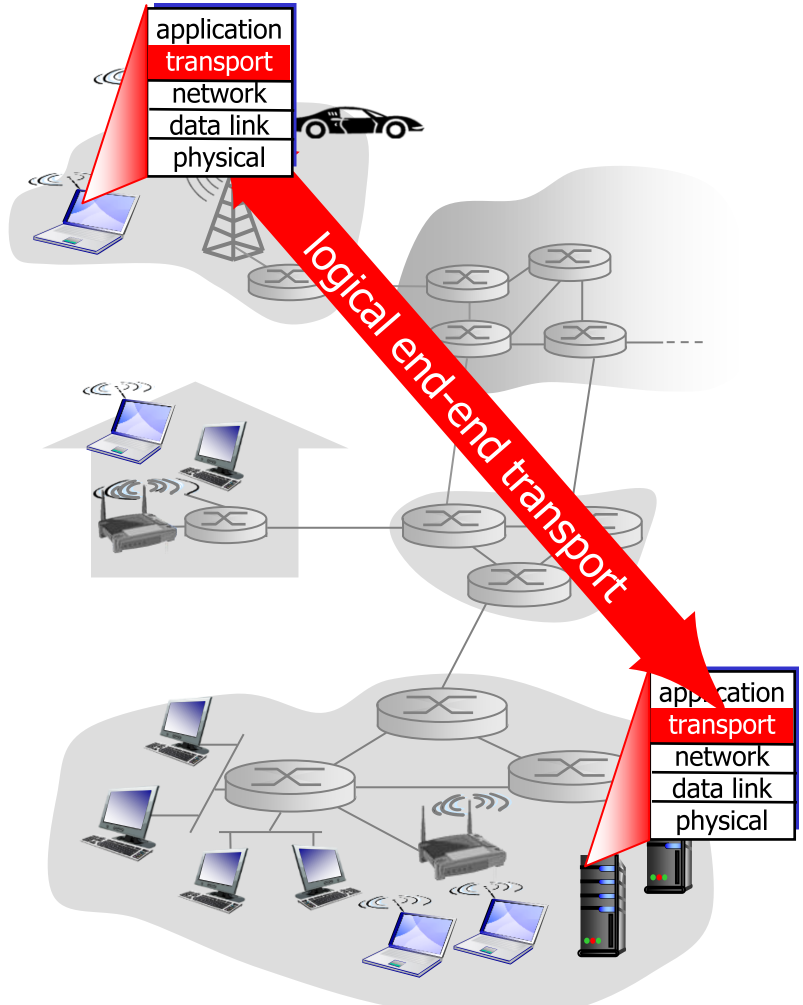
Internet of Things
IO 4041
Transport Layer

Transport Layer

- Transport protocols reside on top of IP
- Applications do not use IP directly,
  - but use the transport protocols to communicate with each other.
- In IP protocol stack, most widely used transport protocols
  - User Datagram Protocol (UDP), and
  - Transport Control Protocol (TCP).

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - **send side:** breaks app messages into *segments*, passes to network layer
  - **rcv side:** reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport Layer

- **UDP** is a best-effort delivery service, which does not add much on top of IP, whereas
- **TCP** is a reliable byte stream that adds a connection abstraction on top of the connectionless IP.
- Although there have been several other transport protocols defined,
  - such as SCTP [229] and DCCP [152],
  - they have as yet to be adopted by the mainstream.

Transport Layer

- Usually, basic unit of transportation is called a Packet
- data from higher layers are transported in these packets
- In UDP, the basic unit of transportation is called a user datagram (or UDP segment)
- In TCP, basic unit of transportation is called a segment (TCP segment)

UDP: User Datagram Protocol [RFC 768]

Best effort delivery service:

- As the underlying IP network does its best to deliver the datagram,
  - but does not guarantee delivery of the datagrams at the destination [may loss]
  - Does not guarantee that the datagrams are delivered in the same order as they were sent [can be delivered out of order]

UDP: User Datagram Protocol [RFC 768]

Connectionless

- no handshaking between UDP sender, receiver
- each UDP segment handled independently of others

Uses

- streaming multimedia apps (loss tolerant, rate sensitive): real time audio/video
- DNS look ups

reliable transfer over UDP

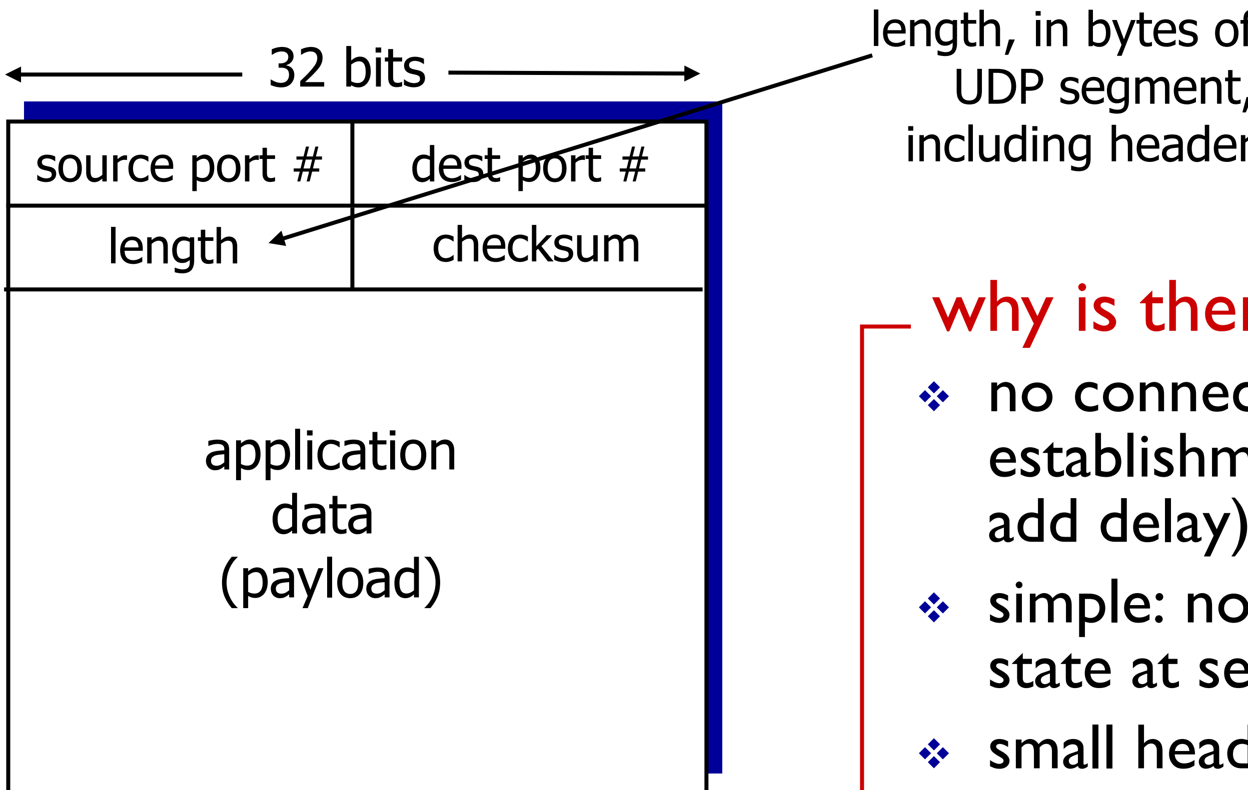
- add reliability at application layer
- application-specific error recovery!

In the context of smart object networks,

UDP [due to its simplicity and lightweight nature]

- **is an exciting choice for quick transportation of sensor data**

UDP: segment header



UDP segment format

why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- **checksum:** addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: example

example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

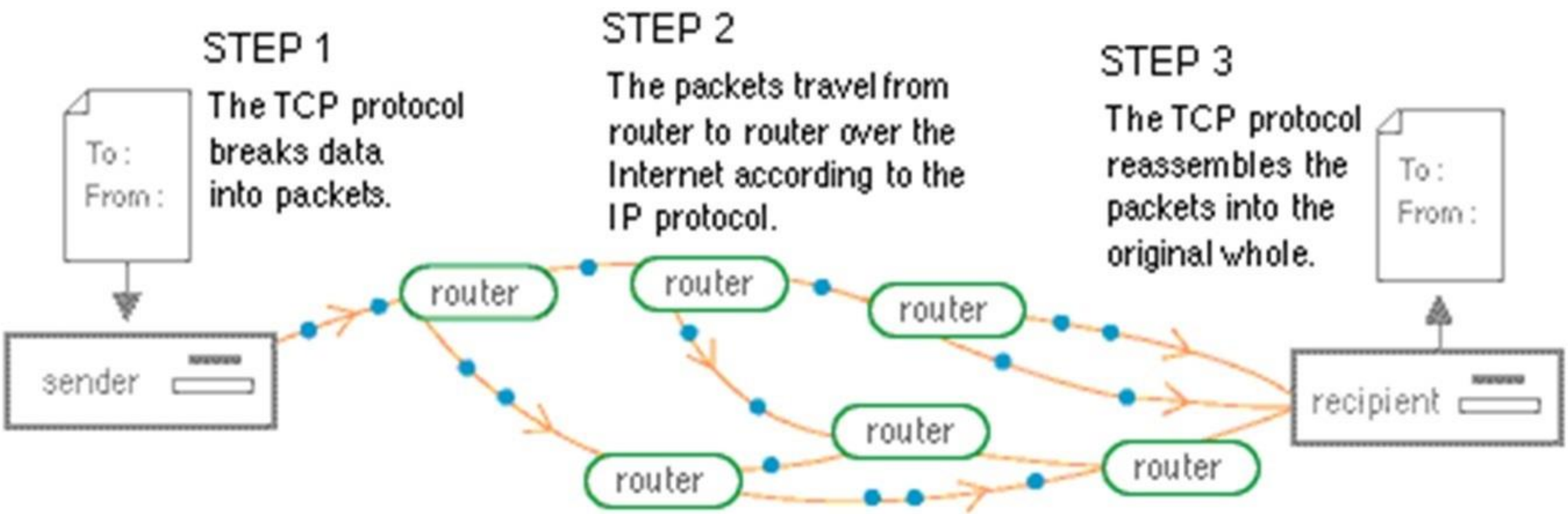
Transmission Control Protocol (TCP)

- A standard that defines how to establish and maintain a conversation so that application layer protocols can share/exchange data
- provides a reliable byte stream on top of the best-effort packet service provided by the IP layer
- It is based on client-server model
  - Client gets a service from a server, i.e., a web page
- Since IoT devices offer a varying traffic pattern
  - So TCP is not suitable for IoT applications
- **uIP**, an implementation of TCP for memory-constrained smart objects.

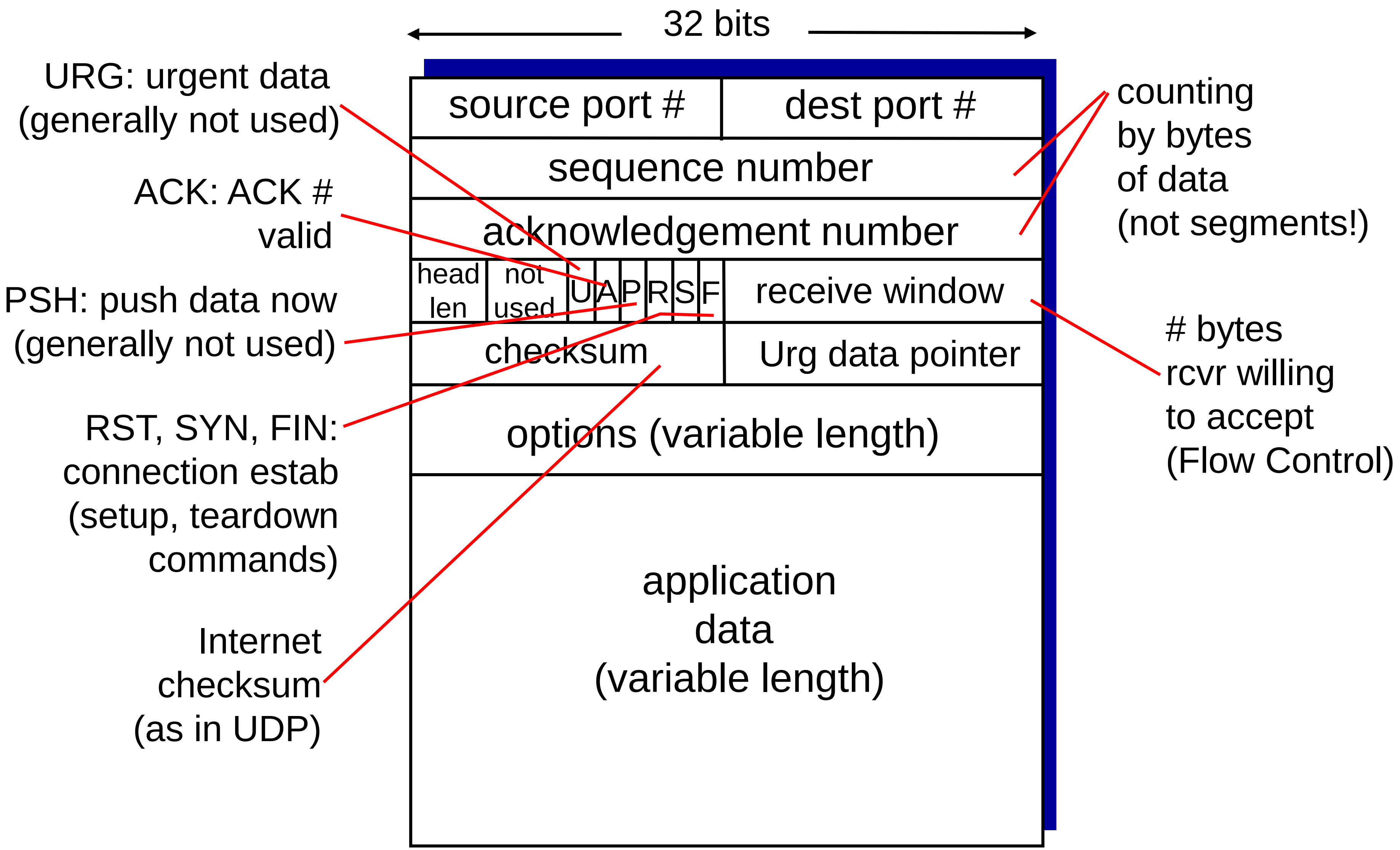
Transmission Control Protocol (TCP)

- **Reliability** is achieved by buffering data combined with positive ACKs and retransmissions
- Before any data are transported, the two connection end points must explicitly set up a connection.
- A connection is identified by the IP addresses and TCP port numbers of the end points
- Many application layer protocols are defined over TCP,
  - such as HTTP (Web), SMTP (e-mail), and XMPP (instant messaging).

Transmission Control Protocol (TCP)



TCP segment structure



TCP seq. numbers, ACKs

sequence numbers:

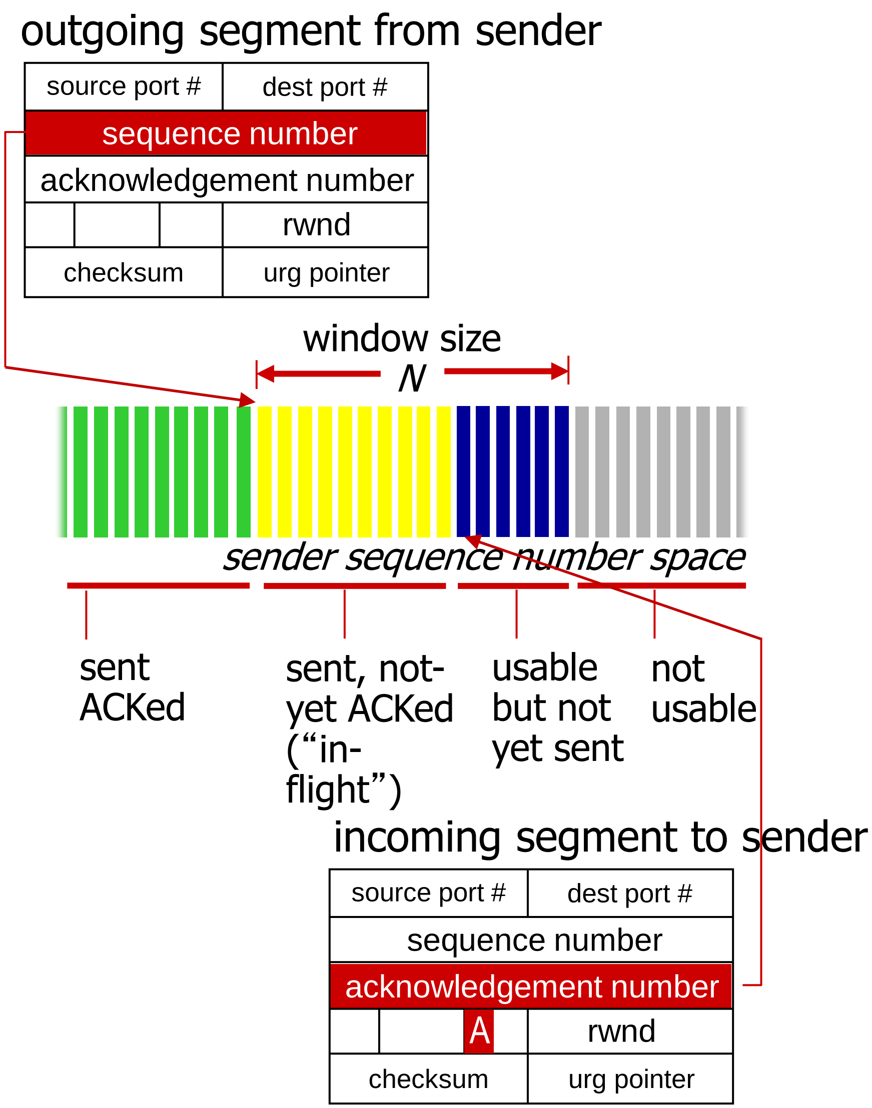
- byte stream "number" of the first byte in segment's data

acknowledgements:

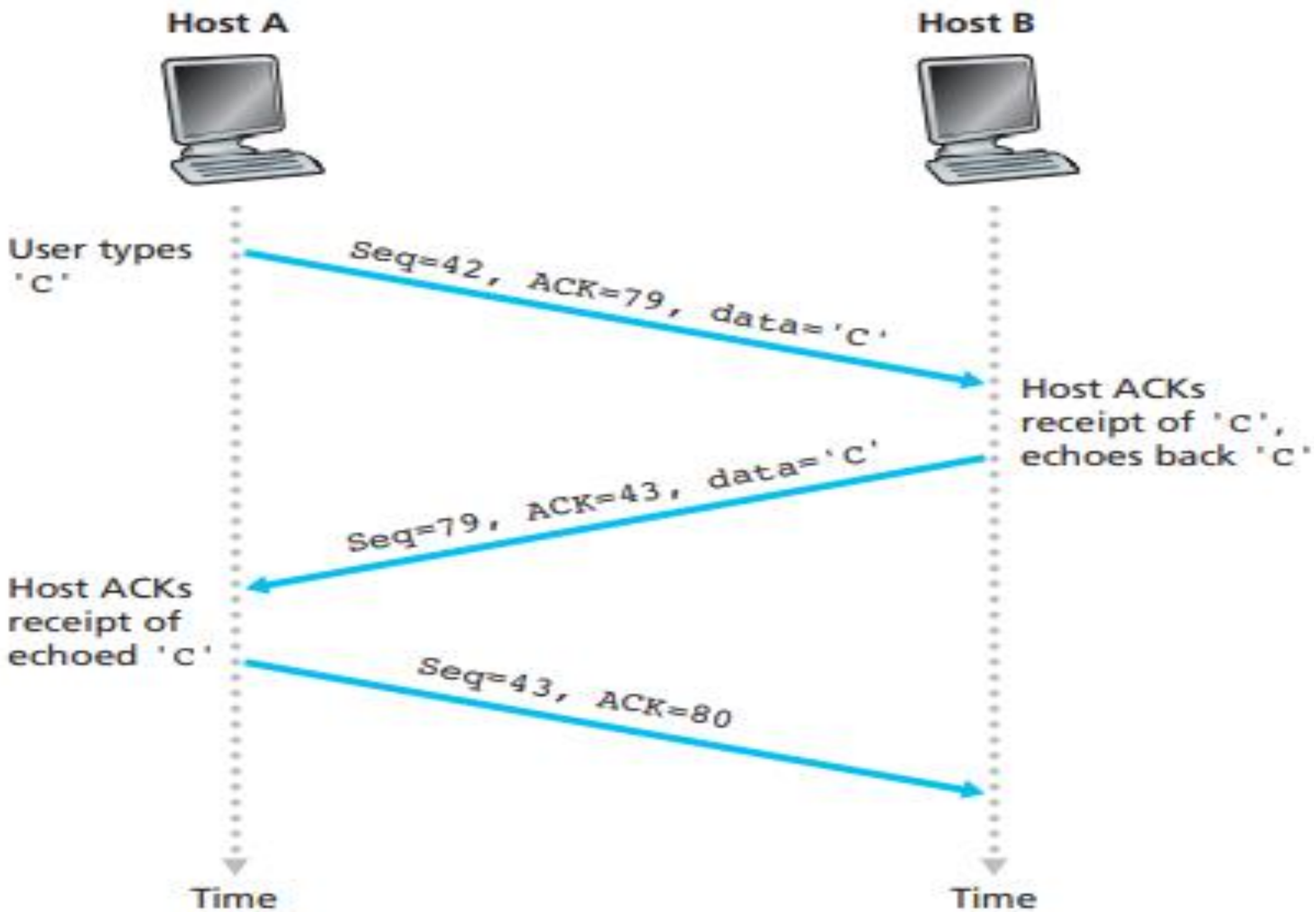
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



A simple Telnet Application over TCP



TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

let's initially consider simplified TCP sender:

- ignore duplicate acks
- ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: `TimeOutInterval`

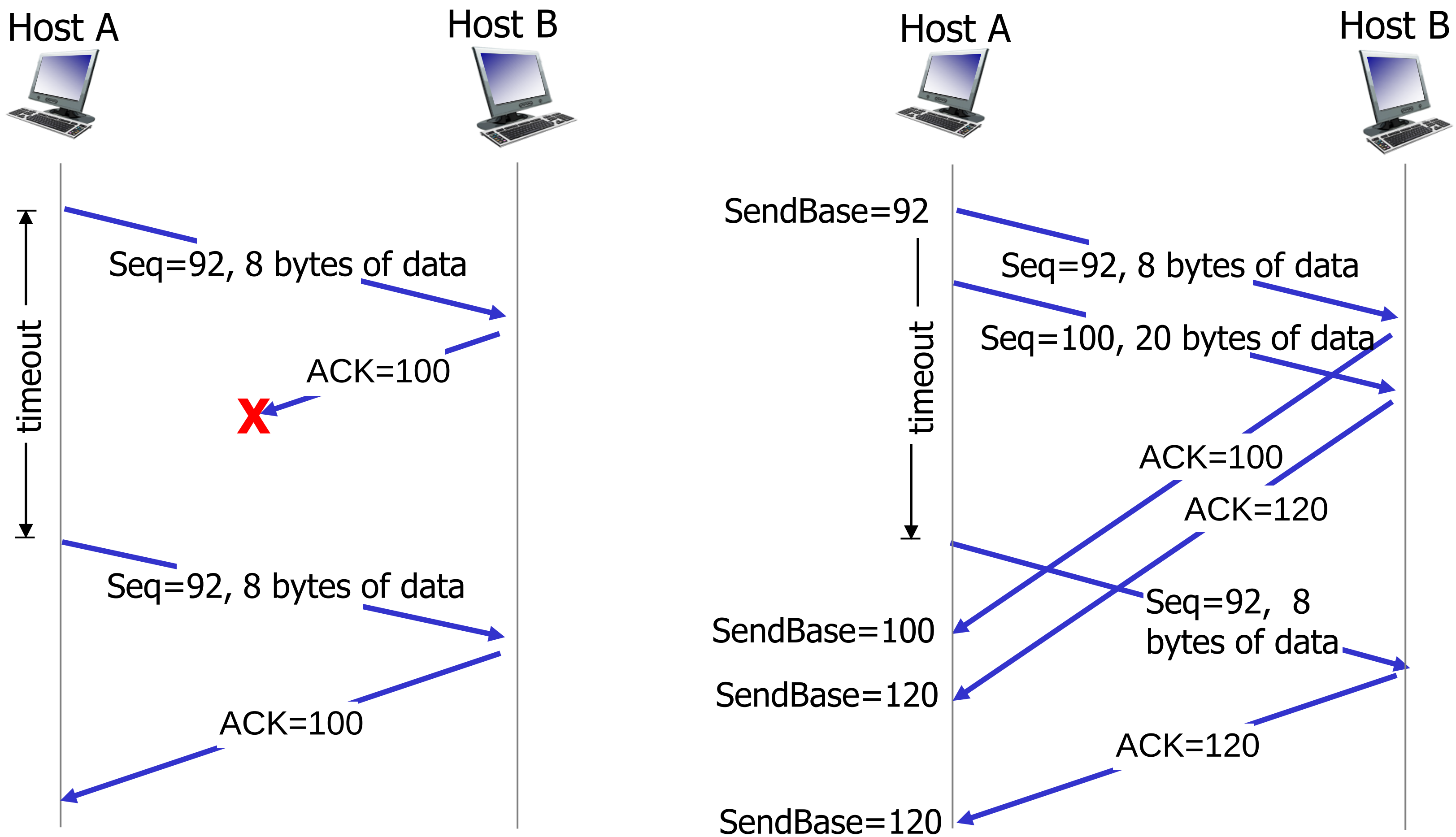
timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

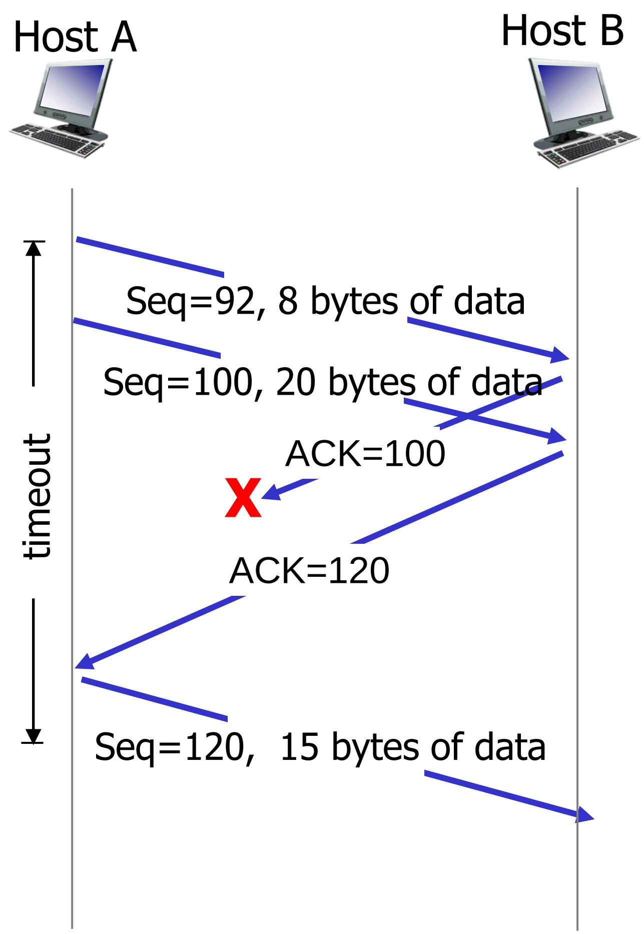
TCP: retransmission scenarios



lost ACK scenario

premature timeout

TCP: retransmission scenarios



cumulative ACK

Doubling the Timeout Interval

length of the timeout interval after a timer expiration?

- whenever the timeout event occurs,
  - TCP retransmits the not-yet-acknowledged segment with the smallest sequence number
  - But each time TCP retransmits,
    - it sets the next timeout interval to twice the previous value

whenever the timer is started after

-either data received from application above,

-Or ACK received

Then TimeoutInterval is derived from the most recent values of EstimatedRTT and DevRTT

The purpose is to control congestion

TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

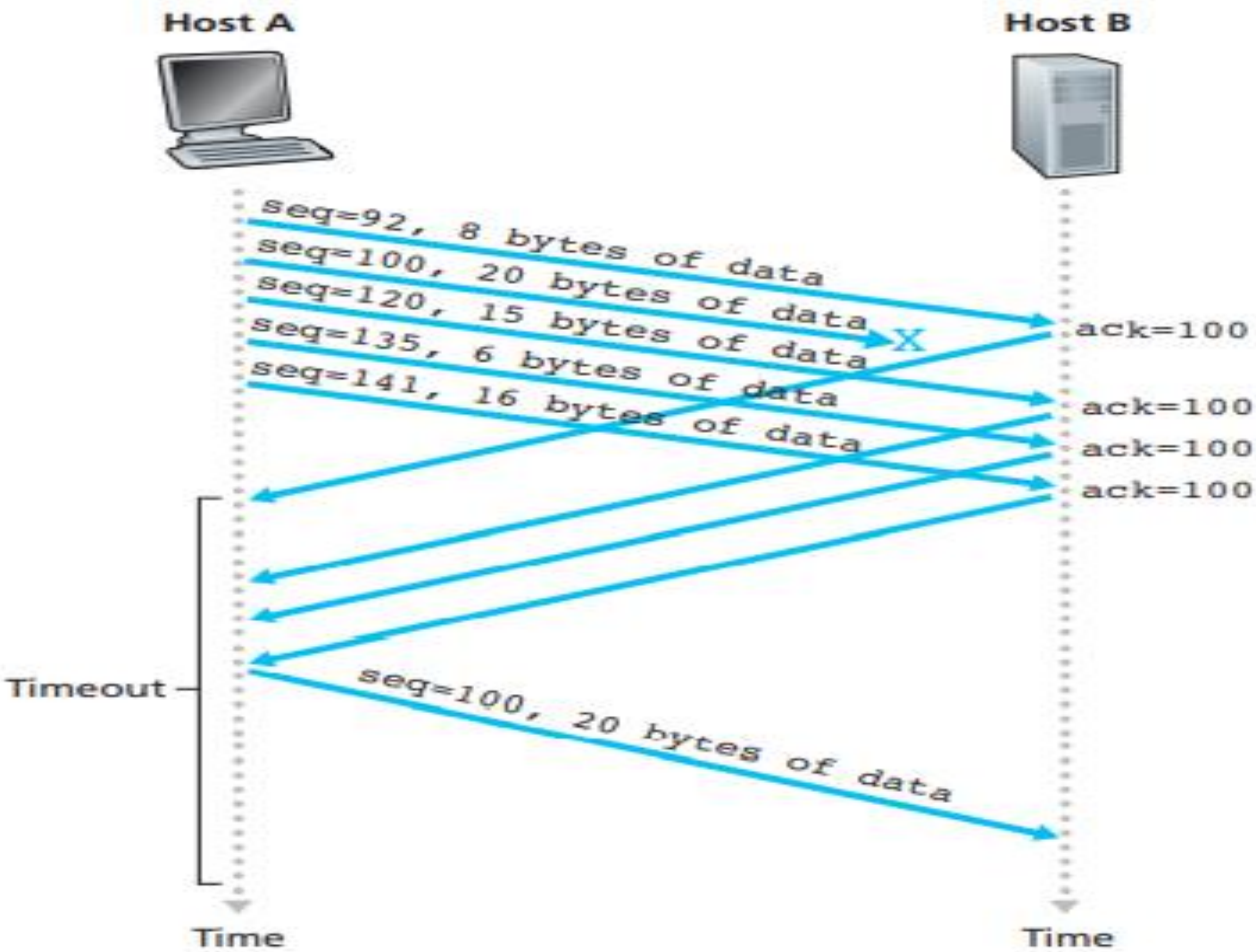
TCP fast retransmit

if sender receives 3 ACKs for same data

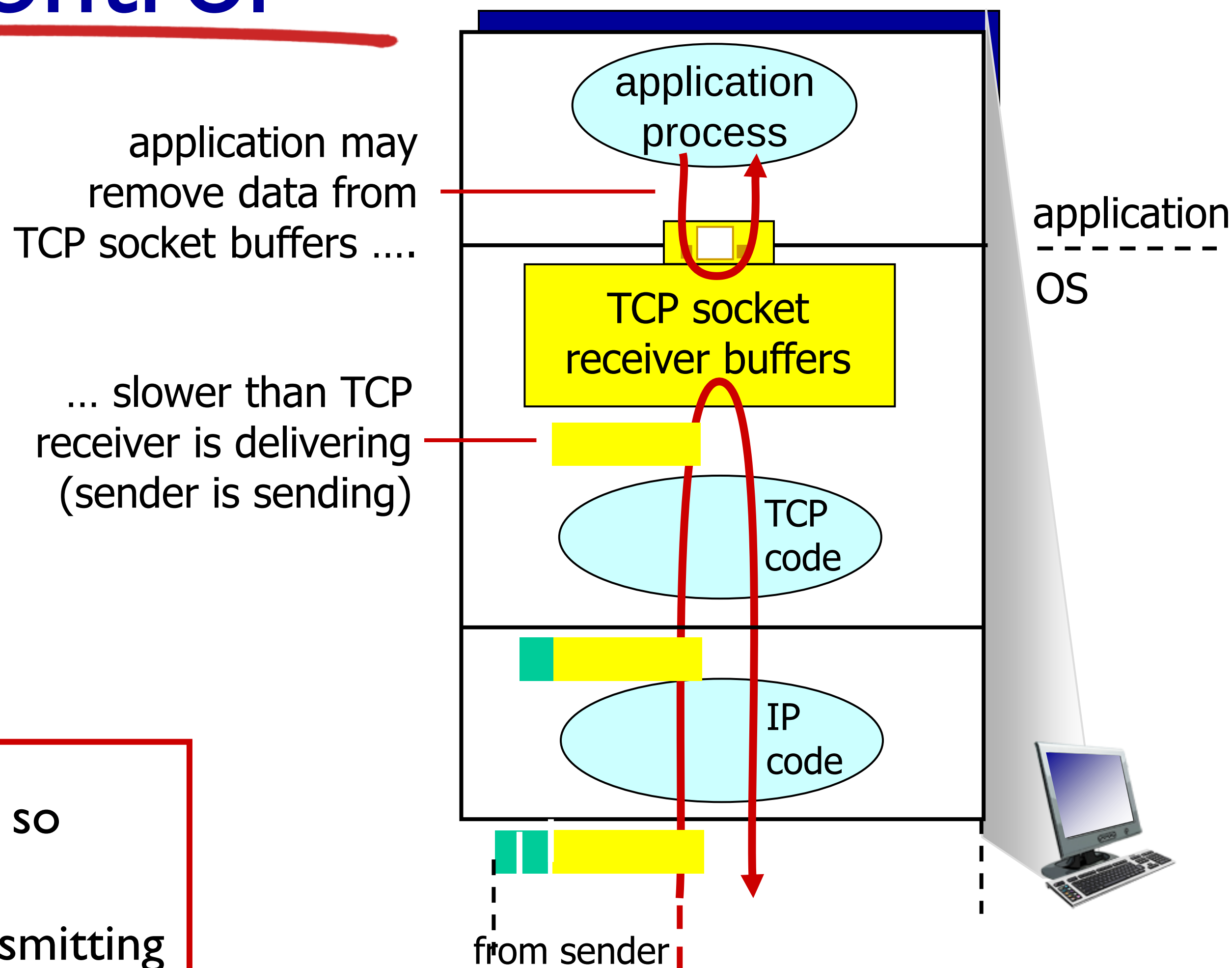
("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



TCP flow control



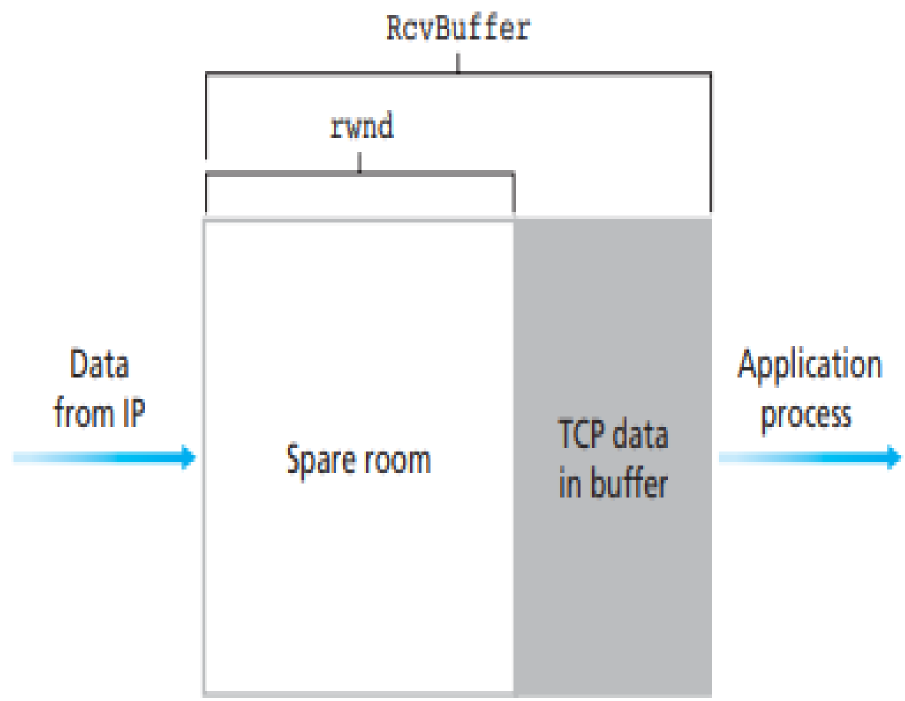
receiver protocol stack

flow control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP flow control

- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size (typical default is 4096 bytes)
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow



receiver-side buffering

TCP flow control

Receive Window is a variable maintained by receiver.

LastByteRcvd – LastByteRead <= RcvBuffer

if RcvBuffer is equal to left side equation, buffer is full
else there is spare room

rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]

LastByteSent – LastByteAcked <= rwnd

Question: What happens when rwnd is 0!

TCP flow control

Question: What happens when rwnd is 0!

- Application process at receiver empties the buffer and
- does not send new segment as receiver has nothing to send (neither data nor ACK).

So **sender** is unaware that space is there in receiver buffer and thus gets blocked

Solution:

- TCP specification requires sender to send segment with one data byte when rwnd is 0.
- Such segments will be ACKed by receiver and new ACKs will eventually contain non-zero rwnd values.

Principles of congestion control

congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

TCP congestion control

- TCP must use end-to-end congestion control rather than network-assisted congestion control
  - since the IP layer provides no explicit feedback to the end systems regarding network congestion

- TCP approach is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion.

TCP congestion control

- If there is little perceived congestion on the path between itself and the destination,
  - then the TCP sender increases its send rate;
- if the sender perceives that there is more congestion along the path,
  - then the sender reduces its send rate.

TCP congestion control

- How does a TCP sender limit the rate at which it sends traffic into its connection?
- How does a TCP sender perceive that there is congestion on the path between itself and the destination?
- What algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

TCP congestion control

- **How does a TCP sender limit the rate at which it sends traffic into its connection?**
- In addition to send buffer, receive buffer and other variables like LastByteRead, rwnd etc)
- Additional variable: congestion window (cwnd)
  - Imposes a constraint on the send rate
- Specifically, the amount of un-ACKed data at a sender may not exceed the minimum of cwnd and rwnd
- LastByteSent – LastByteAcked <= min{cwnd, rwnd}

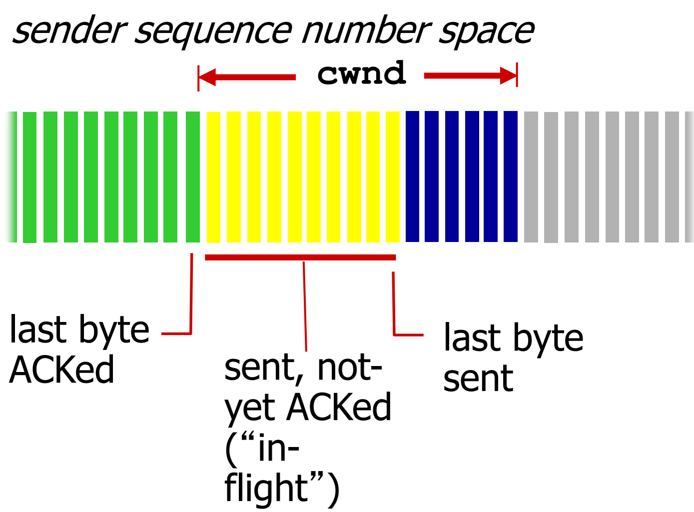
TCP congestion control

- Assume TCP receive buffer is large enough, **ignore rwnd constraint**
  - So, the amount of **un-ACKed** data is solely limited by **cwnd**
- Further, assume sender always has data to send,
  - all segments in the **cwnd** are sent
- This constraint limits the amount of **un-ACKed** data at the sender
  - Thus indirectly limits the sender's send rate.

TCP congestion control

- Consider a connection with negligible loss and packet transmission delays.
- At start of each RTT
  - the constraint permits the sender to send cwnd bytes of data into the connection;
- At the end of the RTT
  - the sender receives ACKs for the data.
- By adjusting the value of cwnd,
  - the sender can adjust send rate

TCP Congestion Control



TCP sending rate:

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

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

- sender limits transmission:

`LastByteSent - LastByteAcked ≤ min{cwnd, rwnd}`

- **`cwnd`** is dynamic, function of perceived network congestion

TCP congestion control

- **How does a TCP sender perceives that there is congestion on the path?**
- **Loss event** at a sender
  - the occurrence of either a timeout or the receipt of three duplicate ACKs from the receiver.
- When there is excessive congestion, then one (or more) router buffers along the path overflows, causing a datagram (containing a TCP segment) to be dropped.
  - The dropped datagram, in turn, results in a loss event at the sender
    - either a timeout or the receipt of three duplicate ACKs
  - which is taken by the sender to be an indication of congestion on the sender-to-receiver path.

TCP congestion control

- **Optimistic case:** Loss event does not occur
- ACKs for Un-ACKed data will be received at sender.
  - So ACKs indicate that all is well and increase its cwnd size (send rate)
- If ACKs arrive at a relatively slow rate,
  - then cwnd will be increased at a relatively slow rate.
- If ACKs arrive at a high rate, then the cwnd will be increased more quickly.
- Arrival of ACK is dependent on the end-end path and/or bandwidth of the link)
- TCP is **self-clocking** as it uses ACKs for increasing its cwnd size

TCP congestion control

- **A lost segment implies congestion**,
  - rate should be decreased when a segment is lost.
- **An ACKed segment indicates all is well**,
  - rate can be increased when an original ACK arrives.
  -
- **Bandwidth probing:** **how much increase or decrease?**

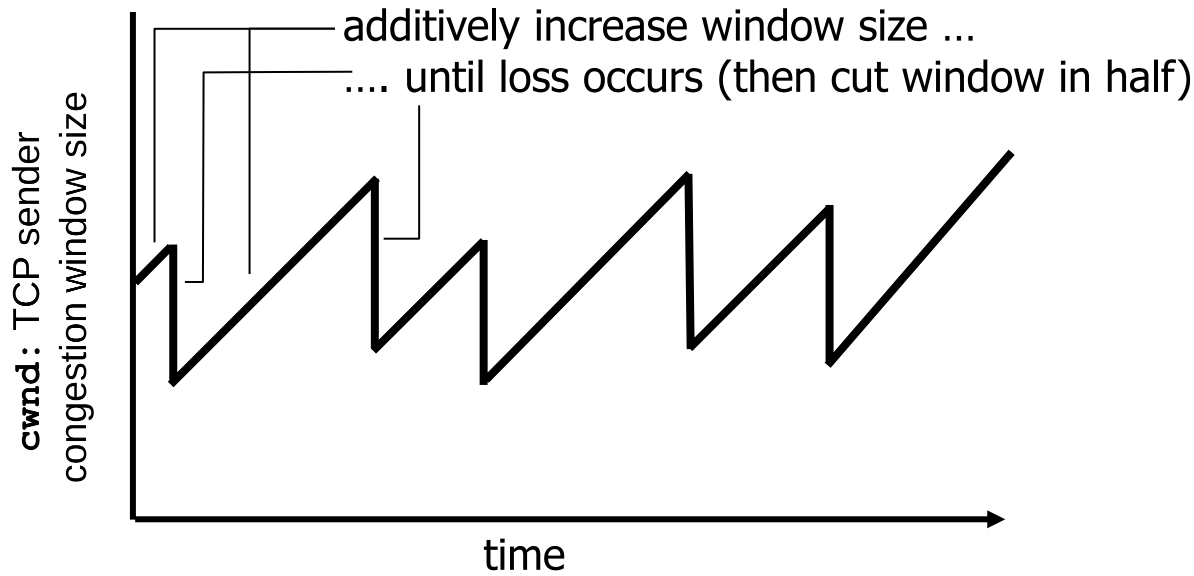
TCP congestion control

- **What algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?**

TCP congestion control: additive increase multiplicative decrease

- *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: probing for bandwidth



TCP congestion control Algorithm

- Described [Jacobson 1988] and is standardized in [RFC 5681].

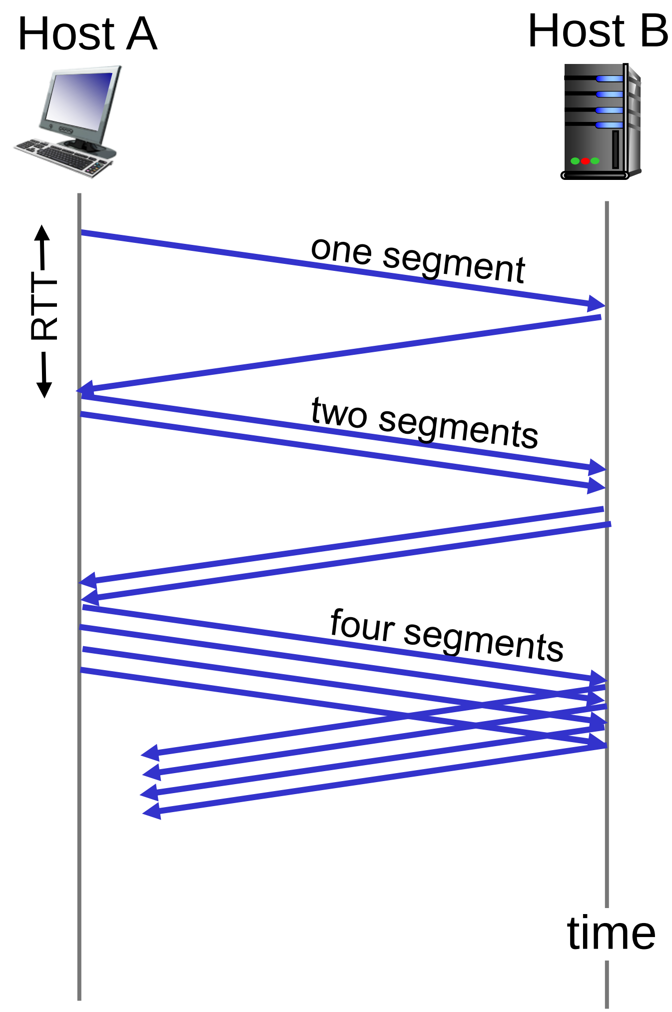
Three major components:

1. slow start [Mandatory]
2. congestion avoidance [Mandatory], and
3. fast recovery [recommended but not required].

- Slow start and congestion avoidance differ in how they increase the size of cwnd in response to received ACKs.
  - slow start increases the size of cwnd more rapidly (despite its name) than congestion avoidance.

TCP Slow Start

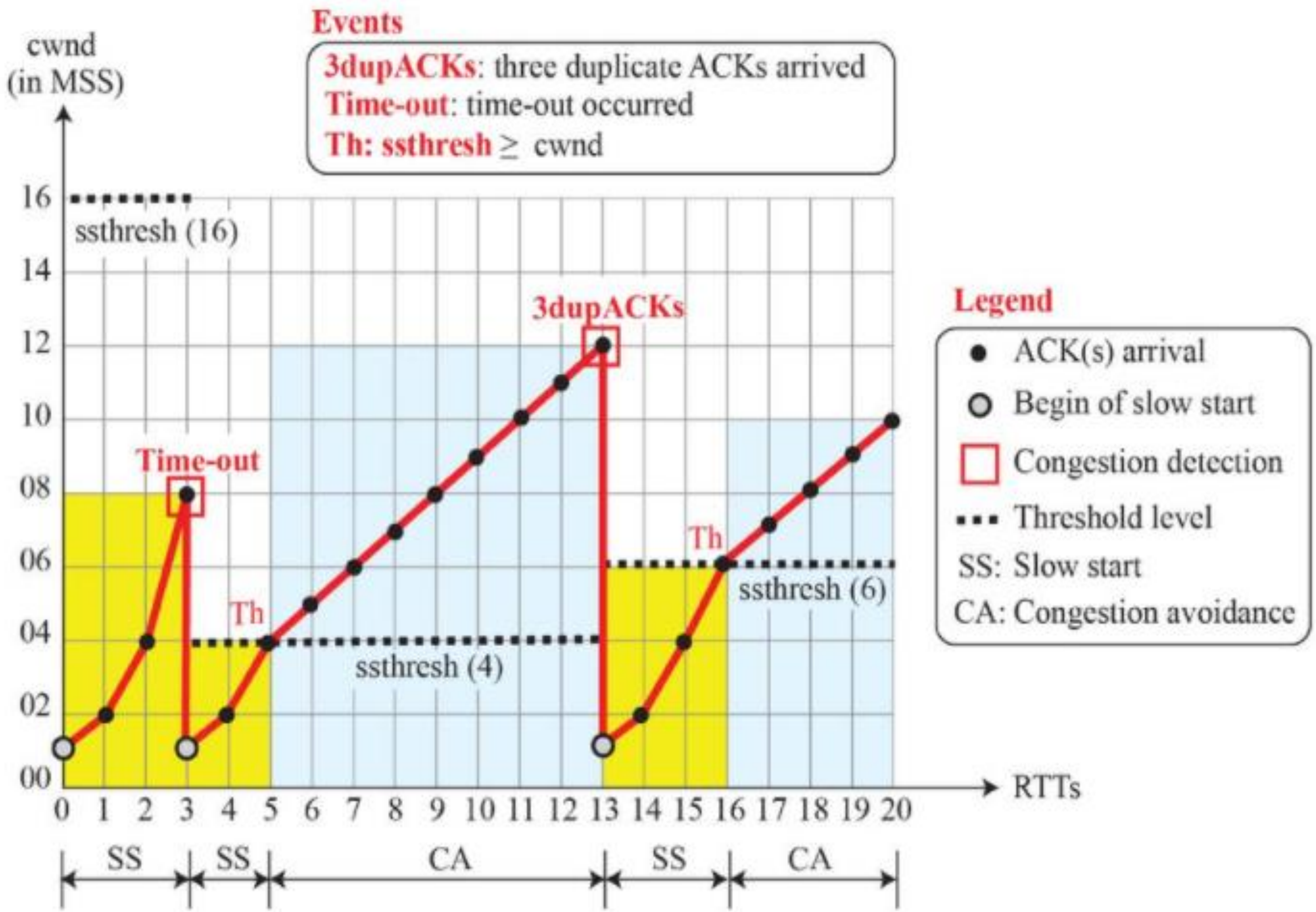
- when connection begins, increase rate exponentially until first loss event:
  - initially **`cwnd`** = 1 MSS
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



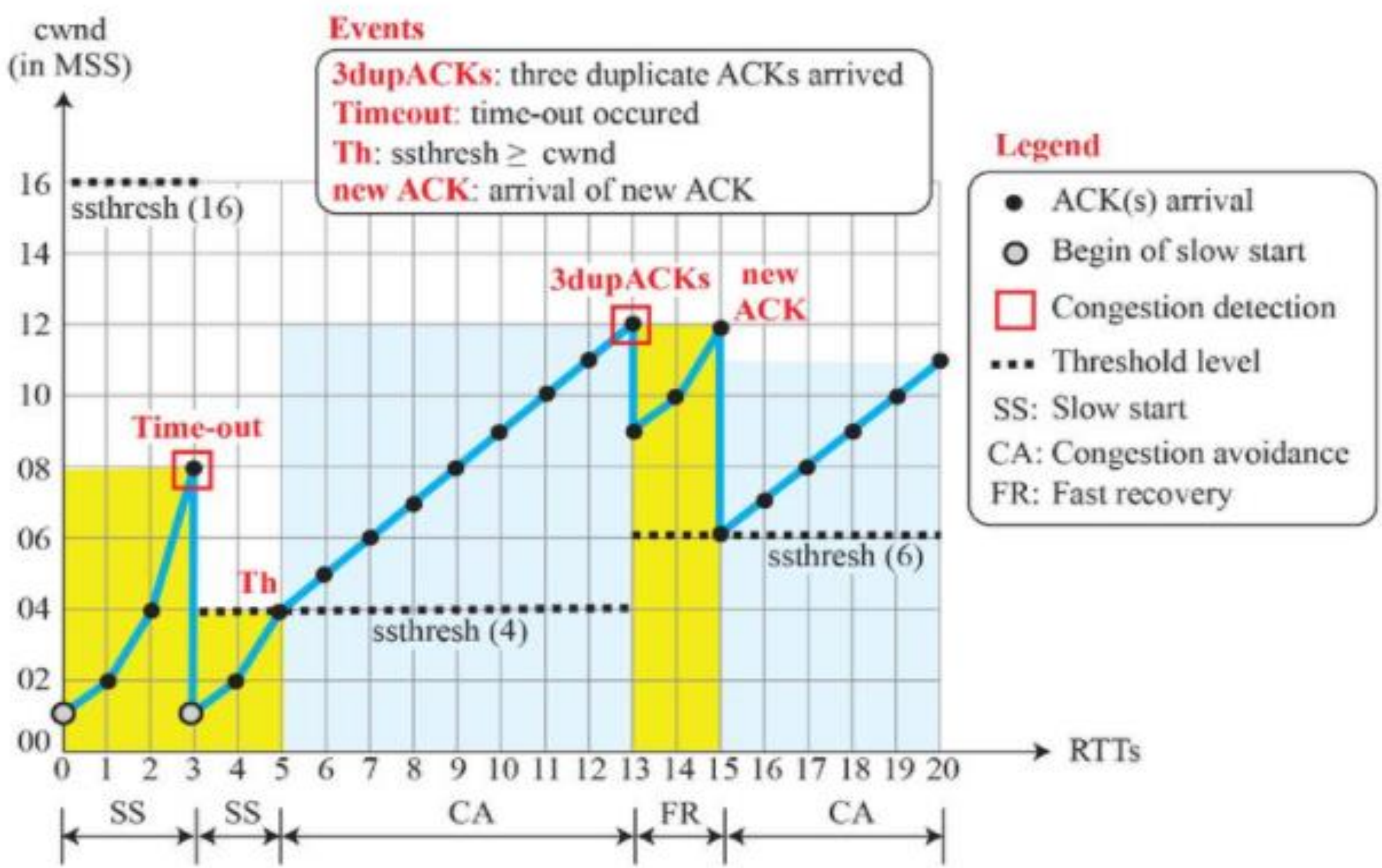
TCP: detecting, reacting to loss

- **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 Tahoe (old TCP) always sets `cwnd` to 1 (timeout or 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: Example



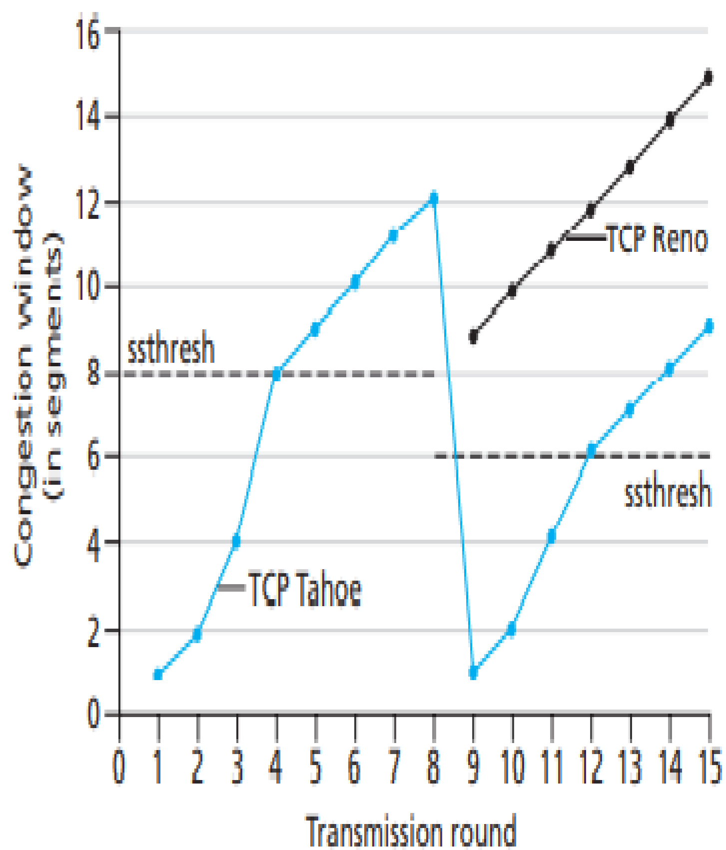
TCP Reno: Example



TCP: switching from slow start to CA

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

TCP: Reliable stream transport

- Reliability is achieved by buffering data combined with positive ACKs and retransmissions
- Before any data are transported, the two connection end points must explicitly set up a connection.
- A connection is identified by the IP addresses and TCP port numbers of the end points
- Many application layer protocols are defined over TCP,
  - such as HTTP (Web), SMTP (e-mail), and XMPP (instant messaging).