnd / 2 + 3$
- On subsequent duplicate ACK, $cwnd = cwnd + 1$
- On new ACK, $cwnd = ss_thresh$

TCP fast retransmit and fast recovery



TCP Saw Tooth Behavior



TCP Variants

- Many implementations of TCP has been done since Reno, some to address some problems, some suited for specific scenarios (ex. links with high bandwidth delay product (Long, fat links) etc.)
 - TCP New Reno (1999)
 - TCP Vegas (1995)
 - TCP SACK (1996)
 - TCP Westwood (2001)
 - Fast TCP (2006)
 - HSTCP (High Speed TCP) (2003)
 - TCP Bic (2004)
 - Cubic TCP
 - Default implementation for Linux