CS 352
Detecting & Reacting to Losses

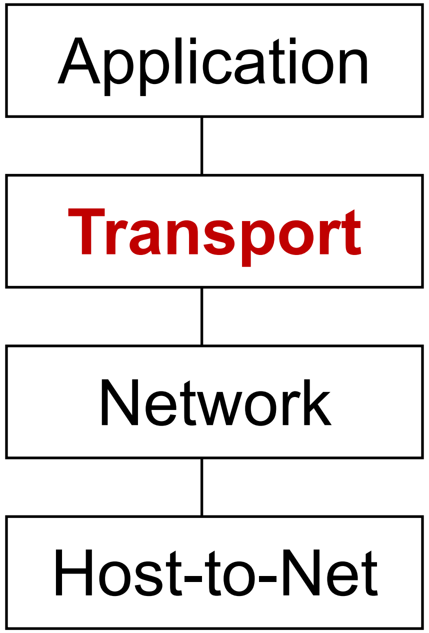
CS 352, Lecture 13.1

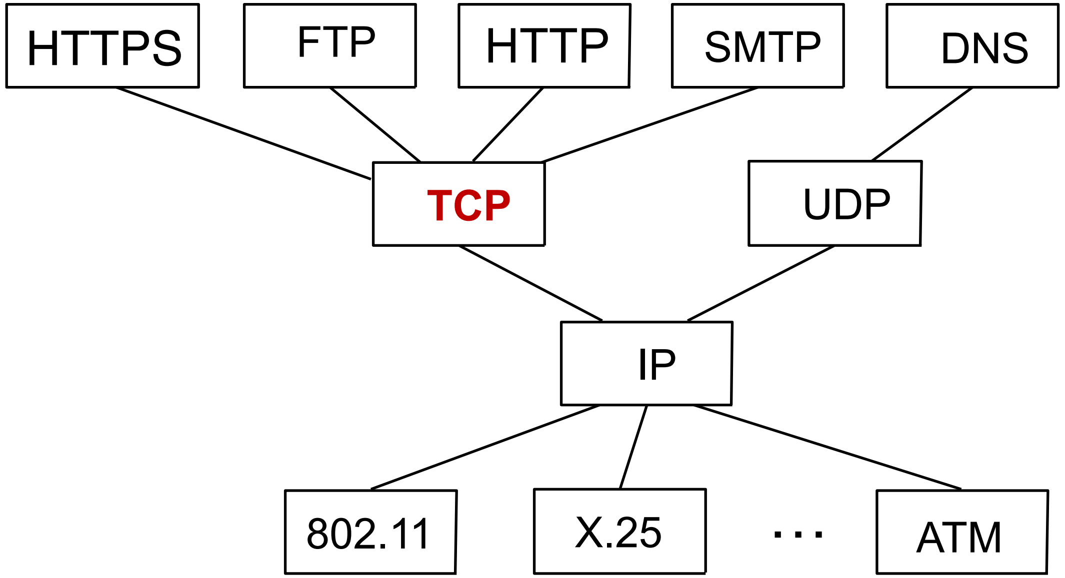
http://www.cs.rutgers.edu/~sn624/352

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

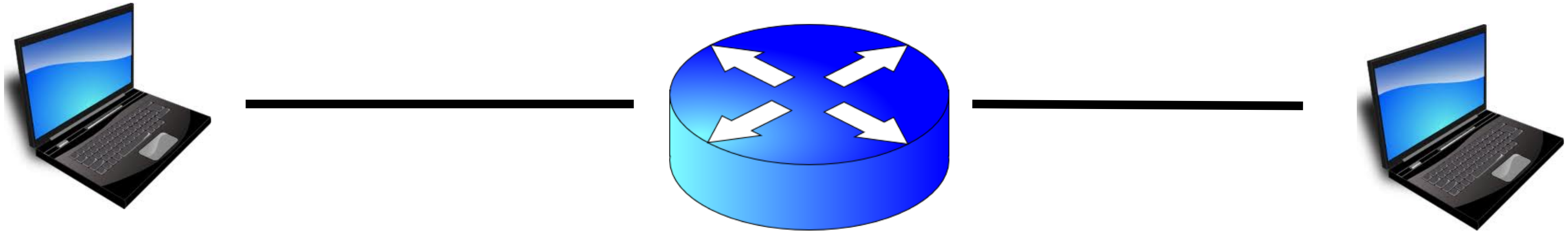
Transport


Application
Transport
Network
Host-to-Net


HTTPS
FTP
HTTP
SMTP
DNS
TCP
UDP
IP
802.11
X.25
...
ATM

How do apps get perf guarantees?

- The network core provides no guarantees on packet delivery



- Transport software on the endpoint oversees implementing guarantees on top of a best-effort network
- Three important kinds of guarantees
  - Reliability
  - Ordered delivery
  - Resource sharing in the network core

Transmission Control Protocol (TCP)

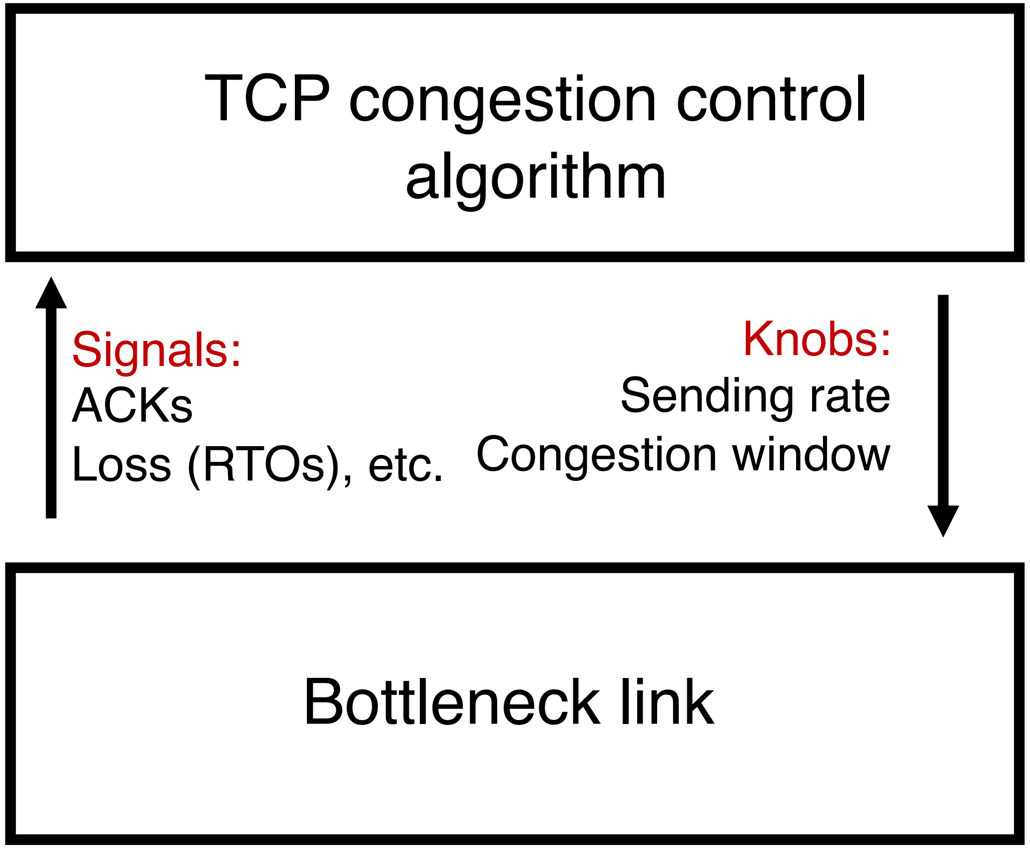
Review: Congestion control so far

- Algorithm by which multiple endpoints efficiently and fairly share bottleneck link

- So far, we've looked at just efficiency.

- Steady state: ACK clocking (keep the pipe full, but don't congest it)

- Getting to steady state:
  - Slow start: exponential increase
  - TCP New Reno: Additive increase
  - TCP BBR: gain cycling & filters



Detecting packet loss

- So far, all the algorithms we've studied have a coarse loss detection mechanism: RTO timer expiration
  - Let the RTO expire, drop `cwnd` all the way to 1 MSS
- Analogy: you're driving a car
  - You're waiting until the next car in front is super close to you (RTO) and then hitting the brakes really hard (set cwnd := 1)
  - Q: Can you see obstacles from afar and slow down proportionately?
- That is, can the sender see packet loss coming in advance?
  - And reduce `cwnd` more gently?

Can we detect loss earlier than RTO?

- Key idea: use the information in the ACKs. How?

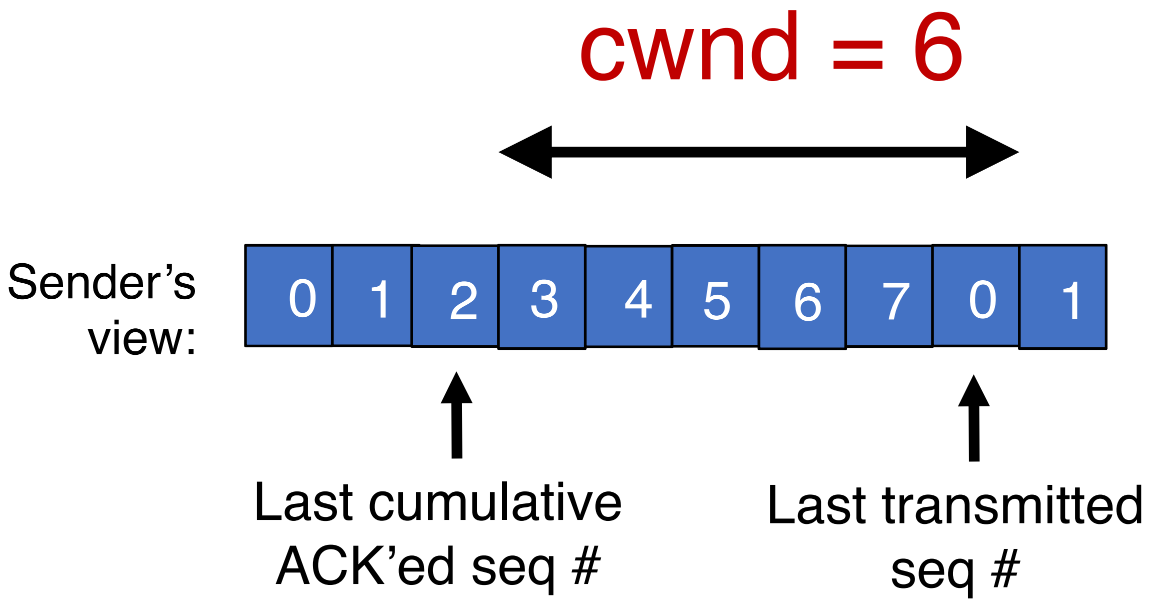
- Suppose successive (cumulative) ACKs contain the same ACK#
  - Also called duplicate ACKs
  - Occur when network is reordering packets, or one (but not most) packets in the window were lost

- Reduce `cwnd` when you see many duplicate ACKs
  - Consider many dup ACKs a strong indication that packet was lost
  - Default threshold: 3 dup ACKs, i.e., triple duplicate ACK
  - Make cwnd reduction gentler than setting cwnd = 1; recover faster

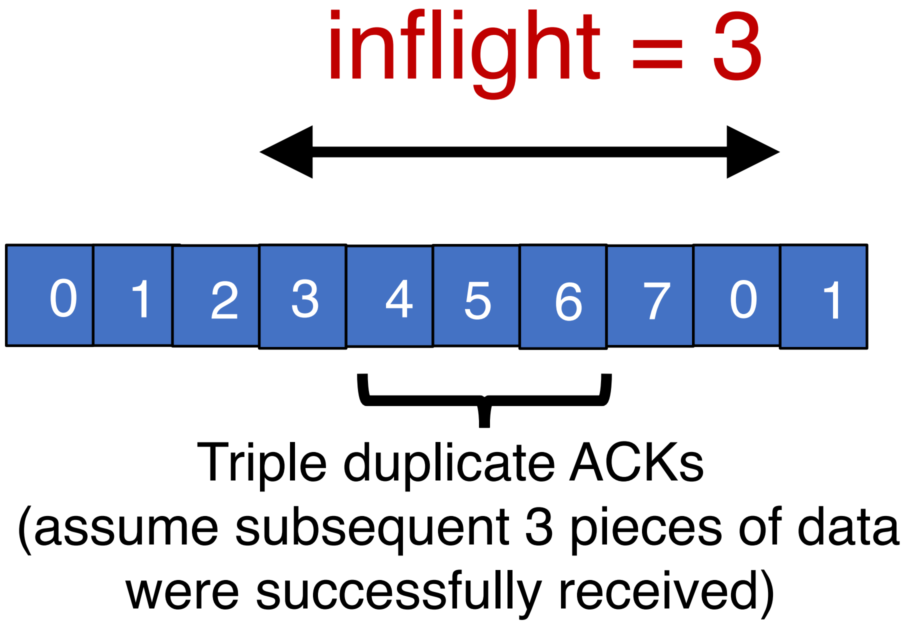
Fast Retransmit & Fast Recovery

Distinction: In-flight versus window

- So far, window and in-flight referred to the same data
- Fast retransmit & fast recovery differentiate the two notions



`cwnd` is the interval between the last cumulatively ACK'ed seq# and the last transmitted seq#



`inflight` is the data currently believed to be in flight.

TCP fast retransmit (RFC 2581)

- The fact that ACKs are coming means that data is getting delivered to the receiver, albeit with some loss.
- Note: Before the dup ACKs arrive, we assume `inflight = cwnd`

- TCP sender does two actions with fast retransmit

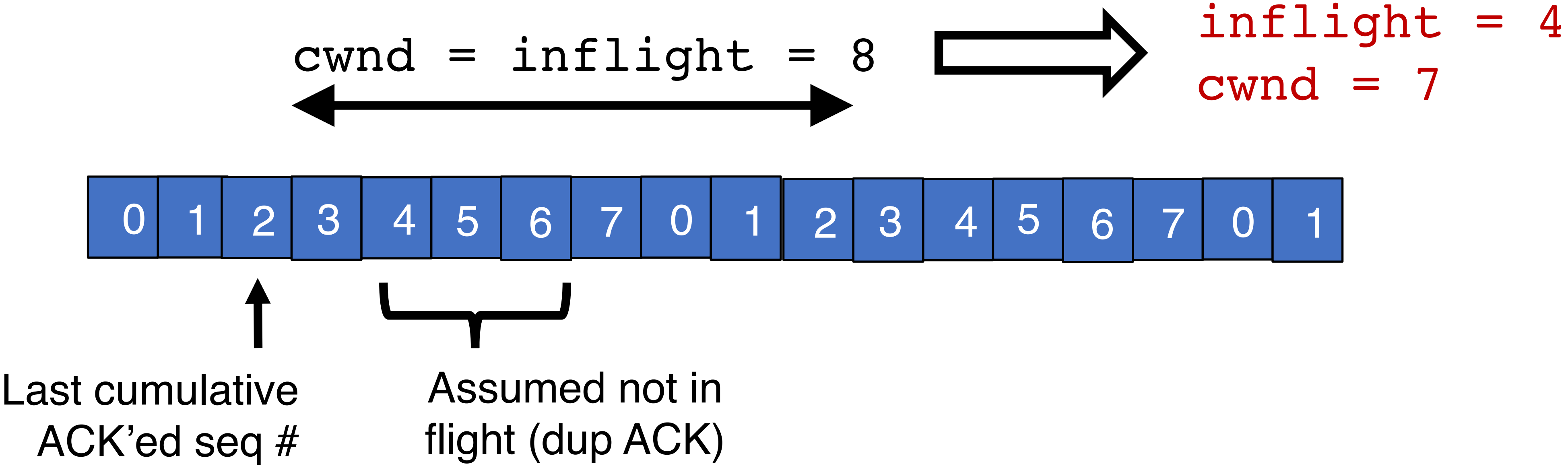
TCP fast retransmit (RFC 2581)

- (1) Reduce the `cwnd` and `in-flight` gently
  - Don't drop `cwnd` all the way down to 1 MSS

- Reduce the amount of in-flight data multiplicatively
  - Set `inflight → inflight / 2`
  - That is, set `cwnd = (inflight / 2) + 3MSS`
  - This step is called multiplicative decrease
  - Algorithm also sets `ssthresh` to `inflight / 2`

TCP fast retransmit (RFC 2581)

- Example: Suppose `cwnd` and `inflight` (before triple dup ACK) were both 8 MSS.
- After triple dup ACK, reduce `inflight` to 4 MSS
- *Assume* 3 of those 8 MSS no longer in flight; set `cwnd` = 7 MSS



TCP fast retransmit (RFC 2581)

- (2) The seq# from dup ACKs is immediately retransmitted
- That is, don’t wait for an RTO if there is sufficiently strong evidence that a packet was lost

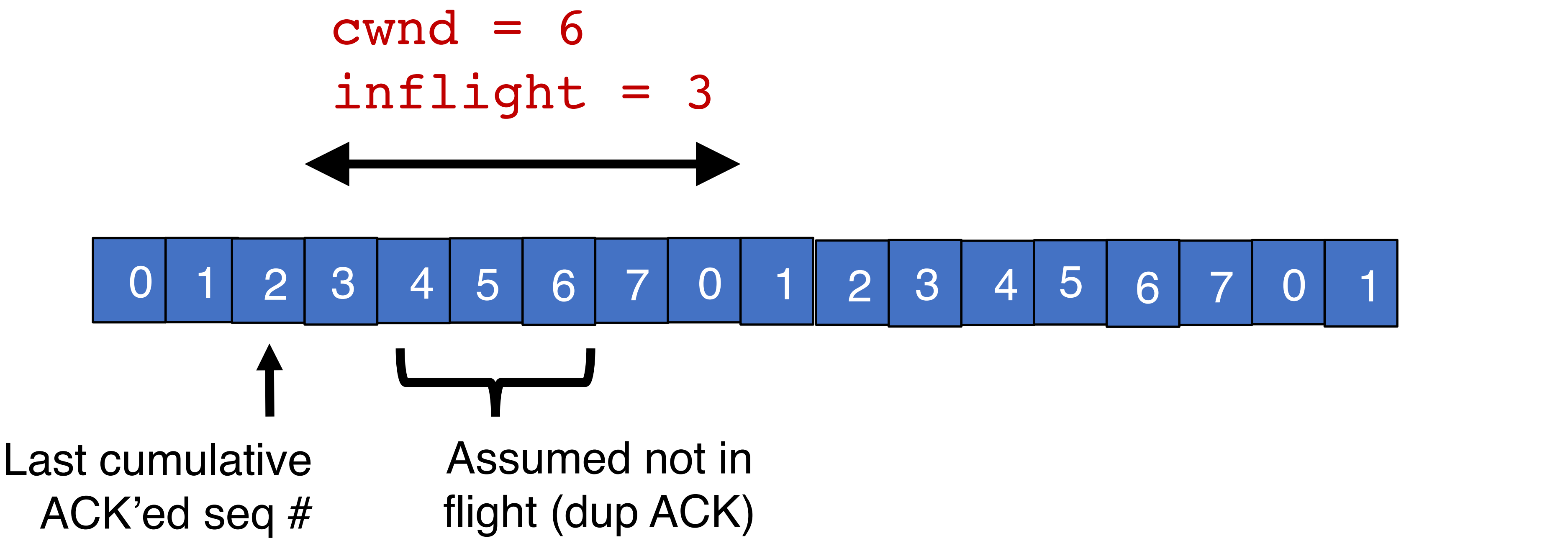
TCP fast recovery (RFC 2581)

- Sender keeps the reduced `inflight` until a new ACK arrives
  - New ACK: an ACK for the seq# that was just retransmitted
  - May also include the (three or more) pieces of data that were subsequently delivered to generate the duplicate ACKs

- Conserve packets in flight: transmit *some* data over lossy periods (rather than no data, which would happen if `cwnd` := 1)

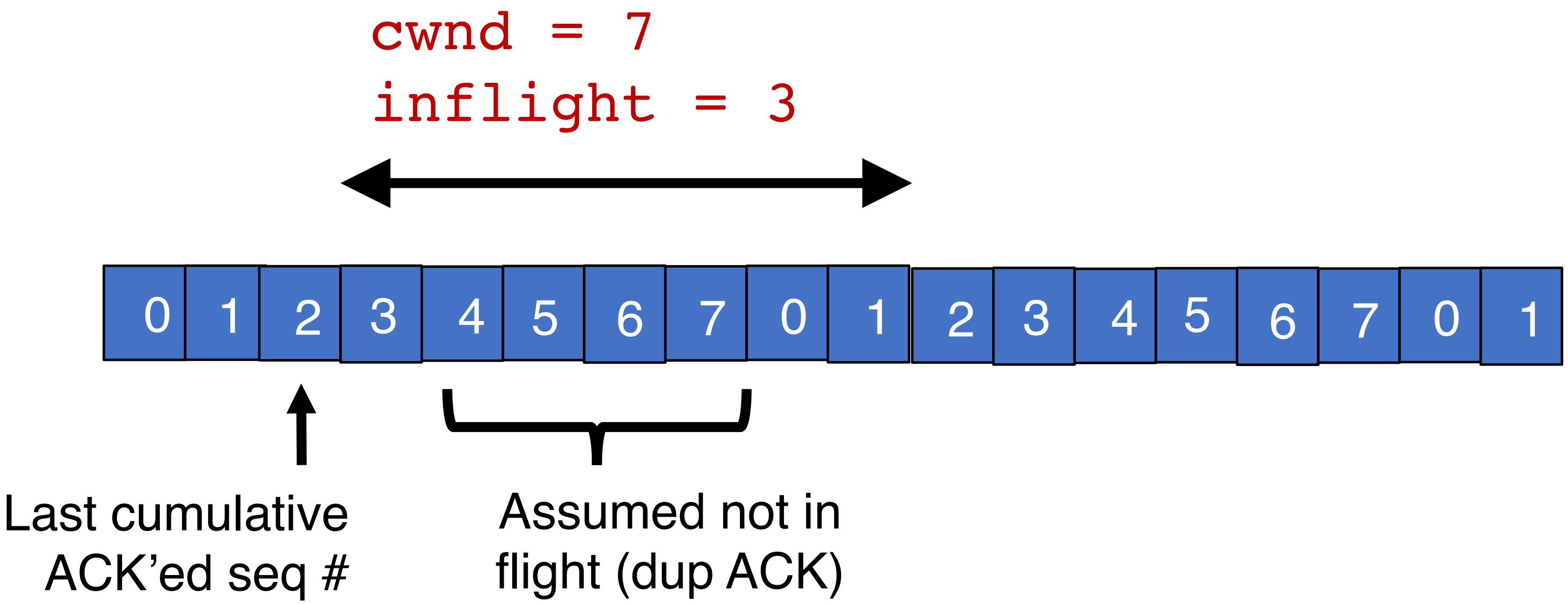
TCP fast recovery (RFC 2581)

- Keep incrementing `cwnd` by 1 MSS for each dup ACK



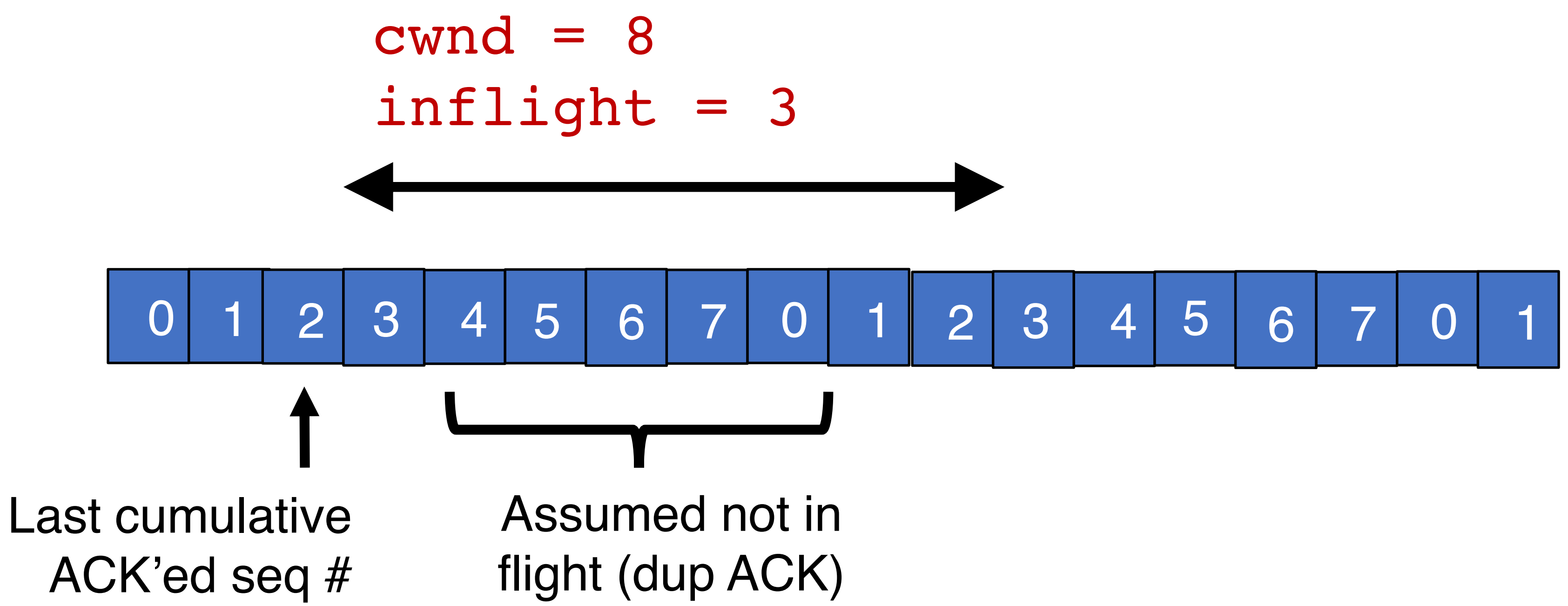
TCP fast recovery (RFC 2581)

- Keep incrementing `cwnd` by 1 MSS for each dup ACK



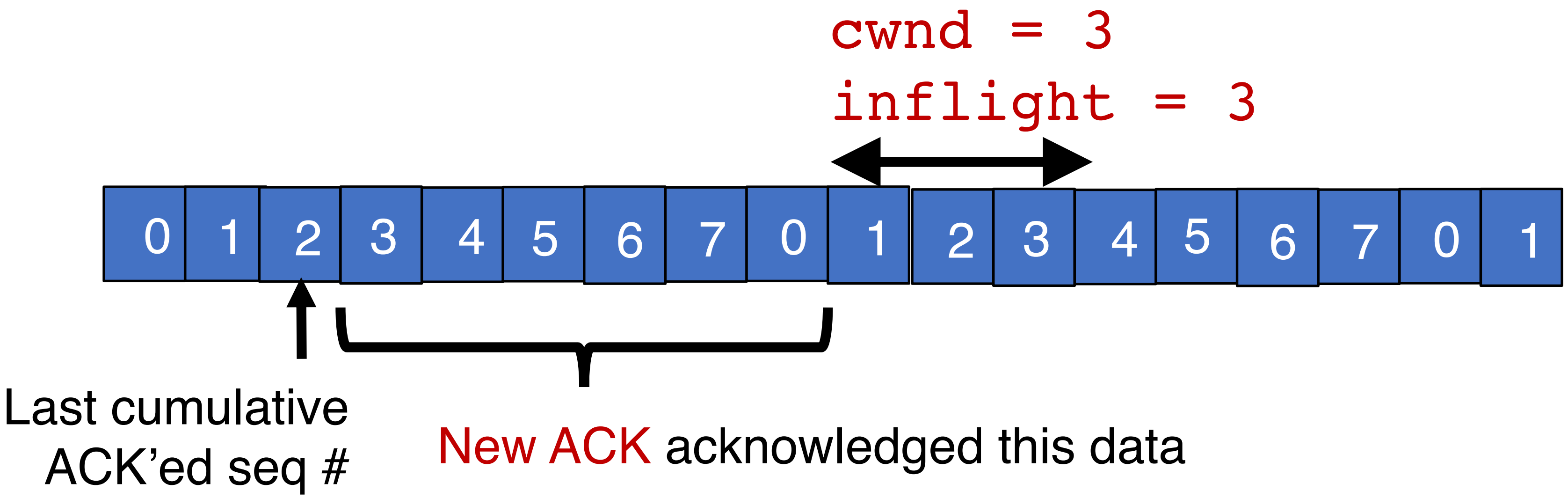
TCP fast recovery (RFC 2581)

- Keep incrementing `cwnd` by 1 MSS for each dup ACK



TCP fast recovery (RFC 2581)

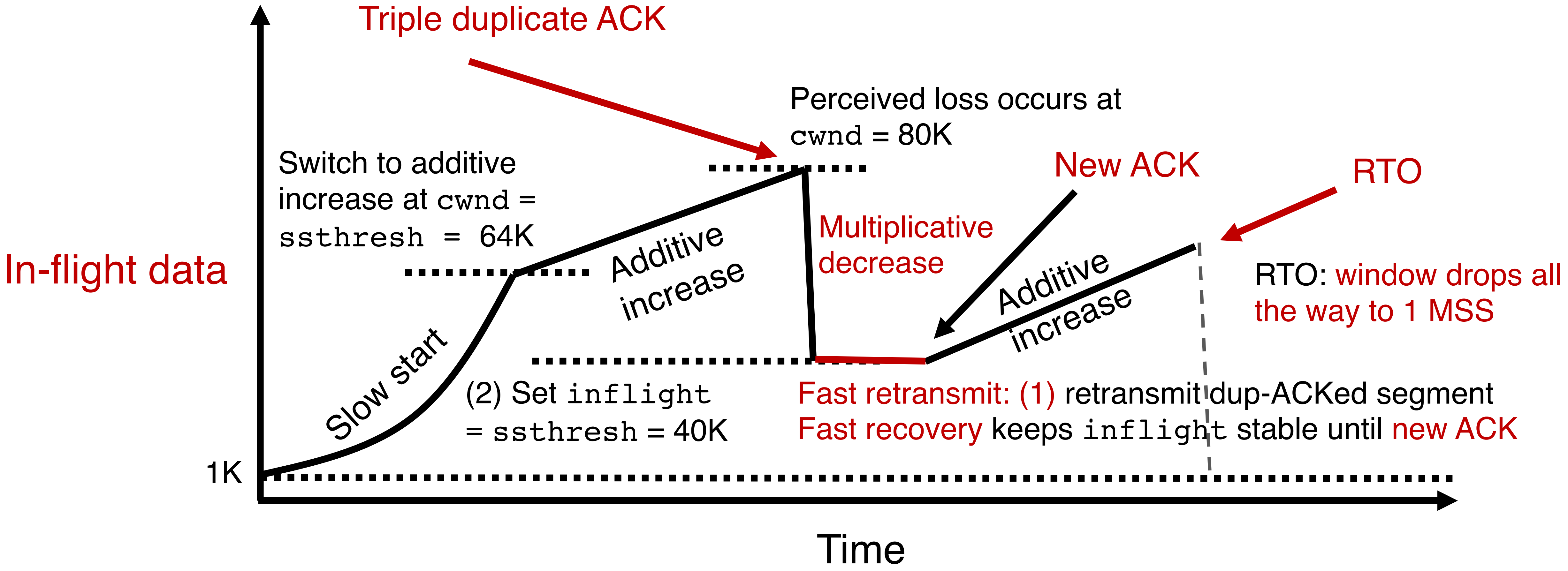
- Eventually a new ACK arrives, acknowledging the retransmitted data and all data in between
- Deflate `cwnd` to half of `cwnd` before fast retransmit.
  - `cwnd` and `inflight` are aligned and equal once again
- Perform additive increase from this point!



Additive Increase/Multiplicative Decrease

Say `MSS` = 1 KByte

Default `ssthresh` = 64KB = 64 `MSS`



TCP New Reno performs additive increase and multiplicative decrease of its congestion window.

In short, we often refer to this as AIMD.

Multiplicative decrease is a part of all TCP algorithms, including BBR.
[It is necessary for fairness across TCP flows.]

Summary of TCP loss detection

- Don't wait for an RTO and then set the `cwnd` to 1 MSS
  - Tantamount to waiting to get super close to the car in front and then jamming the brakes really hard
- Instead, react proportionately by sensing pkt loss in advance

Fast Retransmit

- Triple dup ACK: sufficiently strong signal that network has dropped data, before RTO
- Immediately retransmit data
- Multiplicatively decrease in-flight data to half of its value

Fast Recovery

- Maintain this reduced amount of in-flight data as long as dup ACKs arrive
  - Data is successfully getting delivered
- When new ACK arrives, do additive increase from there on

CS 352
Computing the Retransmit Timeout

CS 352, Lecture 13.2

http://www.cs.rutgers.edu/~sn624/352

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

TCP timeout (RTO)

- Useful for reliable delivery and congestion control
- How to pick the RTO value?
  - Too long: slow reaction to loss
  - Too short: premature retransmissions which are wasteful

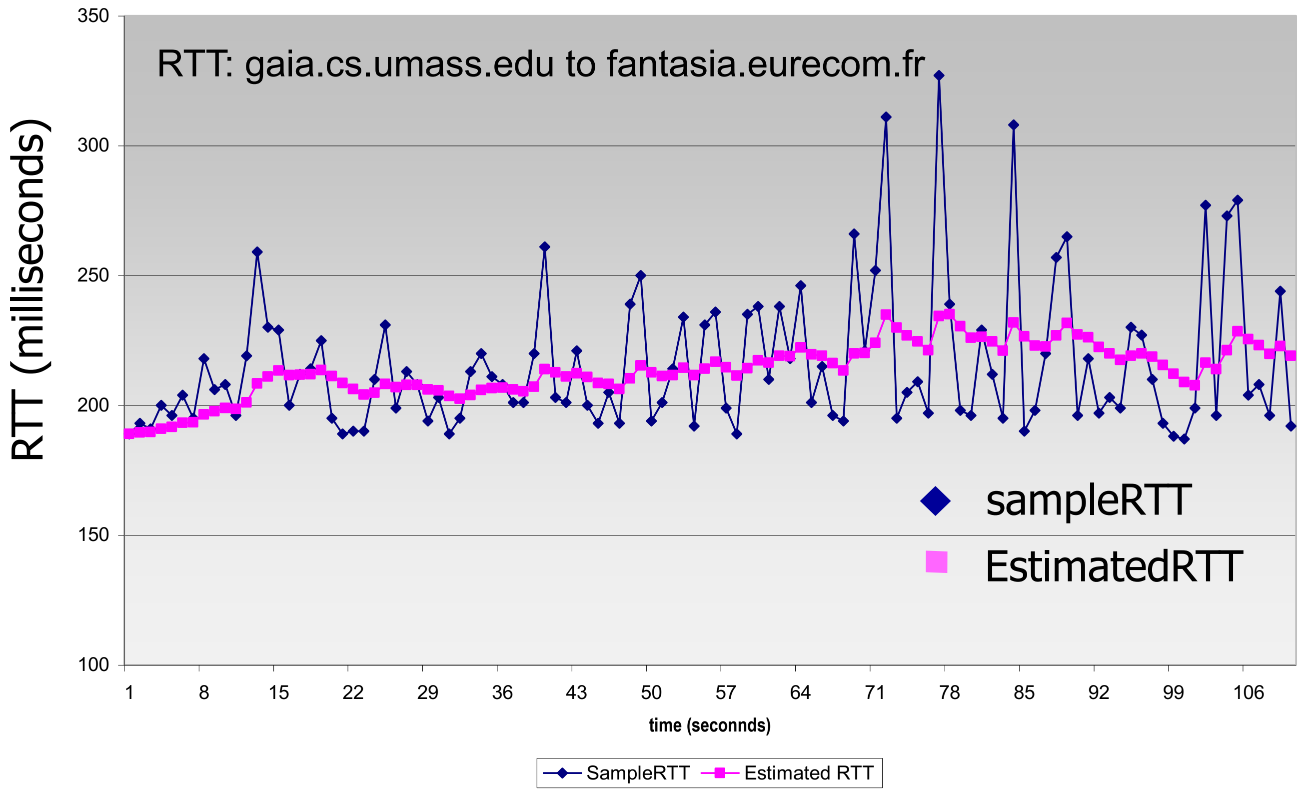
- Intuition: somehow use the observed RTT (`sampleRTT`)
  - Can we just directly set the latest RTT as the RTO?

- RTT can vary significantly!
  - Intermittent congestion, path changes, signal quality changes on wireless channel, etc.

Estimated RTT

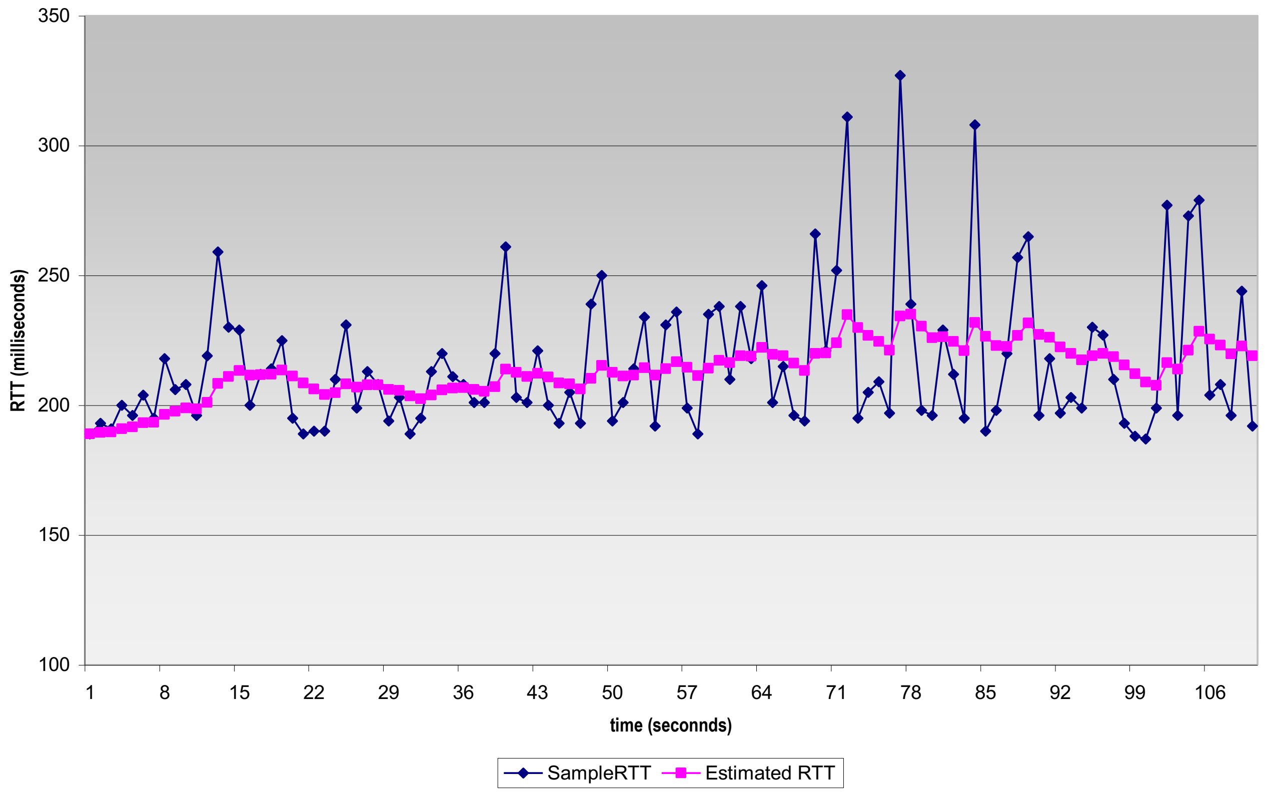
- Exponential weighted moving average (typical alpha = 1/8)

$$\texttt{EstimatedRTT} = (1-\alpha)*\texttt{EstimatedRTT} + \alpha*\texttt{SampleRTT}$$



Timeout == estimated RTT + safety

- Estimated RTT can have a large variance
  - Use a larger safety margin if larger variance



Timeout == estimated RTT + safety

```
DevRTT = (1-β)*DevRTT +
              β*|SampleRTT-EstimatedRTT|

        (typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

safety margin

Managing a single timer

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`

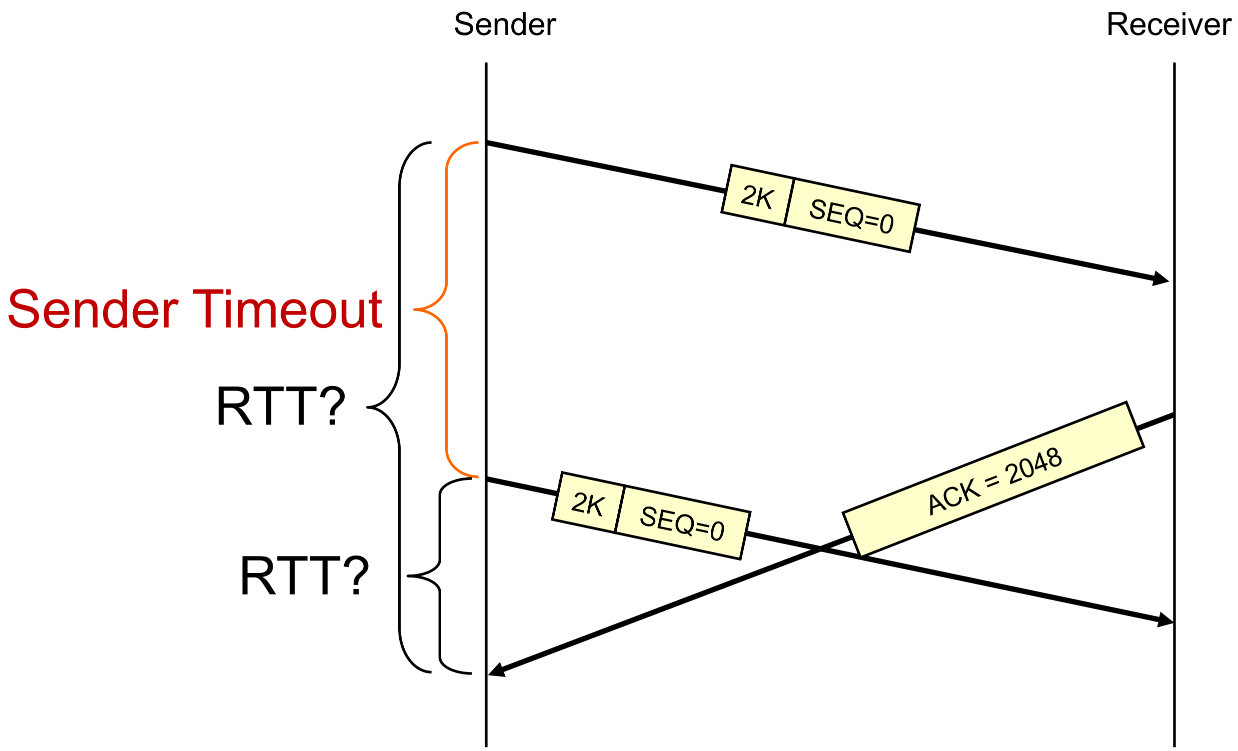
timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

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

Problem with `sampleRTT` calculation



Retransmission ambiguity

- If you retransmitted, how do you measure `sampleRTT` for it?
  - Measure RTT from original data segment?
  - Measure RTT from most recent (retransmitted) segment?
- There could be an error in RTT estimate, since we can't be sure
- One solution
  - Never update RTT measurements based on acknowledgements from retransmitted packets
- Problem: Sudden change in RTT, coupled with many retransmissions, can cause system to update RTT very late
  - Ex: Primary path failure leads to a slower secondary path

Karn's algorithm

- Use back-off as part of `sampleRTT` computation
- Whenever packet loss, RTO is increased by a factor
- Use this increased RTO as RTO estimate for the next segment (not from EstimatedRTT)
- Only after an acknowledgment received for a successful transmission is the timer set to new RTT obtained from EstimatedRTT

CS 352
TCP Connection Establishment

CS 352, Lecture 13.3

http://www.cs.rutgers.edu/~sn624/352

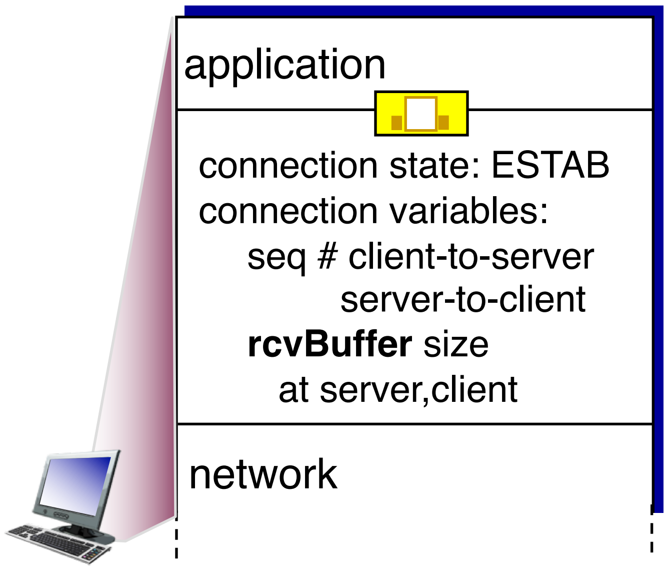
Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

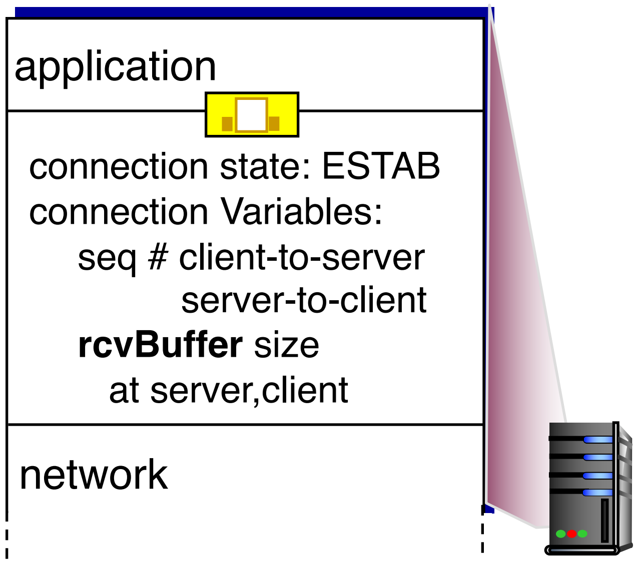
Connection Management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection
- agree on connection parameters



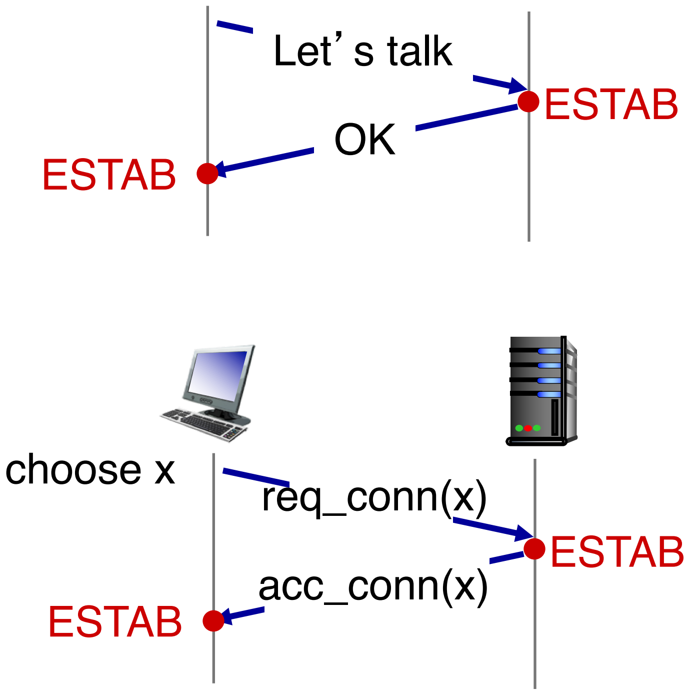

```
Socket clientSocket =
  newSocket("hostname","port
  number");
```




```
Socket connectionSocket =
  welcomeSocket.accept();
```

Agreeing to establish a connection

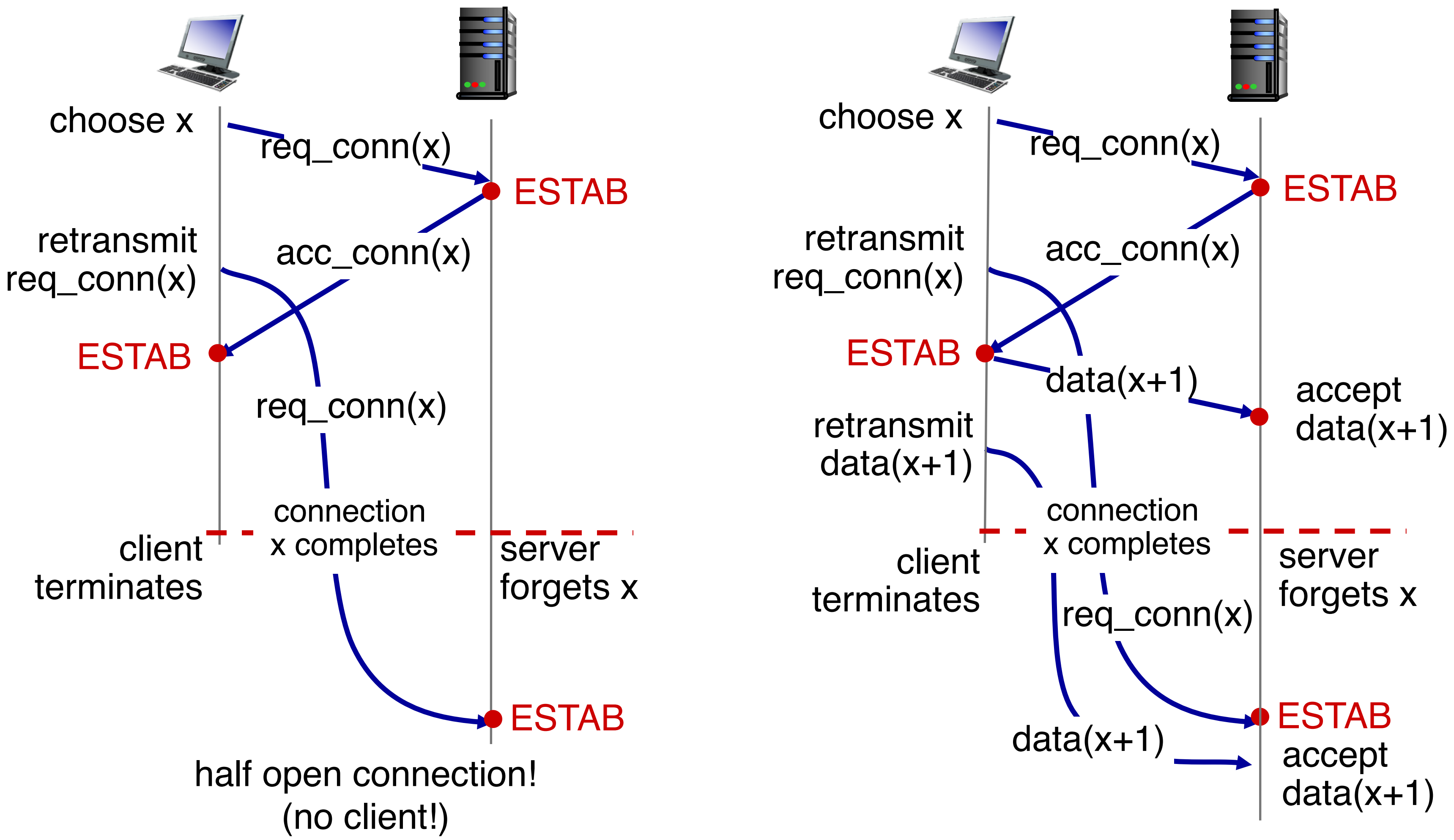
2-way handshake:



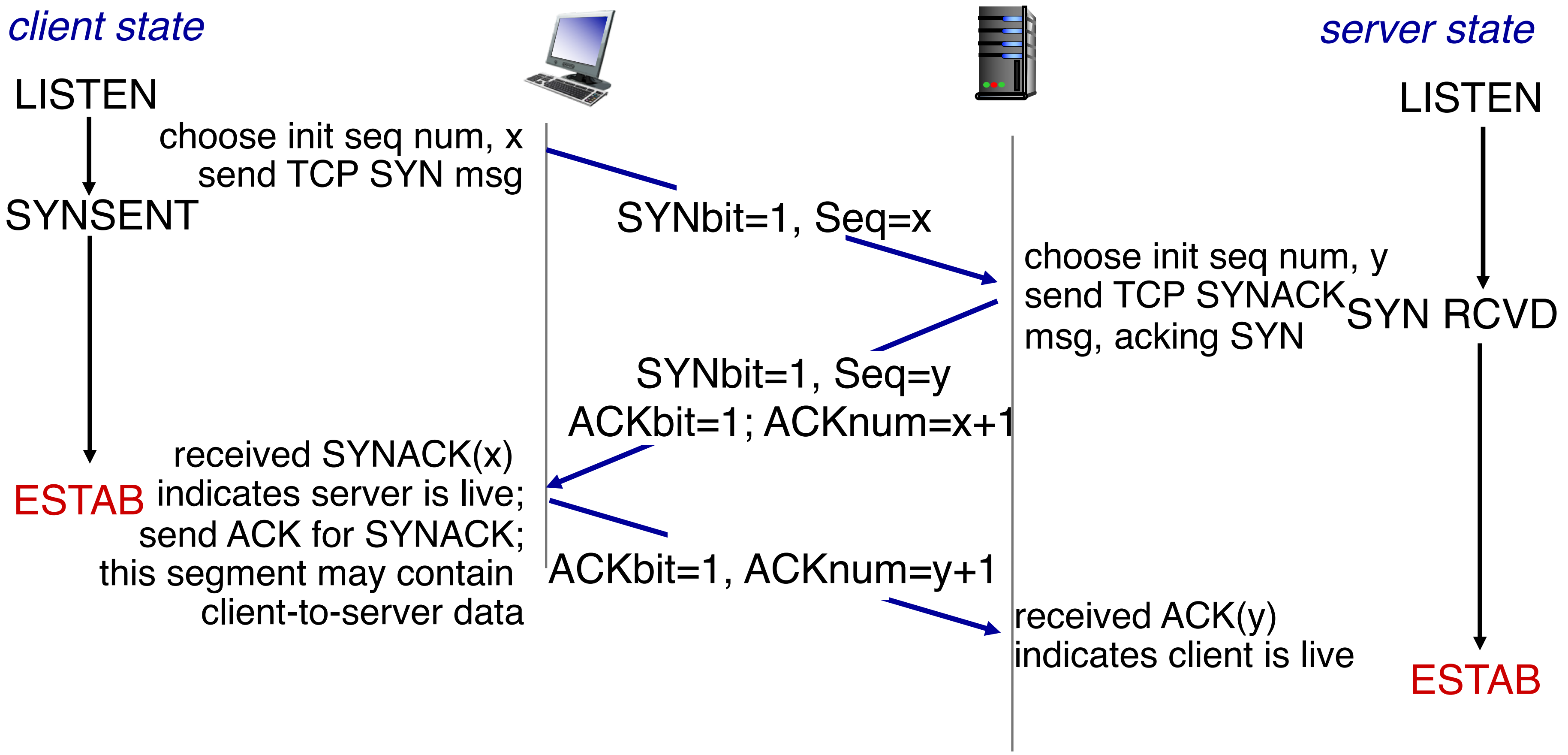
Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can’t “see” other side

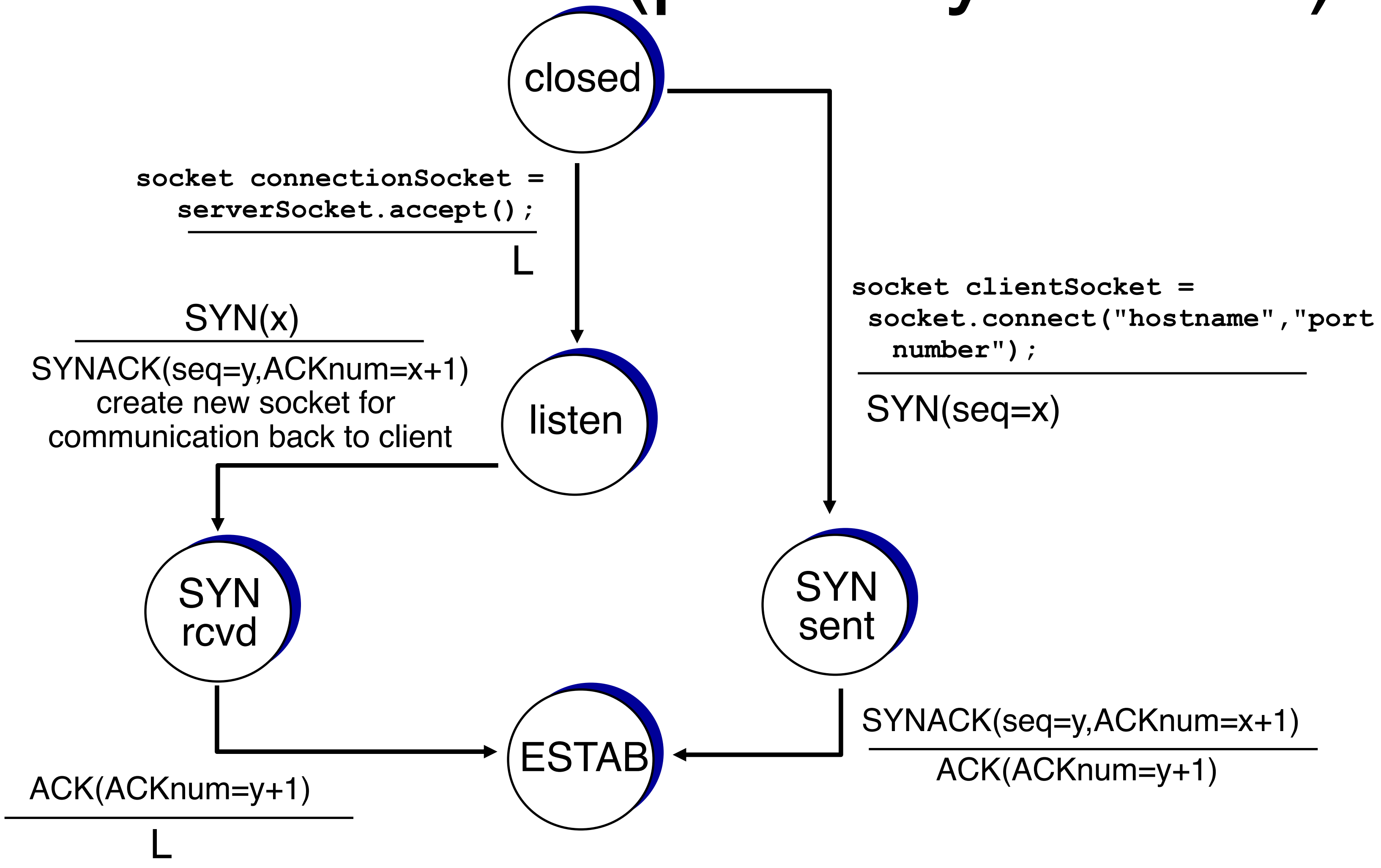
2-way handshake failure scenarios



TCP 3-way handshake



TCP state machine (partially shown)



TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1

- In general, TCP is full-duplex: both sides can send

- But FIN is unidirectional: stop one side of the communication

- 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