

A decorative graphic consisting of a vertical black line and a horizontal grey line intersecting at a point to the left of the text.

EECS 122: Introduction to Communication Networks

Unit 19: UDP/TCP



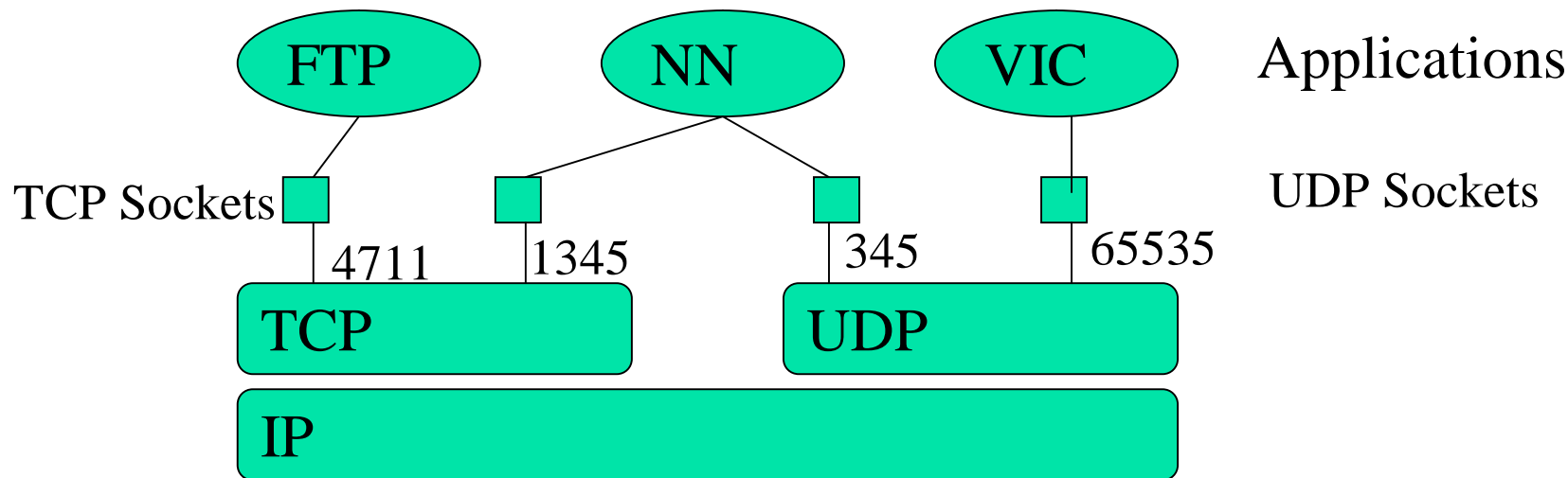
UDP

Introduction

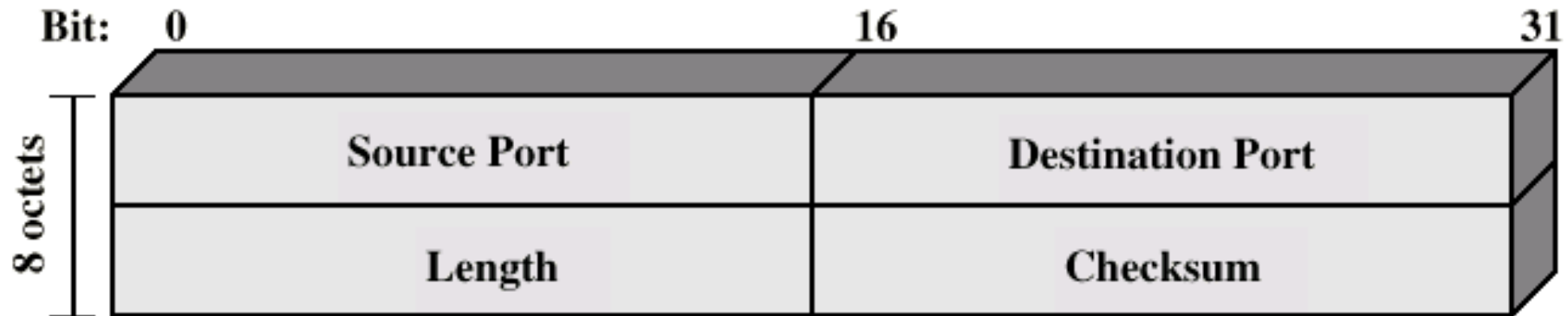
- Network Layer (IP)
 - RFC 791
 - Connection less
 - Packets may be lost, delivered out of order or duplicated
 - Variable delay
- Transport Layer
 - UDP
 - Connection-less
 - RFC 768
 - TCP
 - Connection oriented, reliable
 - RFC 793 (and many more, see draft-ietf-tcpm-roadmap-01.txt)

UDP

- Why is there a second unreliable protocol on top of IP?
- Why should one use an unreliable protocol at all?
- Addressing of applications
 - IP is used to address an interface (host)
 - Protocol identifier of IP header is used to select receiving protocol
 - Ports are used to select the communication end-point (application)



UDP (II)



ÉEnd-to-End Checksum (optional)

ÉTotal length field (redundant, since IP has a length field, too)


ÉMax. total length = 65535 ó 8 ó IP Header length

ÉEach user request is transferred using a single datagram

ÉUDP provides no send buffer but a receive buffer

UDP Checksum

- Ones complement of 16 bit words (as IP)
- Covers header and data plus a 12 byte pseudo header
 - IP addresses, 0, protocol identifier, length
 - Make sure that packet has reached the correct host
- Pad byte in case of an odd packet length (not transmitted)
- Optional
- Receiver has to verify the checksum



TCP

Acknowledgement:

- Most information from the seminal book of Stevens, some slides from the Book of Kurose-Ross.*
- Some slides taken over from Eitan Modiano (MIT)*

Structure

- Where does it fit
- Connection management
- Fields in header
- Byte stream – sequence numbers
- Send events
- Receive events/ACK generation
- Error control – a variant of G- Back _ n
- Timer computation
- Fast retransmit /SACK
- Flow control : when to transmit a segment
- Congestion control
- Throughput

Connection – oriented, Connectionless

- Interface between station and network node
 - Connection oriented
 - User request a reliable service, in order...no duplications
 - Data streams – thins of the conn-oriented socket interface
 - Connectionless
 - Requests handled independently
 - Unreliable transmission, order of delivery not sure, duplications...

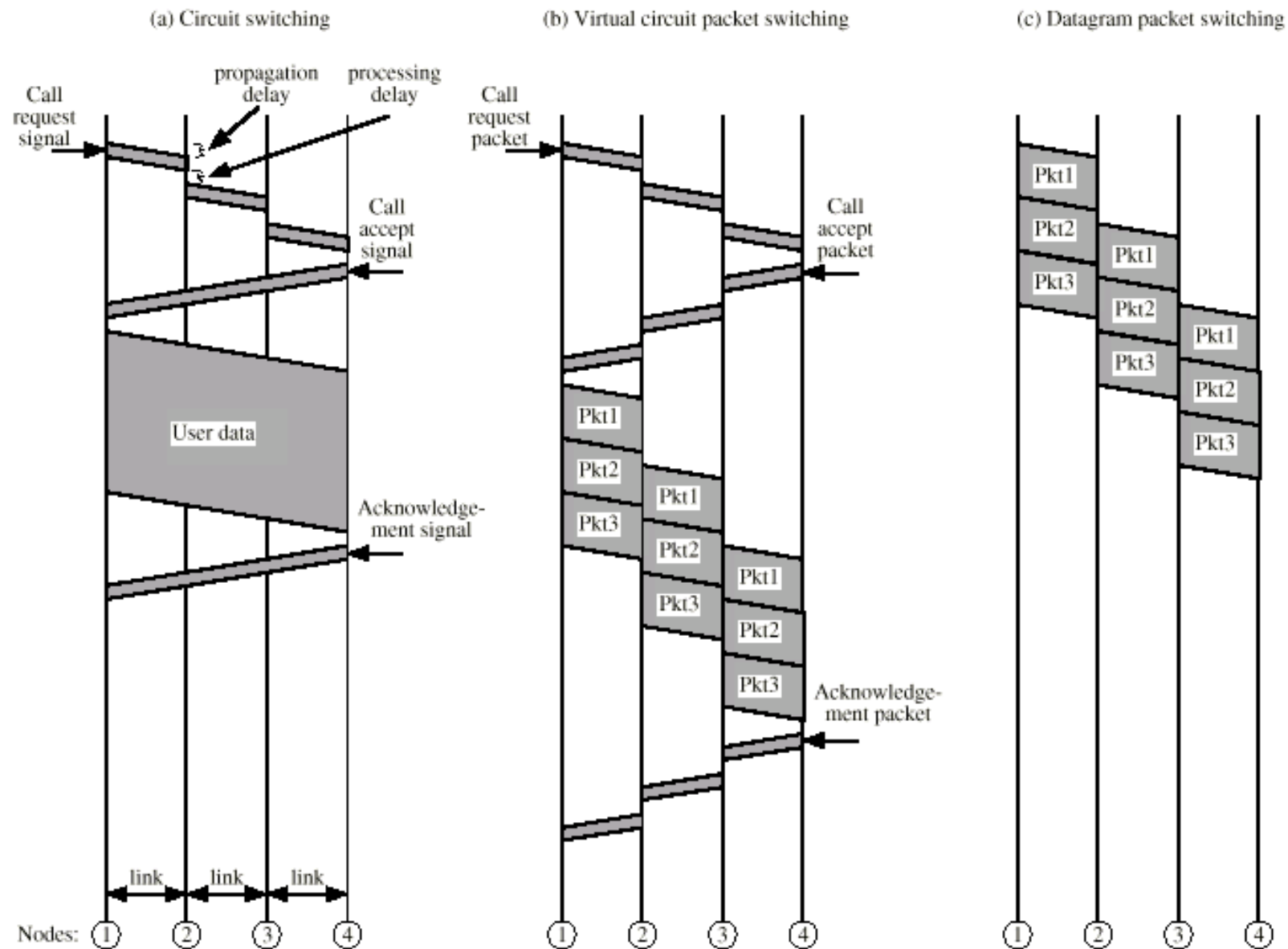
Connection oriented **SERVICE** – how to ?

- In Packet switched networks connection oriented service can – naturally – be offered on top of the Virtual Circuit switching

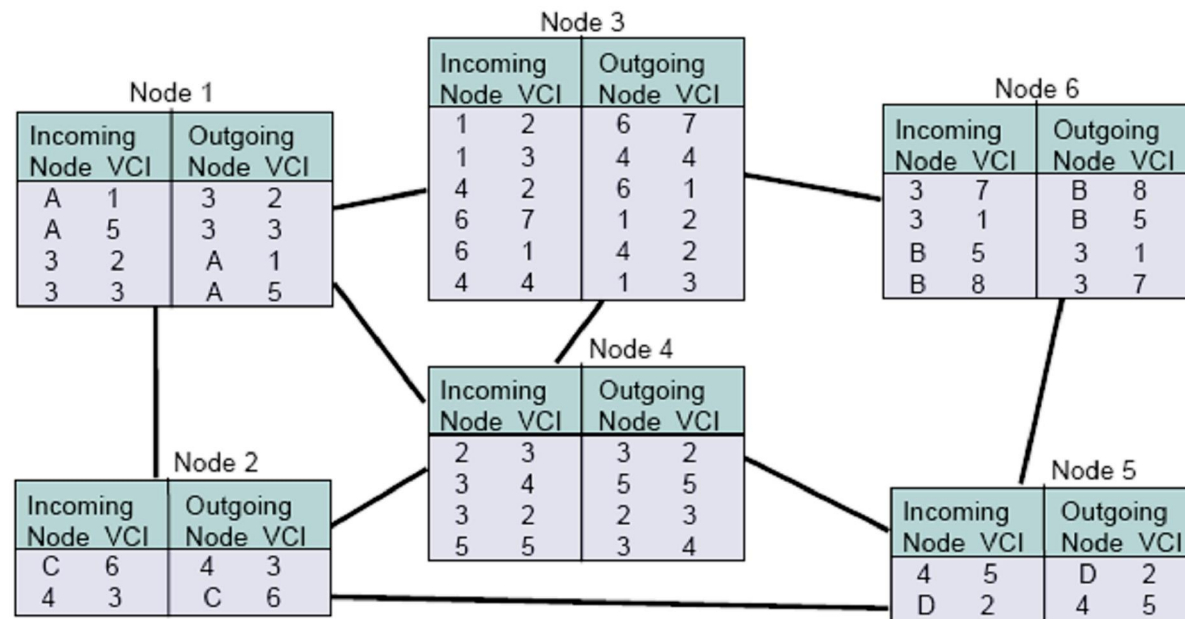
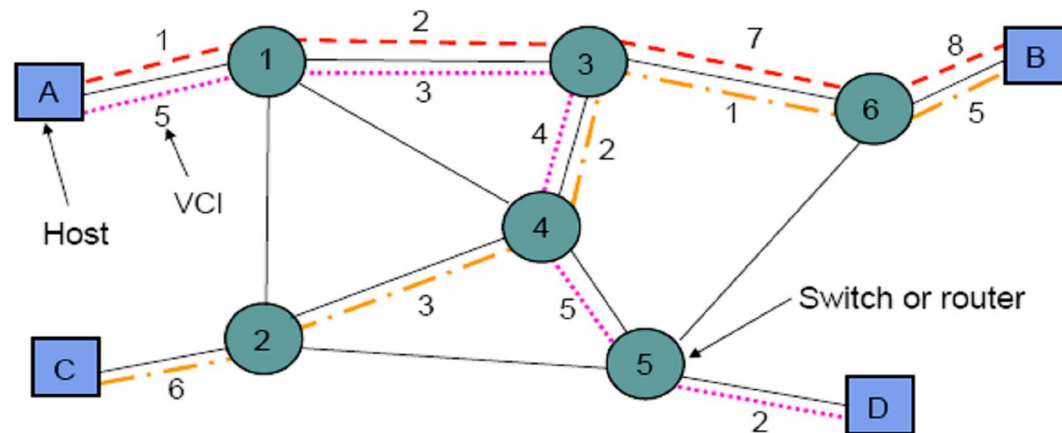
Reminder: Virtual Circuit Packet Switching

- The idea is to combine the advantages of circuit switching with the advantages of datagram switching
- Connection has to be set-up and released:
 - Within connection setup a short (compared to full addresses) connection identifier is assigned per Switch per VC.
 - During the connection set-up resources can be reserved
 - Only the set of ACTIVE VCs has to be searched during forwarding of incoming packets – much smaller table size!
 - This reduces the per packet forwarding processing! – nice for high speed links...
 - **There is, however, state per switch per VC!** Doesn't handle switch crashes well: have to teardown and reinitiate a new circuit

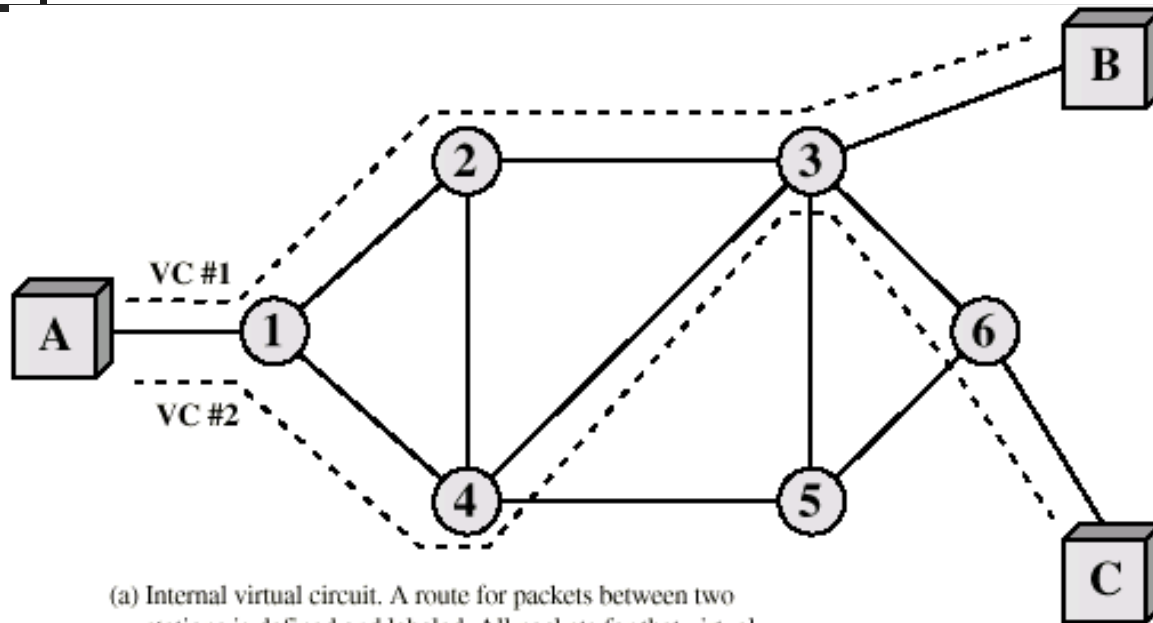
Event Timing



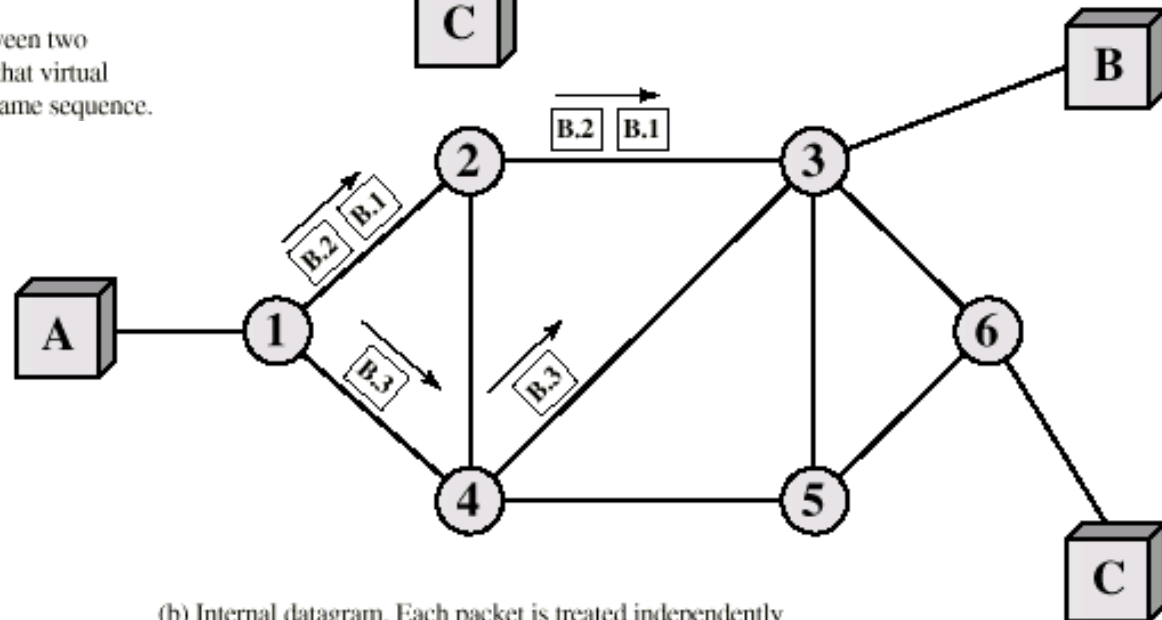
How does forwarding work in VCs?... (Garcia, Ch7)



Virtual Circuit and Datagram Operation



(a) Internal virtual circuit. A route for packets between two stations is defined and labeled. All packets for that virtual circuit follow the same route and arrive in the same sequence.



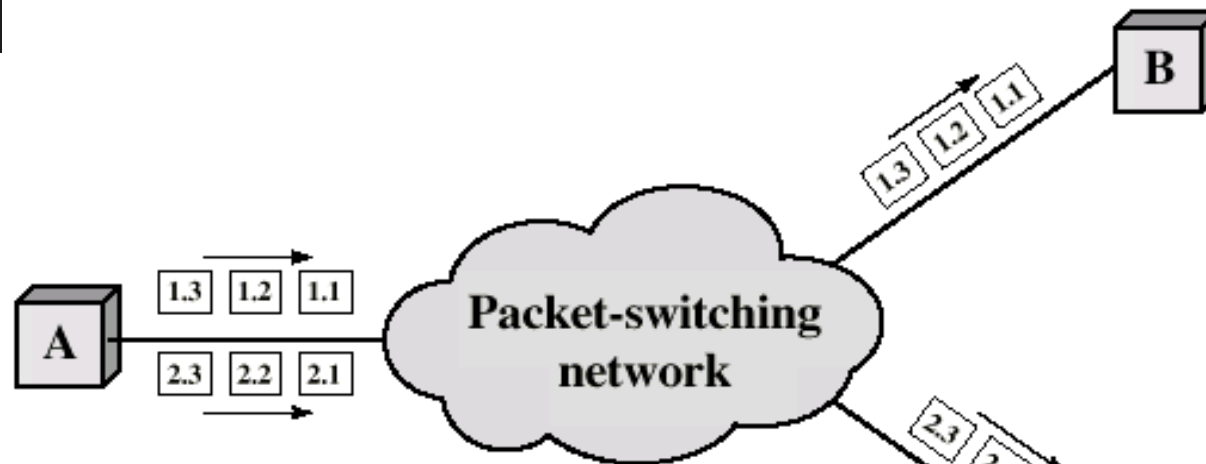
(b) Internal datagram. Each packet is treated independently by the network. Packets are labeled with a destination address and may arrive at the destination node out of sequence.

Connection oriented **SERVICE** – how to ?

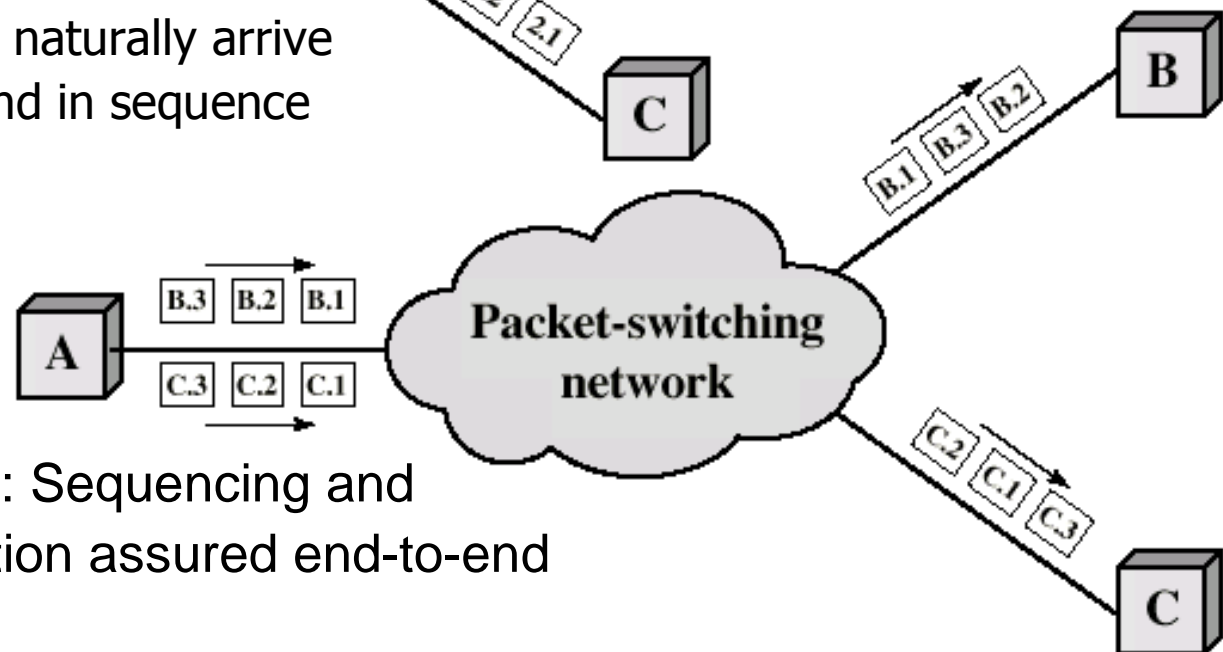
- In Packet switched networks connection oriented service can – naturally – be offered on top of the Virtual Circuit switching
- Connection oriented service can, however, be also offered on top of DATAGRAM switching
 - Connection will be established using datagram's handshake
 - Loss free operation, in-sequence packet delivery will be provided by end-to-end mechanisms between entities establishing the connection
 - Examples **TCP over IP**

Note: Connection oriented . vs. Connectionless pertains to the SERVICE
Virtual Circuit/datagram pertains to the INTERNAL PROTOCOL OPERATION

Connection Oriented with VCs, and with Datagrams



(a) VC used: Packet naturally arrive without losses and in sequence



(b) Datagram used: Sequencing and loss-less operation assured end-to-end

Basic TCP Operation

- **At sender**

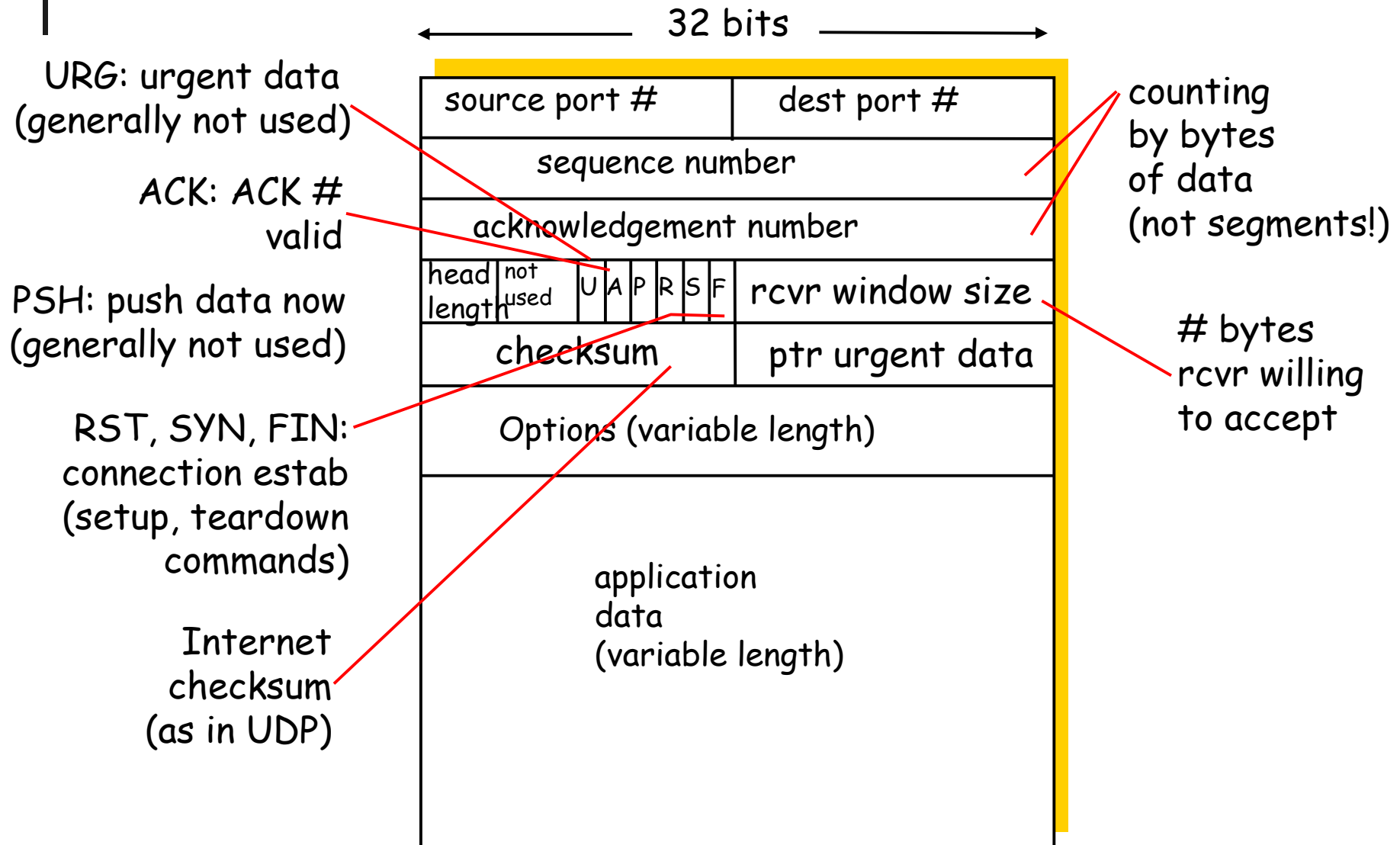
- Application data is broken into TCP segments
- TCP uses a timer while waiting for an ACK of every packet
- Un-ACK'd packets are retransmitted

- **At receiver**

- Errors are detected using a checksum
- Correctly received data is acknowledged
- Segments are reassembled into their proper order
- Duplicate segments are discarded

- Window based retransmission and flow control + Congestion Control...

TCP segment structure

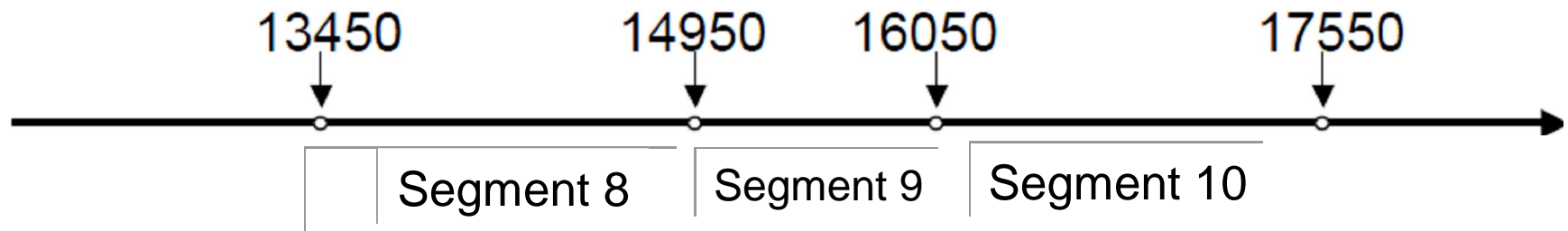


TCP Header Fields

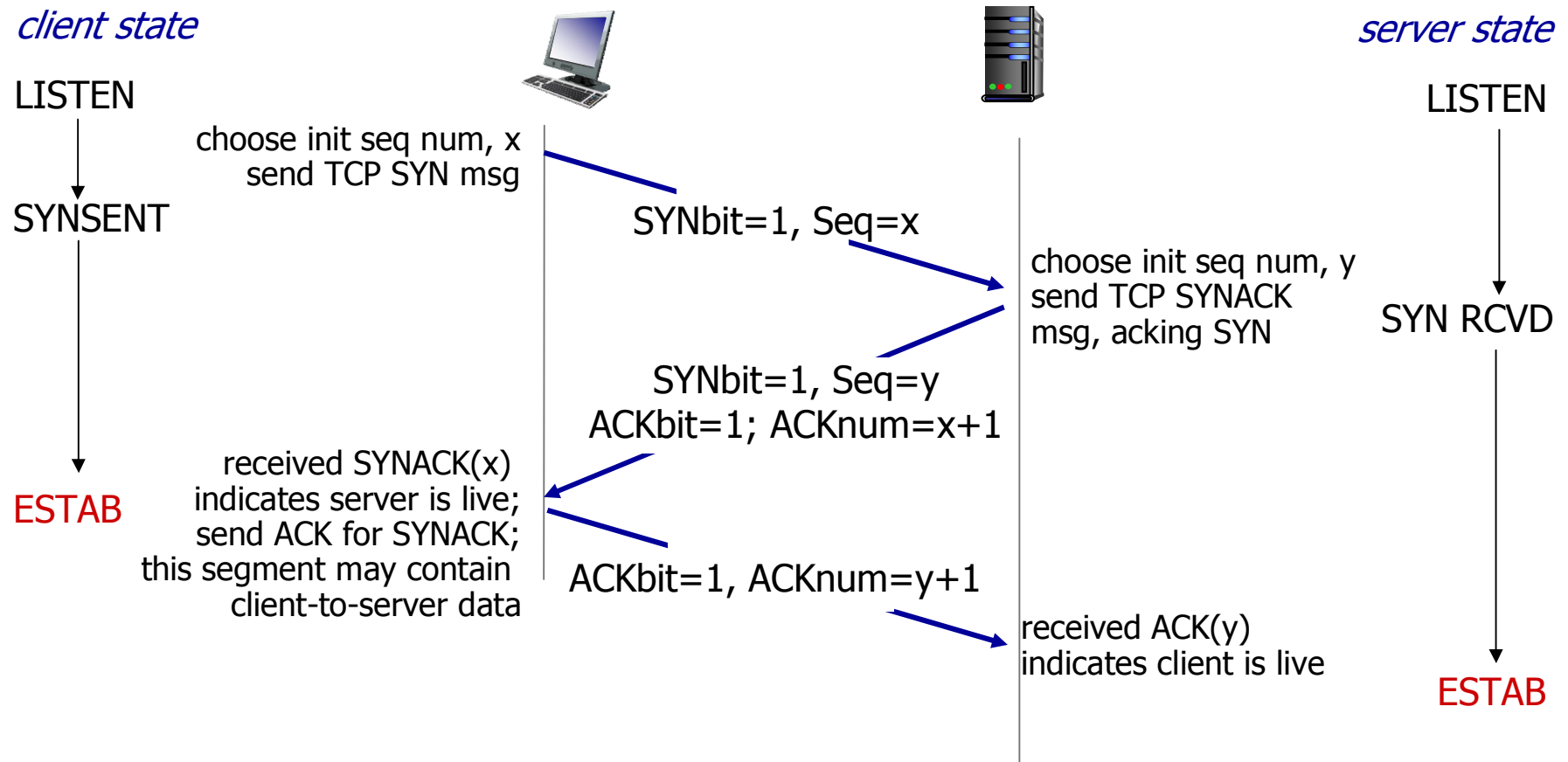
- Port Numbers: Like for UDP
- 32 bit SN uniquely identify the application data contained in the TCP segment
 - SN is in bytes!
 - It identify the first byte of data
- 32 bit RN is used for piggybacking ACK's
 - RN indicates the next byte that the receiver is expecting
 - Implicit ACK for all of the bytes up to that point
- Data offset is a *header length* in 32 bit words (minimum 20 bytes)
- Window size
 - Used for error recovery (ARQ) and as a flow control mechanism
Sender cannot have more than a window of packets in the network simultaneously
 - Specified in bytes
Window scaling used to increase the window size in high speed networks
- Checksum covers the header and data

Sequence Numbers in TCP

- TCP regards data as a “byte-stream”
 - each byte in byte stream is numbered.
- 32 bit value, wraps around
 - initial values selected at start up time
- TCP breaks up byte stream in packets
 - Packet size is limited to the Maximum Segment Size (MSS)
- Each packet has a sequence number
 - seq. no of 1st byte indicates where it fits in the byte stream
- TCP connection is duplex
 - data in each direction has its own sequence numbers



TCP 3-way handshake



TCP Connection Management (1)

Three way handshake:

Step 1: client end system sends TCP SYN control segment to server

- specifies initial seq #
- specifies initial window #

Step 2: server end system receives SYN, replies with SYNACK control segment

- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #
- specifies initial window #

Step 3: client system receives SYNACK

TCP: closing a connection

- client, server each close their side of connection
 - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
 - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: closing a connection

client state

ESTAB

`clientSocket.close()`

FIN_WAIT_1

can no longer
send but can
receive data

FIN_WAIT_2

wait for server
close

TIMED_WAIT

timed wait
for $2 * \text{max}$
segment lifetime

CLOSED



server state

ESTAB

CLOSE_WAIT

LAST_ACK

CLOSED

FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

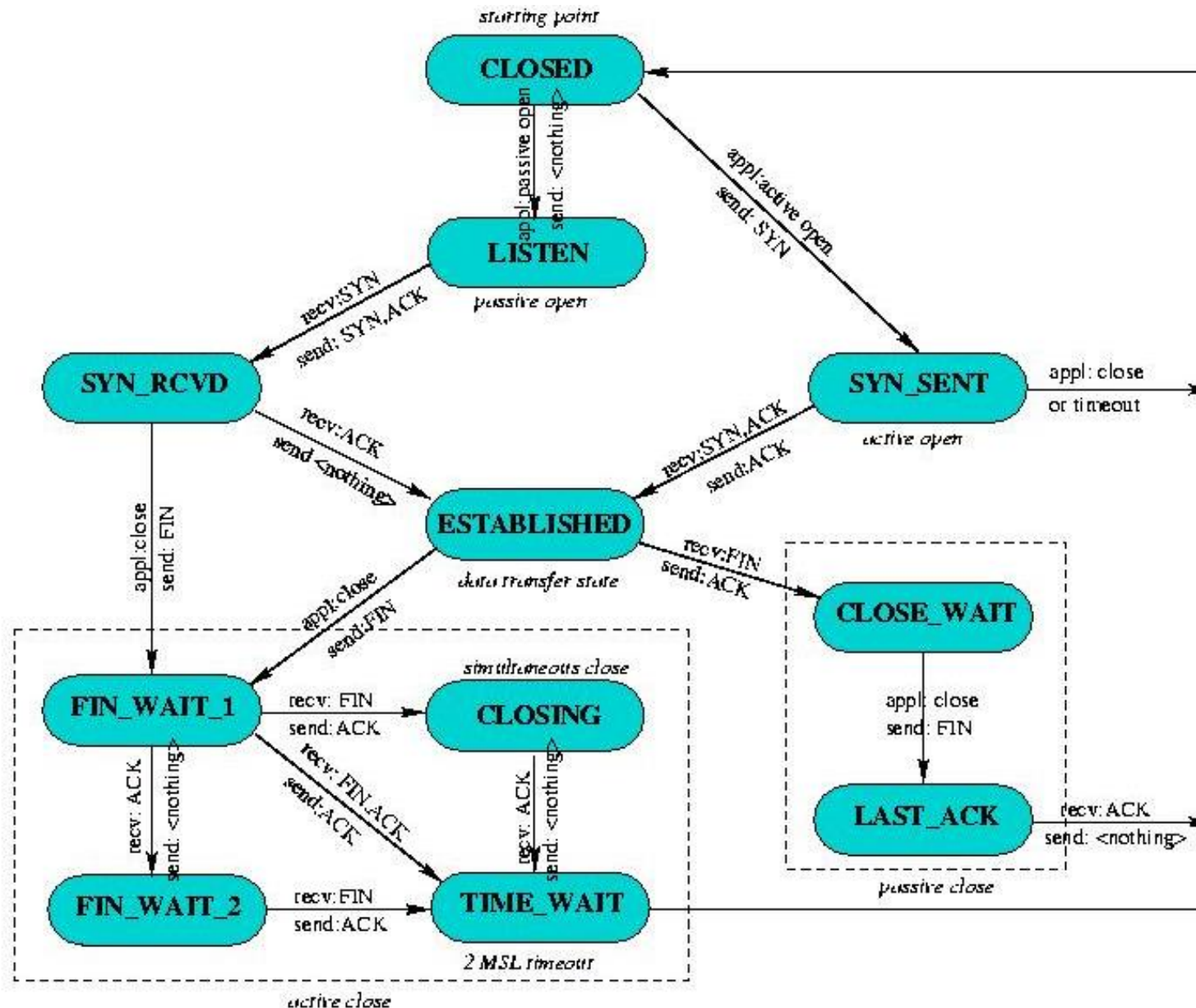
FINbit=1, seq=y

ACKbit=1; ACKnum=y+1

can still
send data

can no longer
send data

TCP state machine



Error Control: A variation of Go-Back_N

- Sliding window with cumulative ACKs
 - Receiver can only return a single “ack” sequence number to the sender
 - Acknowledges all bytes with a lower sequence number
 - Starting point for retransmission
 - Duplicate ACKs sent when out-of-order packet received
- Sender only retransmits a single packet at a time
 - Optimistic assumption: only one that it knows is lost
 - Network is congested \Rightarrow shouldn't overload it
- Error control is based on byte sequences, not packets
 - Retransmitted packet can be different from the original lost packet (e.g., due to fragmentation)

TCP Sender Events

- Data received from application:
 - Create segment with sequence number
 - Sequence number is byte- stream number of first data byte in segment
 - start timer if not already running
 - Think of timer as for oldest un-acknowledged segment*
 - Timer expiration interval: time-out
- Timeout:
 - retransmit segment that caused timeout
 - restart timer
- ACK received:
 - It acknowledges previously unACKed segments
 - update what is known to be ACKed
 - start timer if there are outstanding segments

Fast Retransmit

- When TCP receives a packet with a SN that is greater than the expected SN, it sends an ACK packet with a request number of the expected packet SN
 - This could be due to out-of-order delivery or packet loss
- If a packet is lost then duplicate RNs will be sent by TCP until the packet is correctly received
 - But the packet will not be retransmitted until a Timeout occurs
 - This leads to added delay and inefficiency
- Fast retransmit assumes that if 3 duplicate RNs are received by the sending module that the packet was lost
 - After 3 duplicate RNs are received the packet is retransmitted
 - After retransmission, continue to send new data
- Fast retransmit allows TCP retransmission to behave more like Selective Repeat ARQ

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq # . All data up to expected seq # already ACKed	Delayed ACK . Wait up to 500ms for next segment. If no next segment, send ACK (reduces ACK traffic)
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK , ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK , indicating seq. # of next expected byte (trigger fast retransmit)
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

(SACK)

- **Option** for selective ACKs (SACK) also widely deployed
- Selective acknowledgement (SACK) essentially adds a bitmask of packets received
 - Implemented as a TCP option (extended TCP header)
 - Encoded as a set of received byte ranges (max of 3 or 4 ranges)
- When to retransmit?
 - Packets may experience different delays
 - Still need to deal with reordering
 - Wait for out of order by 3 packets

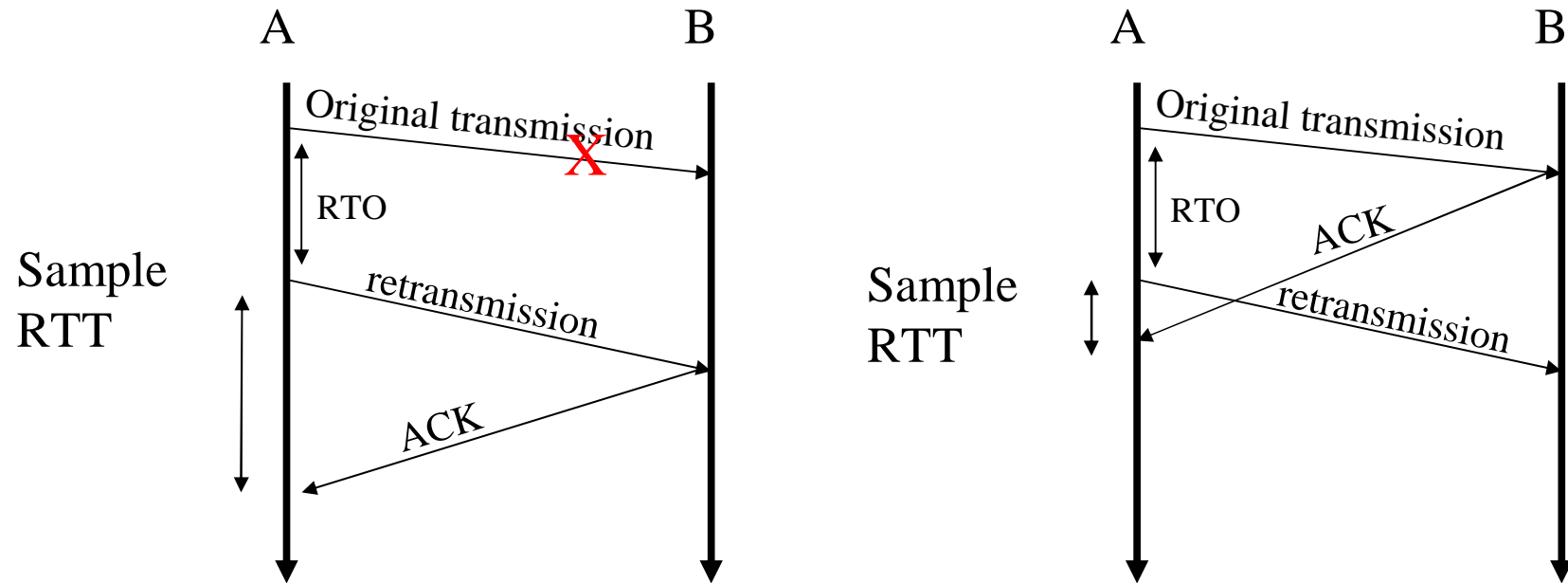
TCP Retransmission Timeout

- TCP uses one timer only
- Retransmission Timeout (RTO) calculated dynamically
 - Based on Round Trip Time estimation (RTT)
 - Wait at least one RTT before retransmitting
 - Importance of accurate RTT estimators:
 - Low RTT → unneeded retransmissions
 - High RTT → poor throughput
 - RTT estimator must adapt to change in RTT
 - But not too fast, or too slow!
 - Spurious timeouts
 - “Conservation of packets” principle – more than a window worth of packets in flight

Retransmission Timeout Estimator

- Round trip times exponentially averaged:
 - $\text{New RTT} = \alpha (\text{old RTT}) + (1 - \alpha) (\text{new sample})$
 - *0.875 for most TCP's*
- Retransmit timer set to β RTT, *where $\beta = 2$ (usually!)*
 - Every time timer expires, **RTO exponentially backed-off**
- Key observation: At high loads round trip variance is high
- Solution (currently in use):
 - Considers: Base RTO on RTT and standard deviation of RTT:
 $\text{RTT} = \text{New RTT} + 4 * \text{rttvar}$
 - $\text{rttvar} = \chi * \text{dev} + (1 - \chi) \text{rttvar}$
 - dev = linear deviation (also referred to as mean deviation)
 - Inappropriately named – actually smoothed linear deviation
 - RTO is discretized into ticks of 500ms ($\text{RTO} \geq 2\text{ticks}$)

Retransmission Ambiguity



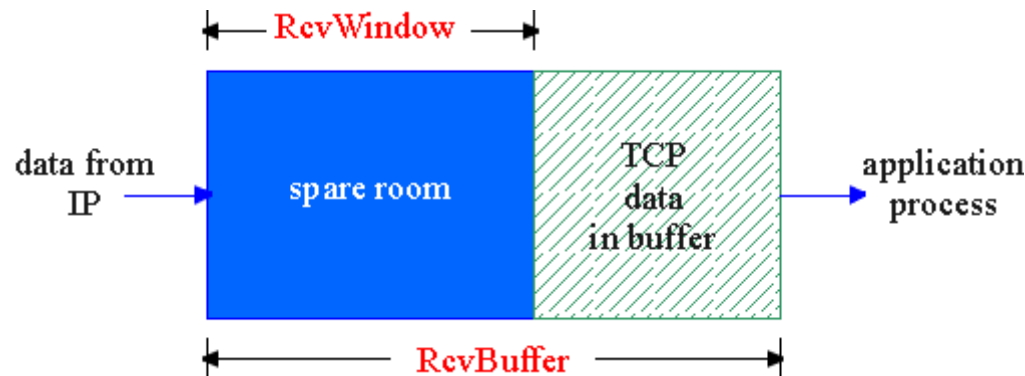
- Karn's RTT Estimator

- If a segment has been retransmitted: Don't count RTT sample on ACKs for this segment
- Double the timeout for next packet!
- Reuse RTT estimate only after one successful transmission

TCP Flow Control: sliding window protocol

TCP is a sliding window protocol

- For window size n , can send up to n bytes without receiving an acknowledgement
- When the data is acknowledged then the window slides forward



sender won't overrun receiver's buffers by transmitting too much, too fast

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

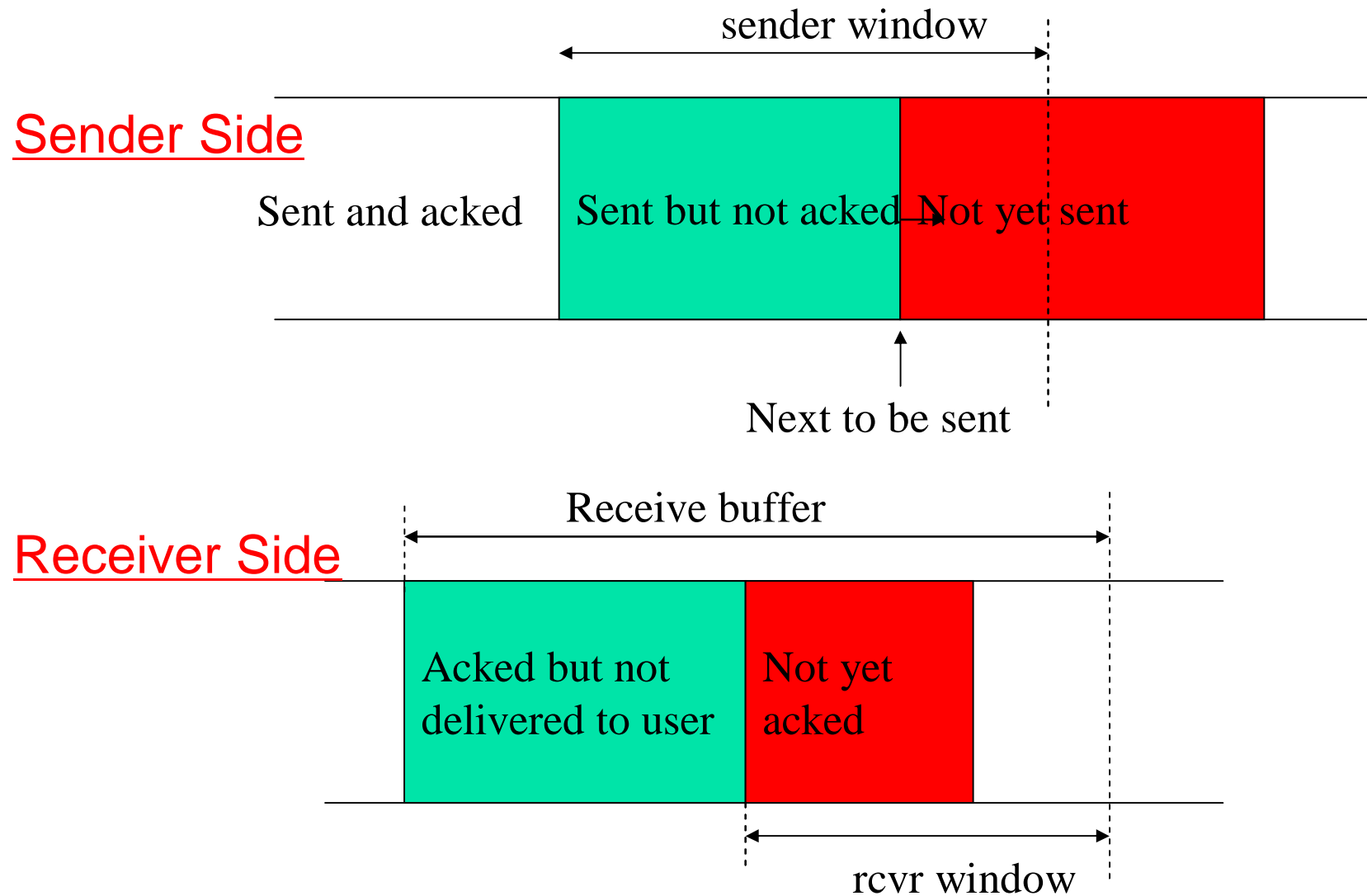
- **rcvr window size** field in TCP segment

sender: amount of transmitted, unACKed data less than most recently-receiver

rcvr window size

(receiver limited operation)

Window Flow Control:



Silly Window Syndrome

- Problem:
 - Receiver opens window a small amount
 - ACK opens $K < \text{MSS}$ bytes (very small amount of data)
- Should sender transmit K bytes?
 - Can be very inefficient as most of the packet will contain header overhead
- If sender is aggressive, sending available window size
 - Results in "silly window syndrome"
 - Small segment size remains indefinitely - very inefficient

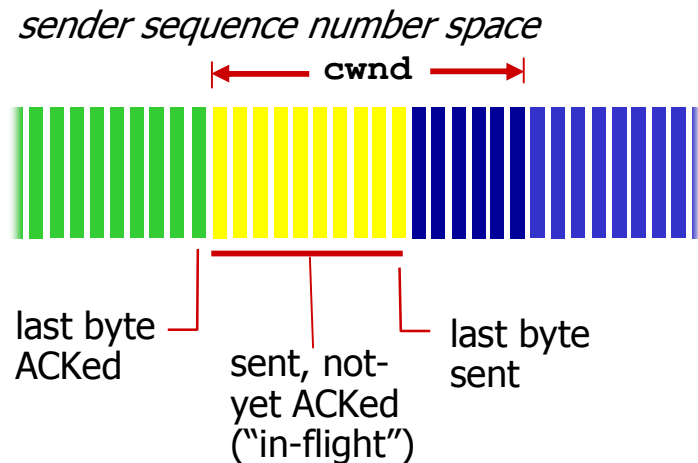
Note that when the receiver receives the small segment, it sends back an ACK (for that small segment), opening the window for another small segment

- Hence a problem when either sender transmits a small segment or receiver opens window a small amount
- Mechanism needed to wait for opportunity for sending larger amount of data.

When to transmit: Nagle Algorithm

- Waiting too long hurt interactive applications (Telnet)
 - Without waiting, risk of sending a bunch of tiny packets
 - silly window syndrome
 - Nagle's Algorithm:
 - Continue to buffer data if some un-acknowledged packets still outstanding
 - If no outstanding data, send segment without delay
 - If more than MSS worth of data, send segment without delay
- a/ Additional implementation details:
- Receiver update of advertise window: avoid small increases in window size
=> avoid very tiny send opportunities
- Applications can disable Nagle's algorithm to avoid long delays
- b/ Implication: if don't have at least MSS worth of data, wait at least one RTT before transmitting new segment:
- TCP's self clocking mechanism

TCP Congestion Control: details



- sender limits transmission:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

- **cwnd** is dynamic, function of perceived network congestion
- BUT CAN YOU JUST START WITH SOME cwnd?

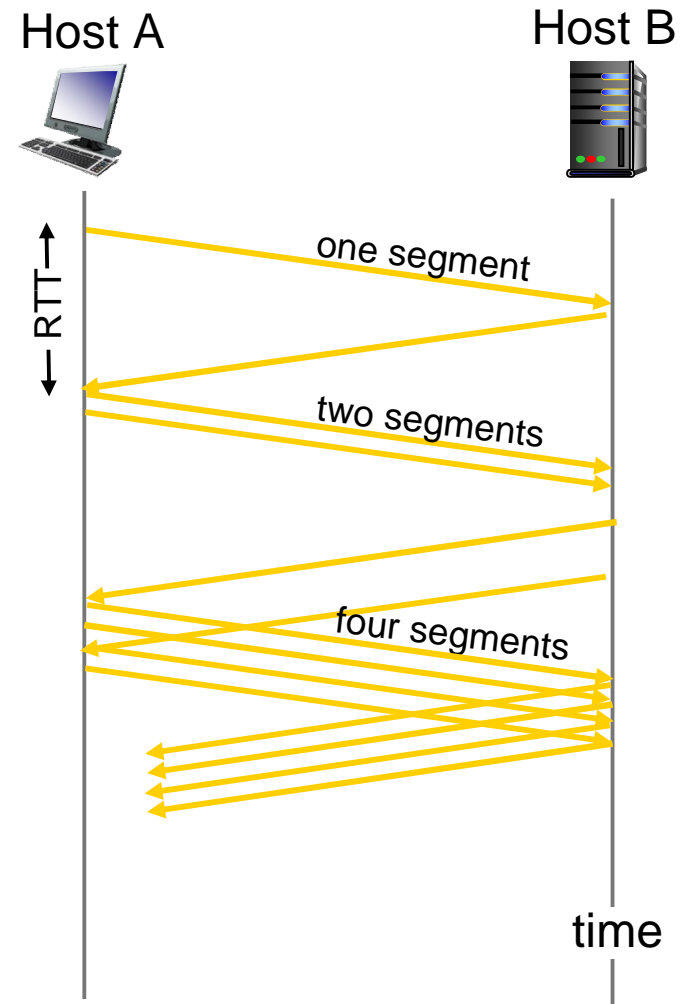
TCP sending rate:

- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
 - initially `cwnd` = 1 MSS
 - double `cwnd` every RTT
 - done by incrementing `cwnd` for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



TCP: detecting, reacting to loss

- loss indicated by timeout:
 - `cwnd` set to 1 MSS;
 - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: TCP RENO
 - dup ACKs indicate network capable of delivering some segments
 - `cwnd` is cut in half window then grows linearly
- TCP Tahoe always sets `cwnd` to 1 (timeout or 3 duplicate acks)

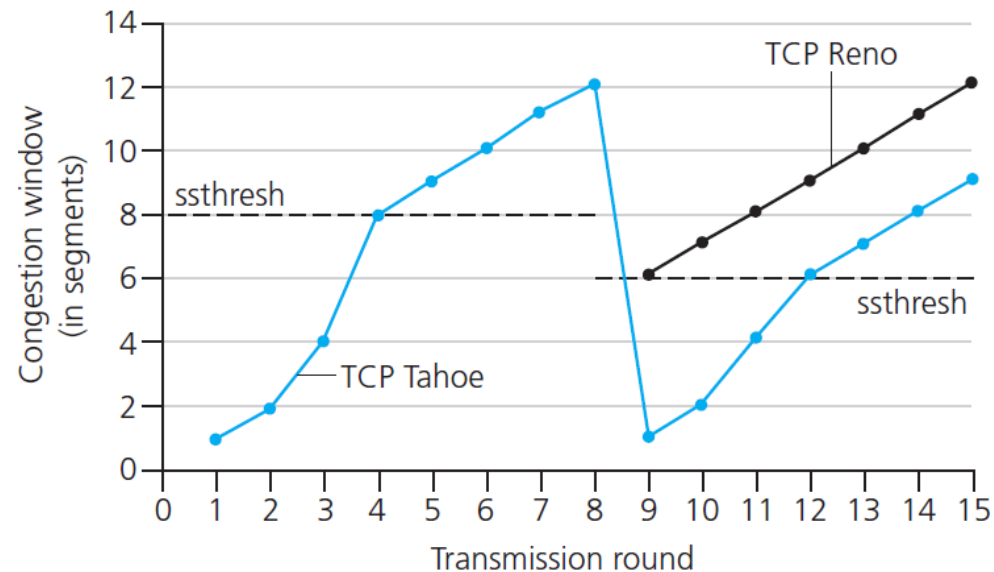
TCP: from slow start to Congestion Avoidance

Q: when should the exponential increase switch to linear?

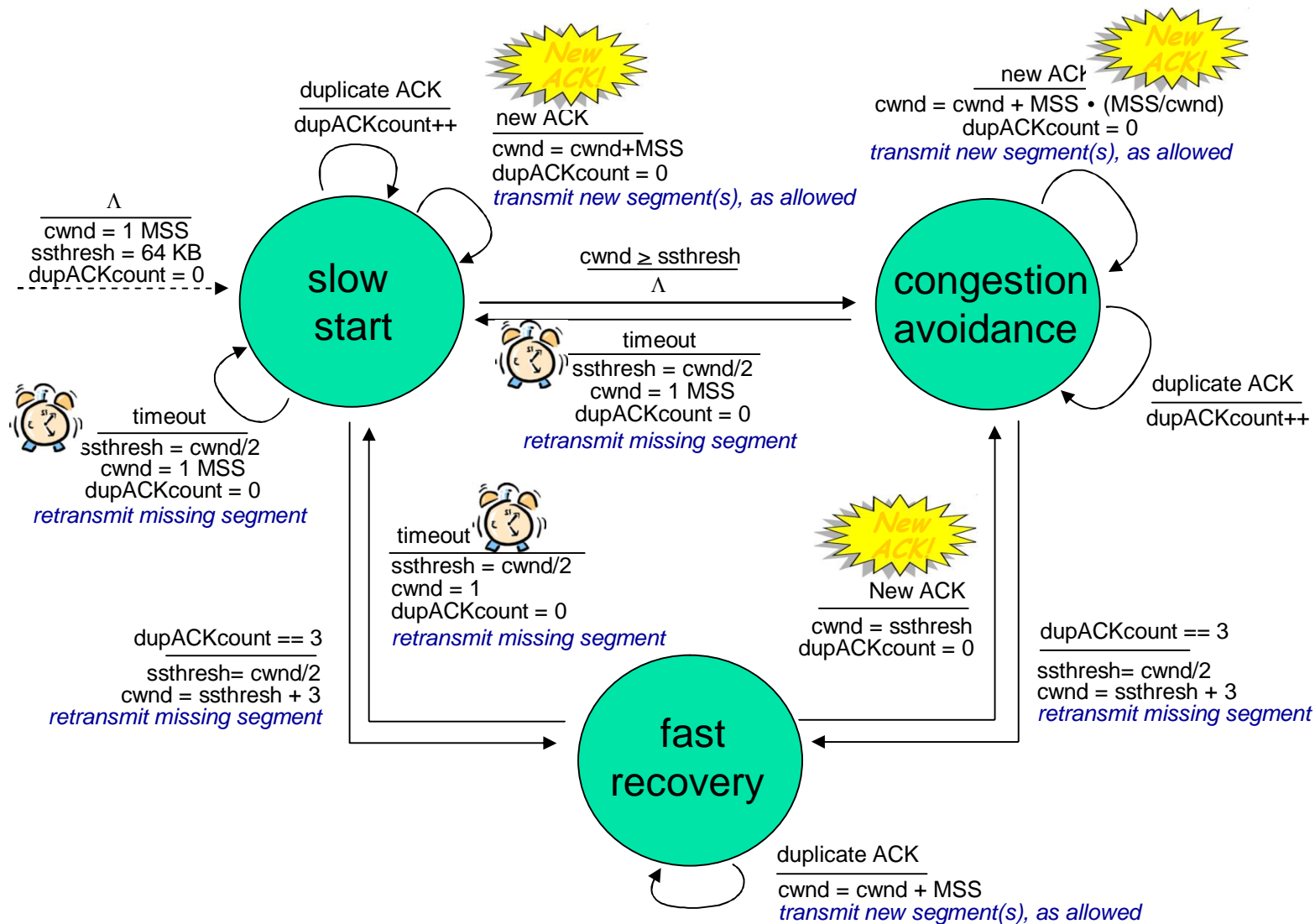
A: when **cwnd** gets to 1/2 of its value before timeout.

Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event



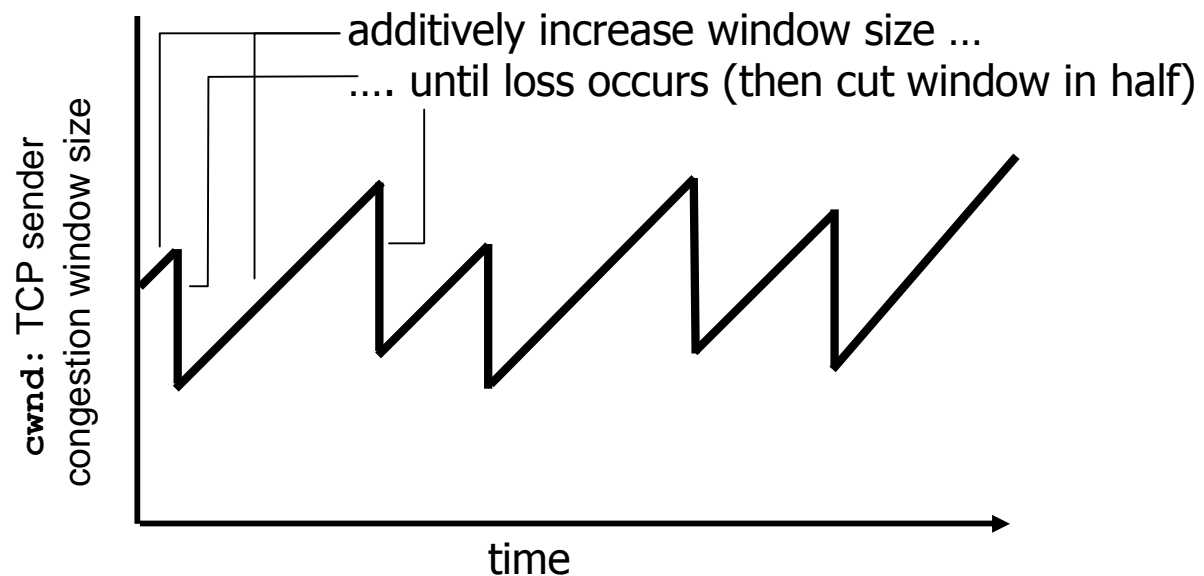
Summary: TCP Congestion Control



TCP congestion avoidance: AIMD

- ❖ *approach*: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
 - *additive increase*: increase **cwnd** by 1 MSS every RTT until loss detected
 - *multiplicative decrease*: cut **cwnd** in half after loss

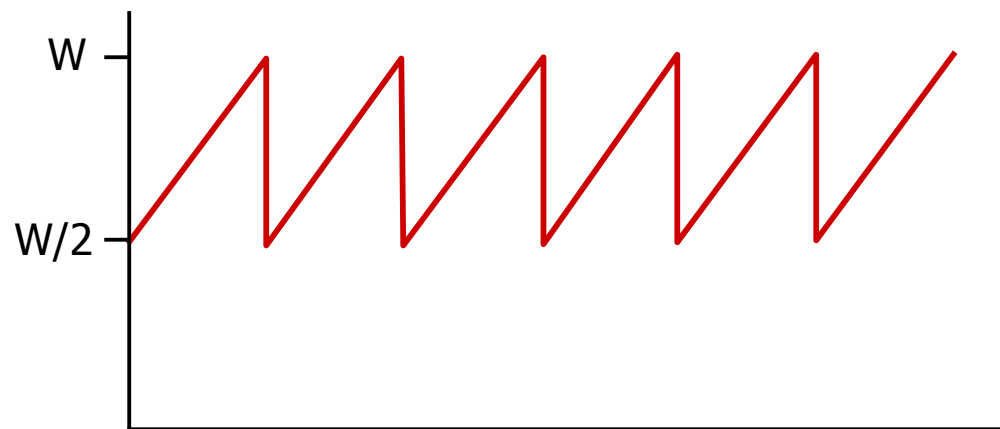
AIMD saw tooth behavior: probing for bandwidth



TCP throughput

- avg. TCP throughput as function of window size, RTT?
 - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
 - avg. window size (# in-flight bytes) is $\frac{3}{4} W$
 - avg. thrupt is $\frac{3}{4}W$ per RTT

$$\text{avg TCP thrupt} = \frac{3}{4} \frac{W}{\text{RTT}} \text{ bytes/sec}$$



Simple TCP model

- Model: single saturated TCP pumping data into bottleneck
- other flows only modeled through packet loss
- Bandwidth as function of packet loss (in packets)

$$B(p) = \frac{1}{RTT} \sqrt{\frac{3}{2bp}} + o(1/\sqrt{p})$$

- Where

RTT – Round trip delay p -- Packet loss rate

b -- Number of packets confirmed by a single ACK

TCP Throughput: An example

Throughput in bit/s for TCP from **XXX** to TU Berlin....

The capacity of the link is like 640 Mbits/s, no other traffic!!!

Calculated by simplified Padhye's formula with
MaxSegmentSize=1460,

<div>Server In: PER</div>	FhG Fokus Berlin (RTT 3.4 ms)	Univ. Stuttgart (RTT 19.2ms)	UC Berkeley (RTT ~170 ms)
PER 0.01	8.9314 e+05	4.1639 e+05	6.8320 e+04 ~64 kbit/s
PER 0.001	1.0338 e+07	2.0187 e+06 ~ 2 Mbit/s	2.3255 e+05
PER 0.0001	3.6920 e+07 ~34 Mbit/s	6.5615 e+06	7.4158 e+05

TCP Futures: TCP over “long, fat pipes”

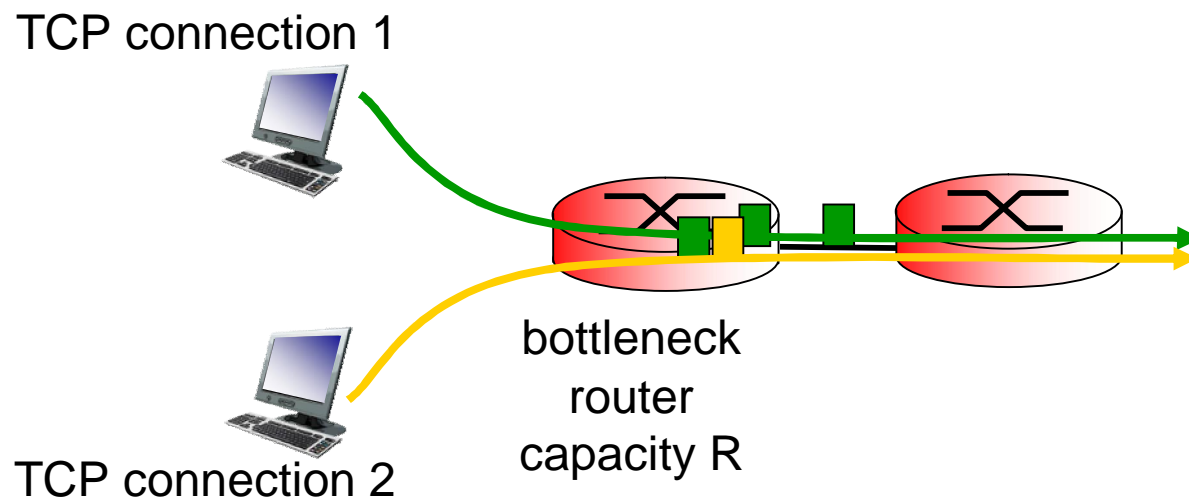
- example: 1500 byte segments, 100ms RTT, acknowledgment after each segment, want 10 Gbps throughput
- requires $W = 83,333$ in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

→ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ – *a very small loss rate!*

TCP Fairness

Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R , each should have average rate of R/K



Fairness (more)

Fairness and UDP

- multimedia apps often do not use TCP
 - do not want rate throttled by congestion control
- instead use UDP:
 - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
 - new app asks for 1 TCP, gets rate $R/10$
 - new app asks for 11 TCPs, gets $R/2$

TCP Fairness?

UDP TCP interactions

- Terminology from RFC 2309:
 - TCP-compatible flow:
 - in steady state, uses no more bandwidth than a conformant TCP under similar conditions
 - unresponsive flow:
 - does not slow down in response to congestion
 - responsive but not TCP-compatible
 - responsive to congestion, but does not compete equally with TCP in a queue with FIFO scheduling