# The Transport Layer: TCP Timeouts and Connection management

CS 352, Lecture 11, Spring 2020

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

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

# Course announcements

- Project 2 will go online today
- Quiz 4 will go online today
  - Due Tue 03/10 at 10 PM

- Mid-term grades will be released this weekend
  - Papers in class on Wednesday

# Review of concepts

- TCP congestion control: need distributed, efficient, fair
  - signals (loss) and knobs (congestion window)
- ACK clocking
- Slow start
- Additive increase and the slow start threshold
- Multiplicative decrease
  - Triple duplicate ACKs and fast retransmit

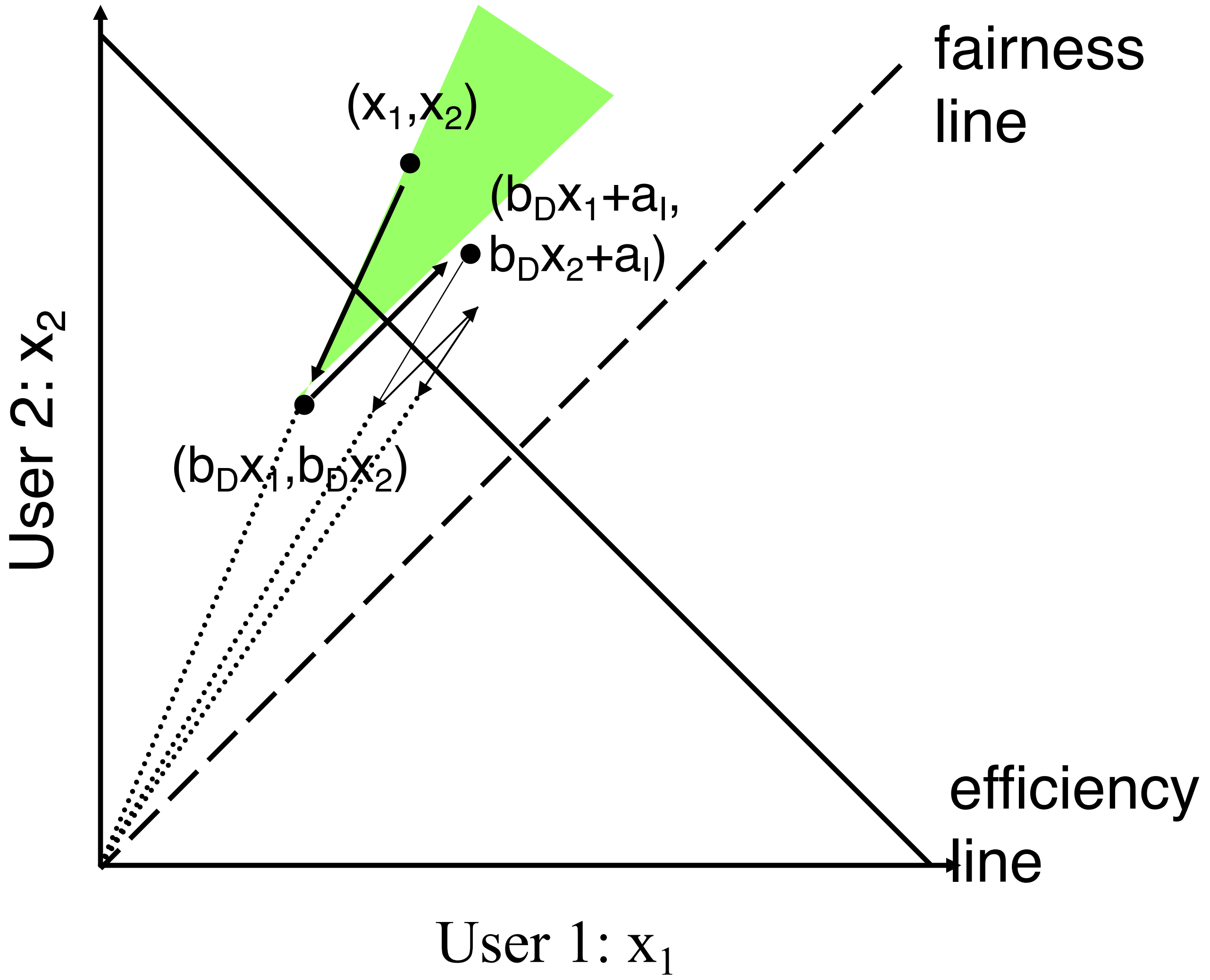
TCP performs additive increase and multiplicative decrease of its congestion window.

This is often termed AIMD.

AIMD results in the so-called TCP sawtooth.

# Why AIMD?

- Converges to fairness
- Can also show it converges to efficiency
  - Intuition: Increments to rate get smaller as fairness increases



# Calculating the TCP timeout

# 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 unnecessary retransmissions

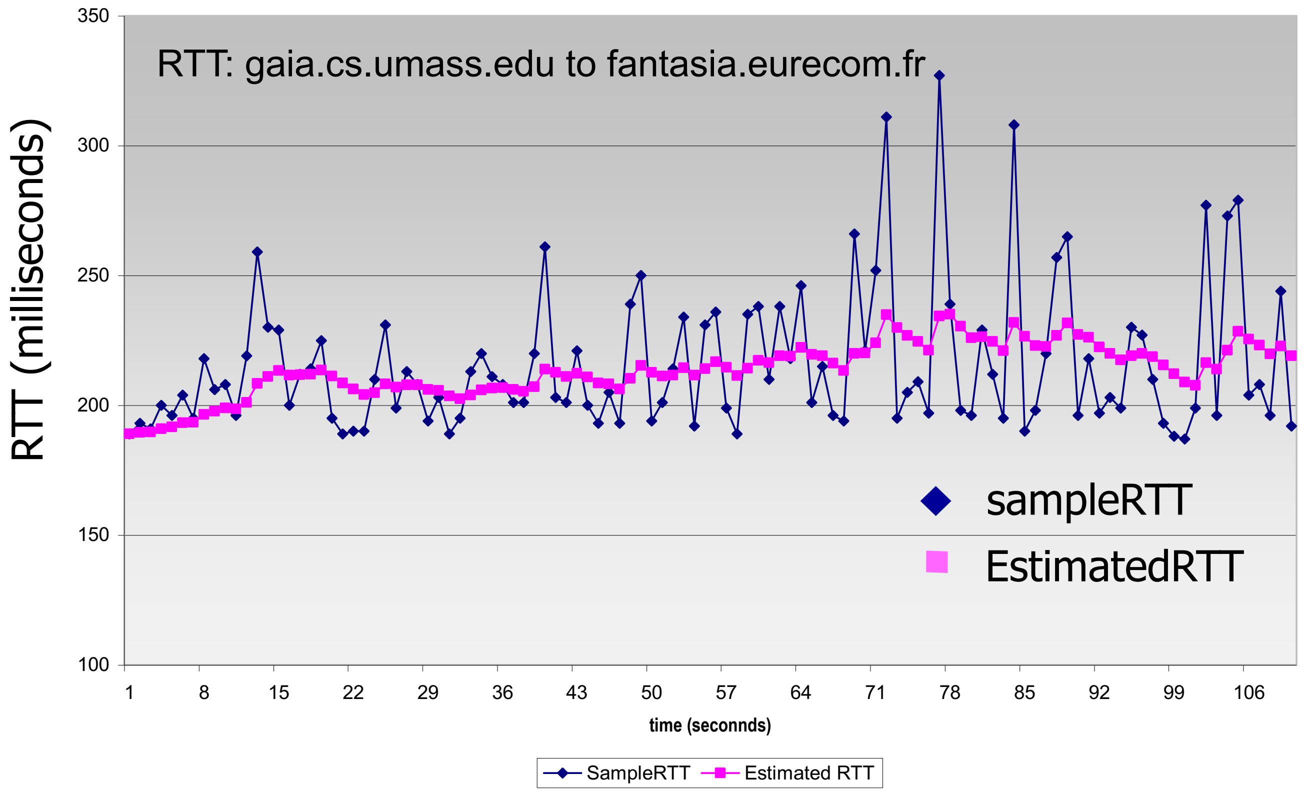
- Intuition: somehow use the observed RTT (`sampleRTT`)
  - Can we just directly set the latest RTT as the RTO?

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

# Estimated RTT

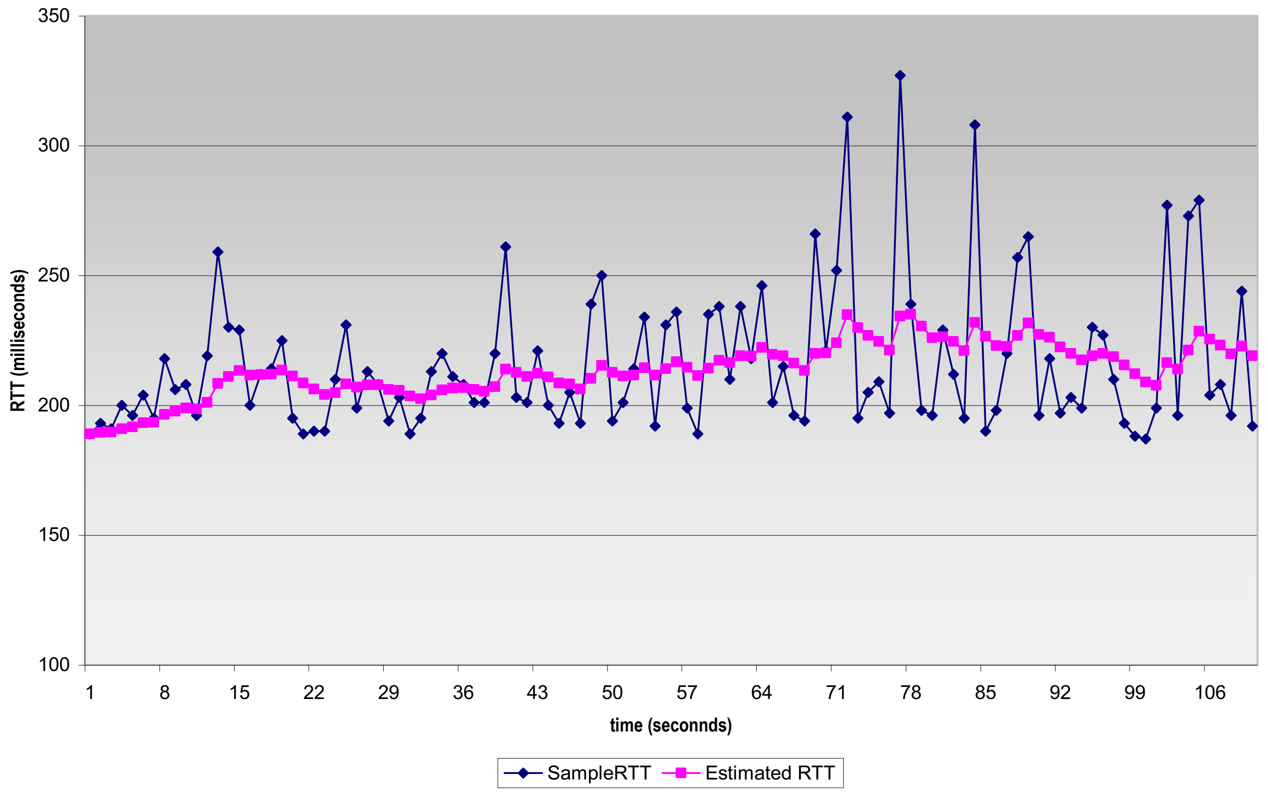
- Exponential weighted moving average (typical alpha = 1/8)

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



# 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`

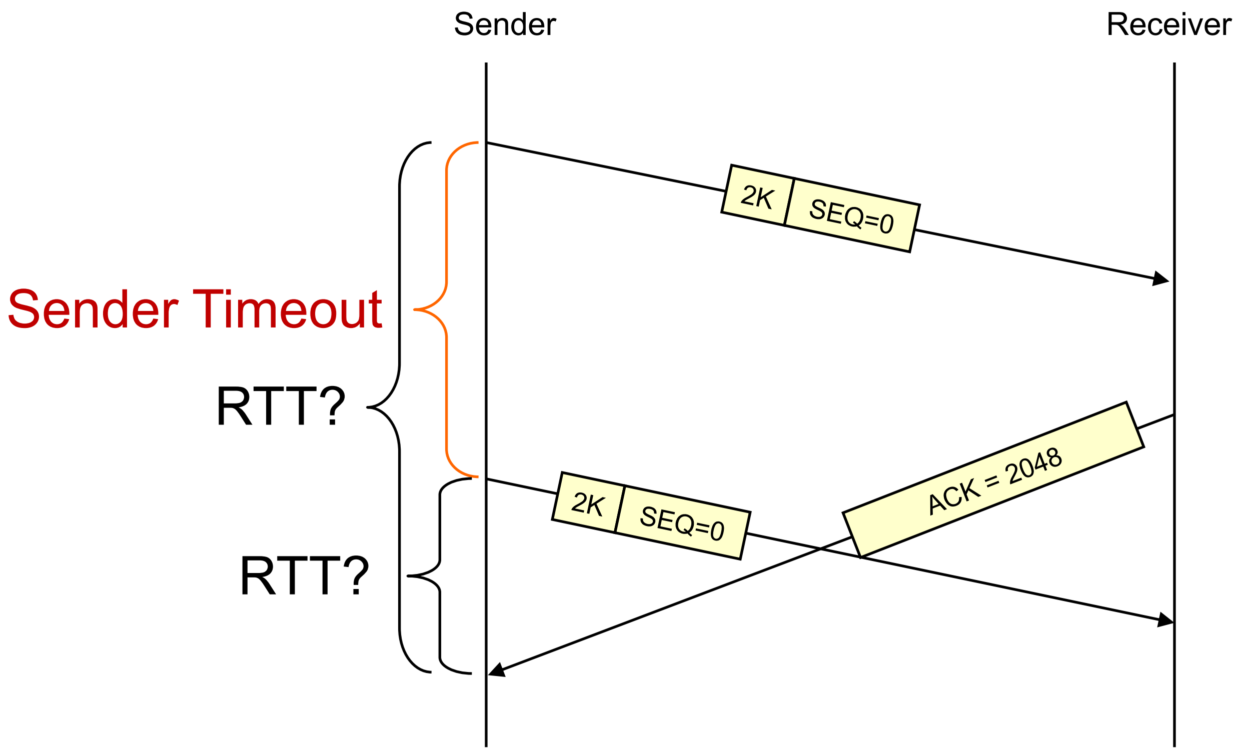
timeout:

- 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

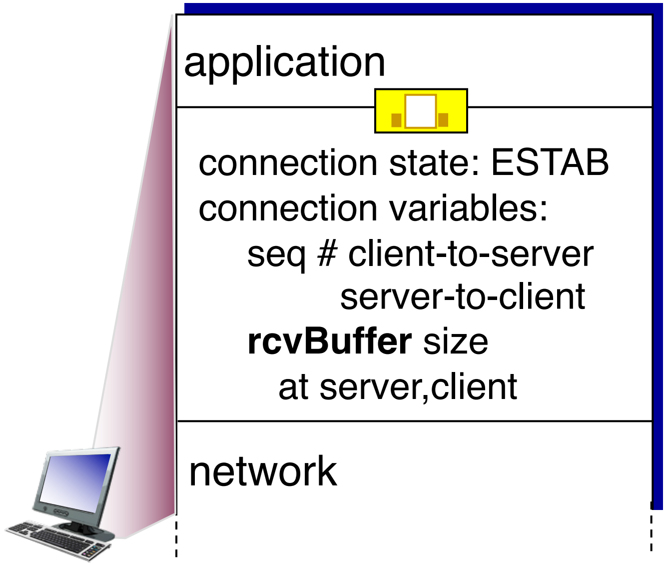
- Use back-off as part of `sampleRTT` computation
- Whenever packet loss, RTO is increased by a factor
- Use this increased RTO as RTO estimate for the next segment (not from EstimatedRTT)
- Only after an acknowledgment received for a successful transmission is the timer set to new RTT obtained from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

# Connection Management

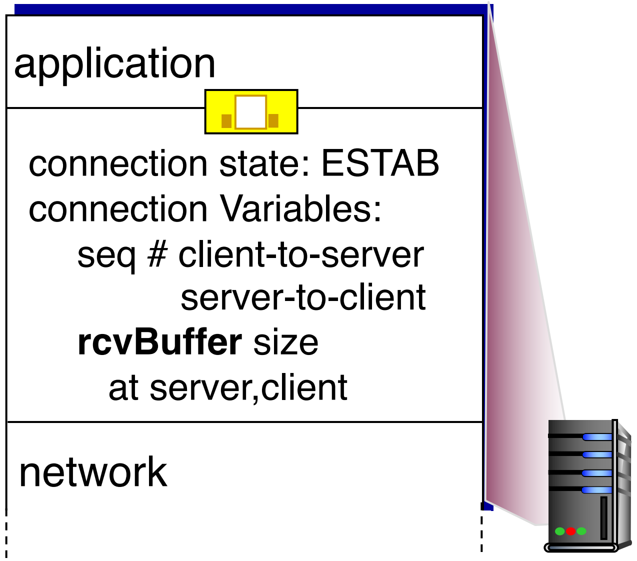
# Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection
- agree on connection parameters



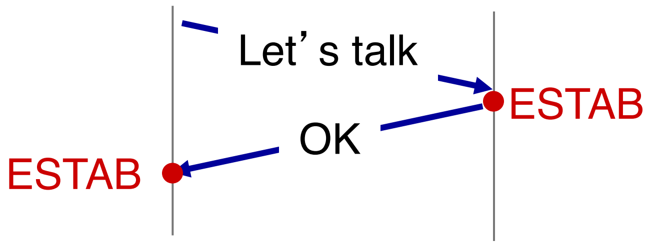

```
Socket clientSocket =
  newSocket("hostname","port
  number");
```

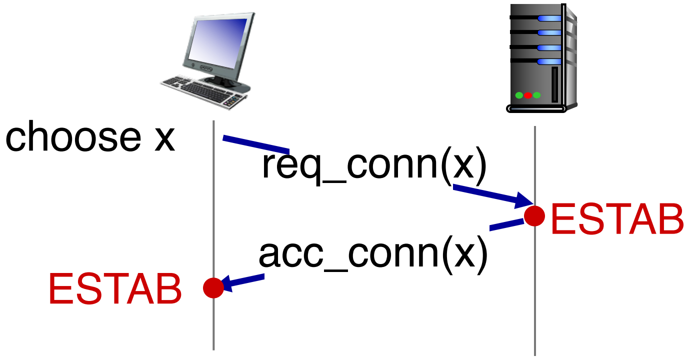



```
Socket connectionSocket =
  welcomeSocket.accept();
```

# Agreeing to establish a connection

2-way handshake:

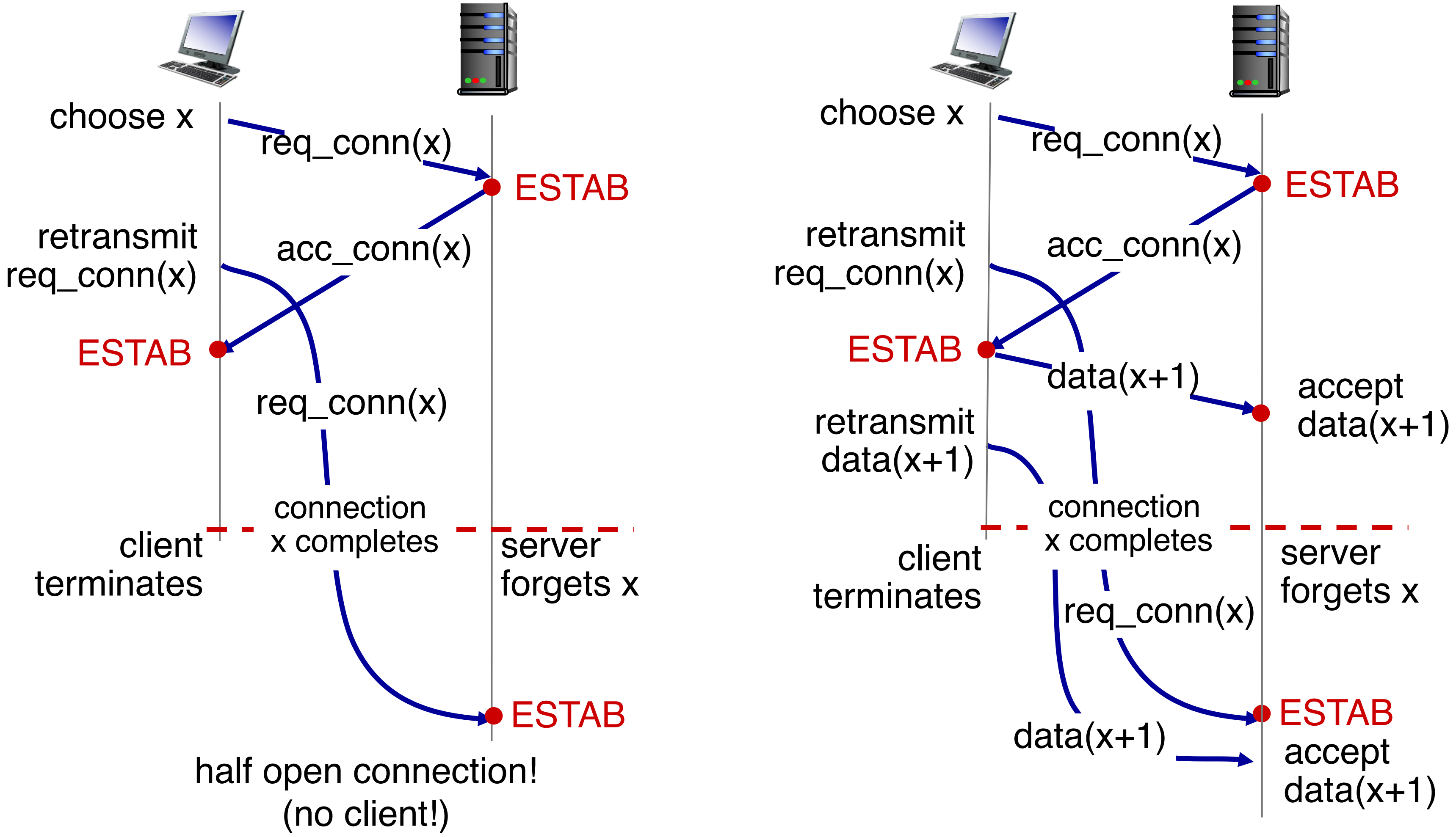


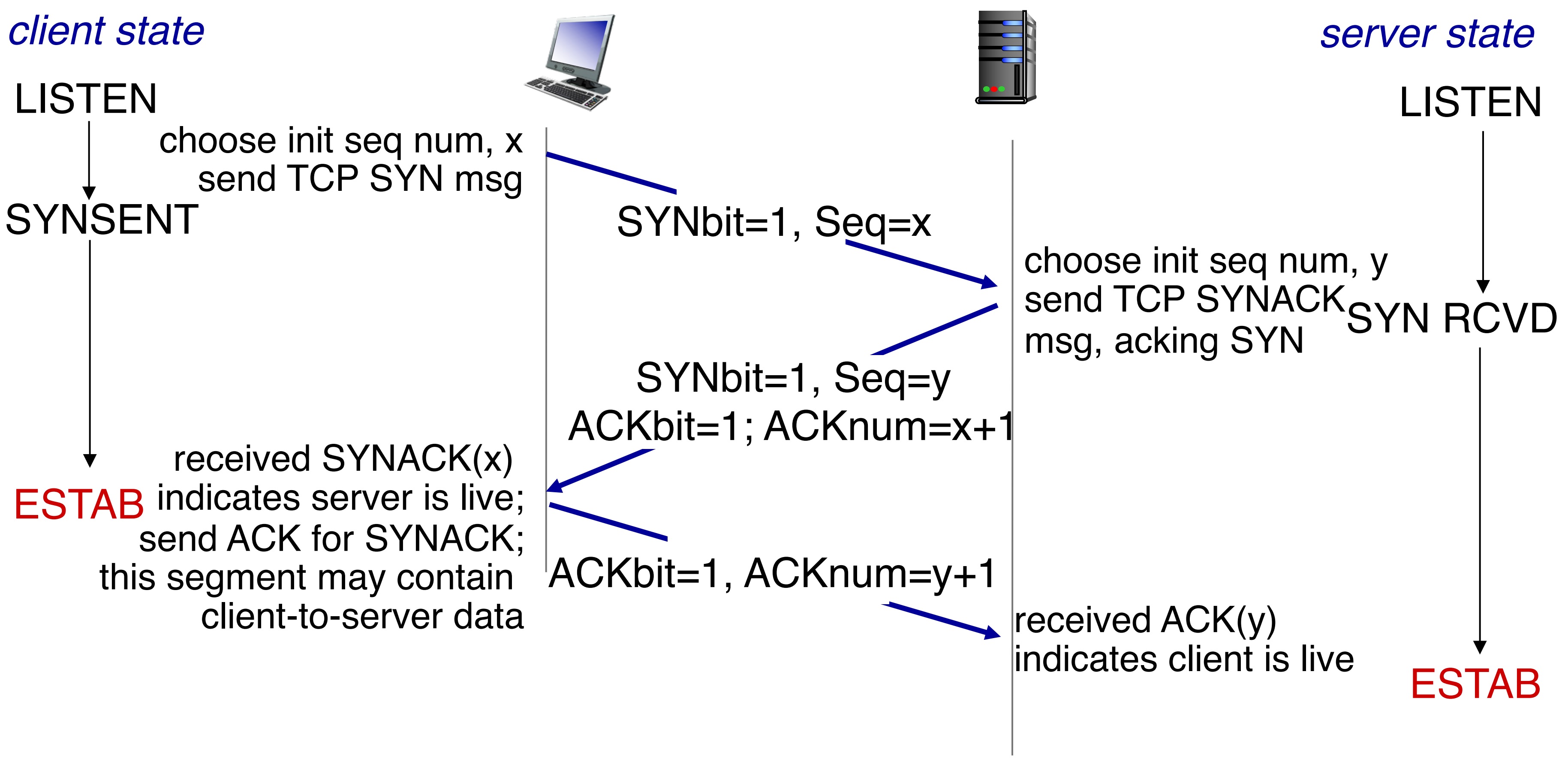

*Q:* will 2-way handshake always work in network?

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

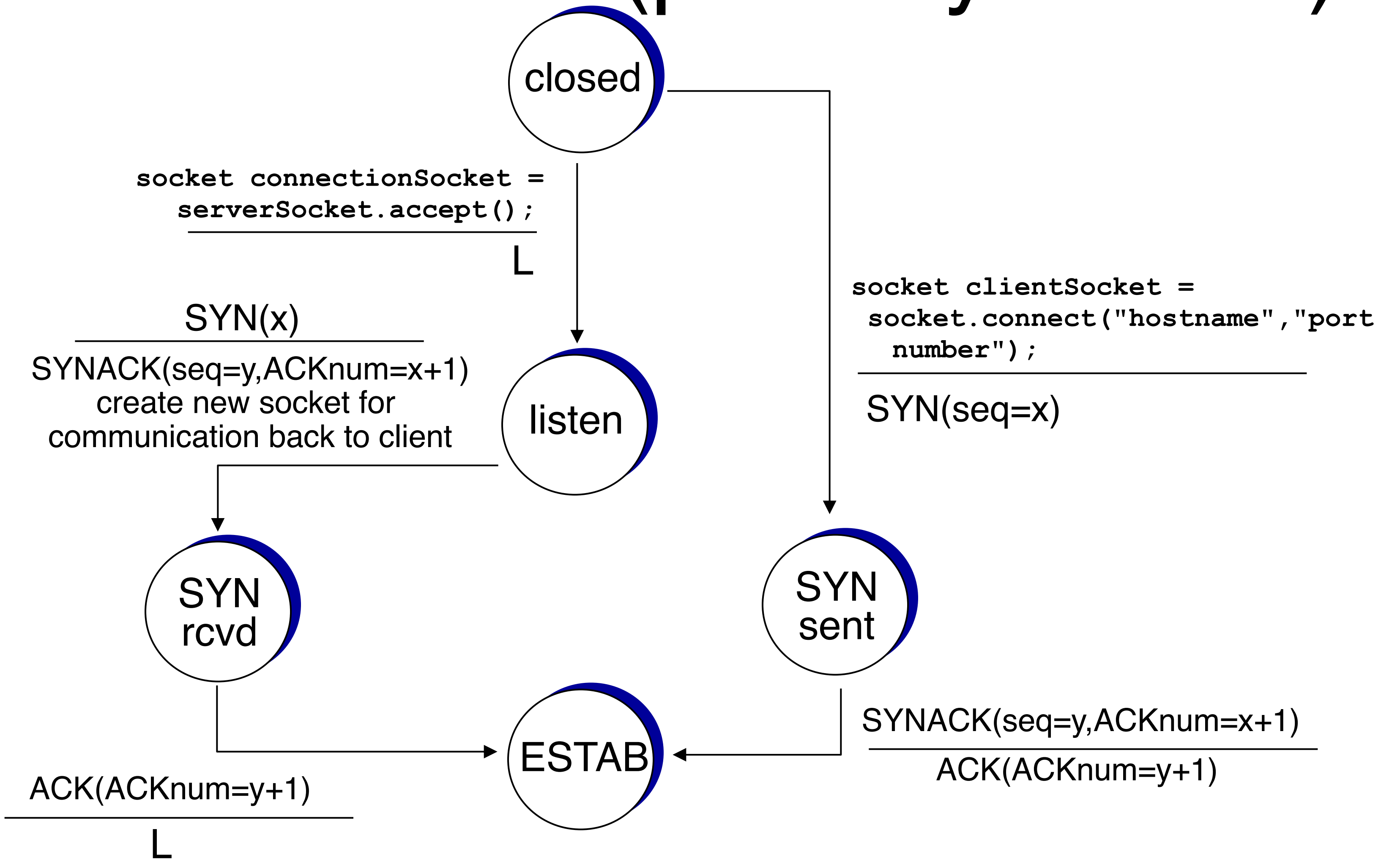
# 2-way handshake failure scenarios



# TCP 3-way handshake



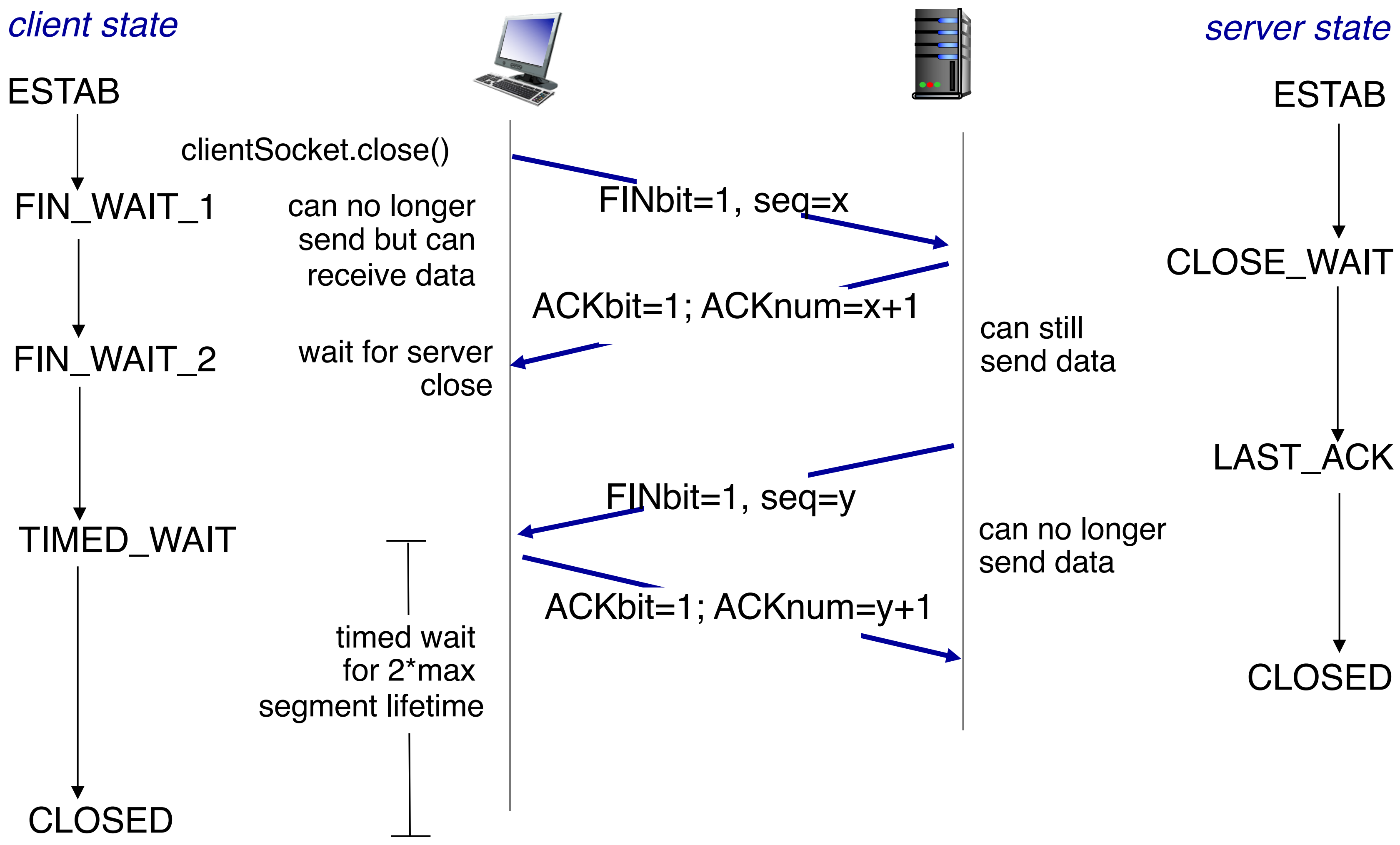
# TCP state machine (partially shown)



# 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



# TCP summary

- Reliability
- Ordering
- Flow control
- Congestion control
- Timeout computation
- Connection management, state machine