Transport Layer

Topics

- Topics:
  - Requirements for a transport protocol
  - End to end issues
  - TCP packet format
  - TCP state transition diagram
  - TCP way of handling:
    - Reliability, ordering
    - Flow Control
    - Congestion control
    - Sequence number
- Books: Peterson/Davie

Requirements from a Transport Protocol

- Transport layer carries data from one application process 1(in host 1) to other application process 2(in host 2)
- Application process has following requirement from underlying transport service:
  - Guaranteed message delivery
  - Same order delivery to destination application process
  - At most one copy
  - Supports arbitrarily large messages
  - Supports synchronization between sender and receiver
  - Receiver should be able to apply flow control
  - Support multiple application use the same transport protocol
  - Security
  - (and other QOS requirement!)

Characteristics of a Best Effort Network

- A typical network nature may have following characteristics:
  - Drop message
  - Re order message
  - Deliver duplicate copies
  - Limit message to some finite size
  - Deliver message after some arbitrarily long delay

  (above characteristics are usually shown by a best effort network)

- A transport protocol provides the required services for an application process under the constraint of network layer

TCP, UDP, RPC

- TCP (RFC 793, …) :
  - multiplexed,
  - reliable,
  - ordered,
  - flow controlled,
  - connection oriented,
  - full duplex, byte stream transport service for Internet
  - TCP also does effective *congestion control* to avoid network overload

- UDP (RFC 768]):
  - multiplexed,
  - message oriented
  - connection-less transport service for Internet
- RPC:
  - can be considered a request-reply type of transport service for internet on top aforementioned two services

UDP: User Datagram Protocol [RFC 768]

- Datagram oriented Internet transport
- “best effort” service; doesn't guarantee any reliability. UDP segments may be:
  - lost
  - delivered out of order to application
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

UDP features

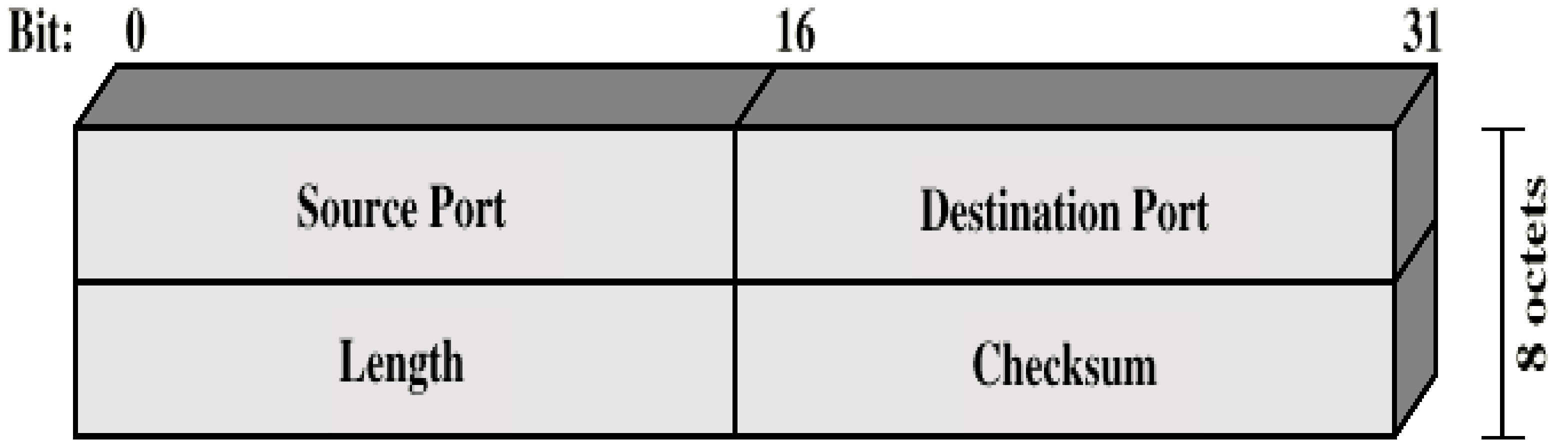
- Simple protocol:
  - no connection establishment delays
  - no connection state at sender, receiver
  - small segment header
  - no congestion control
- reliable transfer over UDP:
  - add reliability at application layer

UDP applications

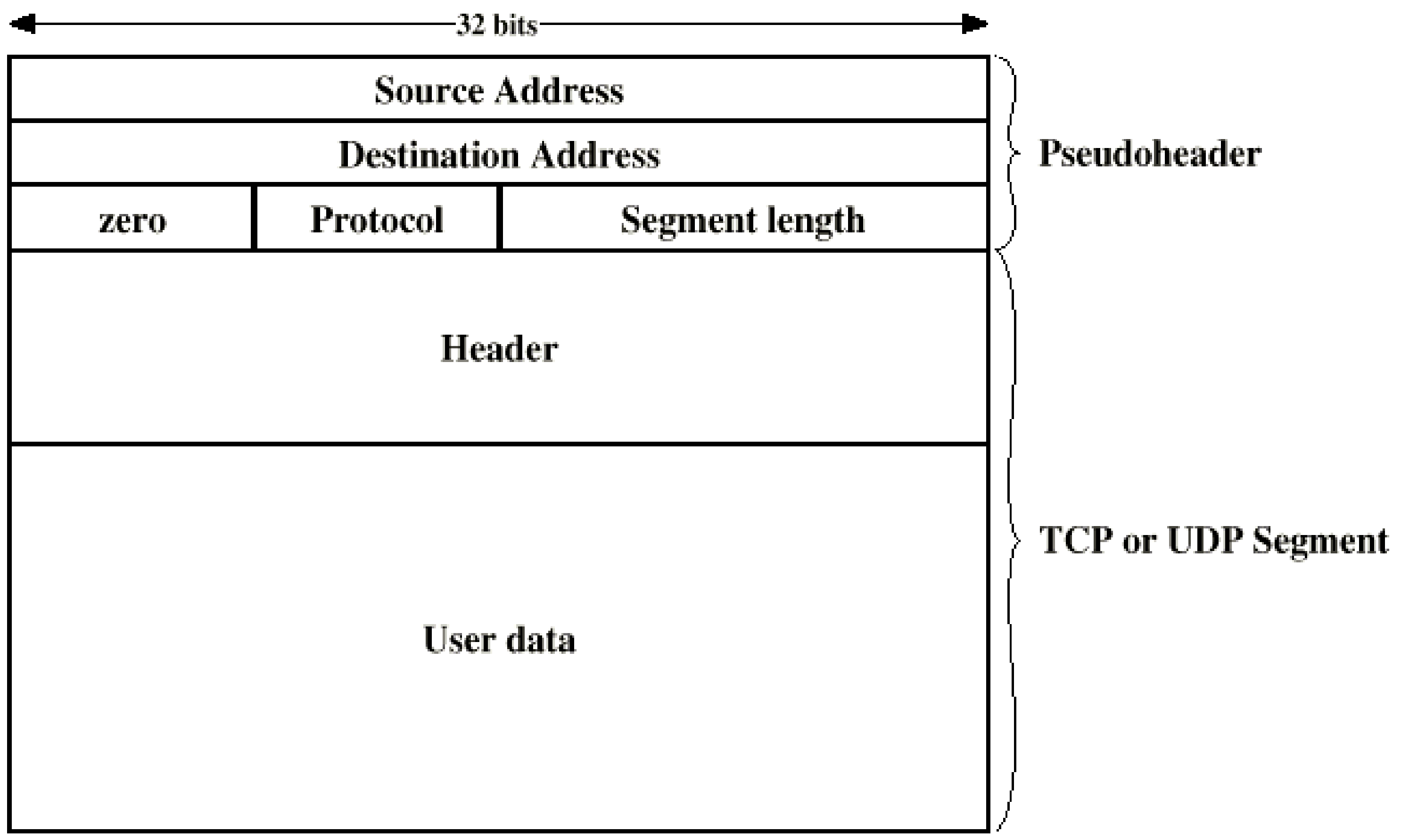
- Useful for Applications such as voice and video, where
  - retransmission should be avoided
  - the loss of a few packets does not greatly affect performance
  - rate sensitive
- other UDP applications
  - DNS; SNMP; NFS

UDP header

- **Length** field is the length of header and data in bytes. Minimum value is 8 bytes



UDP pseudo header



UDP header

- Checksum covers header and data
- 12 byte pseudo-header has some fields from the IP header
  - includes IP address of source and destination, protocol and segment length
  - Double check that data has come to the right destination

End to End Issues

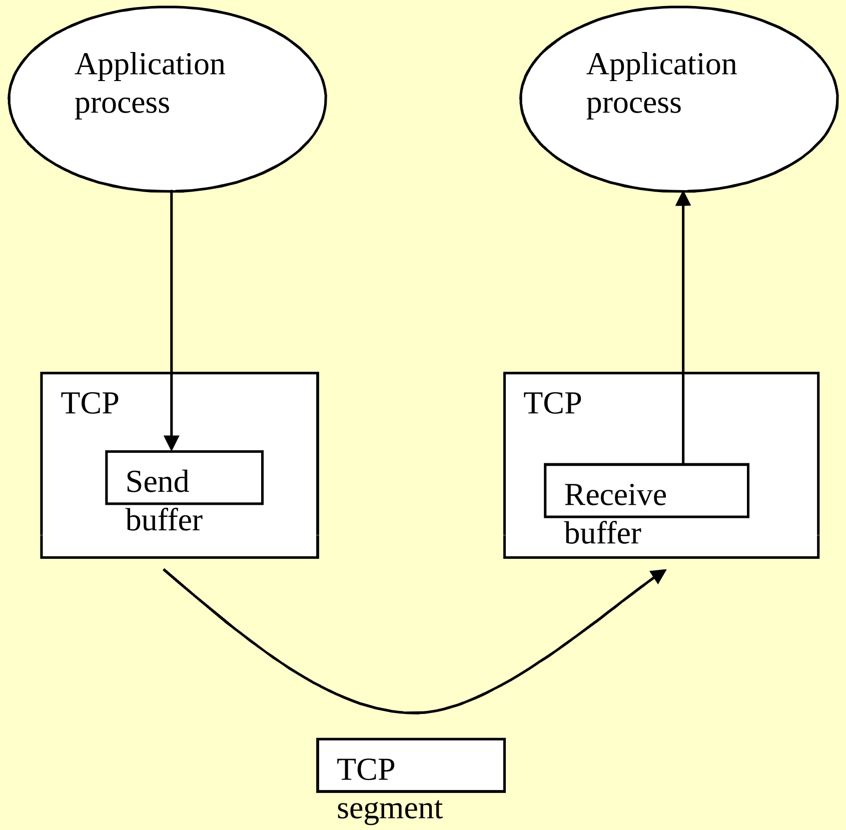
Core of TCP is based on sliding window protocol, similar to what we saw for point to point protocols in DLL. Following are the differences when this works over multihop networks.

- End points may be any two application in any two computers in Internet. This calls for a explicit connection establishment and release process for connection oriented TCP .
- RTT (Round Trip Time i.e. 2* end-to-end delay) is variable. This implies the time-out mechanism for retransmission strategy should be dynamic.
- Re-ordering is not possible in a point to point link while it is very likely in Internet. A packet may be highly delayed (until IP TTL expires) and show up much later. In TCP it handled by *Maximum Segment Life* (MSL) (Typically 120 sec).
- Resource (buffer) dedicated to window algorithm is more or less fixed (approximated by bandwidth delay product) at receiver side for link layer point to point scenario. In internet variable capability machines attaches, so resource allotment should be dynamic
- In point to point sender cannot unknowingly congest the link. In internet it is possible, for some intermediate bottleneck link. This leads to problem of network congestion. Therefore, end to end transport protocol should have a mechanism to detect/avoid congestion.

We will see how TCP handle these issues next.

TCP Functioning

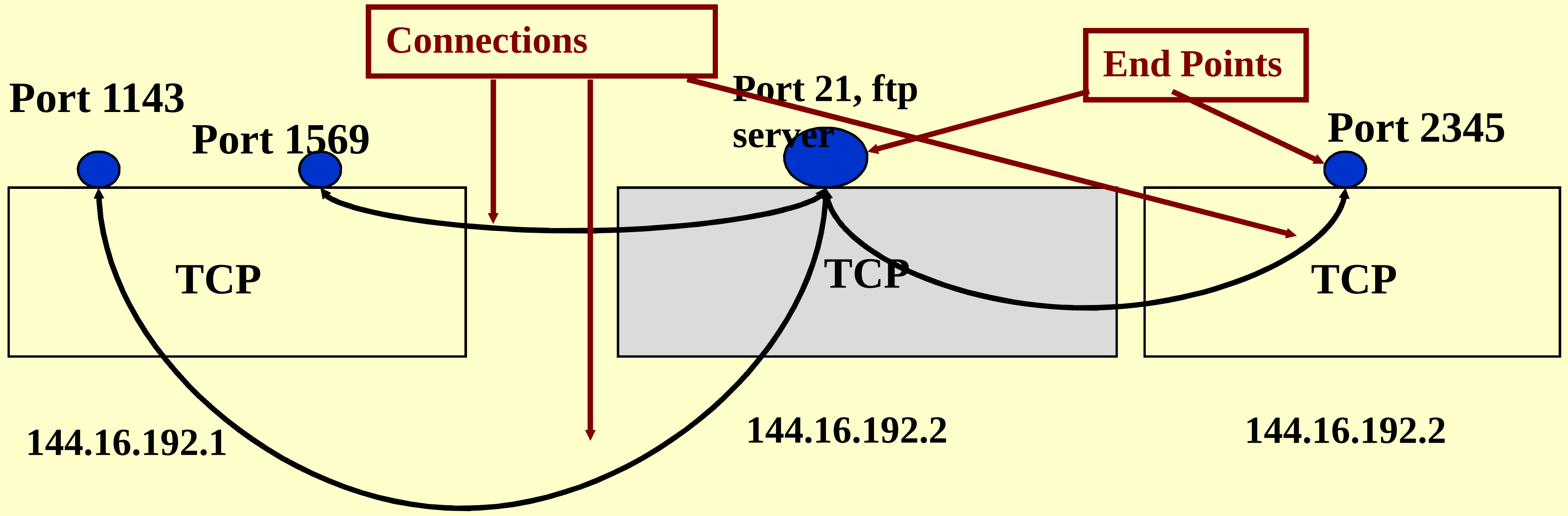
- Application data is broken into what TCP considers the best sized units to send
  - Segment: unit of data passed from TCP to IP
  - MSS: Maximum segment size
- TCP sequences data by associating a sequence number with every byte it sends



TCP functioning (contd..)

- Sending TCP maintains a timer
  - for each segment sent
  - waiting for acknowledgement (ACK)
  - If ACK doesn’t come in time, segment is retransmitted
- Receiving TCP sends an ACK
  - ACK number is the sequence number of the next byte expected
- Receiving TCP
  - Sends ACK
  - Re-sequences the data
  - Discards duplicates
- Congestion and Flow control
  - Sending TCP regulates amount of data to avoid network congestion
  - Receiving TCP prevents fast senders from swamping it

Protocol Ports



- For two processes to communicate, they must be able to address one another
- To identify a process both UDP and TCP use an abstraction called a protocol port
- The port concept gives us the ability to communicate with a particular service on a given system
- This is done by reserving certain ports and having system administrators agree that various services will be mapped to these well-known ports
  - ftp (port 21), telnet (port 23), finger (port 79), login (port 513)

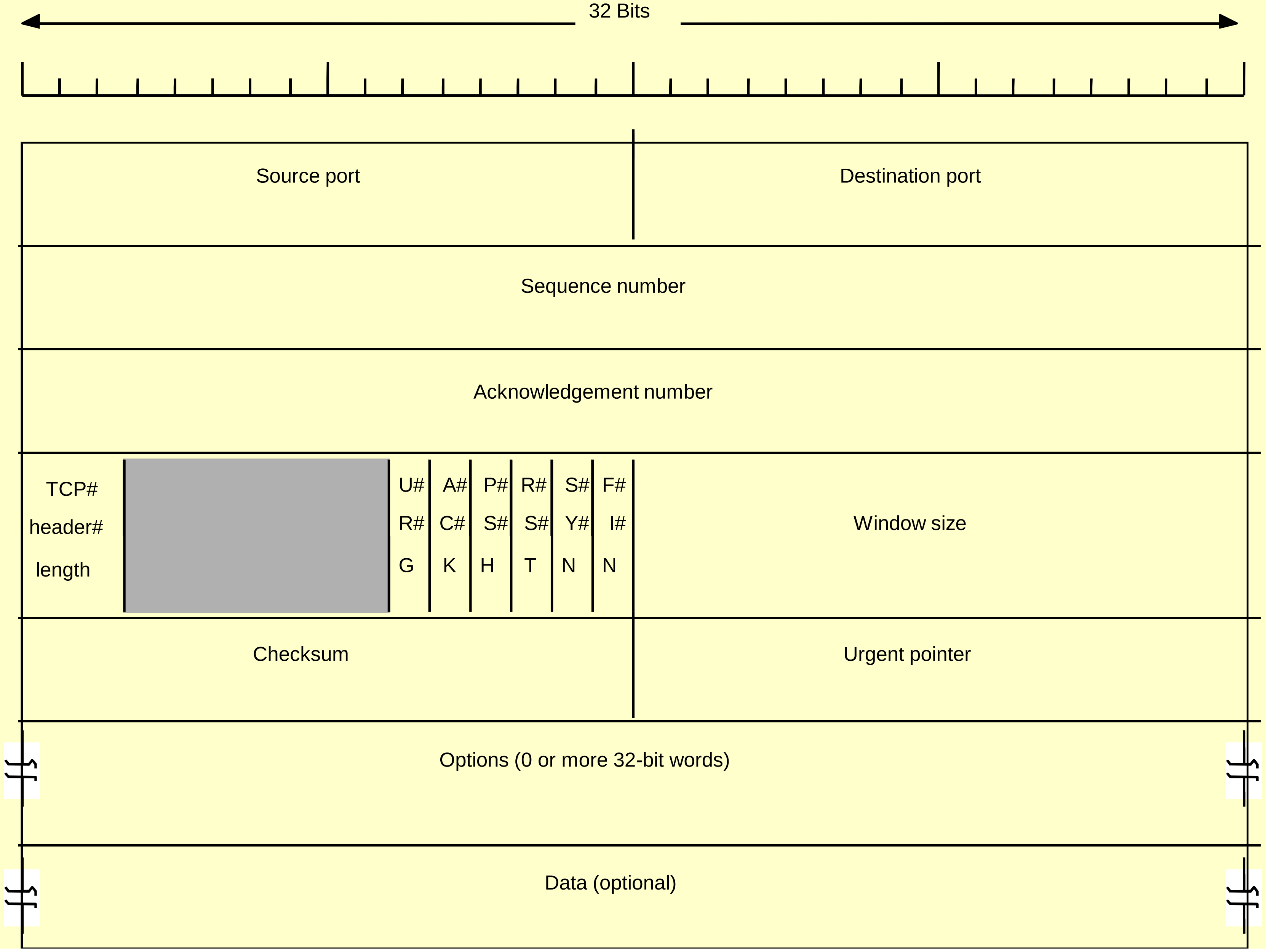
Ports, TCP Connections

- In TCP context, an *End Point* in Internet is a pair <IP address, Port No>

- Two end points participates in a *TCP connection*

- A *Socket* is a programming level abstraction (similar to a file handle) which points to an Internet end point <IP address, port no>

TCP Segment



32 Bits
Source port
Destination port
Sequence number
Acknowledgement number
TCP# header# length
U# R# G
A# C# K
P# S# H
R# S# T
S# Y# N
F# I# N
Window size
Checksum
Urgent pointer
Options (0 or more 32-bit words)
Data (optional)

TCP Segment Size

- The packet exchanged between TCP peers is called TCP segment.
- How many bytes?

  3 mechanisms:

  - When there is MSS amount of data in sending buffer:
    - MSS size is driven by local MTU of link layer (to avoid IP fragmentation) OR
    - Decided by both end's agreement
  - Sending process explicitly asks to do so. PUSH flag. OR
  - A TCP timer that periodically fires.

TCP Header

- Source port, Destination port:
  - These along with source and destination address in IP header makes the TCP de-multiplexer key
  - 16-bit port numbers- 0 to 65535
  - 0 to 1023 are *well-known* ports: assigned to common applications, telnet uses 23, SMTP 25, HTTP 80 etc.

- Sequence no, Acknowledgement, Advertise window:
  - These are related to sliding window mechanism
    - Each byte has *Sequence Number*.
    - The Sequence Number field in TCP segment contain sequence number of first byte in that segment
    - The *Acknowledgement Number (x)* is set by receiver. This value tells
      - » That the receiver has received the segments correctly till x-1, and
      - » what is the next sequence number receiver is expecting .
- Header length:
  - Due to Option Field, TCP has variable length header; this field points to the start of data.
- Checksum:
  - This is computed over TCP header and IP (TCP pseudo Header) host/destination/protocol part

Other Segment Fields

Flags:

- URG: The *urgent pointer* is valid
- ACK: The *acknowledgement number* is valid (i.e. packet has a piggybacked ACK)
- PSH: (Push) The receiver should pass this data to the application as soon as possible
- RST: Reset the connection (during 3-way handshake)
- SYN: Synchronize sequence number to initiate a connection (during handshake)
- FIN: sender is finished sending data

Window Size and Option Field

Advertise Window Field:

- In TCP, the sending window size is variable and depends on available buffer space at receiver end
- Window size controls how much data (bytes), starting with the one specified by the Ack number, that the receiver can accept
- 16-bit field limits window to 65535 bytes. In high speed network this can lead to network under utilization

Options Field:

- This field is optional. Can be used for several purpose such as
  - *Advertise window scale*;

    16 bit can only specify upto 64 Kilo Byte. For specifying higher value, the option field can be used.
  - *MSS Size:*

    Each host can specify MSS. The smaller wins. No specification --> MSS = 536 byte. Each link is expected to carry atleast 536 +20 sized TCP segment.

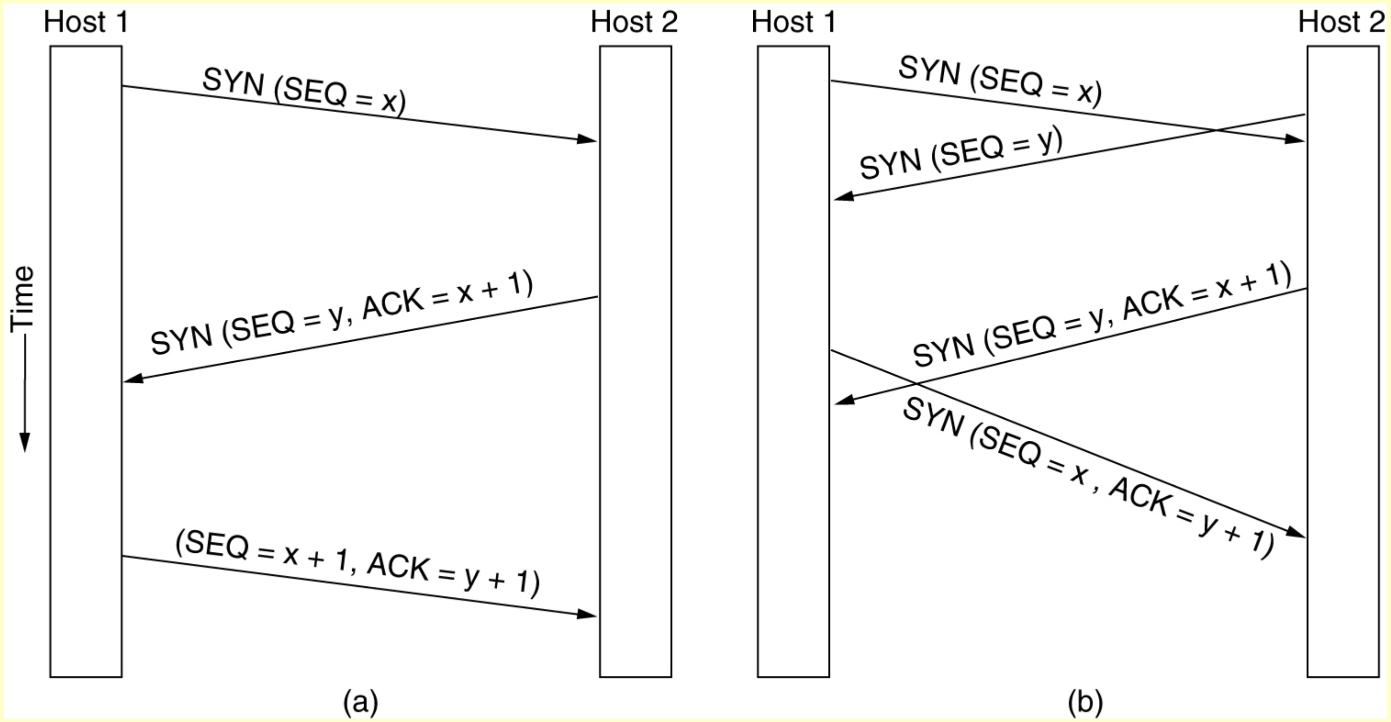
Sequence Number

- Sequence number identifies the byte in the stream between sender & receiver
- Sequence number wraps around to 0 after reaching $2^{32} – 1$
- Acknowledgement number identifies the next byte expected
- Long enough so that
  - >= 2 * receiver size (which is already true 2^32 >>> 64 KB)
  - Should not wrap around within MSL time (120 second currently)
    - This depends on link speed.
    - 1.4 hrs for T1 line (1.5 mbps speed)
    - 57 minutes for Ethernet(10 mbps)
    - 28 second for STS-24 (1.2 gbps)
- Initial Sequence Number for each connection
  - New connection will have different initial sequence number (ISN). Otherwise this can happen
    - host A opens connection to B, source port 512, destination port 612
    - Suppose connection terminates, a new connection opens, A and B assign the same port numbers
    - delayed pkt arrives from old connection

Single format for control and data

- if a node has
  - data to send, as well as ACKs, then it can send both together in one segment - piggybacking
  - an ACK to send, but no data, it can send a separate ACK segment
  - data to send, but no new ACK, it must repeat the last ACK

TCP Connection Establishment


Host 1
Host 2
SYN (SEQ = x)
SYN (SEQ = y, ACK = x + 1)
(SEQ = x + 1, ACK = y + 1)
Time
(a)
Host 1
Host 2
SYN (SEQ = x)
SYN (SEQ = y)
SYN (SEQ = y, ACK = x + 1)
SYN (SEQ = x , ACK = y + 1)
(b)

(a) TCP connection establishment in the normal case.
(b) Call collision.

Three way handshake

- Server does a *passive open*, and Client side does a *active connect*.
- Two party wants to agree on a set of parameters:
  - New Initial Sequence number for both sides, Agreed upon MSS size
  - **Three Way Handshake Steps: (Write it on your copy/register)**
    - *Step 1.*
      - » *Client (active party) send a TCP segment with flag==SYN and sequence number = x, destination port number, MSS*
    - *Step 2.*
      - » *Server responds with flags = SYN + ACK, sequence number = y, Ack = x+1, MSS,*
    - *Step 3.*
      - » *Client responds with flag = ACK, Ack = y+1*

  [Ack is set 1+ sequence number -- to tell what is the next sequence number expected and that all earlier sequence number is received]

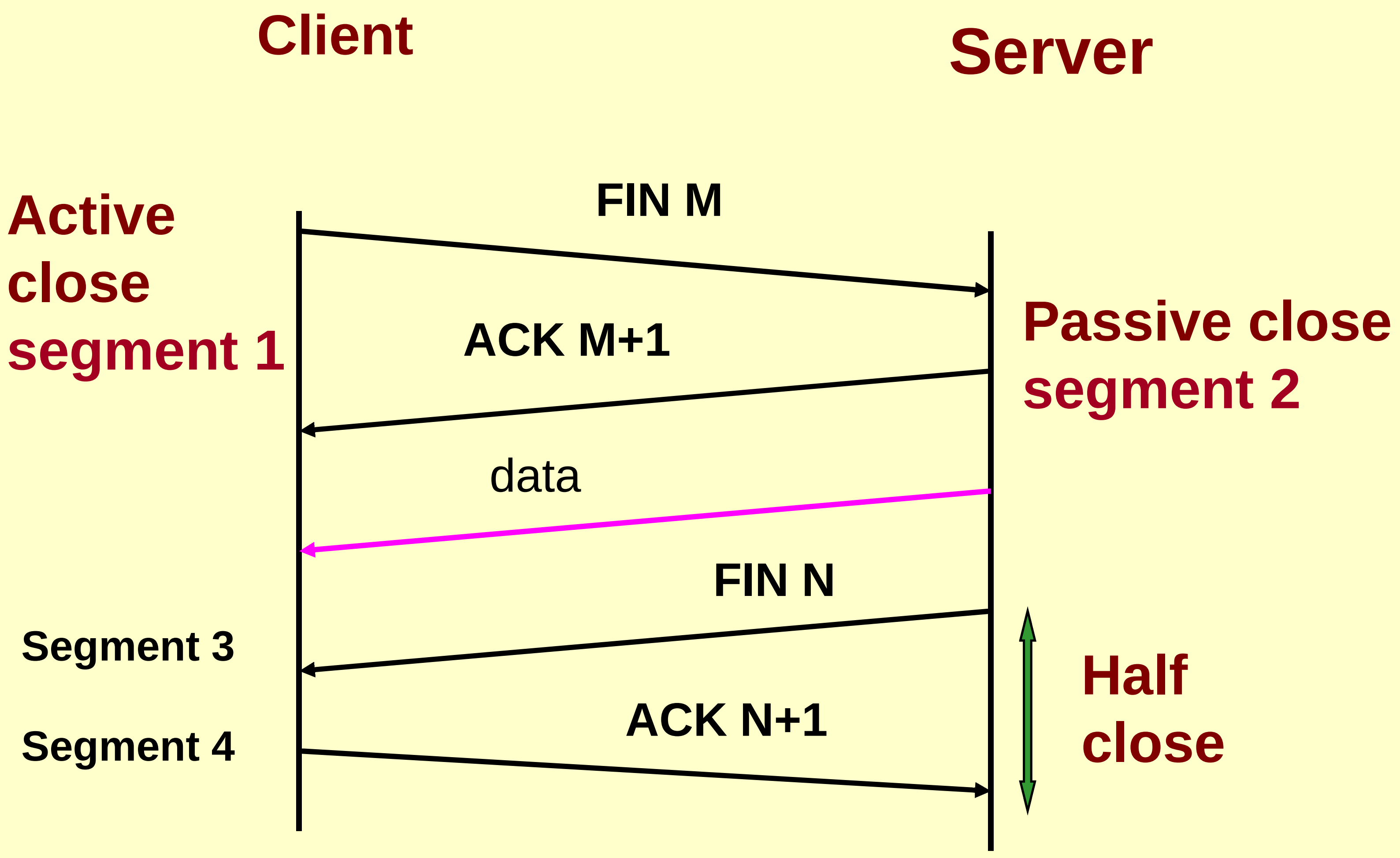
- Why not start with fixed sequence number?
  - Sequence Number (size 32 bit) starts randomly for each connection
  - To avoid two incarnation of same connections reusing the same sequence number too soon to confuse

TCP Connection Establishment (contd..)

Three-way handshake

- Takes care of problems of duplicate SYNs, ACKs
- Initial sequence number is not zero, chosen randomly from 32 bit number
  - This is to avoid duplicate SYNs
  - After a crash, host is required to not reboot for the maximum packet lifetime (that is wait for MSL time before generating or responding any TCP control message like SYN or ACK). This makes packets from previous connections die.
  - The choice of sequence number can be randomized based on clock. Clock is assumed to be running even if host crashes

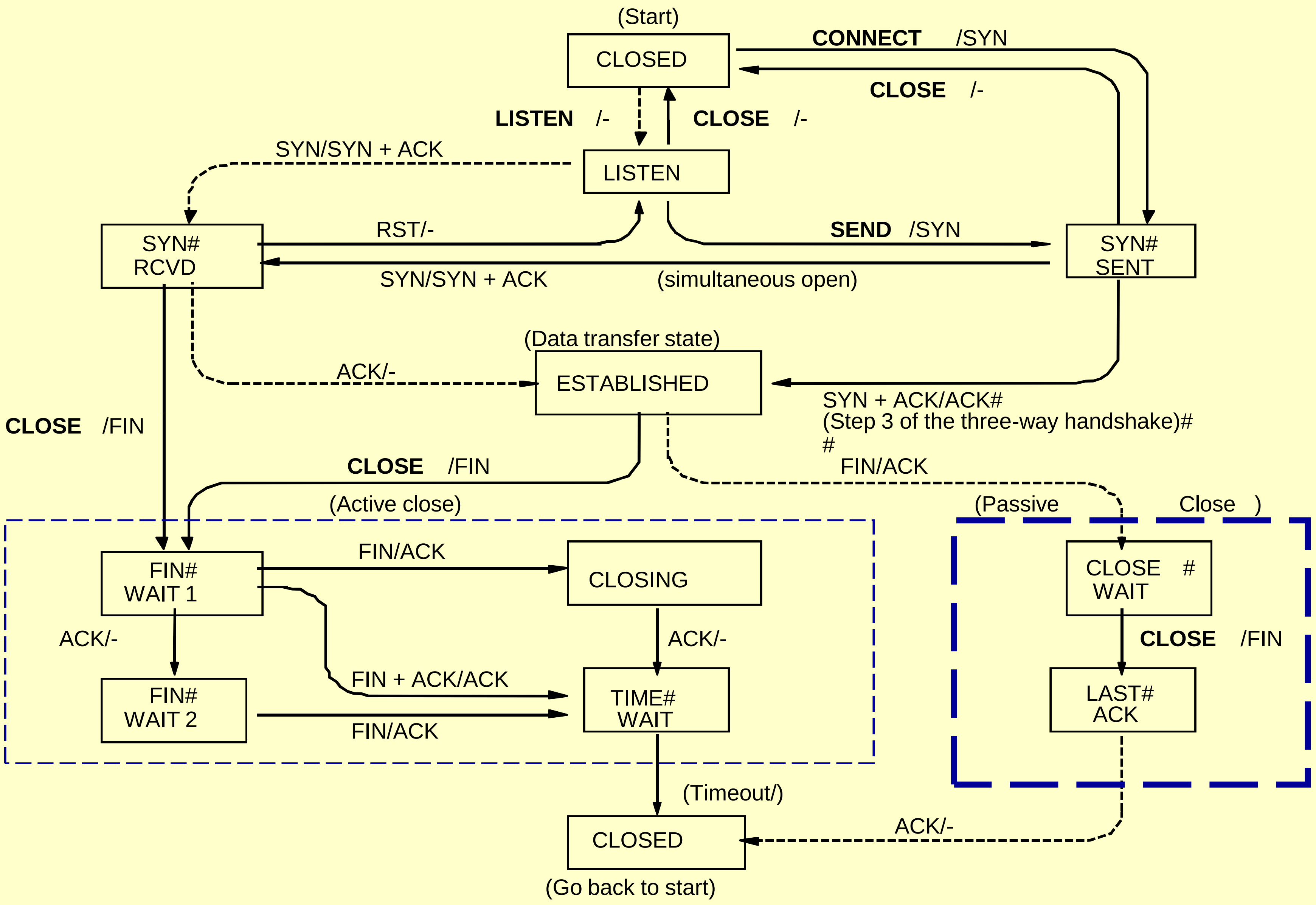
TCP Connection Release



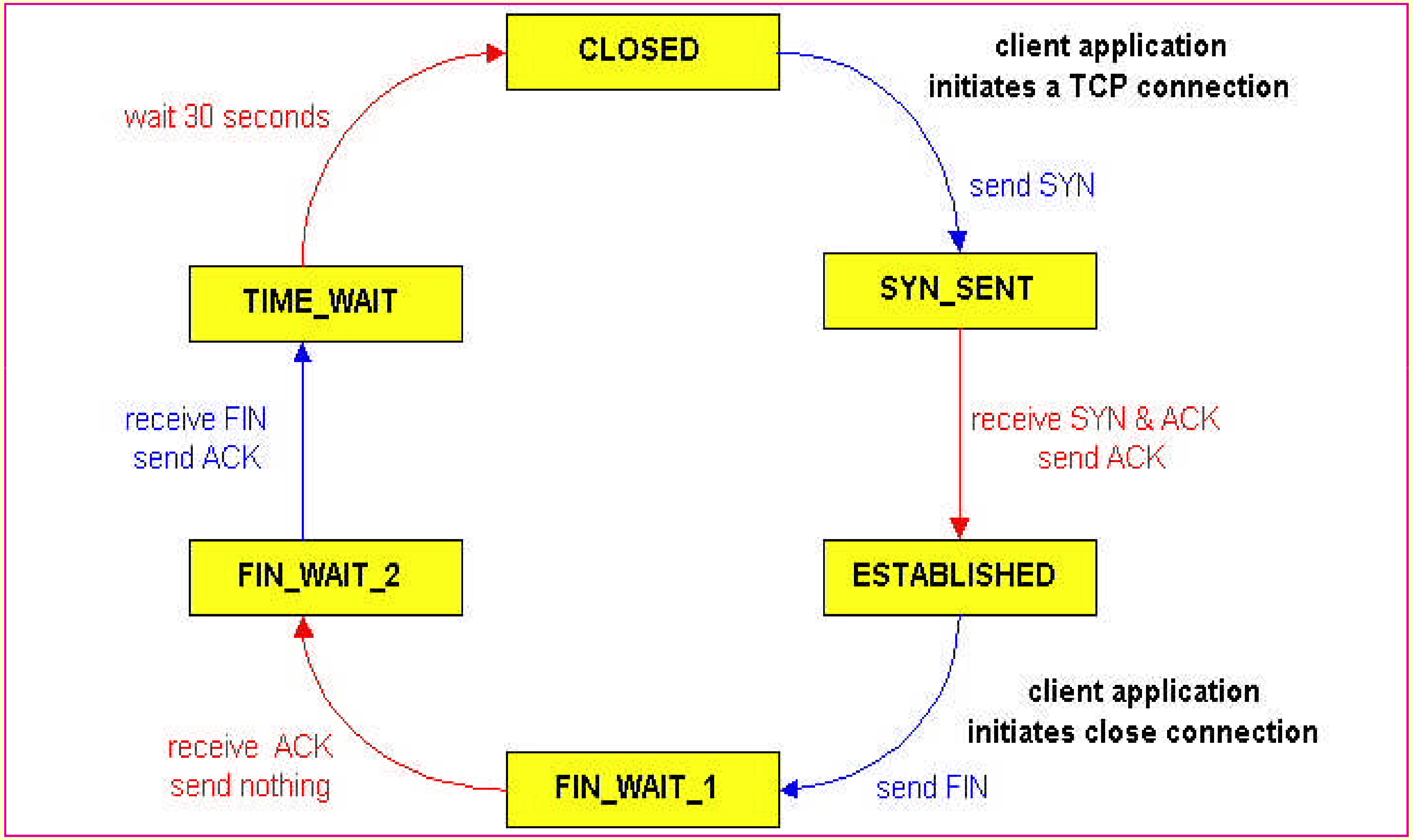
TCP Connection Release

- Two way release is necessary.
  - each end of the full-duplex connection must be closed independently
  - Either side can send FIN when it is done sending data.
  - Both ends maintain state related to connection. On end of data communication these states should be released (if not needed intentionally)
- Half Close (One way close):
  - Side A sends “FIN” (with sequence number)
  - Receive FIN-ACK, shut down.--> Half close.
  - Half Close state: Side A cannot send data to Side B, but vice versa possible
- Full close (Both way close):
  - Side B also send FIN (Can piggy-back on FIN-ACK)
    - Side A Send FIN-ACK in the other direction
  - Lost FINS etc, taken care of by timers.
    - If FIN/ FIN-ACK is lost after several attempts, sender (active closer) releases connection (other side will time-out and release)
  - *Wait for some time*
    - The active closer waits for a fixed time after sending FIN-ACK and before actually closing connection (twice max segment lifetime, MSL)

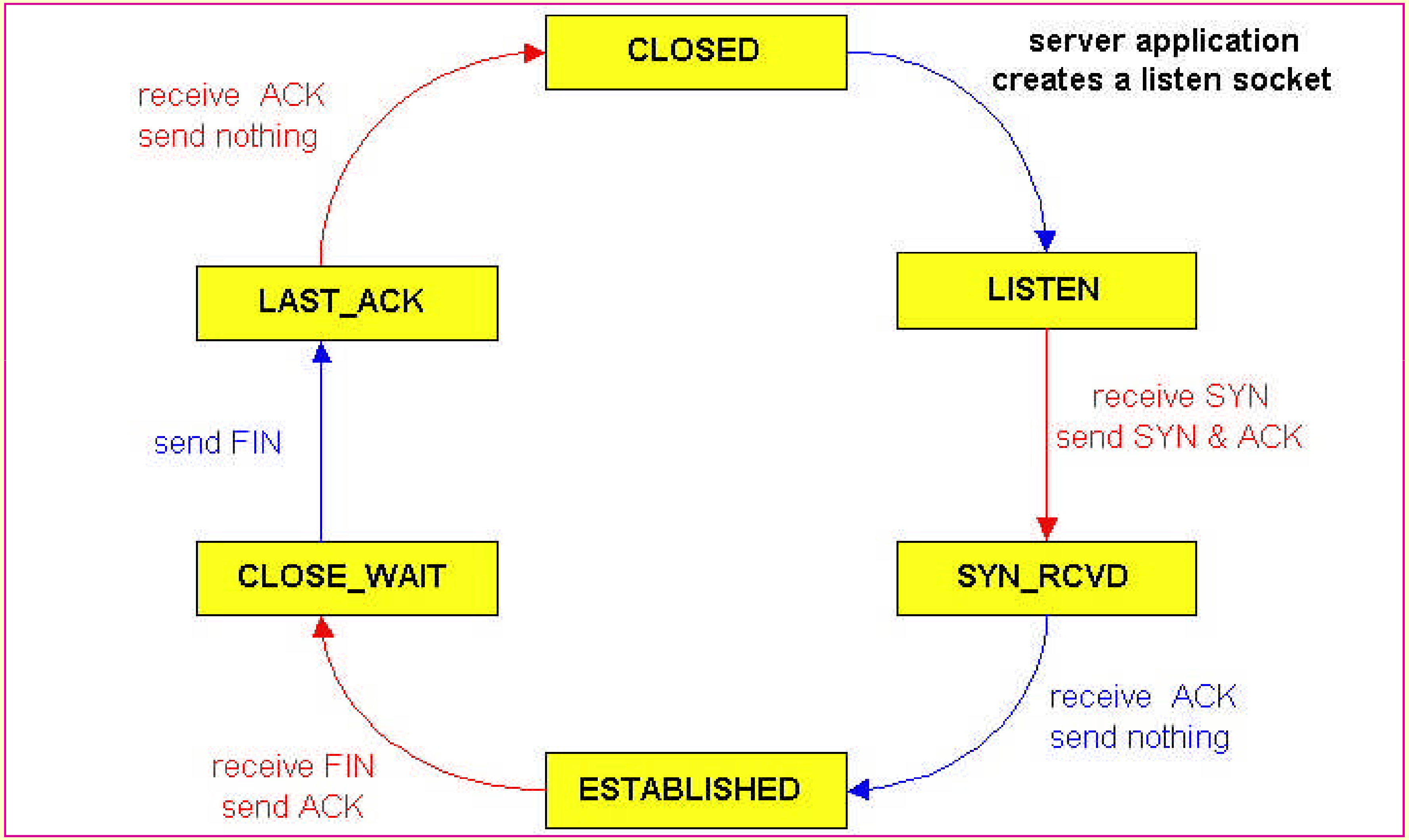
State Transition Diagram for TCP

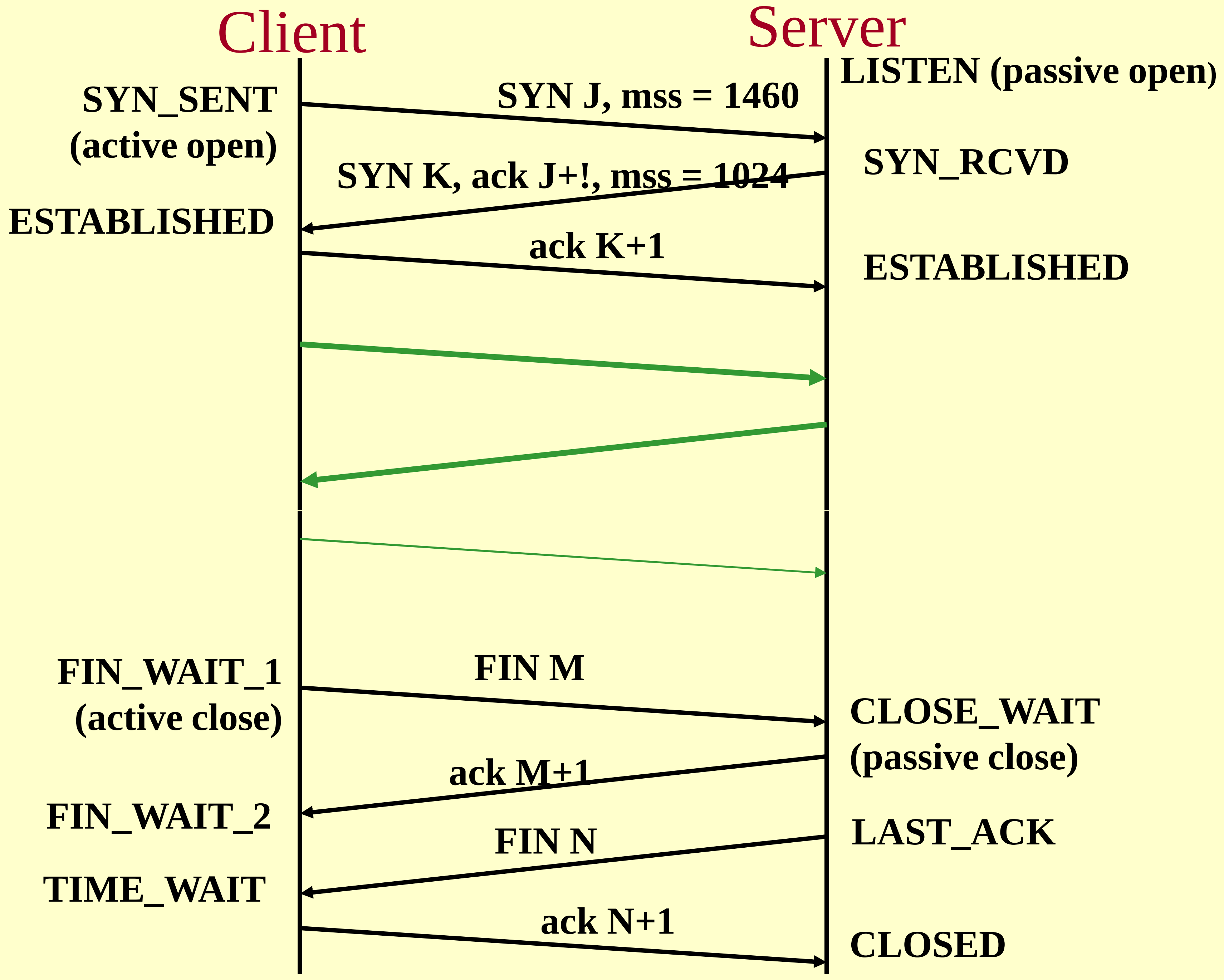


TCP client states



TCP server states




Client
Server
LISTEN (passive open)
SYN_SENT
(active open)
SYN J, mss = 1460
SYN_RCVD
SYN K, ack J+!, mss = 1024
ESTABLISHED
ack K+1
ESTABLISHED
FIN_WAIT_1
(active close)
FIN M
CLOSE_WAIT
(passive close)
ack M+1
FIN_WAIT_2
FIN N
LAST_ACK
TIME_WAIT
ack N+1
CLOSED

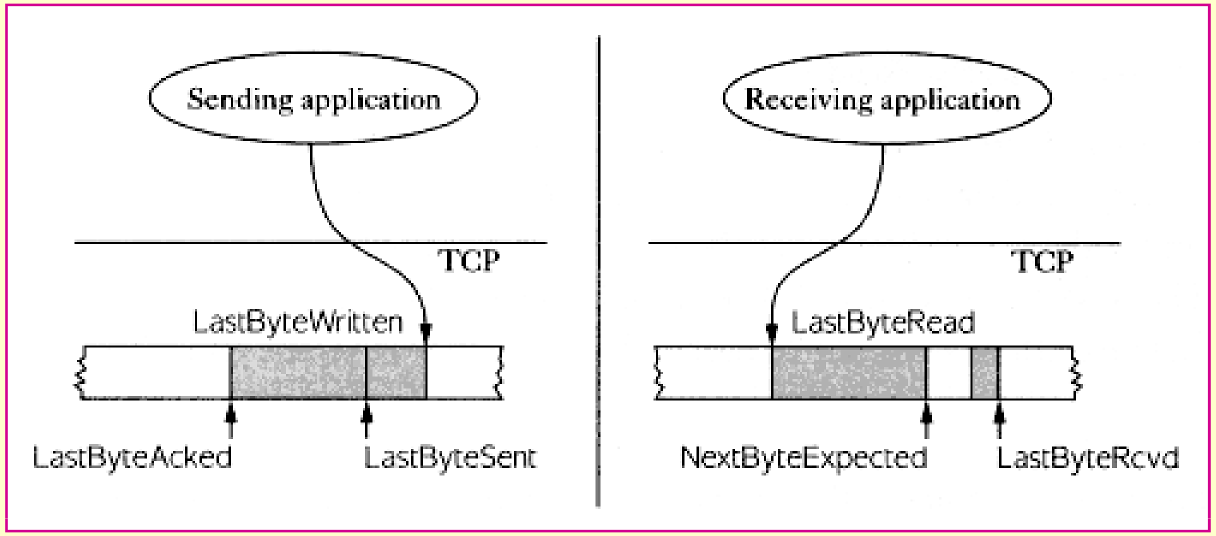
2MSL Time-Wait

- TCP cannot reallocate the socket pair (i.e. the connection) till 2MSL
  - 2MSL wait protects against delayed segments from the previous "incarnation" of the connection
- If server crashes and reboots within 2 MSL wait, it is still safe because TCP delays connections for 1 MSL after reboot

Reliability,Ordered Delivery and Flow Control

- TCP uses the *sliding window protocol* with retransmissions for
  - *reliable, ordered* delivery between two application process and *flow control* between sending and receiving buffers.
- *Reliable, Ordered delivery*
  - The sending and receiving sides of TCP interact in the following manner to implement reliable and ordered delivery:
    - Each byte has a sequence number.
    - ACKs are cumulative
  - *Filling the pipe*
    - TCP window control allows sender to transmit multiple TCP segments without waiting for ACKs
    - Sender’s window size is upper limit on un-ACKed segments
    - Similarly, receiver has a window for buffering (not necessarily the same size as the senders’s)
- *Sending Window size may grow and shrink*
  - Window size minimum of
    - receiver’s advertised window - determined by available buffer space at the receiver
    - congestion window - determined by sender, based on network feedback

Reliability and Ordered Delivery



- On the receiving side, TCP maintains receive buffer to hold un-ordered data or ordered data that the application process has yet read.

- Sending side buffer hold data that has been sent but not yet ack-ed and data that has been written by application process but not yet sent.

Sending Side

- LastByteAcked <=LastByteSent
- LastByteSent <= LastByteWritten
- bytes between LastByteAcked and LastByteWritten must be buffered.

Flow Control

Receiving side

- LastByteRead < NextByteExpected
- NextByteExpected <= LastByteRcvd + 1
- bytes between NextByteRead and LastByteRcvd must be buffered.

Flow Control

- ♣ Sender buffer size : MaxSendBuffer
- ♣ Receive buffer size : MaxRcvBuffer {*Default buffer: 4096 to 16384 bytes, Ideal: window and receiver buffer = bandwidth-delay product*}

Receiving side

- ♣ LastByteRcvd - NextBytteRead <= MaxRcvBuffer
- ♣ AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - NextByteRead)
  - *(Size of AdvertiseWindow depends on how fast application is consuming data)*

Sending side

- ♣ LastByteSent - LastByteAcked <= AdvertisedWindow
- ♣ EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)
- ♣ LastByteWritten - LastByteAcked <= MaxSendBuffer
- ♣ Block sender if (LastByteWritten - LastByteAcked) + y > MaxSendBuffer
  - (local application is overflowing buffer)

- In TCP receiver throttles the sender by advertising a window that is no larger than the receiving buffer can handle at the moment.

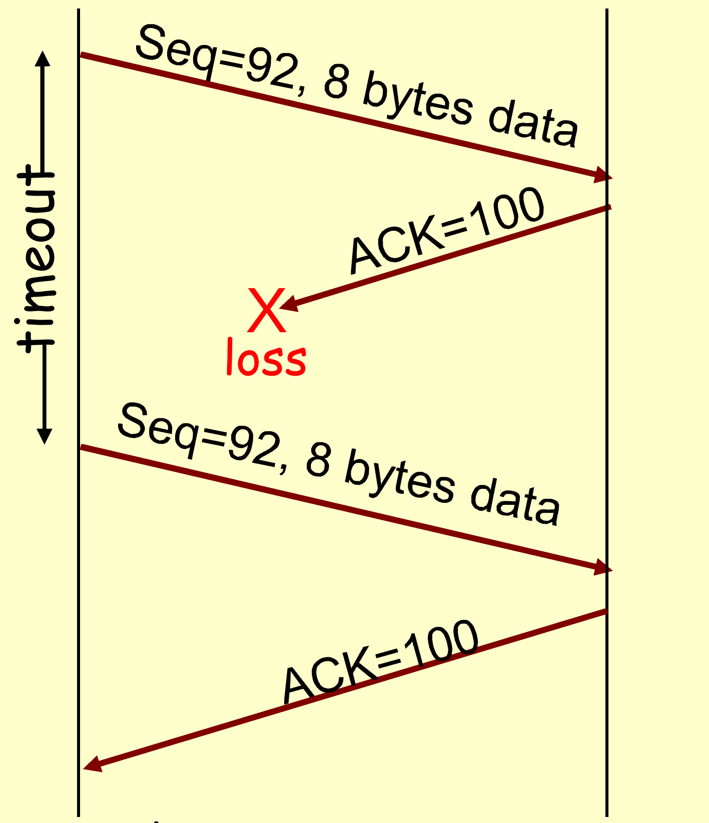
Cumulative and duplicate ACKs

- A cumulative ACK is generated only on receipt of a new in-sequence segment at the receiver
  - Receiver may delay the sending of an ACK
- A duplicate ACK is generated whenever an out-of-order segment arrives at the receiver

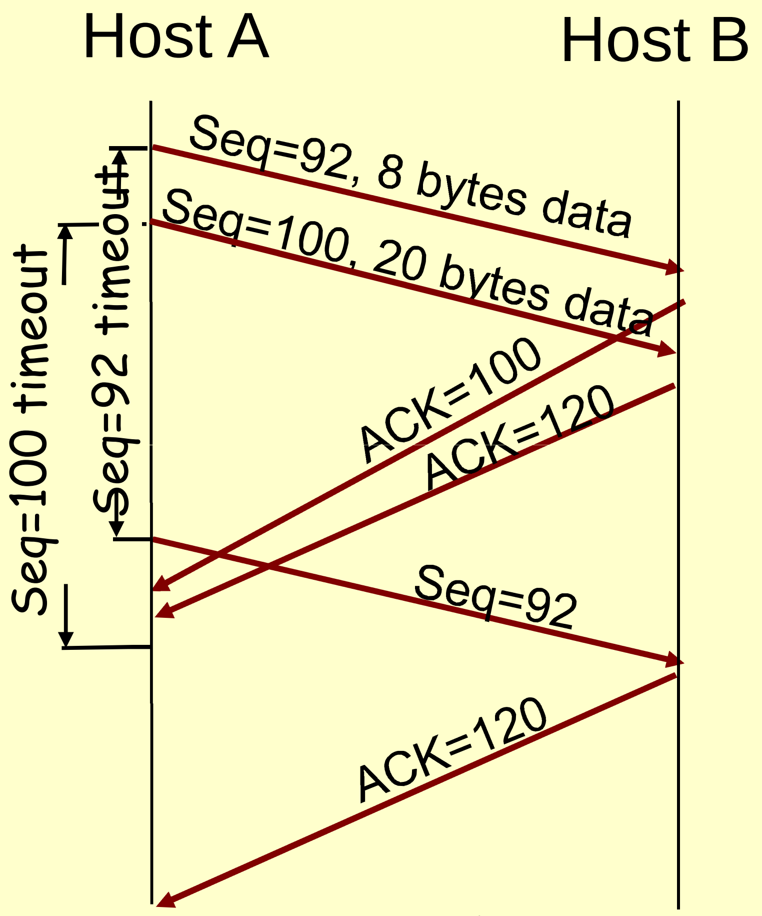
TCP ACK generation [RFC 1122, 2581]

- in-order segment arrival, everything else already ACKed: Delayed ACK (500 ms)
- in-order segment arrival, one delayed ACK pending: Cumulative ACK
- out-of-order segment arrival, higher-than-expect seq. #: Duplicate ACK
- arrival of segment that partially or completely fills gap: Cumulative ACK

TCP: retransmission scenarios



lost ACK scenario



premature timeout, cumulative ACKs

Retransmission

- Default:
  - Cumulative ACKs, Duplicate ACKs
  - go-back-N retransmission
    - N is equal to size of sending window
- Optimization:
  - Selective ACK (SACK)
    - need to specify ranges of bytes received (requires large overhead)
  - Selective retransmission

Timeouts and retransmission

- TCP manages four different timers for each connection
  - retransmission timer: when awaiting ACK
  - persist timer: keeps window size information flowing
    - Sender uses persist timer to probe receiver when AdvertisedWindow=0.
    - Persist Timer bounded between 5s and 60s. It does exponential backoff like other timers
  - keepalive timer: when other end crashes or reboots
    - Optional timer
      - Not necessary, because “connection” defined by endpoints
      - Connection can be “up” as long as source/destination “up”
    - Found in most implementations
      - Used to detect idle clients or half-open connections and de-allocate server resources (Example: telnet, ftp)
  - 2MSL timer: for the TIME_WAIT state

RTT estimation for Retransmission

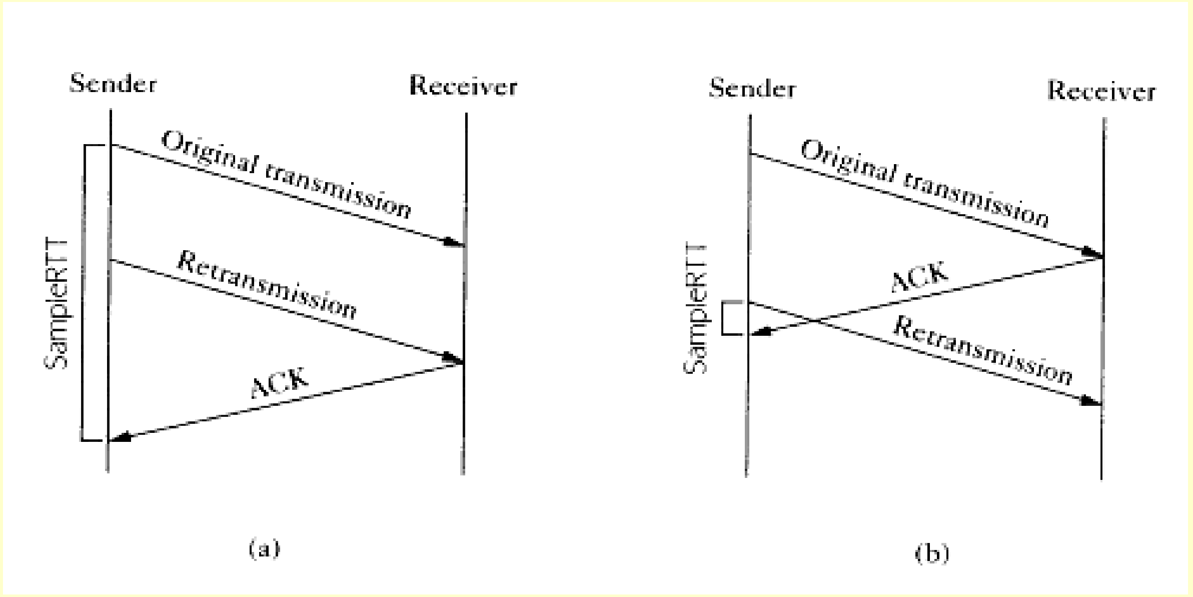
- RTT Estimation
  - TCP guarantees reliable delivery and so it retransmits each segment if an ACK is not received in a certain period of time. TCP sets this timeout as a function of the RTT it expects between the two ends of the connection.
- Accurate timeout mechanism is important for TCP sliding window mechanism
  - Too long: Under-utilization of link
  - Too short: Wasteful retransmission resulting into duplication and congestion
- Fixed: Choose a timer interval a priori;
  - useful if system is well understood and variation in packet-service time is small (true for point to point DLL window control mechanisms)
- Adaptive: Choose interval based on past measurements of RTT
  - Given the range of possible RTT's between any pair of hosts in the Internet, as well as the variation in RTT between the same two hosts over time, choosing an appropriate timeout value is not that easy. To address this problem, TCP uses an adaptive retransmission mechanism. We describe this mechanism and how it has evolved over time.

Exponential averaging filter

- Measure SampleRTT for segment/ACK pair
- Compute weighted average of RTT
  - EstimatedRTT = α PrevEstimatedRTT + $(1 - \alpha)$ SampleRTT
  - RTO = β * EstimatedRTT
- Typically $\alpha = 0.9$; $\beta = 2$ (original TCP recommendation)

  - The parameter α is selected to smooth the EstimatedRTT . A small RTT tracks changes in RTT but too heavily influenced by temporary fluctuations.
  - A large α is more stable but perhaps not quick enough to adapt to real change

RTO: Karn-Partridge Algorithm



- During retransmission, it is unclear whether an ACK refers to a packet or its retransmission
- Karn/Partridge's simple approach for RTT estimation:
  - Don't take sample for retransmitted segment in the previous formula.
  - Restart RTO only after an ACK received for a segment that is not retransmitted.
- Karn/Partridge also introduced Exponential Backoff:
  - Each time a segment is retransmitted , the Timeout (RTO) is set to double of last Timeout value
  - Motivation is like that of Ethernet: Congestion is the most likely cause of loss. Hence TCP should not react aggressively to a timeout

RTT: Jacobson-Karel algorithm

- Previous RTT computation was not helping reduce network congestion
- It does not take variance of Sample RTT s into account

♦ Jacobson-Karel [Use mean and deviation of RTT]

Tanenbaum:
$D = \alpha D + (1-\alpha)(RTT - SampleRTT)$,
Timeout= RTT +4*D

- Measure the sample RTT as before

  - ***Difference = SampleRTT - EstimatedRTT***
  - ***EstimatedRTT = EstimatedRTT + (d * Difference)***
  - ***Deviation = Deviation + d (|Difference| - Deviation)),***
    - where ***d*** is a fraction between 0 and 1

  Consider variance when setting timeout value

  - ***Timeout = u * EstimatedRTT + q * Deviation,***
    - where $u = 1$ and $q = 4$ (based on experience)

  - Thus when *Deviation* is small RTO is close to *EstimatedRTT* , large *Deviation* causes the *Deviation* term to dominate

Interactive Input

- Small segments cause congestion
  - Rlogin generates 41-byte packets which can cause congestion
- Delayed ACKs are used to reduce number of segments {Nagle algorithm [RFC 896] }
  - Delay-ACK Timer: typically 200ms
  - If TCP connection has outstanding data that has not been ACKed, small segments cannot be sent until ACKs come in. {Nagle algorithm [RFC 896] }
  - May be turned off if appropriate
  - For example, X Window server

Silly-Window syndrome

- Slow application receiver
  - TCP advertises small windows
  - Sender sends small segments
- Nagle algorithm extension
  - Receiver advertises window only if size is MSS or half the buffer
  - Sender typically sends data only if a full segment can be sent

TCP Congestion Control

- CongestionWindow:
  - limits amount of data in transit
  - MaxWin = MIN (CongestionWindow, AdvertisedWindow)
  - EffectiveWin = MaxWin - (LastByteSent - LastByteAcked)
- Congestion Control:
  - On detecting a packet loss, TCP sender assumes that network congestion has occurred
  - On detecting packet loss, TCP sender drastically reduces the congestion window
  - Reducing congestion window reduces amount of data that can be sent per RTT
  - TCP interprets timeouts as congestion, not transmission error induced packet loss.

AIMD: Additive increase Multiplicative decrease

- AIMD
  - Source infers congestion upon RTO
  - Increase Congestion Window (linearly, by 1 per RTT) when congestion goes down
  - Decrease Congestion Window (multiplicatively, by factor of 2) when congestion goes up
  - Provides fair sharing of links

Slow start and Congestion avoidance

- AIMD may be too conservative

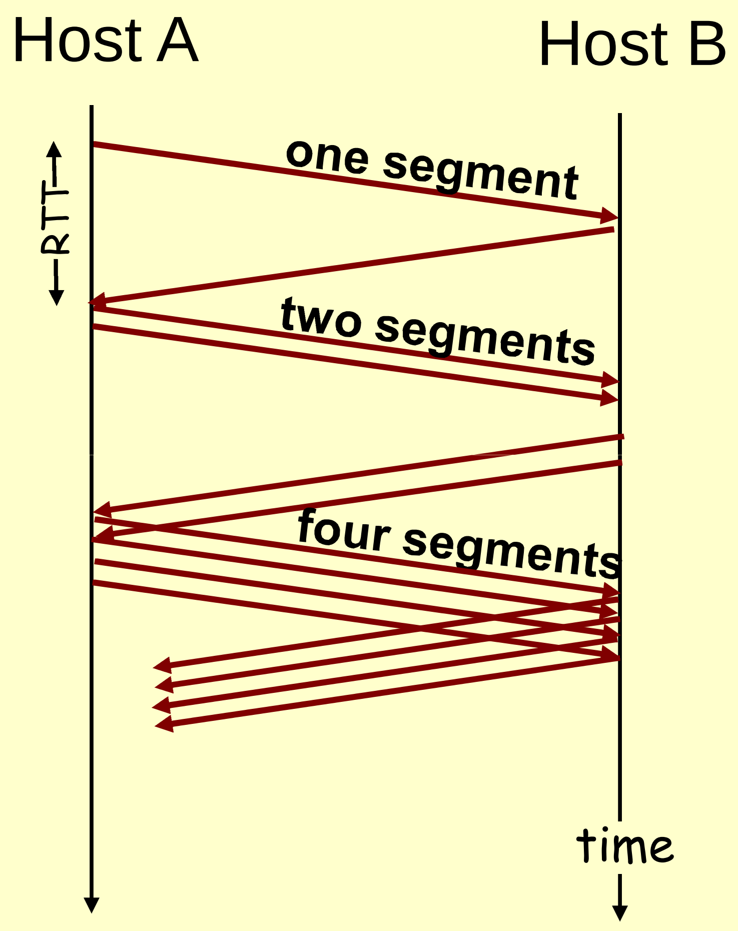
- Slow Start
  - Increase CongestionWindow exponentially upto a threshold (ssthresh).
- Congestion Avoidance
  - Increase cwnd linearly after ssthresh

- On Timeout
  - the congestion window is reduced to the initial value of **1 MSS**
  - Set ssthresh = 1/2 * current CongestionWindow
  - more precisely,

    ssthresh = maximum of min(cwnd,receiver's advertised window)/2 and 2 MSS
  - Slow Start in initiated

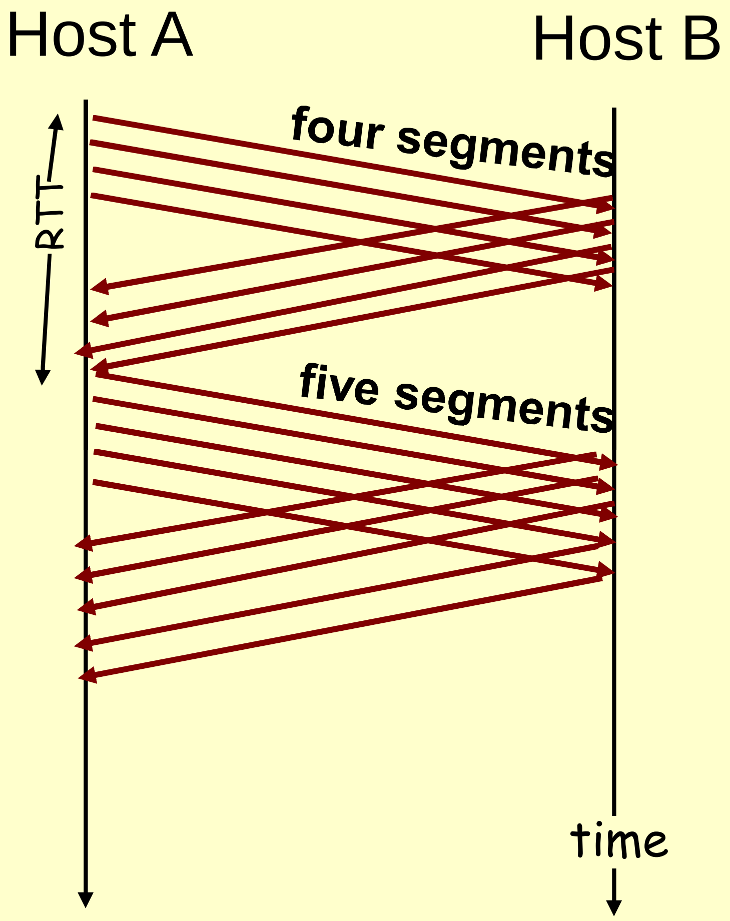
Slow start phase

- initialize:
  - CongestionWindow = 1
- for (each ACK)
  - CongestionWindow ++
- until
  - loss detection OR
  - CongestionWindow > ssthresh

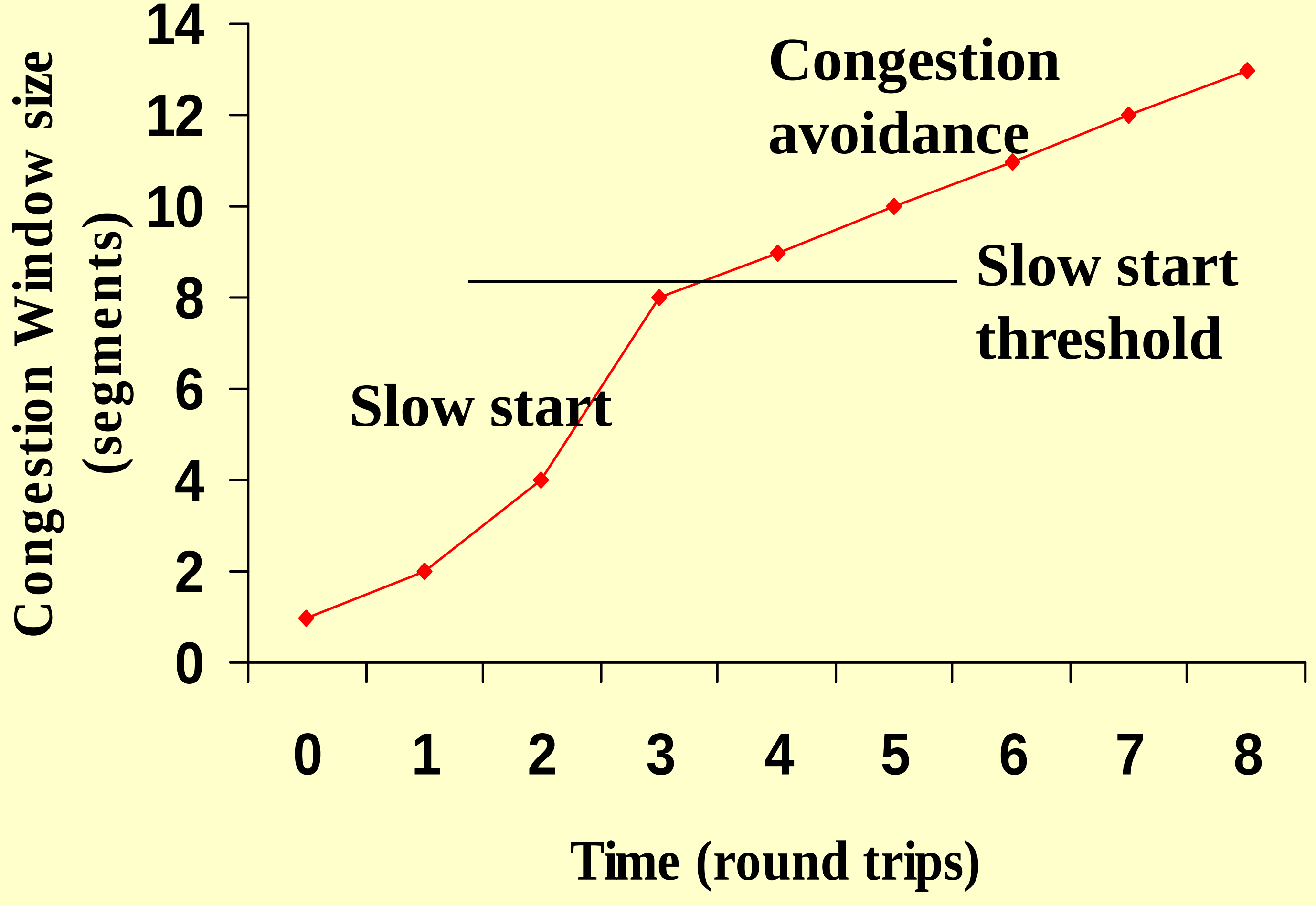


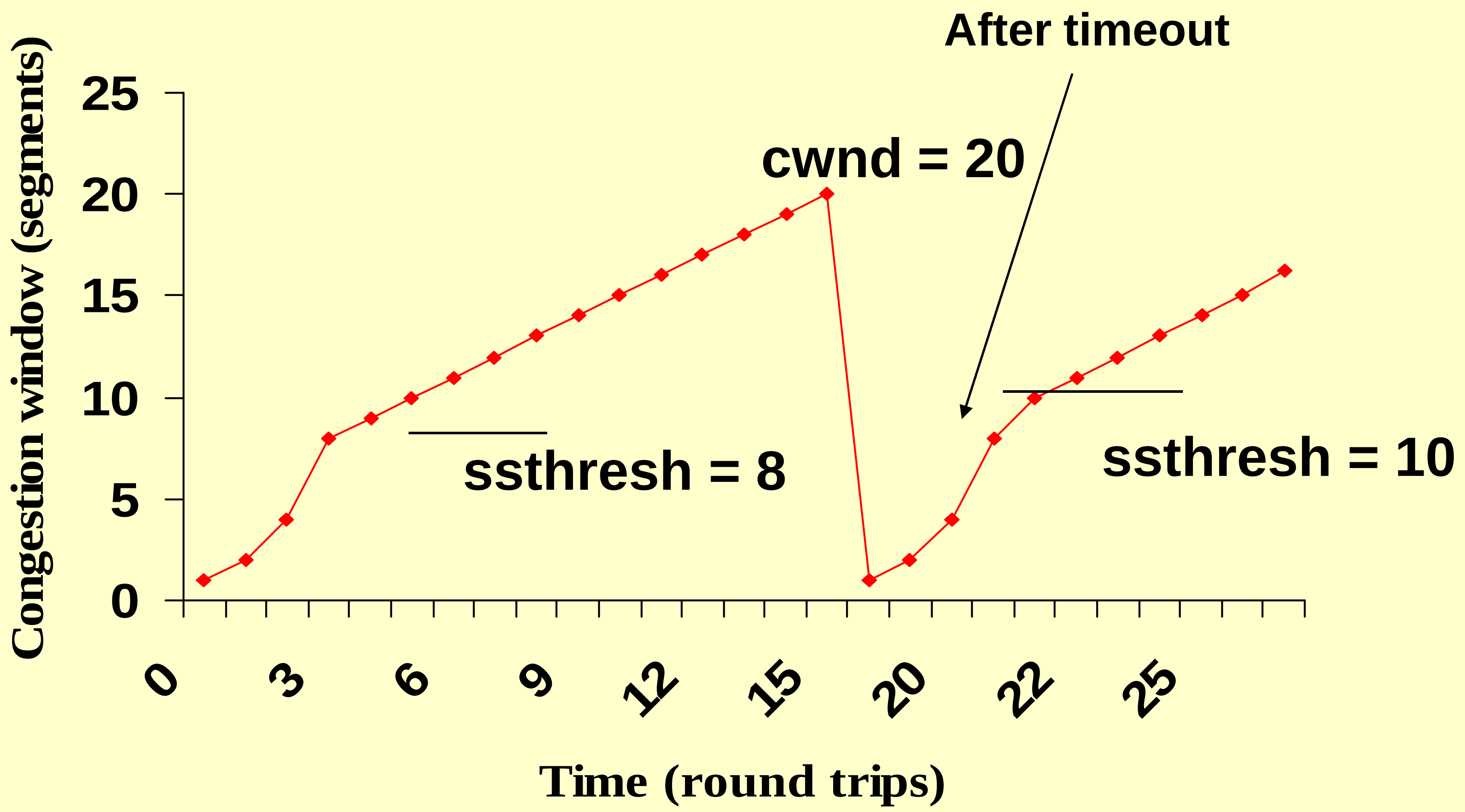
Congestion Avoidance Phase

```
/* CongestionWindow > threshold */
•   Until (loss detection) {
 with every ACK:
 CongestionWindow ++
  }
•   ssthresh = Cwnd/2
•   CongestionWindow = 1
•   perform slow start
```



Slow Start Congestion Avoidance




After timeout
cwnd = 20
ssthresh = 8
ssthresh = 10
Congestion window (segments)
25
20
15
10
5
0
0
3
6
9
12
15
20
22
25
Time (round trips)

Fast retransmit and Fast recovery

- Waiting for TCP sender timeouts leads to idle periods
- Fast retransmit:
  - use duplicate ACKs to trigger retransmission
    - On each arrival of out-of-order packet receiver send one duplicate ack.
    - Sender waits for atleast 3 duplicate acks. Because the packet may be delayed, not lost.
    - Sender sends the lost packet
  - This is a situation when a packet is lost, but latter packets get through
  - Slow start follows timeout
    - timeout occurs when no more packets are getting across
  - Fast retransmit follows duplicate ACKs

Fast retransmit and Fast Recovery

- Fast recovery
  - remove the slow start phase;
  - go directly to half the last successful cwnd

  - ssthresh = min(cwnd, advertised window)/2 (at least 2 MSS)
  - retransmit the missing segment (fast retransmit)
  - cwnd = ssthresh + number of dupacks

  when a new ack comes: cwnd = ssthresh
  - enter congestion avoidance

  - 20% improvement over what could otherwise have been achieved
  - This technique can eliminate close to half of course grained timeouts (but not completely)

TCP Tahoe

- Slow Start, Congestion Avoidance
- Detects congestion using timeouts
- Initialization
  - cwnd initialized to 1;
  - ssthresh initialized to 1/2 MaxWin
- Upon timeout
  - ssthresh = 1/2 cwnd, cwnd = 1
  - enter slow start

TCP Reno

- Fast Retransmit, Fast recovery
- Detects congestion loss using timeouts as well as duplicate ACKs
- On timeout, TCP Reno behaves same as TCP Tahoe
- On fast retransmit
  - skips slow start and goes directly into congestion avoidance phase
  - ssthresh = 1/2 cwnd; cwnd = ssthresh

Topics

- Topics:
  - Congestion Avoidance
  - Router based congestion avoidance
    - RED, (DecBIT)
  - Source based congestion avoidance
  - TCP Vegas
- Books: Peterson/Davie

Congestion Avoidance

- We have seen TCP's approach:
  - detect congestion after it happens
  - increase load trying to maximize utilization until loss occurs
- Alternatively:
  - we can try to predict congestion and reduce rate before loss occurs
  - this is called **congestion avoidance**
- Two approaches for Congestion Avoidance :
  - Source based congestion avoidance
    - TCP Vegas
  - Router based congestion avoidance
    - RED, DECbit

TCP Vegas

Increasing RTT

- We have seen that Timeout is considered as indication of congested state.
- RTT is a good measure of load in network.
- Increment in RTT is a indicator of increased load in bottleneck router, which will soon become a ‘congested state’
- Sending packets in loaded state will probably only increase queue length in the bottleneck router.

♦ TCP Vegas

- controls it’s window size based achieved sending rate
- It measures data rate for a certain RTT, where this RTT is that value of RTT when flow is not congested. This data rate is considered as expected data rate for next measurement.
- If for the next RTT cycle for a packet, if this rate is falling then reduce the congestion window size, else if it is increasing, increase the congestion window size.

TCP Vegas

BaseRTT

- Define *BaseRTT* to be the minimum of all RTTs when flow is not congested. That is when all acks returns within a round trip time.
- Initially it is set to the RTT of first packet

We assume that we are not overflowing router queue---

- Compute ***ExpectedRate = Current CongestionWindow/BaseRTT***

Next we see that if this rate is achieved in next RTT period--

- Compute current sending rate ActualRate with a distinguished packet and measure the RTT of this packet
  - Record sending time of a distinguished packet and then count how many bytes have send to network by TCP till the acknowledgement of this distinguished packet returns.
- Computer ***Diff = ExpectedRate - ActualRate ,***
- Change RTT (Diff < 0)
  - If ActualRate > ExpectedRate , BaseRTT changes to NewRTT, cwnd increases linearly(?)
- If ActualRate < ExpectedRate, (Diff >=0) Change Congestion Window as follows:
- Define $\alpha < \beta$, corresponding to too little or too much change
  - If Diff < α increase cwnd linearly in next RTT
  - If Diff > β decrease cwnd linearly in next RTT
  - If α <Diff, < β , don’t do anything.

TCP Vegas (contd)

- Choice of α and β :
  - When actual rate is close to expected rate, that is diff is on the left side of α (say x), TCP can put (α -x) more packets to network.That means, we become optimistic about network for extra (α -x) packets. It is expected that these extra bytes can be accomodated by router queue and will not lead to dropping situation.
  - When actual rate is far below expected rate, that is diff is on the right side of β, then congestion window should decrease to avoid congestion build up.
  - When actual rate is only (β - α), β > α, far from expected rate , we do not remain optimistic about network. The network is already accommodating β - α extra packets.
  - Example:
  - On a connection with BaseRTT of 100 ms, packet size of 1 KB, α = 3 KB and β= 6 Kb , means the connection can occupy 3 extra buffers of network, but should not occupy more than 6 extra buffer in next RTO. In practice, β=3 and α= 1

- Linear decrease with fall of rate is not conflicting with multiplicative decrease. Multiplicative decrease happens when timeout occurs.

Router congestion notification

- Router has unified view of queuing behavior
- Routers can distinguish between transient and persistent queuing delays
- Routers can inform source of congestion by explicit on implicit mechanisms
  - DECbit: explicit way by setting congestion bit on in packets from source
  - RED: implicitly by dropping packet. It is developed in conjunction with TCP

Random early detection (RED)

- Observation:
  - transient congestion per RTT time
  - TCP detects congestion from loss - after queues are exhausted in Drop tail scheme. Single source may suffer. Also, allowing queue to build up to its fullest extent increases end to end delay.
  - Better to drop before the congestion builds up and implicitly notify user to slow down
  - What the criterions for when to drop?
- Aim:
  - keep throughput high and delay low, do not reach to the limits of queue and drop indiscriminately like drop tail mechanism.
  - Keep average queue length
  - accommodate bursts, do not drop for transient queue build up

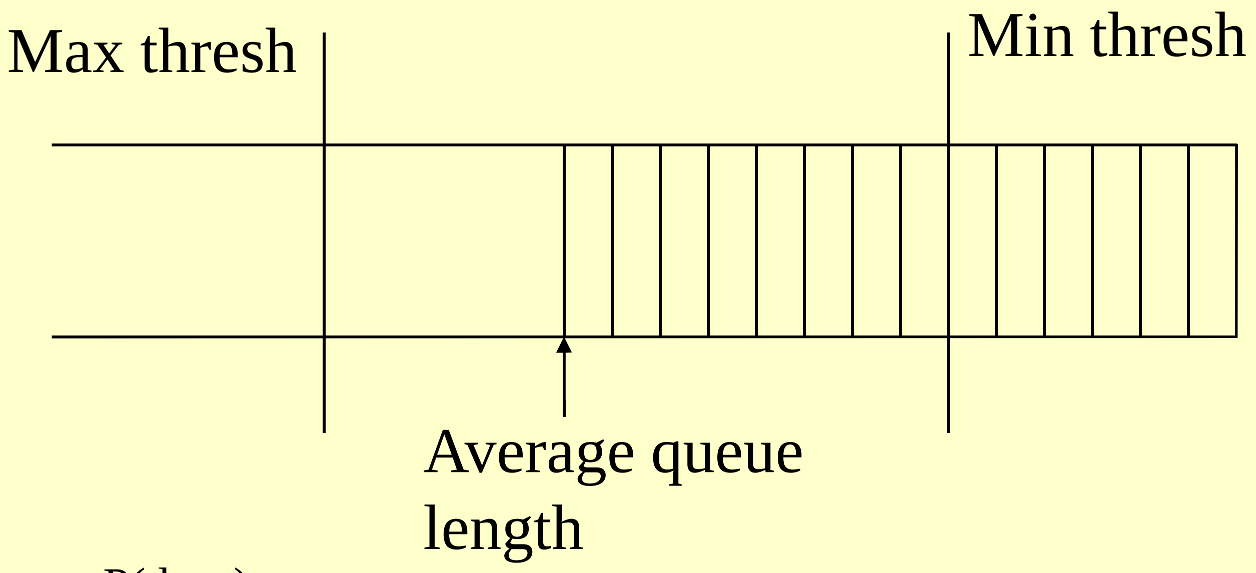
Why we need active queue management (rfc2309)

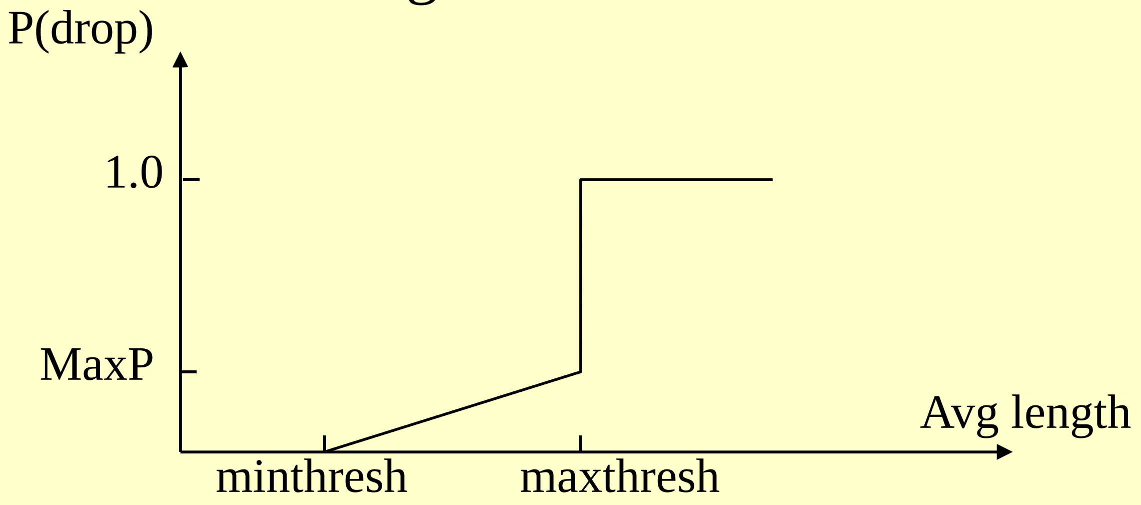
- Simple way is drop tail.
- Lock-out problem
  - drop-tail allows a few flows to monopolize the queue space, locking out other flows.
- Full queues problem:
  - drop tail maintains full or nearly-full queues during congestion;
    - Increased delay
    - queue length limits should reflect the size of bursts we want to absorb, not steady-state queuing

RED algorithm

- Maintain running average of queue length
- if avg < min_{th} do nothing
  - low queuing, send packets through
- if avg > max_{th}, drop packet
  - protection from misbehaving sources
- else drop packet with a probability proportional to queue length
  - notify sources of incipient congestion

RED operation





RED algorithm

- Maintain weighted running average of queue length
  - avgLen= (1-w_q) avgLen + w_qSampleLen
- For each packet arrival
  - calculate average queue size avg
  - if min_{th} <= avg < max_{th}
    - calculate probability Pa
      - drop the arriving packet with probability Pa
  - else if max_{th} <= avg
      - mark the arriving packet

Packet dropping probability

- Dropping probability is calculated in way that it depends on queue length and the number of packets not dropped since last packet drop
- dropping probability based on queue length
  - $Pb = max_p * (avg - min_{th}) / (max_{th} - min_{th})$
- Just marking based on Pb can lead to clustered dropping. (can cause multiple drop in a single connection, which is unfair, similar to tail drop). Dropping should be evenly distributed over timeline.

- Better to bias Pb by history of undropped packets
  - $Pb = Pb/(1 - count*Pb)$, where count the number of packets not dropped since the last drop.

Parameters

- Standard weighted running average: avgLen= (1-w_q) avgLen + w_qSampleLen
  - Can filter out the short term, bursty changes and can detect the long term sustained congestions more appropriately.
- Min threshold (min_{th})and and Max threshold (max_{th}):
  - If the traffic is fairly bursty, then, min_{th} should be larger to maintain link utilization at acceptable high rate
  - The difference between the two threshold should be large enough to accommodate typical increase in average queue length in one RTT.
  - Setting max twice of min is seen to be good for today’s internet.

- w_q :
  - Should be chosen in a way that it filters out changes happen one RTT.

Improvements

- Drop Tail
- Early random drop [Hashem 89]
  - Whenever aggregate queue length exceeds a cartain level, each arriving packet is dropped with certain probability. Intuition is that misbehaving sources are sending more packets, so dropping a packet with randomly probably drops more packets from the misbehaving source
- RED [Sally Floyed 93]:
  - Packet drops probability is function of queue length rather than instateneous queue length. This allows small burst to pass through. It drops packet only during sustained overheads.
  - Packet drop probability is linear function of mean queue length. Drop probability increases slowly as mean queue length increases. This prevents severe reaction to mild overload (as with early random drop)
  - Substantially improves the performance of network with co-operative TCP flows, and that the probability that a connection loses packets is roughly proportional to it’s actual throughput share.
  - Does not have a bias against bursty source like Drop tail. This is important for a TCP connection as they send packets is burst when they start.