# C6 Transport Layer (UDP and TCP)

## 1. Transport Layer

- Provides end-to-end service for transferring data between processes (process-to-process communication)
- Only implemented at end hosts (routers use only until network layer)

**Port Numbers**

- Single transport layer used to support multiple application processes through the use of ports; transport layer is said to perform multiplexing / de-multiplexing
  - Multiplexing: gathering data from multiple processes and passing it to a single network layer
  - De-multiplexing: delivering of data from single network layer to different processes correctly

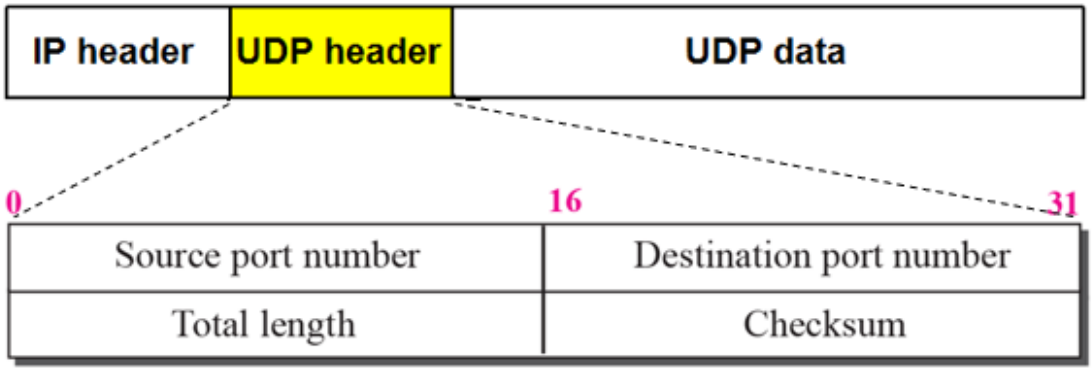
**Protocols**

- User Datagram Protocol (11, base 16)
  - Unreliable, connectionless
  - Datagram oriented
  - Simple
- Transmission Control Protocol (06, base 16)
  - Reliable
  - Stream oriented
  - Complex
  - Resends if lost
- Application layer is aware that UDP sends each message as a datagram and views TCP as a channel for sending stream of bytes and **not aware** that bytes are sent in blocks called segments

```
- UDP in java

    request = new DatagramPacket();

- TCP in java

    outStream = socket.getOutputStream();
    request = outStream.write(message);
```

## 2. User Datagram Protocol (UDP)

- Used for applications that do not need reliable communications
  - Broadcasting, advertising messages to users
  - Sending live video streams over Internet
- Provides process-to-process communication service for applications to use

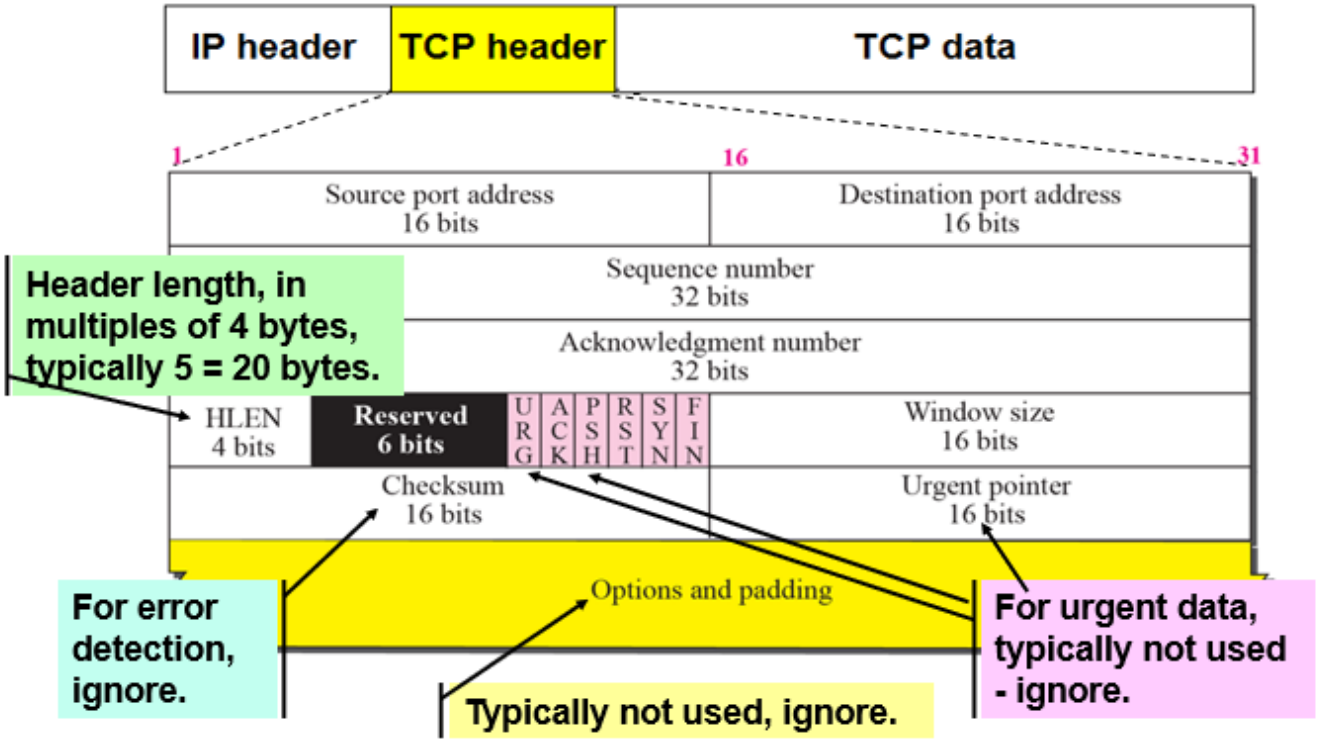



## 3. Transmission Control Protocol (TCP)

- Used to support applications requiring reliable communications
- Features
  - Connection Management: Connection must be setup before data exchange can be performed
  - Flow Control: Sender will not overwhelm receiver
  - Error Control: Receiver detects errors, sender retransmits error packets
  - Congestion Control: During transmission, sender detects network usage (congestion) and adjust transmission rate

### Header Format

- Sequence Number (SN)
  - Each TCP connection will start with a different SN called Initial Sequence Number
  - Position of each data byte in byte stream is labelled from ISN+1, and cycle back to 0 once reaching $2^{32} - 1$
    - i.e. first byte = $(\text{ISN} + 1) \text{mod} (2^{32} - 1)$
  - SN indicated the position of the first byte in each segment
- Acknowledgement Number (AN)
  - AN of the next data byte expected from sender
  - Also imply all bytes up to AN-1 have been received correctly
- Window size (W)
  - Indicates number of bytes (credits) counting from AN that the receiver is ready to accept




## 3.1 Connection Management

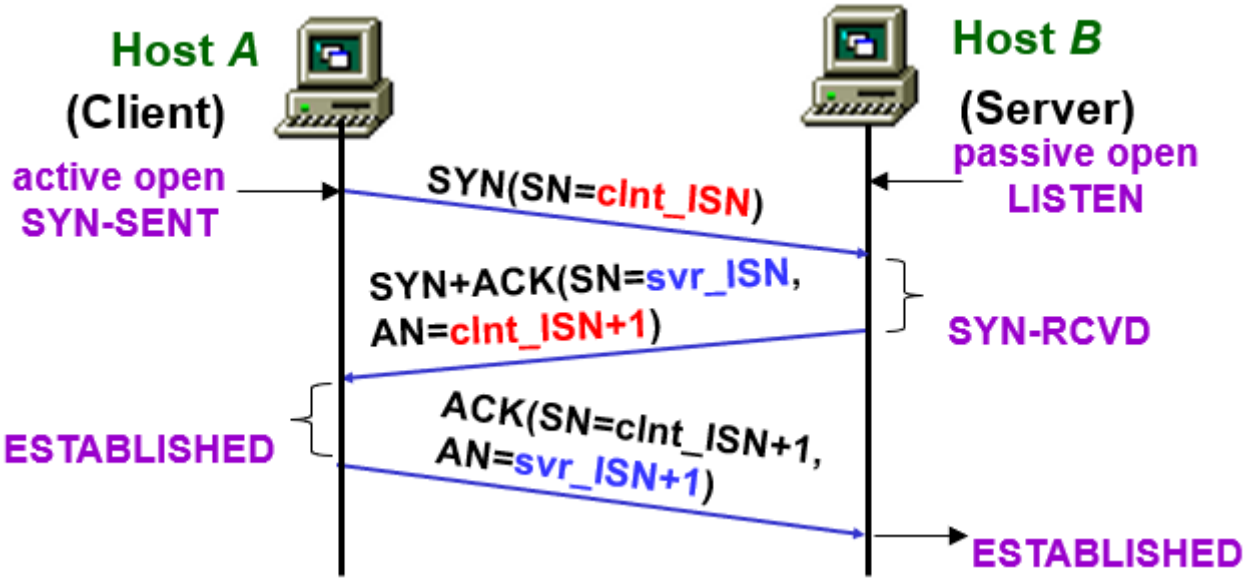
- Connection establishment serves to
  - Ensure both ends are ready to communicate
  - Establish initial sequence number (ISN)
  - Exchange parameter (e.g. window size in bytes)
  - Allocate resources to support connection
- Starts with a synchronisation (SYN) request

### Control bits

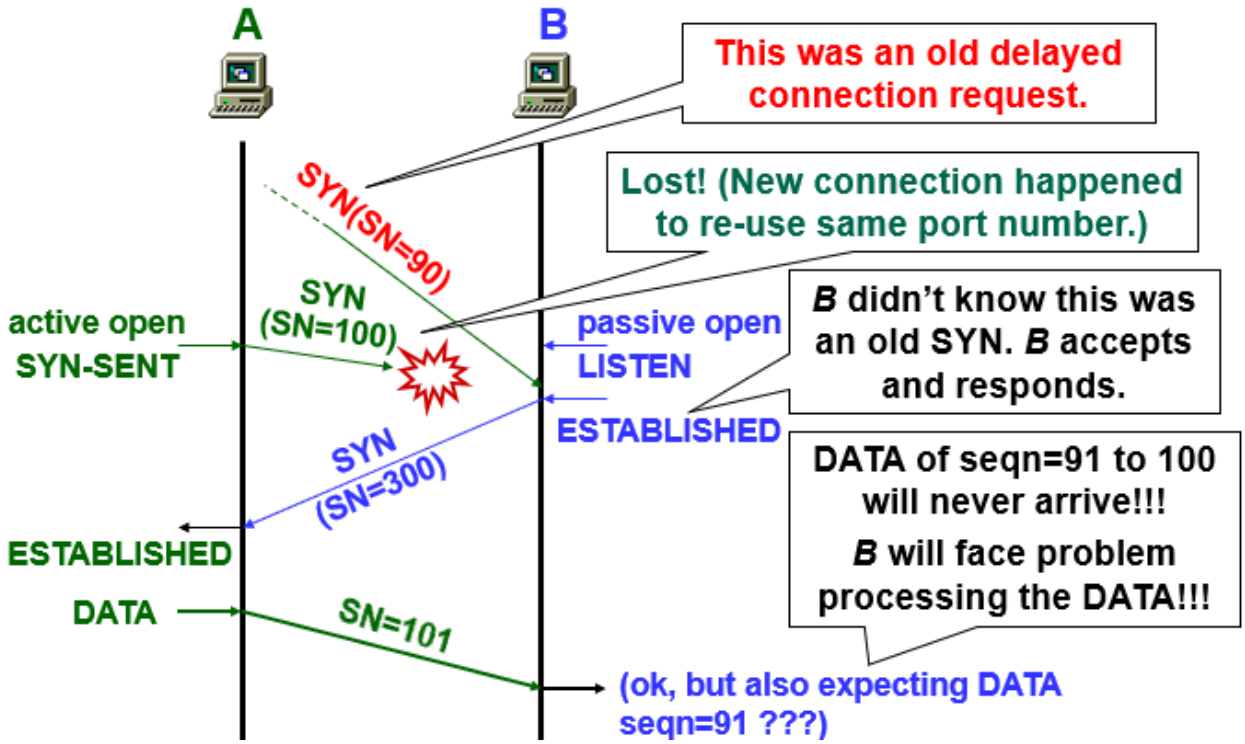
- Control bits can be found in the pink section of TCP header, URG and PSH usually not used
  - SYN set to synchronise sequence numbers for connection establishment
  - FIN set to terminate connection
  - RST set to reset connection when error occurs during connection establishment
  - ACK set to show acknowledgement number is valid

### Connection Establishment

- Uses **3-way handshake** with positive acknowledgements

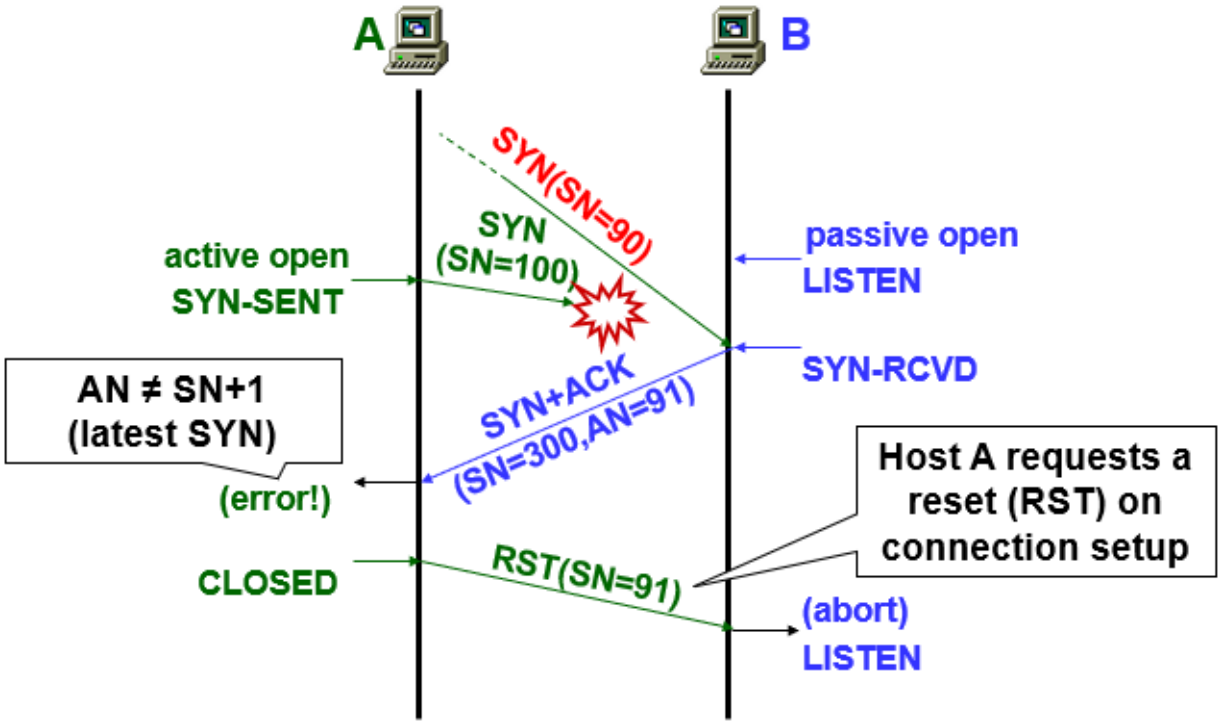



- **2-way handshake** is not used
  - TCP resends packets assumed delayed / lost due to time out, may lead to duplicate packets



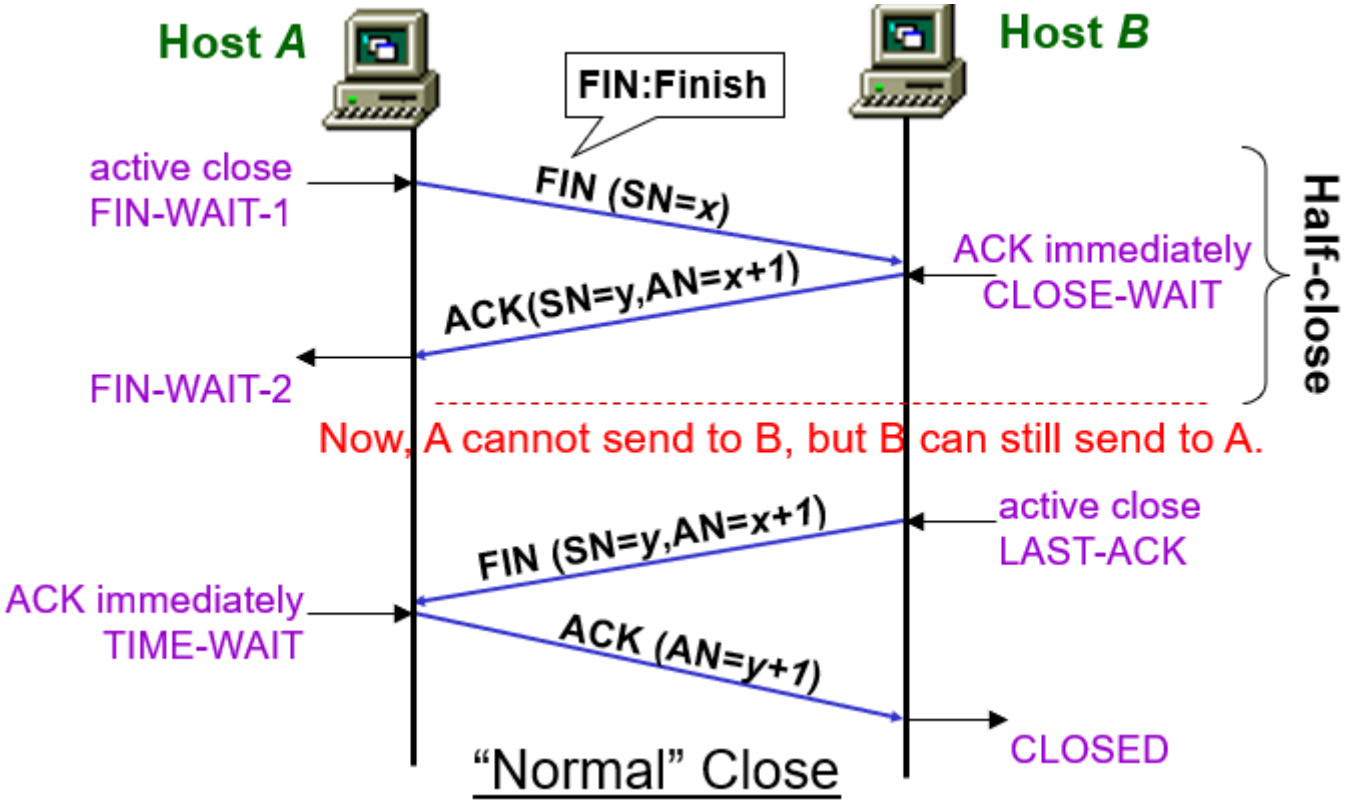

- 3-way handshake solves the above problem

- Synchronisation becomes reliable if connection request positively acknowledged

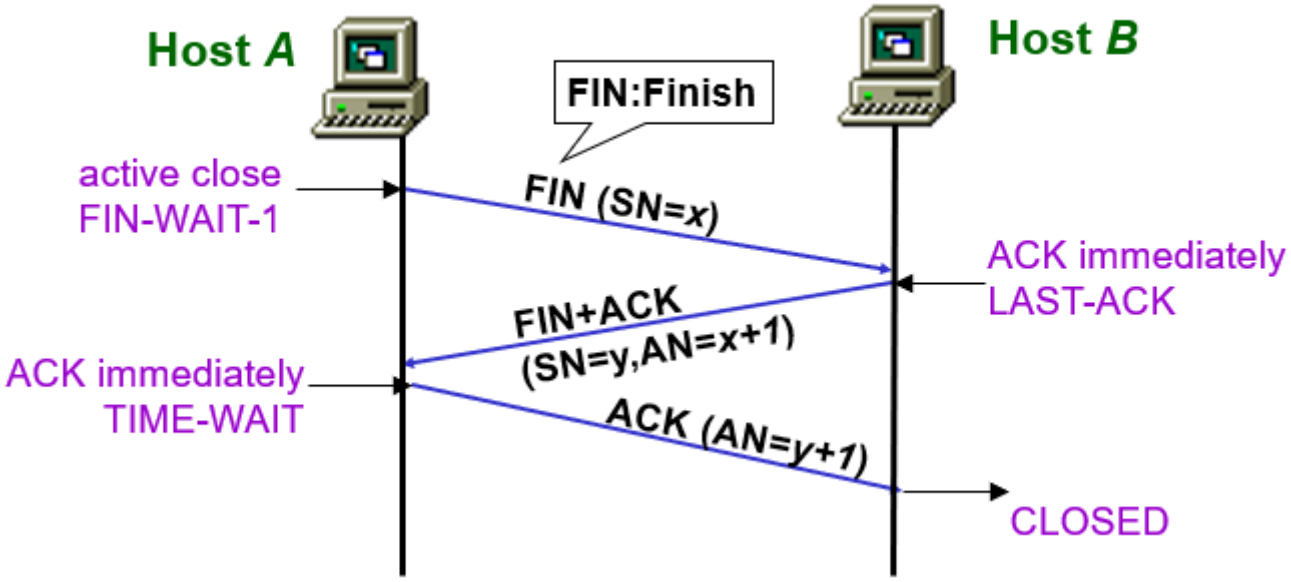



  - B sends acknowledgement for late arriving SN=90 which raises an error in A, leading to a reset request

**Connection Termination**

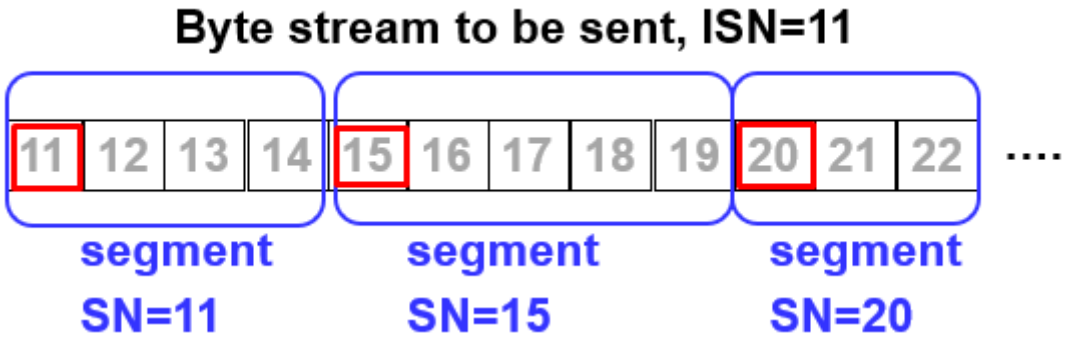



or



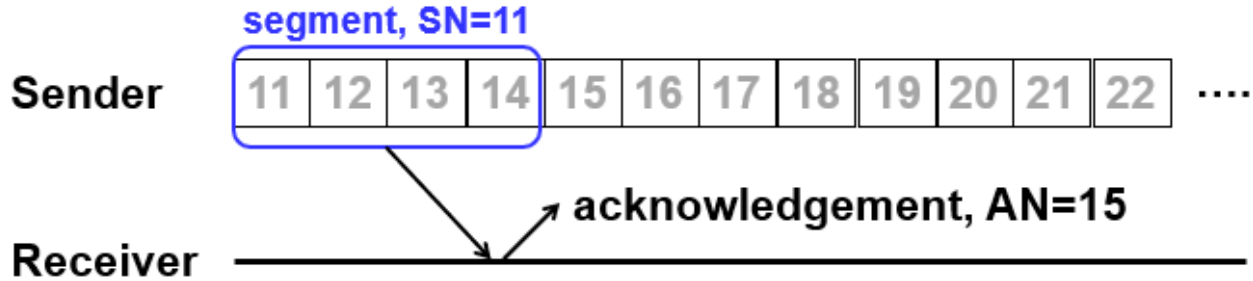

### Sequence Number

- A SYN and FIN segment **does not** carry data but consumes one SN
- A SYN+ACK, FIN+ACK **does not** carry data but consumes one SN
- ACK segment **does not** carry data and does not consume any sequence number
- SN of a TCP segment indicates the position of its 1st data byte in a data stream



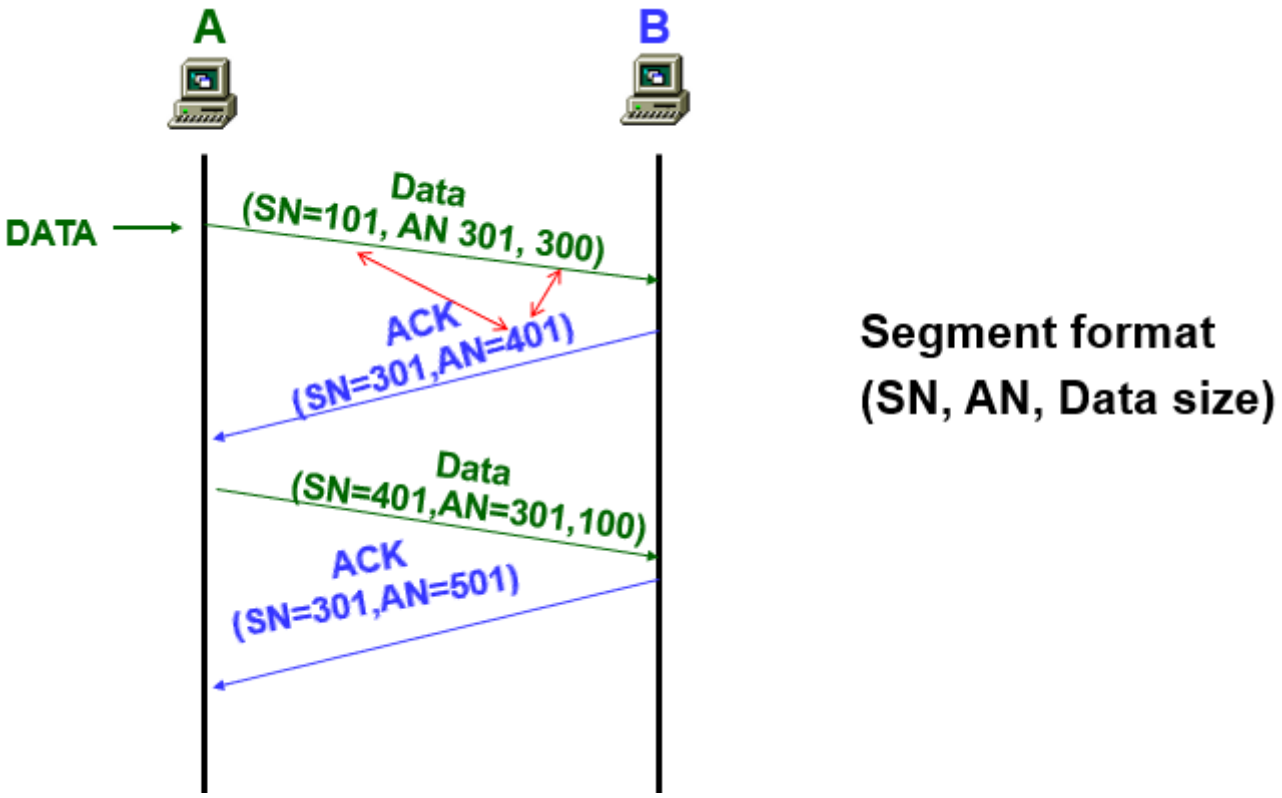

### Acknowledgement Number & Window Size

- AN is SN of next byte expected from sender and implies all bytes up to AN-1 has been received correctly, similar to ACK



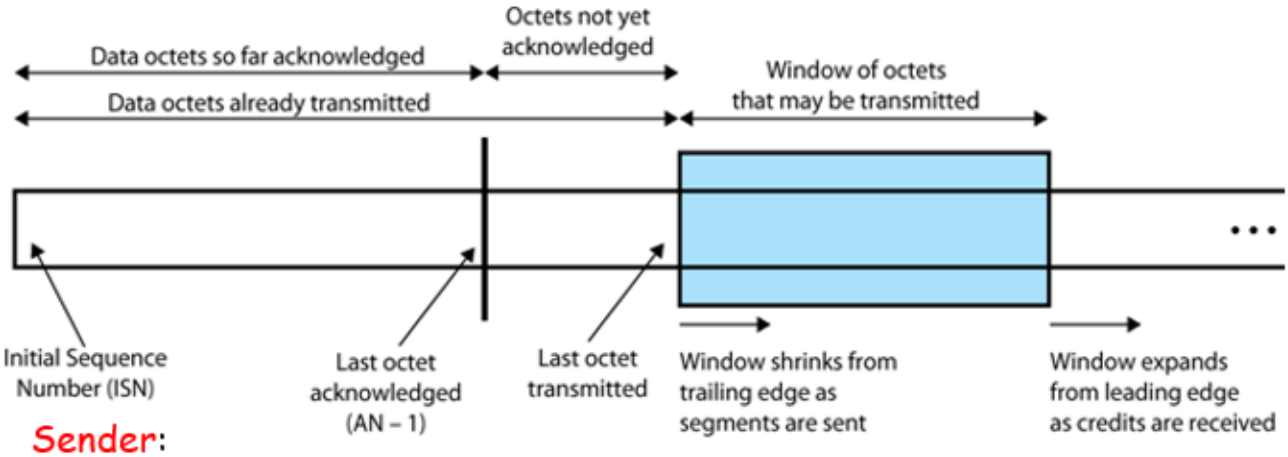

- W is the number of bytes counting from AN that the receiver is ready to accept

### Data transfer format




## 3.2. Flow Control

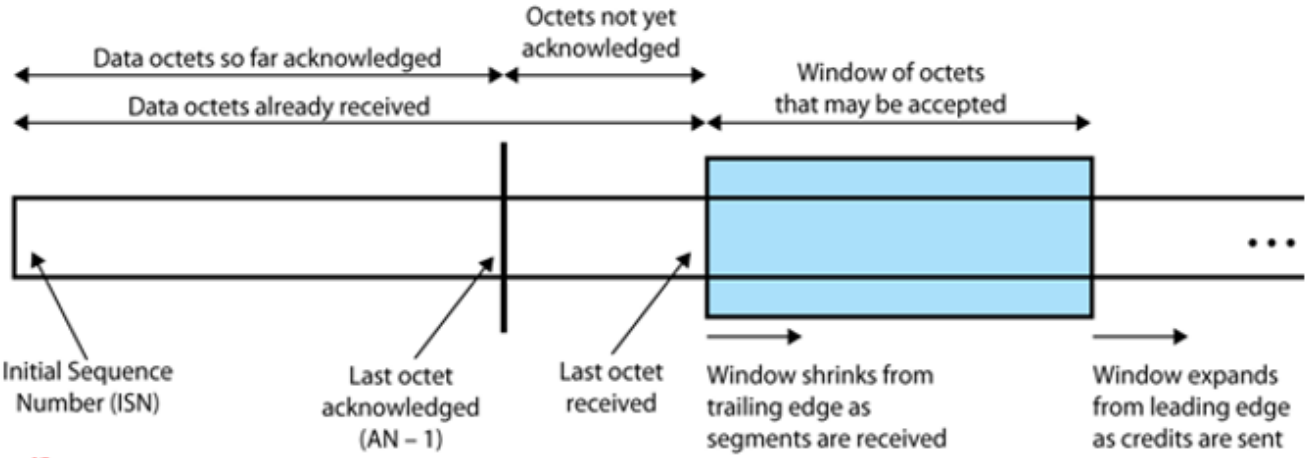
- Used to ensure sender won't overrun receiver's buffers by transmitting too much too fast, using a concept similar to sliding-window flow control in data link layer.
- Multiple segments are allowed to be in transit at the same time
- **Sender**




Sender:

- Maintain a blue window representing bytes that can be transmitted without ACK
- When segment is sent, shrink blue window from trailing edge
- Stop sending when blue window size = 0
- When ACK is received, new blue window size = W bytes starting from AN.

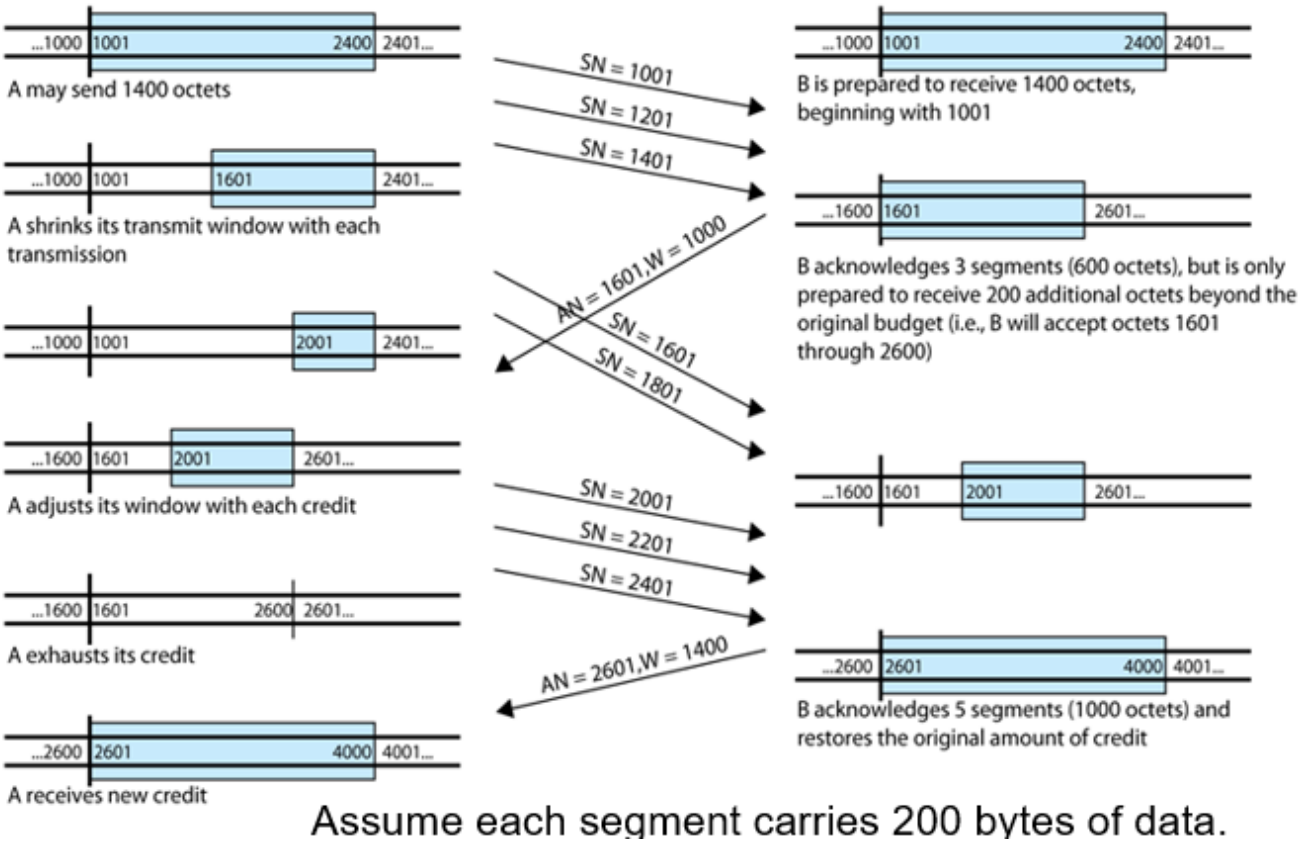
- **Receiver**




Receiver:

- Maintain a blue window representing bytes ready to accept
- When segment is received, shrink blue window from trailing edge
- If NOT ready to accept more segments, send ACK with credit W = remaining blue window size
- If ready to accept more segments, send ACK with W > remaining blue window size, and expand blue window from leading edge

- **Example**

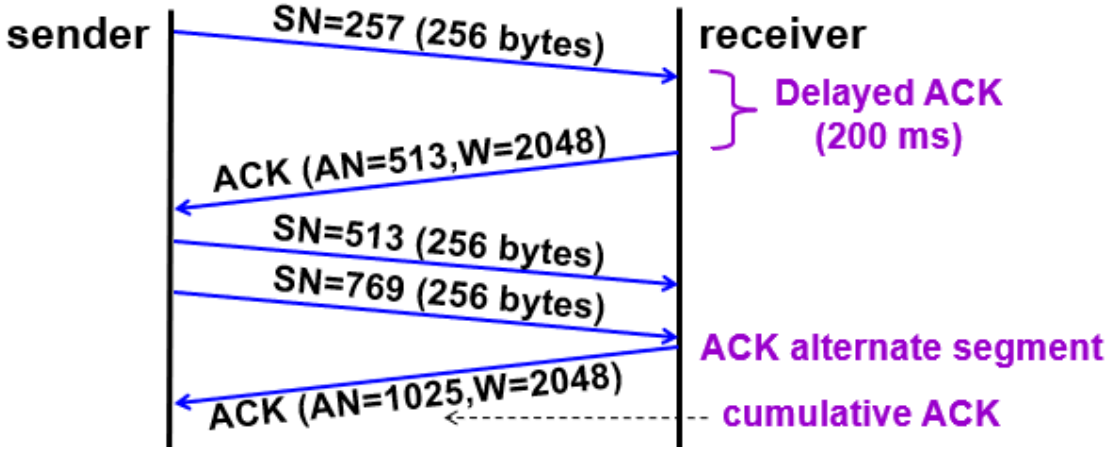



Assume each segment carries 200 bytes of data.

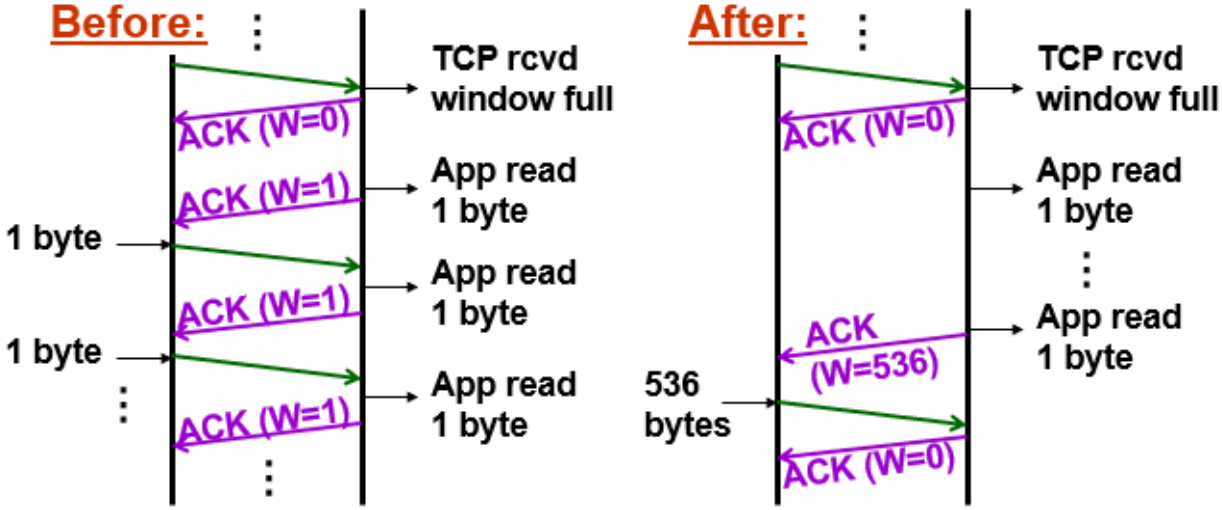
## Enhancement methods

- **Delayed ACK**
  - Wasteful to send ACK only segment (40 bytes TCP + IP headers)
  - Delayed ACK waits for a while before sending cumulative ACK for segments received

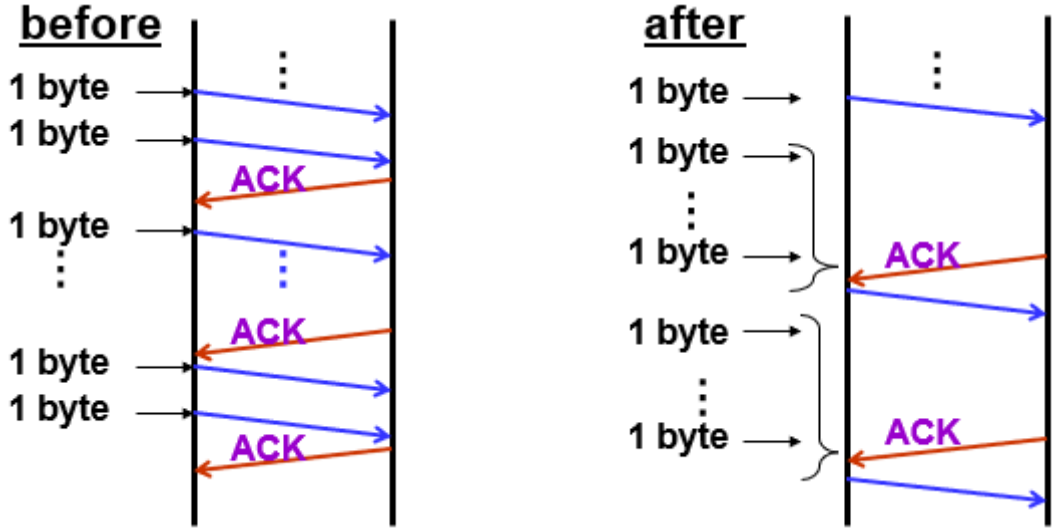
- Maximum < 500ms to avoid error-control timeout re-sent
- ACK every alternate segment received




- **Avoiding Silly Window Syndrome at Receiver**
  - Wasteful for receiver to keep ACK with small window when sender can send more
  - Receiver ACK W=0 instead of small window size until free buffer gets large




- **Avoiding Silly Window Syndrome at Sender**
  - Wasteful for sender to keep sending small segments when receiver can receive more
  - Send 1st small segment, buffer the rest and send them together when ACK is received



## 3.3 Error Control

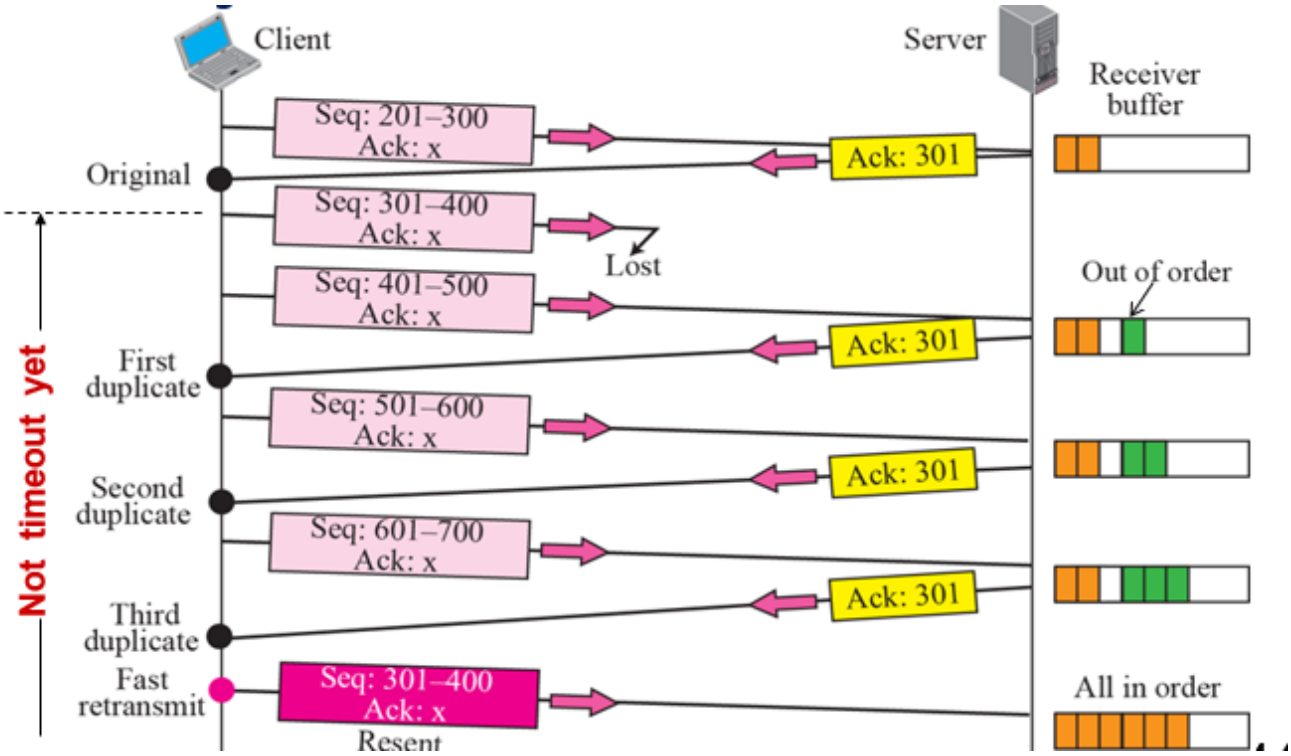
- Used to guarantee reliable service to application layer even when IP is unreliable, using a concept similar to Selective-Reject in data link layer
- **Error types**
  - Out-of order segments; detected based on SN in TCP header
    - Re-order and ACK
  - Duplicated segments; detected based on SN
    - Discard and ACK
  - Corrupted segments; detected based on checksum in TCP header
    - Discard, wait for timeout retransmission
  - Lost segments
    - Wait for timeout retransmission

### Retransmission timer

- Having too short of a timer may lead to unnecessary retransmission and having too long of a timer may lead to a long period time required to discover a lost segment
- Measure round trip time (RTT) and compute smoothed RTT which is then used to derive Retransmission Timeout (RTO)
- **Jacobson's algorithm**
  - Initialise RTO as 1s
  - Measure first RTT
    - SRTT(smoothed RTT) = RTT, RTTVar(variation) = RTT/2
    - RTO = SRTT + 4xRTTVar
  - After each subsequent RTT measured
    - $\mathrm{RTTVar} = (1-\beta)\mathrm{RTTVar} + \beta|\mathrm{SRTT} - \mathrm{RTT}|, \beta = \frac{1}{4}$
      $\mathrm{SRTT} = (1-\alpha)\mathrm{SRTT} + \alpha\mathrm{RTT}, \alpha = \frac{1}{8}$
      $\mathrm{RTO} = \mathrm{SRTT} + 4\mathrm{RTTVar}$
  - Minimum RTO is 1s, maximum at least 60s
- **Karn's algorithm** to measure RTT
  - Each TCP connection measures the RTT from sending a segment to receiving its corresponding ACK
  - Typically only one measurement at any point in time
  - If a segment is retransmitted due to timeout, ignore its measured RTT because it is ambiguous whether the ACK is for 1st or retransmission
  - When retransmission occurs, set RTO = 2xRTO

### Enhancement (Fast Retransmit)

- Fast retransmit if receiving 3 duplicate ACKs instead of waiting for timeout of lost segments




## 3.4 Congestion Control

- Used to prevent senders from sending too much traffic to the network such that it becomes overly congested and useless, done by implementing congestion control algorithm with a congestion window which controls the amount of traffic that a connection can send
- Congestion collapse if network congestion is ignored, transmissions are dominated by retransmissions.
- TCP assumes there is a congestion when LOSS events occurs
  - Timeout / receiving duplicate ACKs possibly due to queuing of buffer overflow at routers
- **TCP congestion control**
  - Phases: Slow start + Congestion avoidance
  - Parameters
    - `cwnd` (congestion window) - measured in number of segments, each with Maximum Segment Size
    - `ssthresh` (slow start threshold) - defines the point to transit from slow start to congestion avoidance phase
  - Maximum bytes that can be sent without ACK = min(Window size, `cwnd` x MSS)

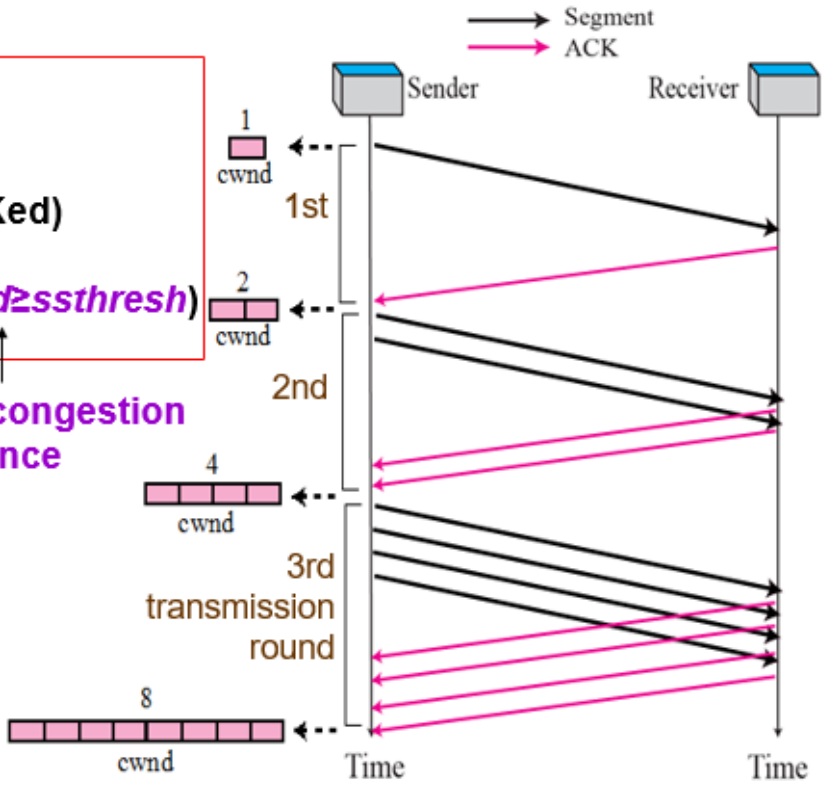
- **Slow Start Phase**

**Slow start**

initialize: *cwnd* = 1
do
  for (each segment ACKed)
    *cwnd*++
until (loss event OR *cwnd*≥*ssthresh*)

Go to recovery

Go to congestion avoidance



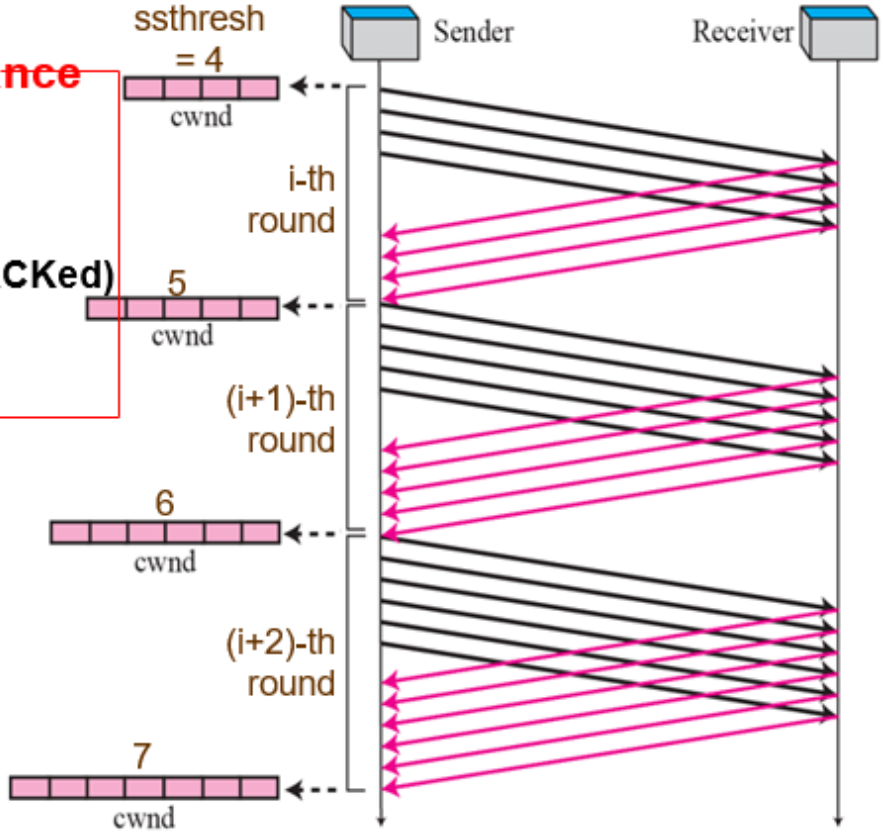

- ***cwnd*** increases exponentially at each transmission round (not so slow after all!)

- **Congestion Avoidance Phase**

**Congestion avoidance**

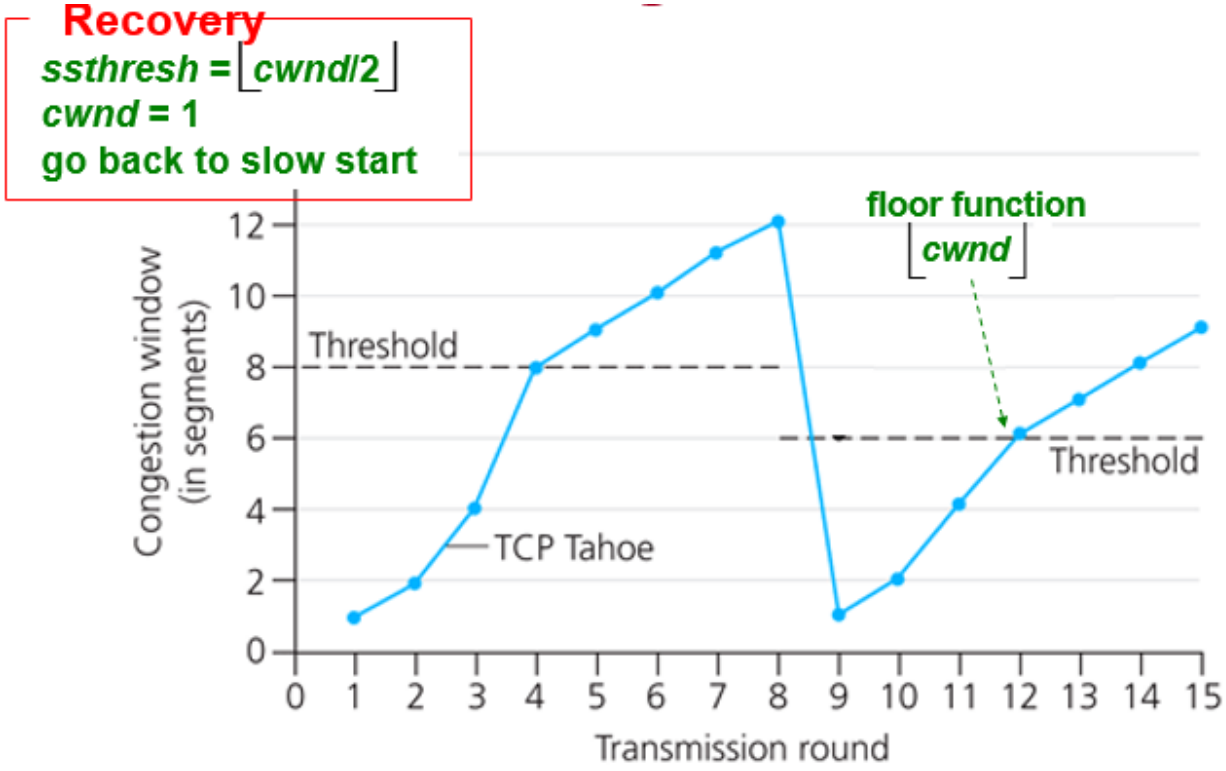
/* slow start is over */
/* *cwnd* ≥ ssthresh */
do
  for (every segment ACKed)
    *cwnd* += 1/⌊*cwnd*⌋
until (loss event)

Go to recovery

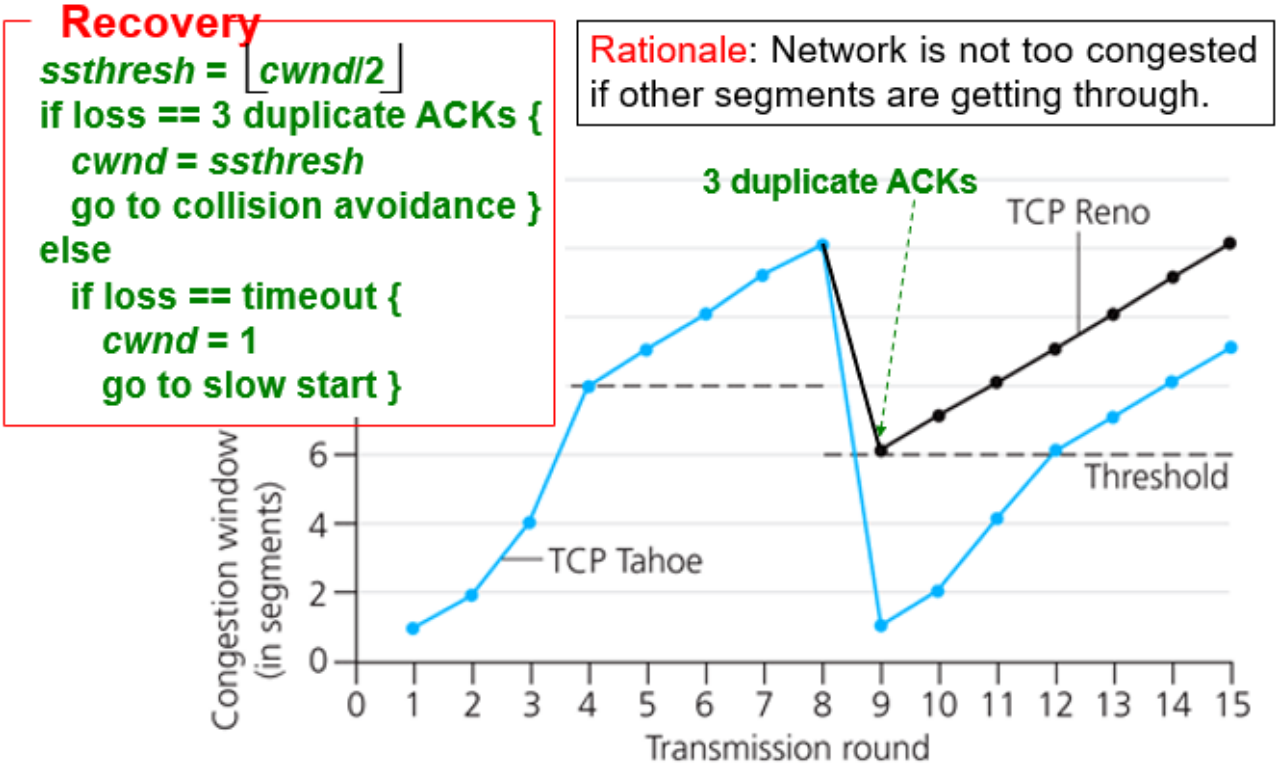



- In contrast, ***cwnd*** only increases linearly at each transmission round.

- **Tahoe algorithm**




- **Reno algorithm**




  - During slow start phase, congestion window is doubled at every transmission round until it reaches the threshold where congestion avoidance (linear increase) phase starts
  - (**blue line**) When congestion occurs (when no ACK is detected), all senders ideally cuts congestion window size to minimum, and threshold is moved to half the congestion window size when congestion started to occur in the previous cycle

- **TCP throughput**

- $(\text{cwnd} \times \text{MSS})/\text{RTT} = \text{Throughput}$, assuming no buffer constraint and $((\text{cwnd} \times \text{MSS} \times 8)/\text{Transmission rate}) <<< RTT$
- Throughput given in Mbps / Gbps (bits per second, so have to multiply by 8 if using bytes)
- If $\text{throughput}$ instead of $\text{cwnd}$ is given, $(\text{cwnd} \times \text{MSS})$ is a constant provided throughput is limiting factor, i.e. throughput given is not the maximum bandwidth of a link.
  - e.g.

**Table Q4**

| Round trip Time (RTT) milliseconds | Throughput (Gbps) |
|---|---|
| 10 | 1 |
| 100 | |
| 200 | 0.06144 |
| 350 | |

You have noted the following information

(i) Maximum Segment Size is 6000 Bytes
(ii) Maximum Transfer Unit of the server is set at 8000 Bytes.
(iii) Transmission bandwidth is 1 Gbps

  - Throuput for RTT = 10ms = 1Gbps (bandwidth) so do not use 10ms to find $(\text{cwnd} \times \text{MSS})$
- Recall RTT is minimum time taken, so if given a list, take the shortest RTT
- Recall MTU = MSS + IP header (20 Bytes) + TCP header (20 Bytes)

# 4. Questions

### Find transmission time to send data of size X

- Given link speed, window size W, X and RTT, find the bottleneck (comes from either transmission speed or RTT), by using W / link speed
  - If transmission time > RTT, link speed is the bottleneck, then find the time taken to transmit X over link speed
  - Otherwise, W is limiting factor, use W / RTT to find transmission rate and then use transmission rate to find time taken to transmit X over link speed

### Fill in time sequence diagram

(b) Figure Q4 shows the interaction between a Client and Server with TCP as the transport layer protocol.
Assume the following conditions:

(i) The Initial Sequence Number (ISN) for the Client TCP is 200, and the ISN for the Server TCP is 600;
(ii) Both receiving window sizes of the Client and Server are fixed to 2048 bytes;
(iii) The TCP uses slow start for congestion control. The Maximum Segment Size is 256 bytes. The initial congestion window is 1 MSS;
(iv) The communication link is ideal, so no error retransmissions;
(v) Delayed ACK is implemented for the TCP, and one ACK must be sent for every two received data packets.

Please complete the time sequence diagram by filling in the TCP control and data packet transmissions. Indicate clearly the Sequence Number (SN) and the Acknowledgement Number (AN) in each packet. Include SYN or FIN flags if they are set.

(15 marks)

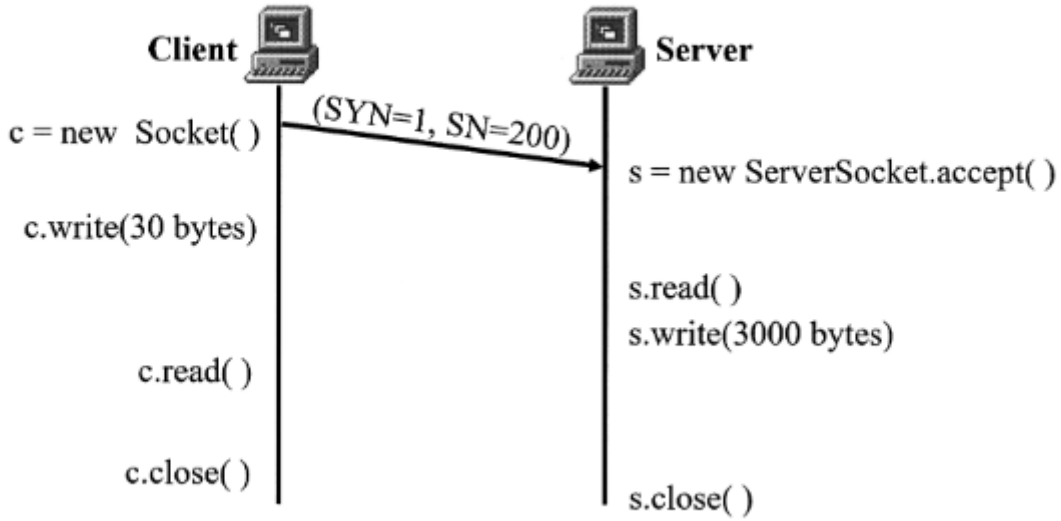



**Figure Q4**

- If there is nothing to be sent, SN stays the same
- ACK=1 every time Client / Server wants to ACK
- AN to request for next segment
- W to indicate window size
- Take note of
  - How many partitions can be sent until sender reads first ACK (Window size)
  - How many times must data be partitioned (MSS)

- How often ACKs need to be sent