Transport Layer - III

Transmission Control Protocol

Transmission Control Protocol (TCP)

- TCP was specifically designed to provide a reliable, end-to-end byte stream over an unreliable **internetwork**.

- **Internetwork** – different parts may have widely different topologies, bandwidths, delays, packet sizes and other parameters

- **TCP** dynamically adapts to properties of the internetwork and is robust in the face of many kinds of failures.

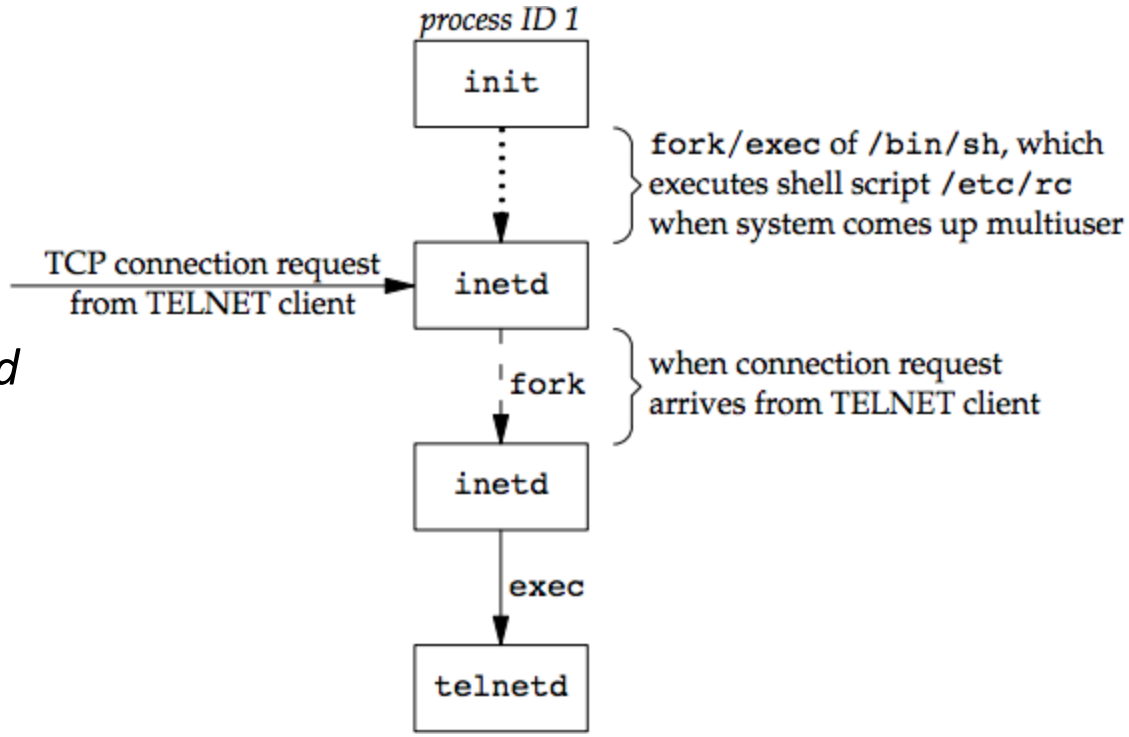
- RFC 793 (September 1981) – Base protocol
  - RFC 1122 (clarifications and bug fixes), RFC 1323 (High performance), RFC 2018 (SACK), RFC 2581 (Congestion Control), RFC 3168 (Explicit Congestion notification)

TCP Service Model

Uses **Sockets** to define an end-to-end connection (Source IP, Source Port, Source Initial Sequence Number, Destination IP, Destination Port, Destination Initial Sequence Number)

Unix Model of Socket Implementation:

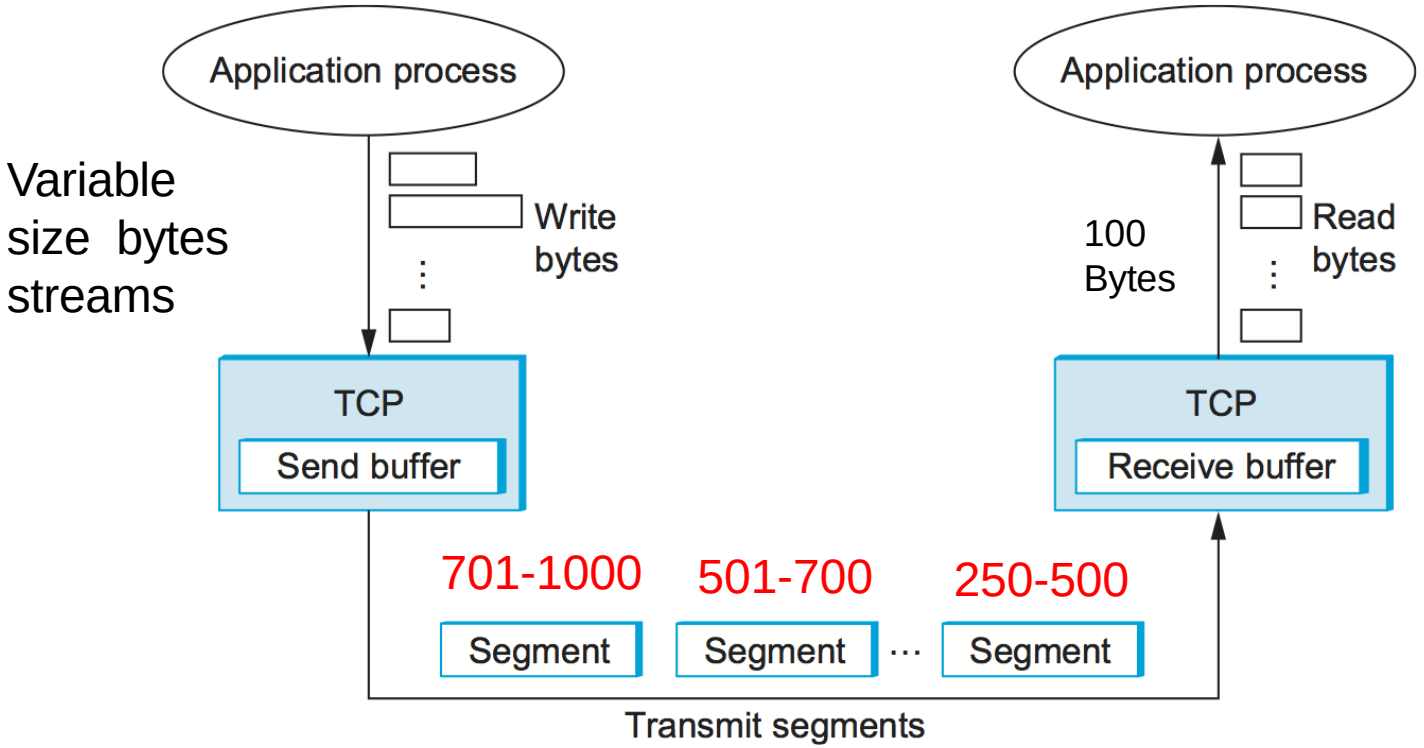
- A single daemon process, called **Internet Daemon (inetd)** runs all the times at different **well known ports**, and wait for the first incoming connection
- When a first incoming connection comes, *inetd* forks a new process and starts the corresponding daemon (for example *httpd* at port 80, *ftpd* at port 21 etc.)



All TCP connections are full-duplex and point-to-point. TCP does not support multicasting or broadcasting

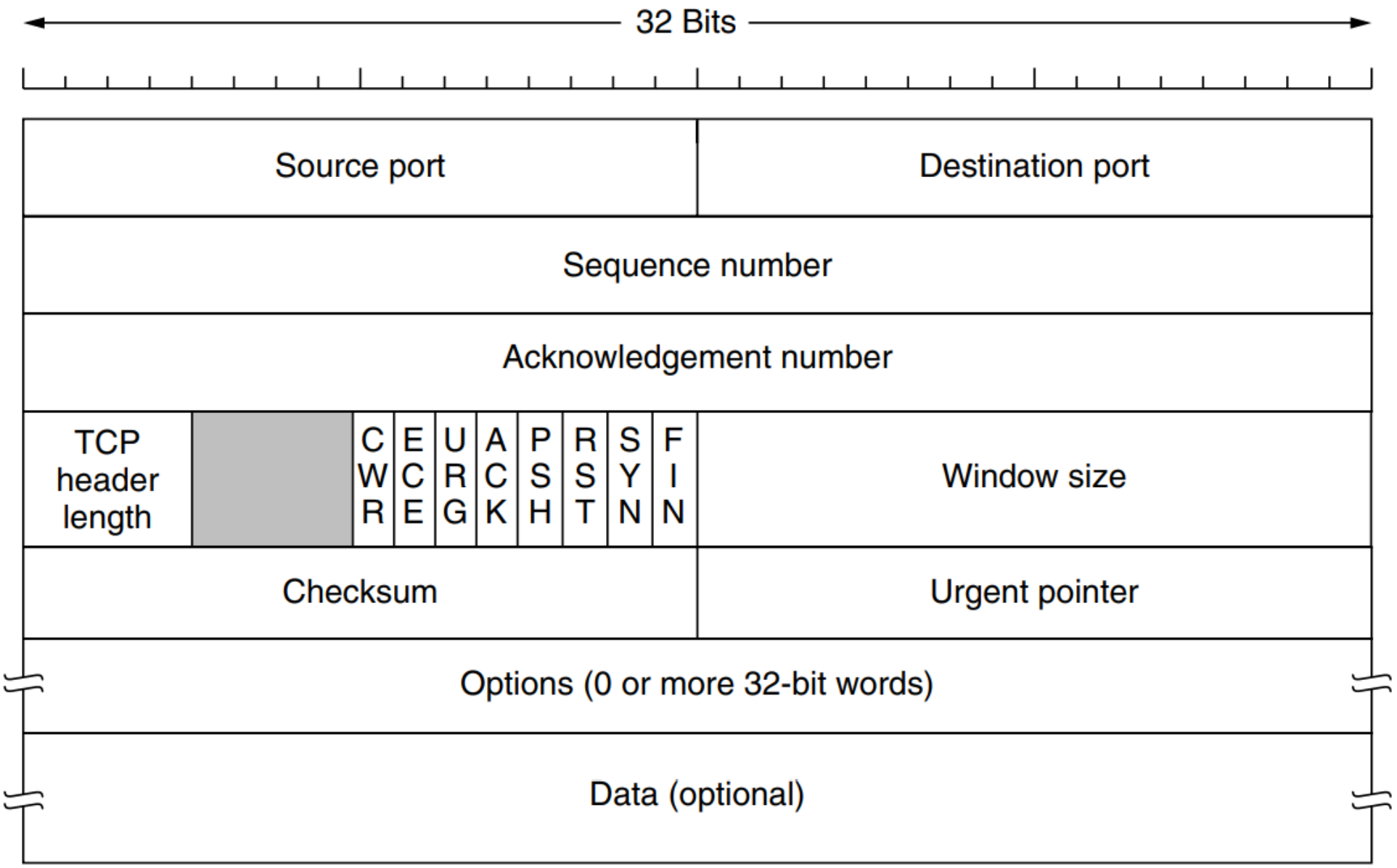
TCP Service Model

- A TCP connection is a **byte stream**, not a message stream
- Message boundaries are *not preserved end-to-end*



There is no way for the receiver to detect the unit(s) in which the data were written by the sending process.

The TCP Protocol- The Header



TCP Header

- Every segment begins with a fixed-format, 20-byte header.
- The fixed header may be followed by header options.
- After the options, if any, up to 65,535 – 20 – 20 = 65,495 data bytes may follow, where the first 20 refer to the IP header and the second to the TCP header.
- Segments without any data are legal and are commonly used for acknowledgements and control messages.

TCP Header: Port Numbers

- The Source port and Destination port fields identify the local end points of the connection.
- A TCP port plus its host's IP address forms a 48-bit unique end point.
- The source and destination end points together identify the connection.
- This connection identifier is called a 5 tuple because it consists of five pieces of information: the protocol (TCP), source IP and source port, and destination IP and destination port.

TCP Sequence Number and Acknowledgement Number

- 32 bits sequence number and acknowledgement number
- Every byte on a TCP connection has its own 32 bit sequence number – a **byte stream** oriented connection
- TCP uses sliding window based flow control – the acknowledgement number contains next expected byte in order, which acknowledges the **cumulative bytes** that has been received by the receiver.
  - ACK number 31245 means that the receiver has correctly received up to 31244 bytes and expecting for byte 31245

TCP Segments

- The sending and receiving TCP entities exchange data in the form of **segments**.
- A TCP segment consists of a fixed 20 byte header (plus an optional part) followed by zero or more data bytes.
- TCP can accumulate data from several `write()` calls into one segment, or split data from one `write()` into multiple segments
- A segment size is restricted by two parameters
  - IP Payload (65515 bytes)
  - Maximum Transmission Unit (MTU) of the link

Other fields in TCP Header

- The TCP header length tells how many 32-bit words are contained in the TCP header.
- This information is needed because the Options field is of variable length, so the header is, too.
- CWR and ECE are used to signal congestion when ECN (Explicit Congestion Notification) is used, as specified in RFC 3168.
- ECE is set to signal an ECN-Echo to a TCP sender to tell it to slow down when the TCP receiver gets a congestion indication from the network.
- CWR is set to signal Congestion Window Reduced from the TCP sender to the TCP receiver so that it knows the sender has slowed down and can stop sending the ECN-Echo.

Other fields in TCP Header

- URG is set to 1 if the Urgent pointer is in use.
- The Urgent pointer is used to indicate a byte offset from the current sequence number at which urgent data are to be found.
- The ACK bit is set to 1 to indicate that the Acknowledgement number is valid. This is the case for nearly all packets. If ACK is 0, the segment does not contain an acknowledgement, so the Acknowledgement number field is ignored.
- The PSH bit indicates PUSHed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency).
- The SYN bit is used to establish connections. The connection request has SYN = 1 and ACK = 0 to indicate that the piggyback acknowledgement field is not in use. The connection reply does bear an acknowledgement, however, so it has SYN = 1 and ACK = 1.

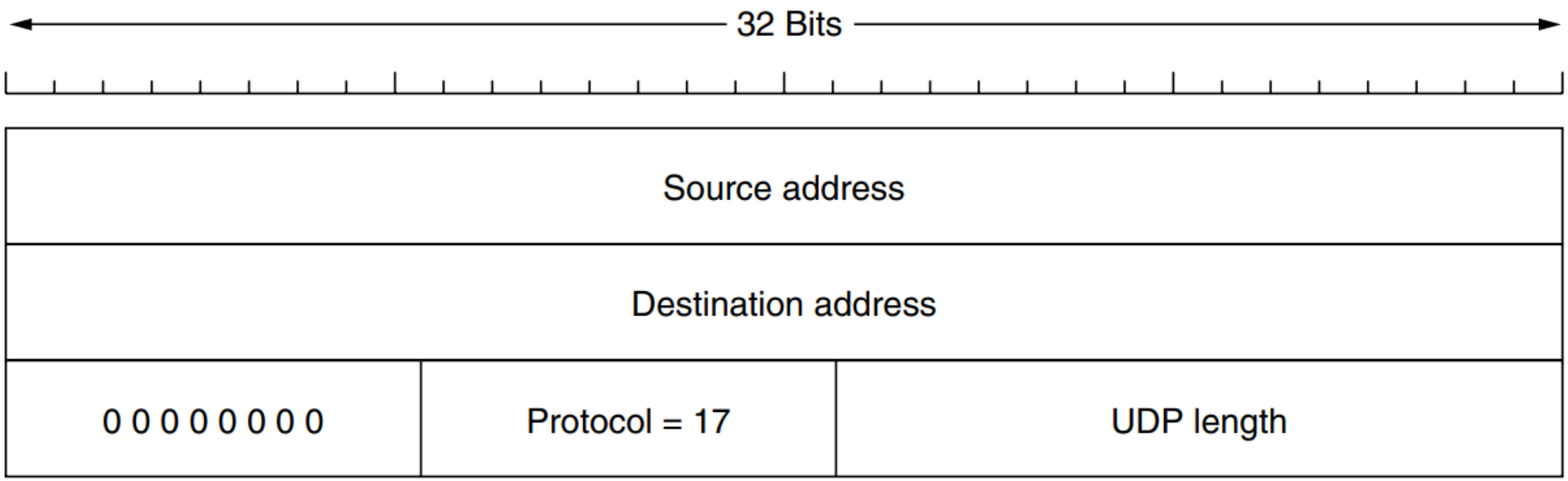
TCP Window size

- Flow control in TCP is handled using a variable-sized sliding window.
- The Window size field tells how many bytes may be sent starting at the byte acknowledged.
- A Window size field of 0 is legal and says that the bytes up to and including Acknowledgement number − 1 have been received, but that the receiver has not had a chance to consume the data and would like no more data for the moment.
- The receiver can later grant permission to send by transmitting a segment with the same Acknowledgement number and a nonzero Window size field.

TCP Checksum

- A Checksum is also provided for extra reliability.

- It checksums the header, the data, and a conceptual pseudo header with the protocol number for TCP (6) and the checksum is mandatory.



IPV4 Pseudoheader included in UDP checksum

TCP Header Options

- Option to negotiate ***MSS***, *window scale* option to scale window size.
- The ***timestamp option*** carries a timestamp sent by the sender and echoed by the receiver.
- It is included in every packet, and used to compute round-trip time samples that are used to estimate when a packet has been lost.
- On a fast connection, the sequence number may wrap around quickly, leading to possible confusion between old and new data.
- The ***PAWS (Protection Against Wrapped Sequence numbers)*** scheme discards arriving segments with old timestamps to prevent this problem.
- Finally, the ***SACK (Selective ACKnowledgement)*** option lets a receiver tell a sender the ranges of sequence numbers that it has received.

How a TCP Segment is Created

- Write() calls from the applications write data to the TCP sender buffer.
- Sender maintains a dynamic window size based on the flow and congestion control algorithm

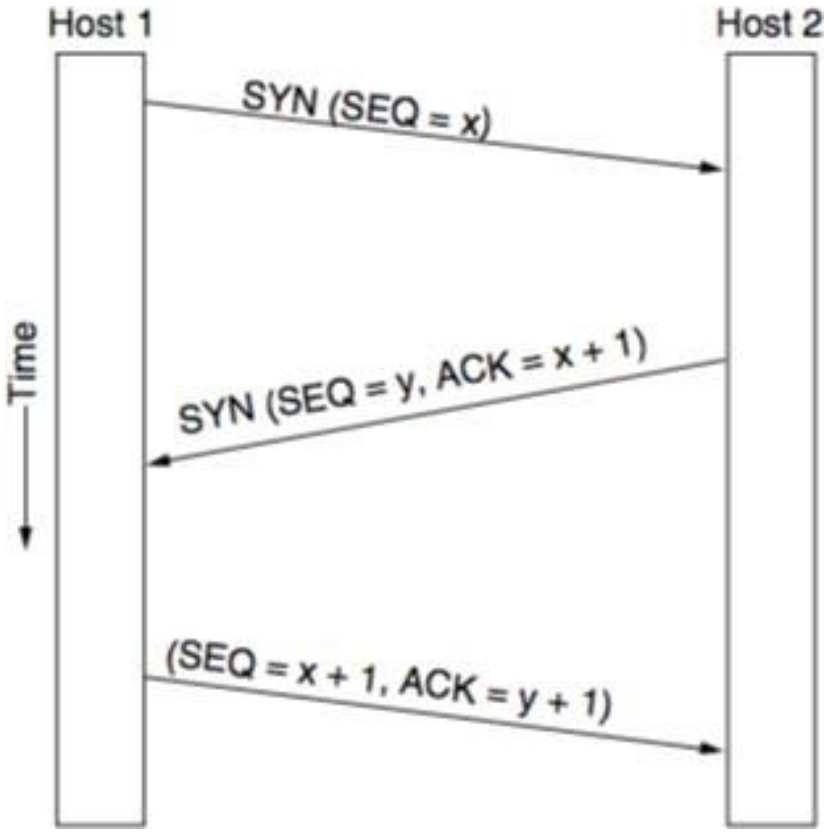
- Modern implementations of TCP uses **path MTU discovery** to determine the MTU of the end-to-end path (uses ICMP protocol), and sets up the **Maximum Segment Size** (**MSS**) during connection establishment
  - May depend on other parameters (buffer implementation).

Challenges in TCP Design

- Segments are constructed dynamically, so retransmissions do not guarantee the retransmission of the same data segment – a retransmission may contain additional data or less data

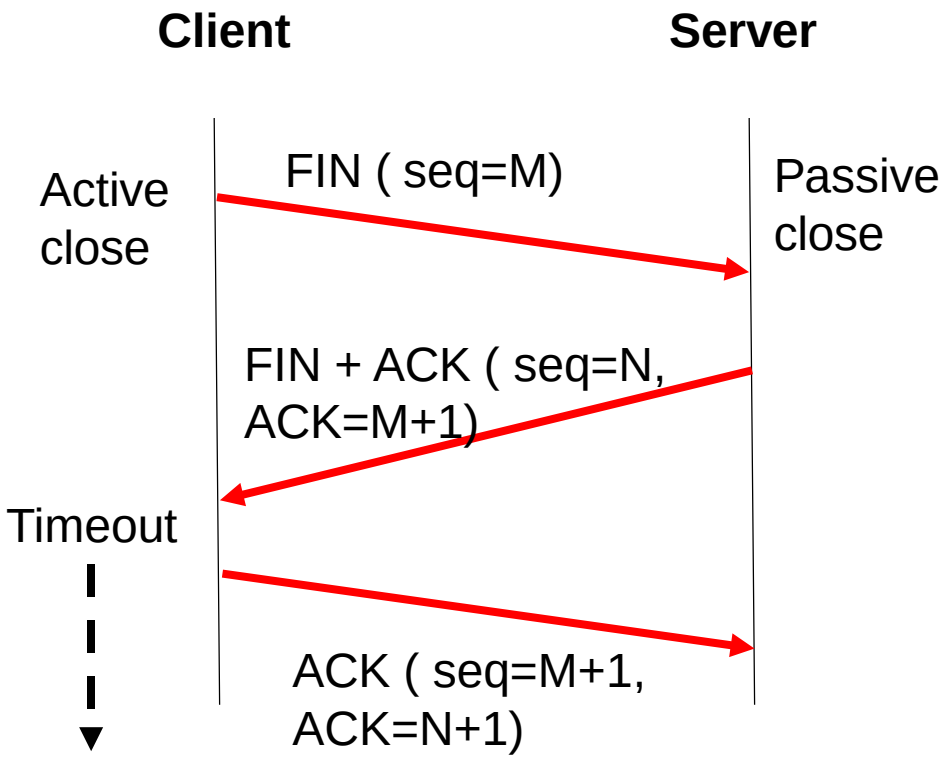
- Segments may arrive out-of-order. TCP receiver should handle out-of- order segments in a proper way, so that data wastage is minimized.

TCP Connection Establishment

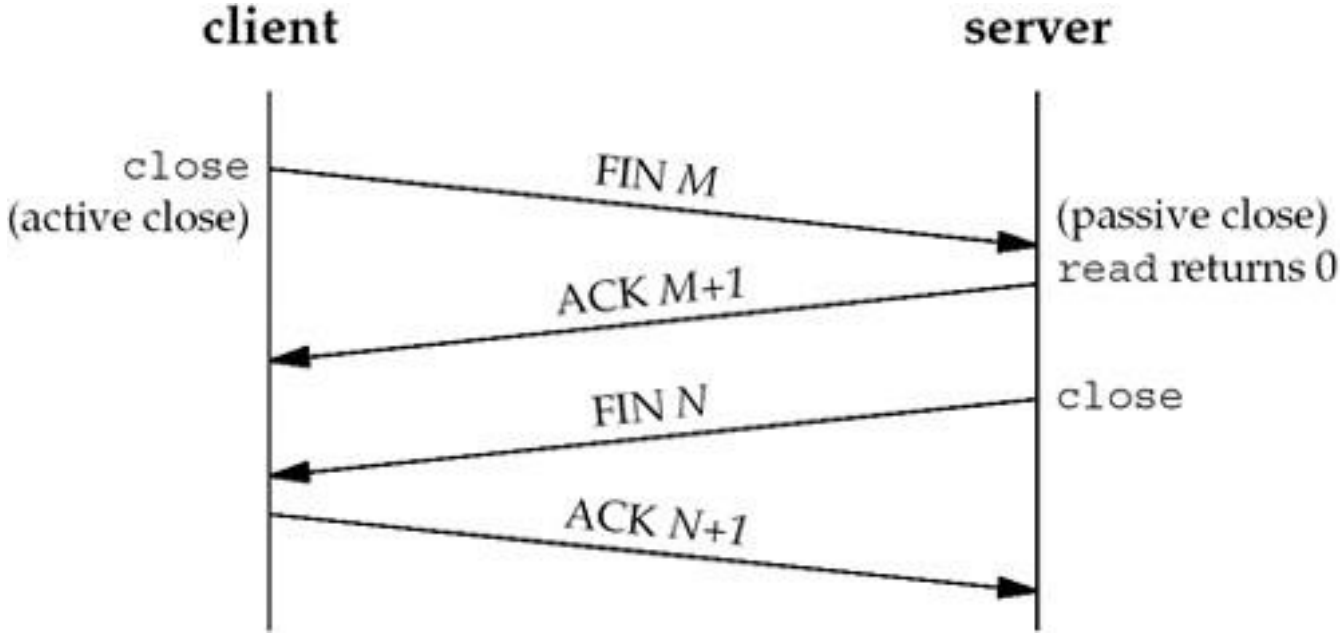


- **How to choose the initial sequence number?**
  - Protect delayed duplicates, generate the randomly chosen ISN for every connection.
  - Original implementation of TCP used a clock based approach, the clock ticked every 4 microseconds, the value of the clock cycles from 0 to 2^{32}-1. The value of the clock gives the initial sequence number
- **TCP SYN flood attack**
  - Solution: Use cryptographic function to generate sequence numbers

TCP Connection Release

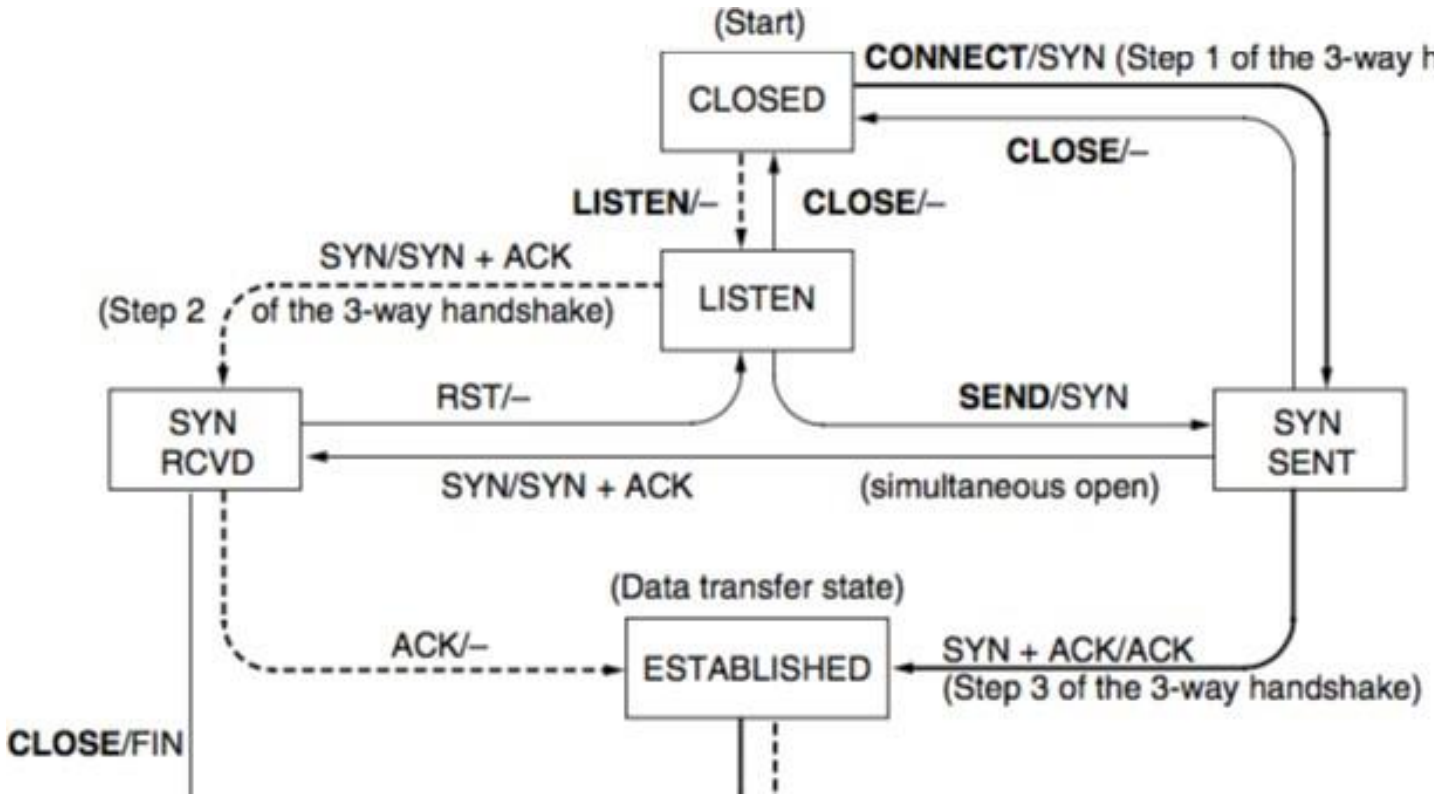


Case - I



Case - II

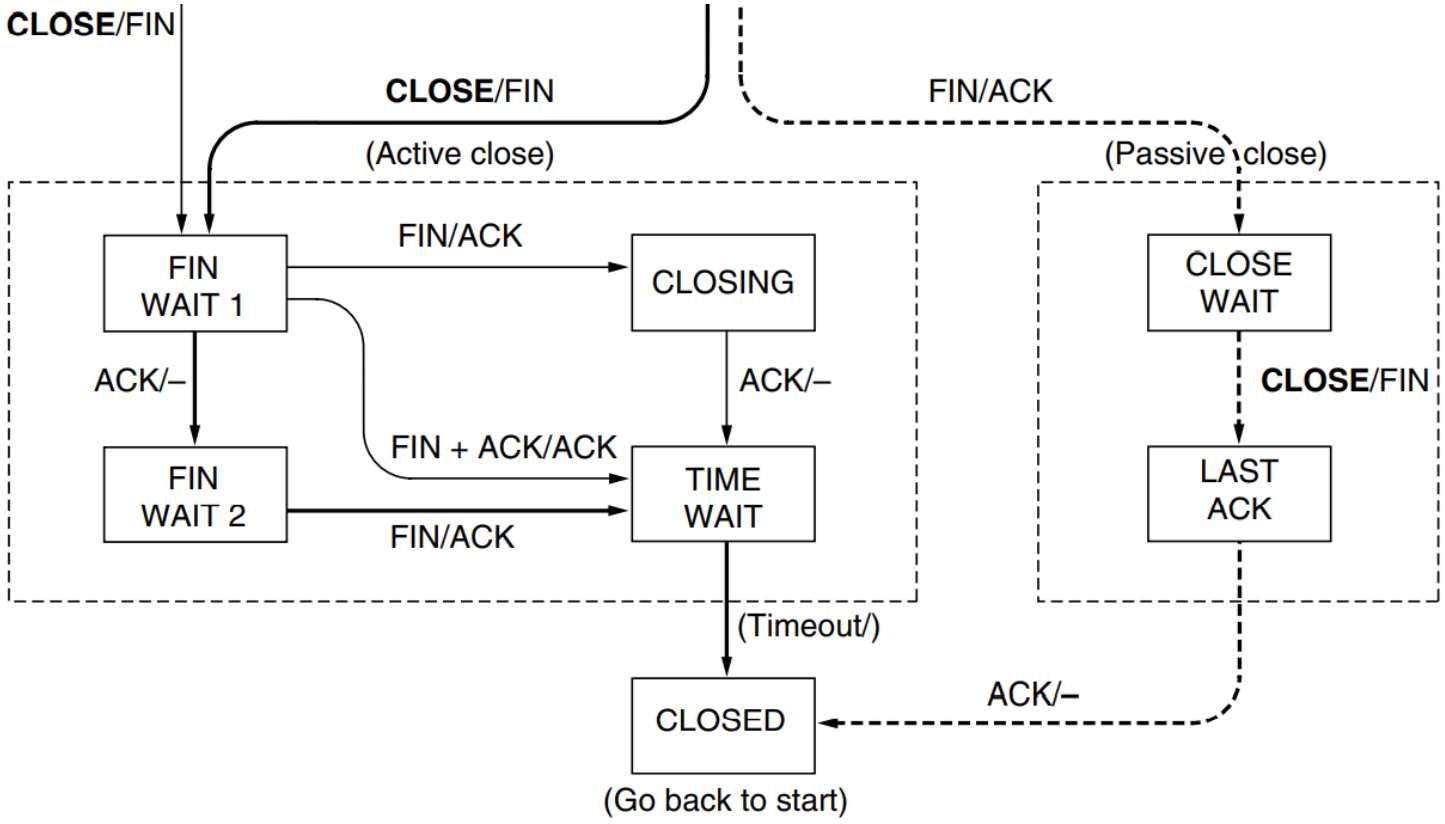
TCP State Transition Diagram – Connection Modelling



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

Event/Action
---- Server
____Client

TCP State Transition Diagram – Connection Modelling



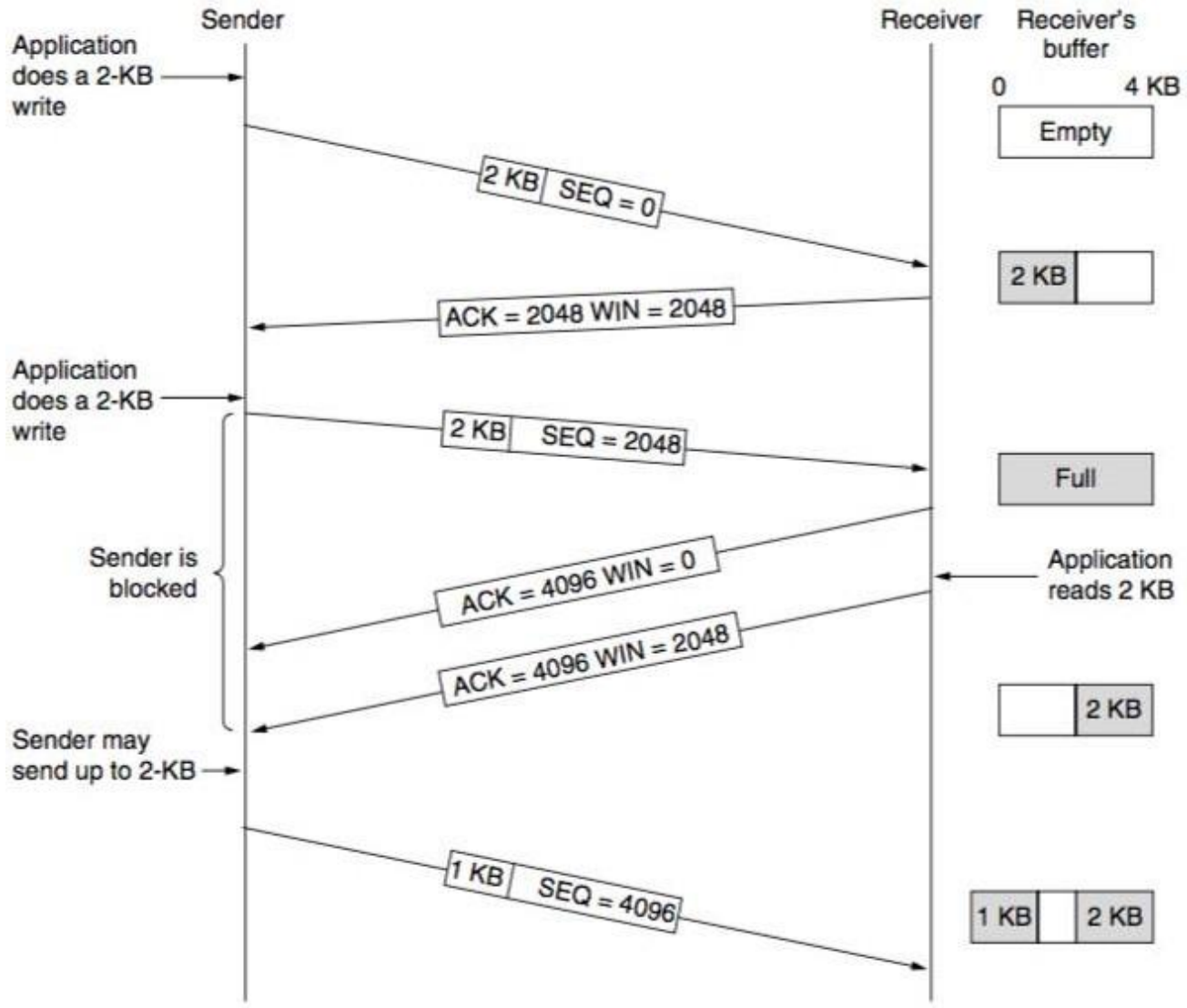
State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said 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

Event/Action
---- Server
____Client

TCP Sliding Window

Flow control rationale:

The receiver should advertise the available buffer space (receiver window size) to the sender so that to restrict it from sending too much data beyond the receiver's capacity.



TCP Sliding Window

- When the window is 0, the sender may not normally send segments, with 2 exceptions.
  - First, urgent data may be sent, for example, to allow the user to kill the process running on the remote machine.
  - Second, the sender may send a 1-byte segment to force the receiver to reannounce the next byte expected and the window size. This packet is called a window probe.

- Senders are not required to transmit data as soon as they come in from the application.

- Neither are receivers required to send acknowledgements as soon as possible.

Delayed Acknowledgments

- Consider a telnet connection, that reacts on every keystroke.

- In the worst case, whenever a character arrives at the sending TCP entity, TCP creates a 21 byte TCP segment, 20 bytes of header and 1 byte of data. For this segment, another ACK along with window update is sent when the application reads that 1 byte. This results in a huge wastage of bandwidth.

- **Delayed acknowledgements:** Delay acknowledgement and window updates for up to 500 msec in the hope of receiving few more data packets within that interval.

However, the sender can still send multiple short data segments**. Inefficient!!**

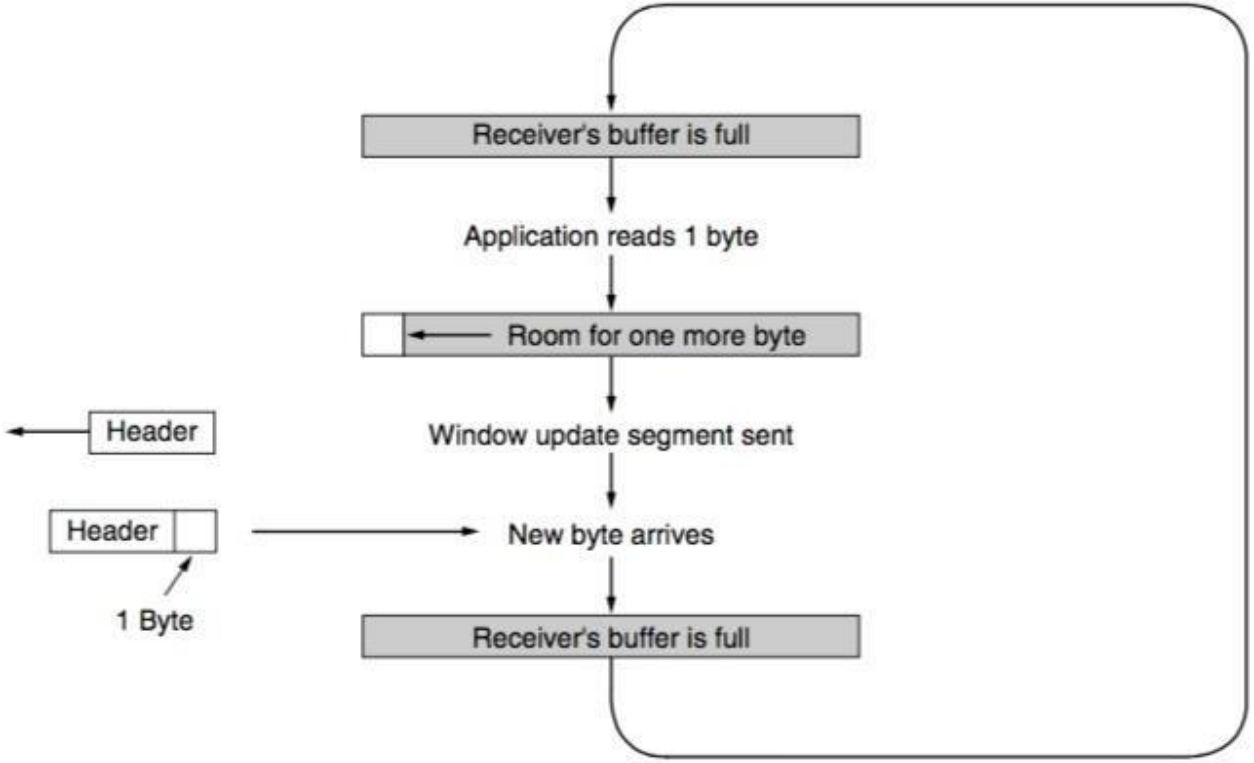
Nagle's Algorithm

- When data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged.
- Then send all buffered data in one TCP segment and start buffering again until the next segment is acknowledged.
  - **Only one short packet can be outstanding at any time.**
- **Do we want Nagle's Algorithm all the time?**
- **Nagle's Algorithm and Delayed Acknowledgement**
  - Receiver waits for data and sender waits for acknowledgement – results in starvation

Silly Window Syndrome

- Data are passed to the sending TCP entity in large blocks, but an interactive application on the receiver side reads data only 1 byte at a time.

Clark's solution: Do not send window update for 1 byte. Wait until sufficient space is available at the receiver buffer.



Handling Short Segments

- Nagle's algorithm and Clark's solution to silly window syndrome are **complementary**
- **Nagle's algorithm:** Solves the problem caused by the sending application delivering data to TCP a byte at a time
- **Clark's solution:** Receiving application fetching the data up from TCP a byte at a time

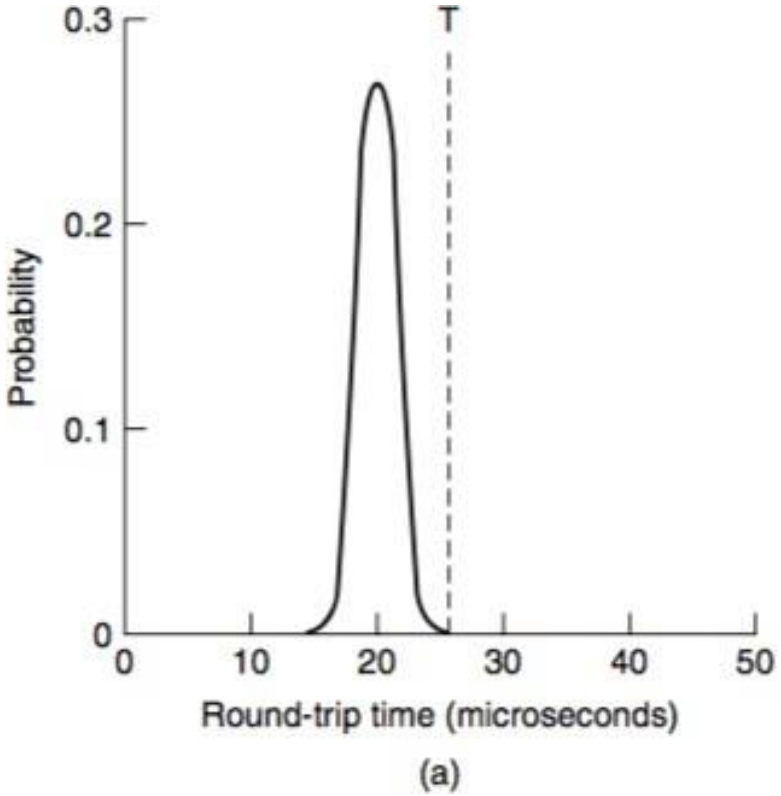
Handling Out of Order in TCP

- TCP buffers out of order segments and forward a duplicate acknowledgement to the sender.
- **Acknowledgement in TCP – Cumulative acknowledgement**
- Receiver has received bytes 0, 1, 2, _, 4, 5, 6, 7
    - TCP sends a cumulative acknowledgement with ACK number 3, acknowledging everything up to byte 2
    - Once 4 is received, a duplicate ACK with ACK number 3 (next expected byte) is forwarded
    - After timeout, sender retransmits byte 3
    - Once byte 3 is received, it can send another cumulative ACK with ACK number 8 (next expected byte)

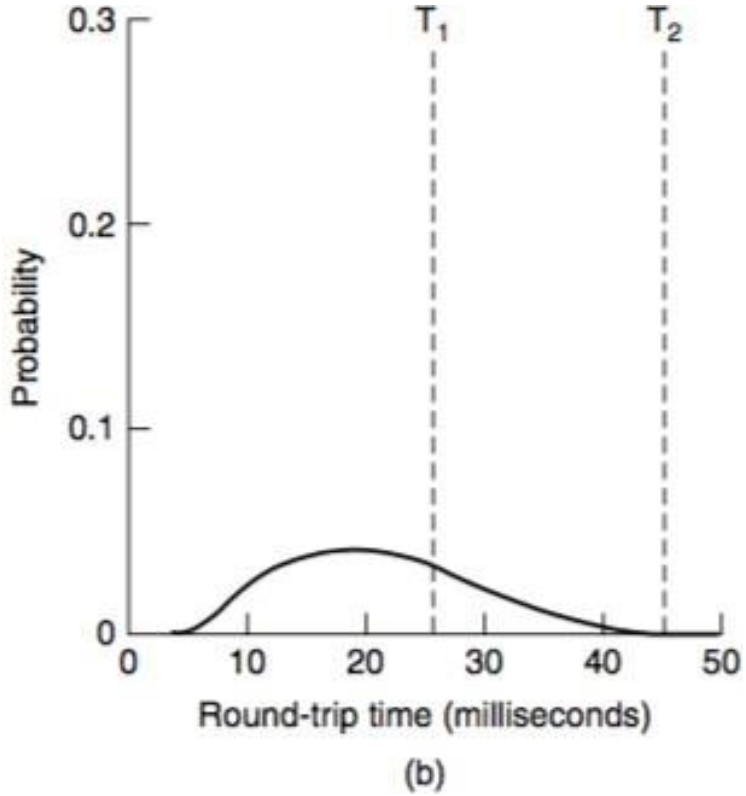
TCP Timer Management

- **TCP Retransmission Timeout (RTO)**: When a segment is sent, a retransmission timer is started
  - **If the segment is acknowledged before the timer expires, the timer is stopped**
  - **If the timer expires before the acknowledgement comes, the segment is retransmitted**
- **What can be an ideal value of RTO ?**
- **Possible solution**: Estimate RTT, and RTO is some positive multiples of RTT
- **RTT estimation is difficult for transport layer – why?**

RTT at Data Link Layer vs. RTT at Transport Layer



Data Link Layer



Transport Layer

RTT Estimation at Transport Layer

- Use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance.
- **Jacobson's algorithm (1988) - used in TCP**
  - For each connection, TCP maintains a variable, ***SRTT (smoothed Round Trip Time)*** – best current estimate of the round trip time to the destination
  - When a segment is sent, a timer is started (both to see how long the acknowledgement takes and also to trigger a retransmission if it takes too long)
  - If the ACK gets back – measure the time (say, *R*)
  - Update SRTT as follows

$$SRTT = \alpha\, SRTT + (1 - \alpha)\, R$$ **(Exponentially Weighted Moving Average – EWMA)**

α is a smoothing factor that determines how quickly the old values are forgotten. Typically $\alpha = 7/8$

Problem with EWMA

- Even given a good value of SRTT, choosing a suitable RTO is non-trivial.
- Initial implementation of TCP used RTO = 2SRTT
- Experience showed that a constant value was too inflexible, because it failed to response when the **variance went up (RTT fluctuation is high) – happens normally at high load**
- **Consider variance of RTT during RTO estimation.**

RTO Estimation

Update RTT variation (*RTTVAR*) as follows.

$$\boldsymbol{RTTVAR = \beta\ RTTVAR + (1 - \beta)|SRTT - R|}$$

Typically β = ¾

RTO is estimated as follows,

$$\boldsymbol{RTO = SRTT + 4 \times RTTVAR}$$

- **Why 4 ?**
  - **Somehow arbitrary**
  - **Jacobson's paper is full of clever tricks – use integer addition, subtraction and shift – computation is lightweight**

Kam's Algorithm

- One problem that occurs with gathering the samples, R, of the round-trip time is what to do when a segment times out and is sent again.
- When the acknowledgement comes in, it is unclear whether the acknowledgement refers to the first transmission or a later one. Guessing wrong can seriously contaminate the RTO.

- **Karn's algorithm:**
  - Do not update estimates on any segments that has been retransmitted
  - The timeout is doubled each successive retransmission until the segments gets through the first time

Other TCP Timers

- **Persistent TCP Timer:** Avoid deadlock when receiver buffer is announced as zero
  - **After the timer goes off, sender forwards a probe packet to the receiver to get the updated window size**
- **Keepalive Timer:** Close the connection when a connection has been idle for a long duration
- **TCP TIME_WAIT:** It runs for twice the maximum packet lifetime to make sure that when a connection is closed, all packets created by it have died off.

TCP Congestion Control

- Based on implementation of AIMD using a window and with packet loss as the binary signal
- TCP maintains a **Congestion Window (CWnd) –** number of bytes the sender may have in the network at any time
- **Sending Rate = Congestion Window / RTT**
- Congestion window is maintained in addition to the flow control window, which specifies the number of bytes that the receiver can buffer.
- **Sender Window (SWnd) = Min (CWnd, RWnd)**
- RWnd – Receiver advertised window size

TCP Congestion Control: What this means?

- If the receiver says ‘‘send 64 KB’’ but the sender knows that bursts of more than 32 KB clog the network, it will send 32 KB.

- On the other hand, if the receiver says ‘‘send 64 KB’’ and the sender knows that bursts of up to 128 KB get through effortlessly, it will send the full 64 KB requested.

1986 Congestion Collapse

- In 1986, the growing popularity of Internet led to the first occurrence of congestion collapse – a prolonged period during which goodput dropped precipitously (more than a factor of 100)

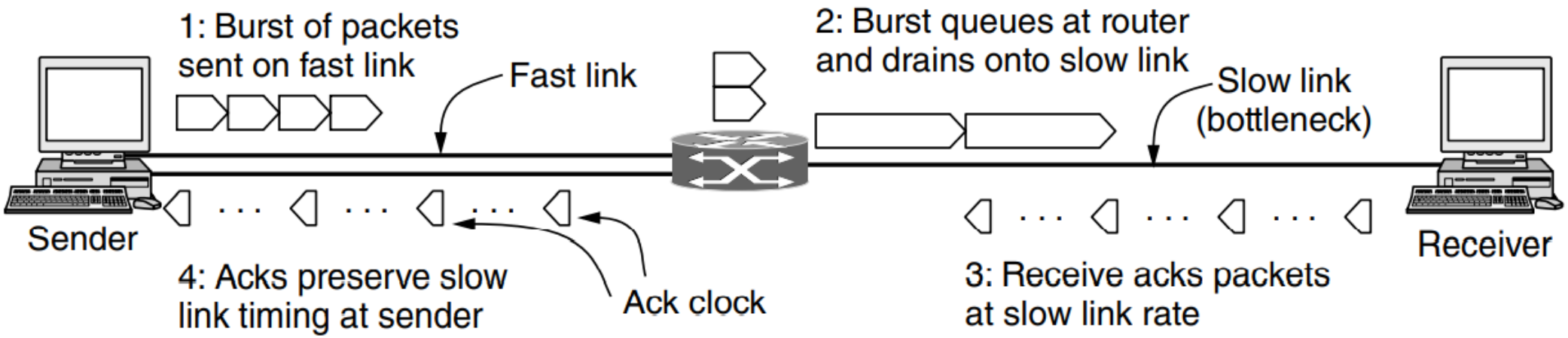
- Early TCP Congestion Control algorithm – Effort by Van Jacobson (1988)

- **Challenges for Jacobson** – Implement congestion control without making much change in the protocol (made it instantly deployable)

- **Packet loss is a suitable signal for congestion – use timeout to detect packet loss. Tune CWnd based on the observation from packet loss**

Adjust CWnd based on AIMD

- **One of the most interesting ideas – use ACK for clocking**



- **ACK returns to the sender at about the rate that packets can be sent over the slowest link in the path.**
- **Trigger CWnd adjustment based on the rate at which ACK are received.**

Increase Rate Exponentially at the Beginning - The Slow Start

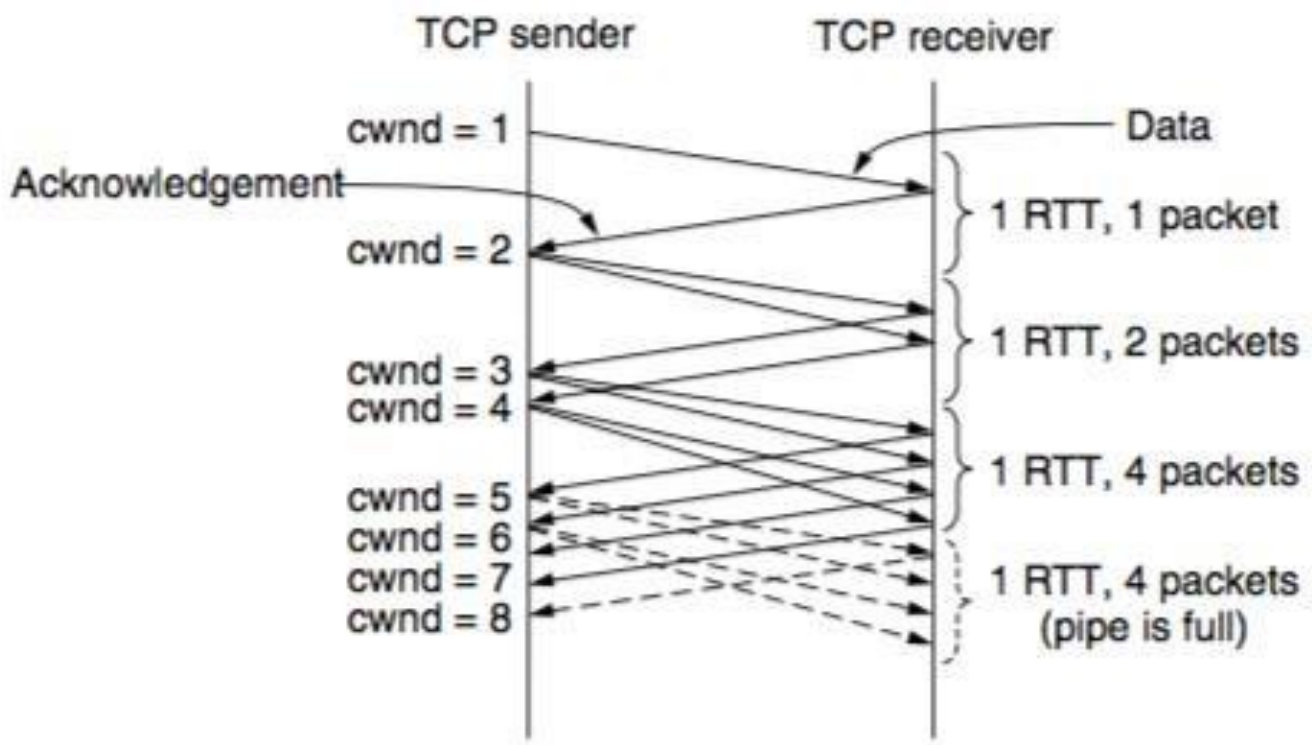
- AIMD rule will take a very long time to reach a good operating point on fast networks if the CWnd is started from a small size.

- A 10 Mbps link with 100 ms RTT
  - Appropriate CWnd = BDP = 1 Mbit
  - 1250 byte packets -> 100 packets to reach BDP
  - CWnd starts at 1 packet, and increased 1 packet at every RTT
  - 100 RTTs are required 10 sec before the connection reaches to a moderate rate

- **Slow Start - Exponential increase of rate to avoid slow convergence**
  - **Rate is not slow at all !**
  - **CWnd is doubled at every RTT**

TCP Slow Start

- Every ACK segment allows two more segments to be sent
- For each segment that is acknowledged before the retransmission timer goes off, the sender adds one segment's worth of bytes to the congestion window.



Slow Start Threshold

- Slow start causes exponential growth, eventually it will send too many packets into the network too quickly.
- To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold (ssthresh).**
- Initially ssthresh is set to BDP (or arbitrarily high), the maximum that a flow can push to the network.
- Whenever a packet loss is detected by a RTO, the ssthresh is set to be half of the congestion window

Additive Increase (Congestion Avoidance)

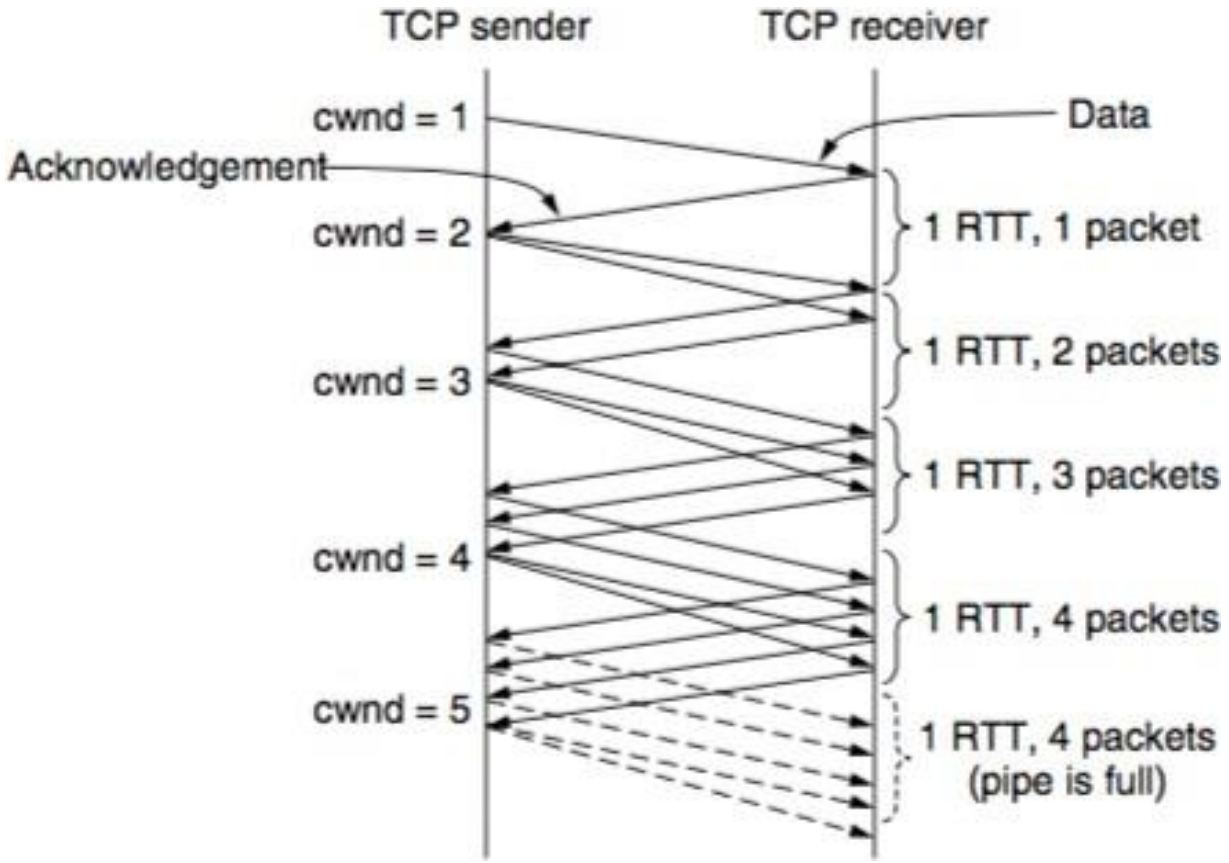
- Whenever ssthresh is crossed, TCP switches from slow start to additive increase.

- Usually implemented with an partial increase for every segment that is acknowledged, rather than an increase of one segment per RTT.

- A common approximation is to increase Cwnd for additive increase as follows:

$$CWnd = Cwnd + \frac{MSS \times MSS}{CWnd}$$

Additive Increase – Packet Wise Approximation



Triggering a Congestion

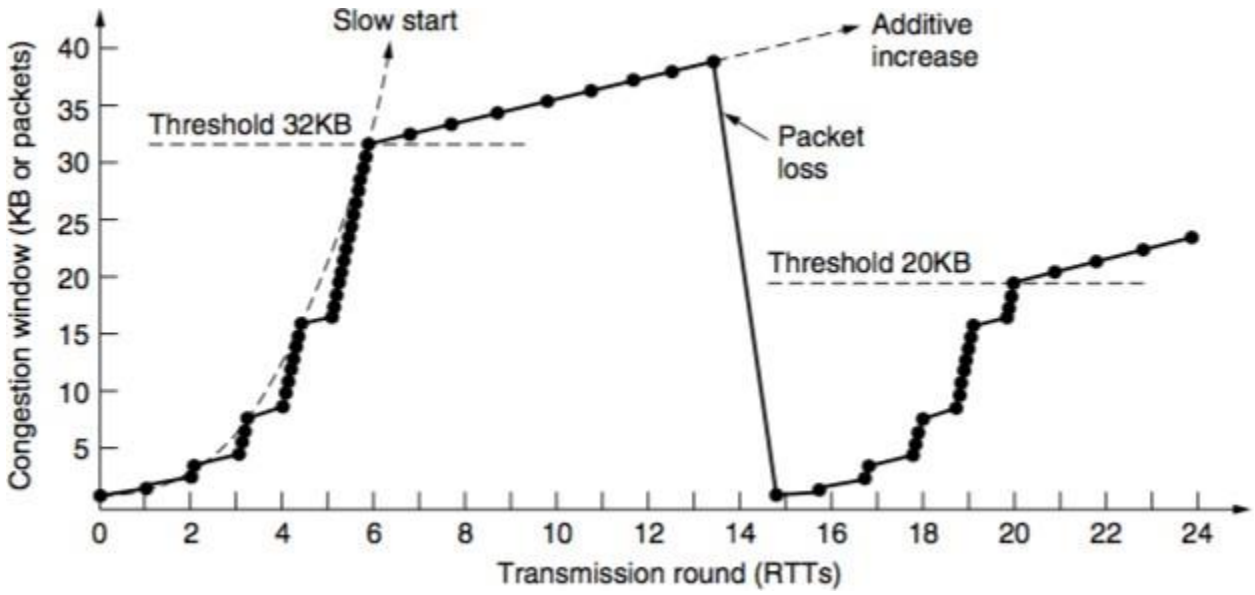
- Two ways to trigger a congestion notification in TCP – (1) RTO, (2) Duplicate ACK
- **RTO**: A sure indication of congestion, however time consuming
- **Duplicate ACK:** Receiver sends a duplicate ACK when it receives out of order segment
  - **A loose way of indicating congestion**
  - **TCP arbitrarily assumes that THREE duplicate ACKs (DUPACKs) imply that a packet has been lost – triggers congestion control mechanism**
  - The identity of the lost packet can be inferred – **the very next packet in sequence**
  - **Retransmit the lost packet and trigger congestion control**

Fast Retransmission – TCP Tahoe

Use THREE DUPACK as the sign of congestion

Once 3 DUPACKs have been received,

- Retransmit the lost packet (**fast retransmission)** – takes one RTT
- Set ssthresh as half of the current CWnd Set CWnd to 1 MSS



Fast Recovery - TCP Reno

- **Once a congestion is detected through 3 DUPACKs, do TCP really need to set CWnd = 1 MSS ?**

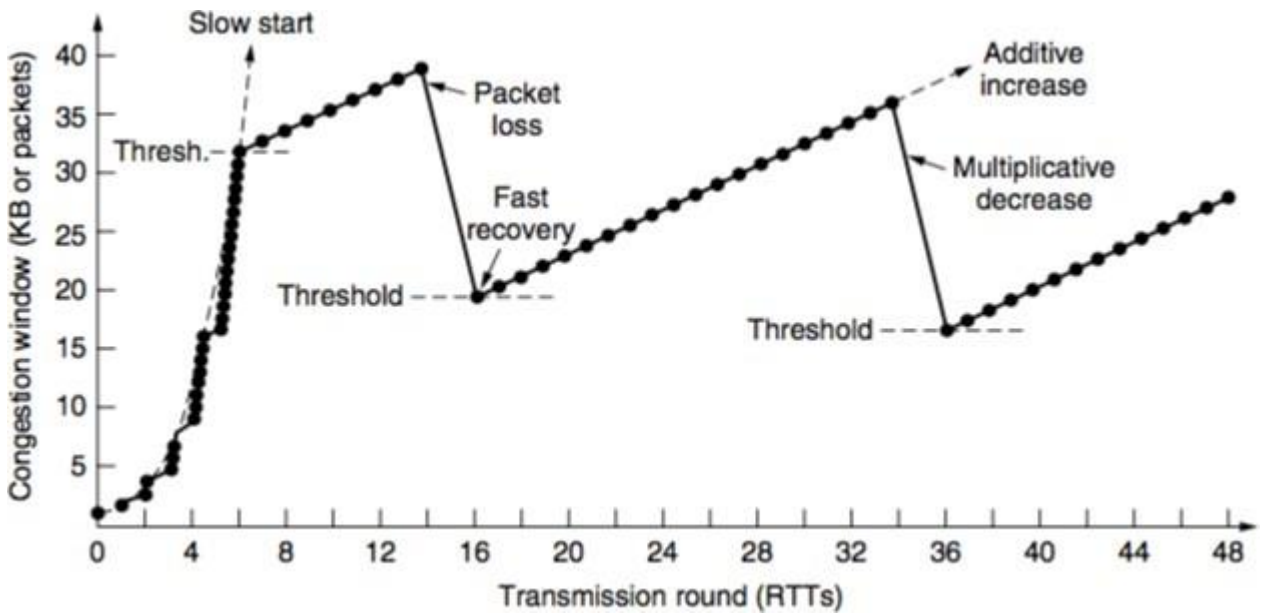
- DUPACK means that **some segments are still flowing in the network**– a signal for temporary congestion, but not a prolonged one

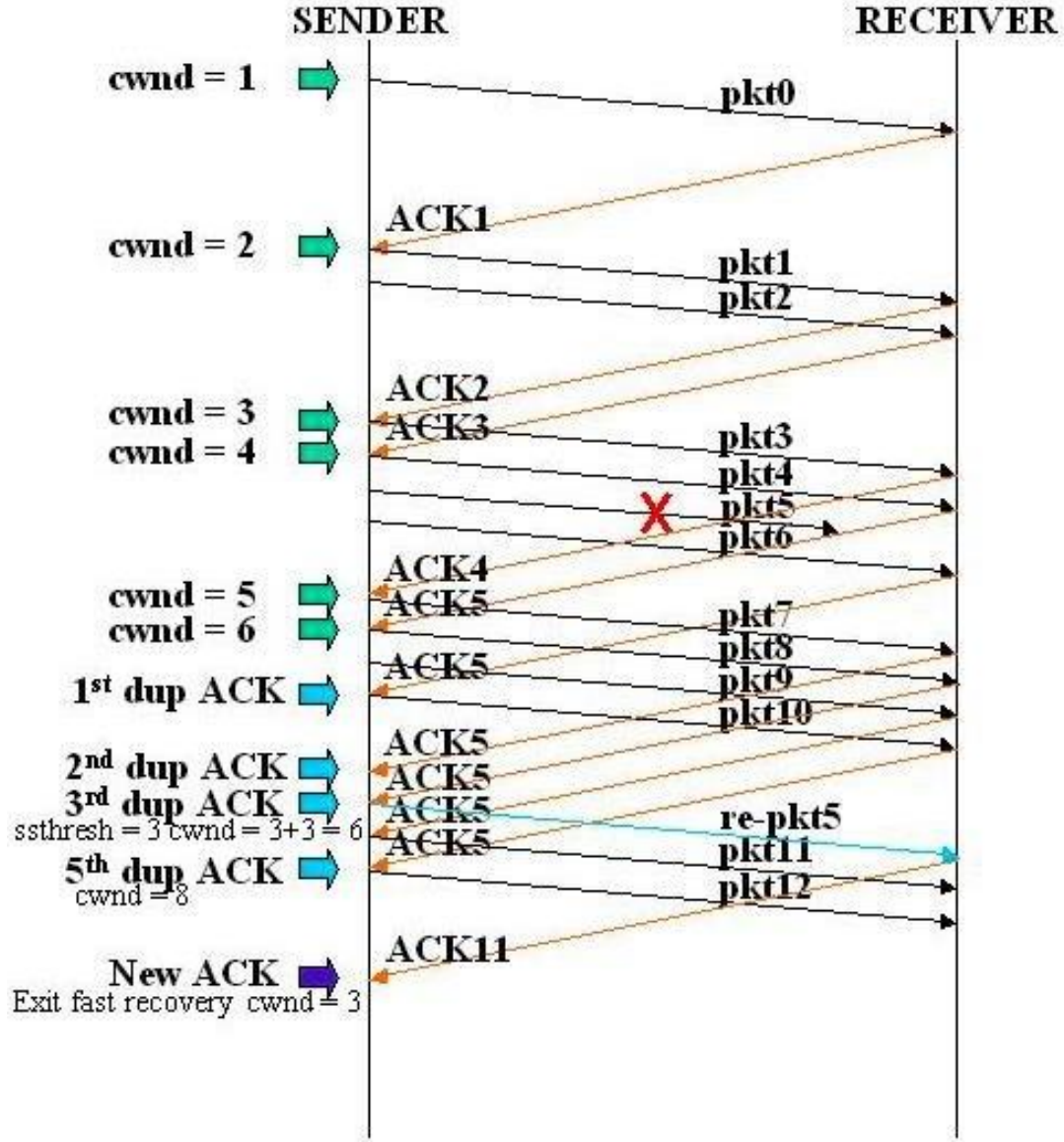
- Immediately transmit the lost segment (**fast retransmit)**, then transmit additional segments based on the DUPACKs received **(fast recovery)**

Fast Recovery - TCP Reno

- **Fast recovery:**
  1. set ssthresh to one-half of the current congestion window. Retransmit the missing segment.
  2. set cwnd = ssthresh + 3.
  3. Each time another duplicate ACK arrives, set cwnd = cwnd + 1. Then, send a new data segment if allowed by the value of cwnd.
  4. Once receive a new ACK (an ACK which acknowledges all intermediate segments sent between the lost packet and the receipt of the first duplicate ACK), exit fast recovery.
  5. This causes setting cwnd to ssthresh (the ssthresh in step 1). Then, continue with linear increasing due to congestion avoidance algorithm.

Fast Recovery - TCP Reno





Next we will see another Transport Layer Protocol UDP.