CS 352
Bandwidth-Delay Product

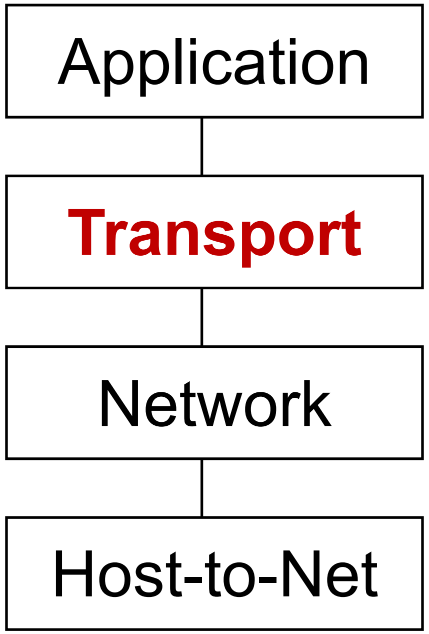
CS 352, Lecture 13.1

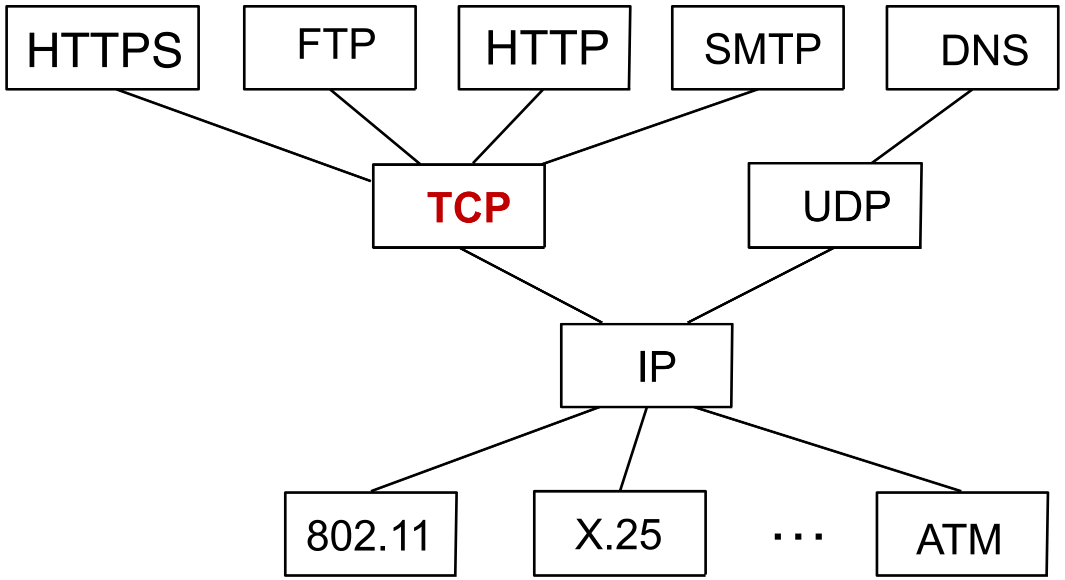
http://www.cs.rutgers.edu/~sn624/352

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

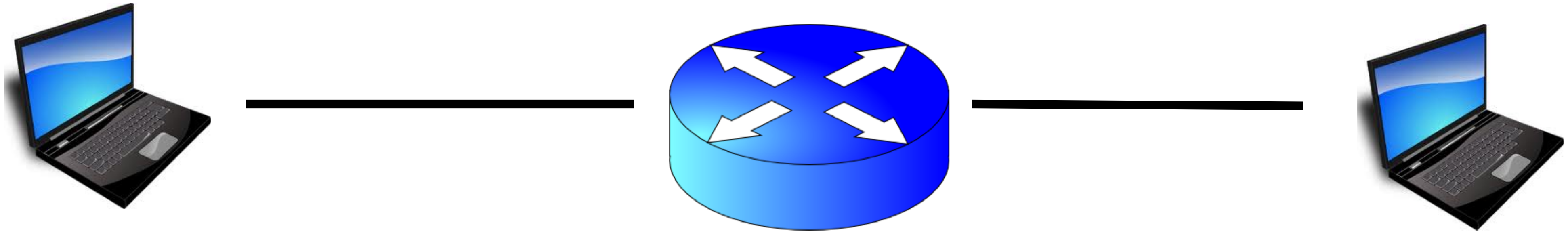
Transport


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

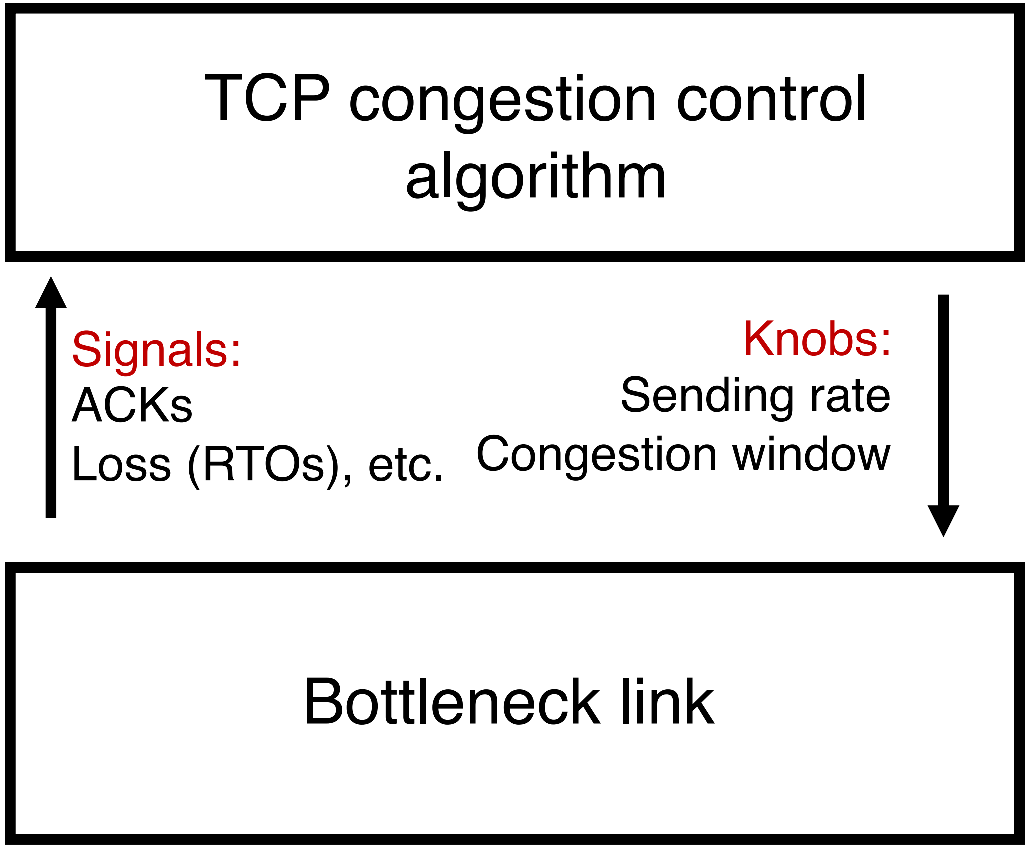


- Transport software on the endpoint oversees implementing guarantees on top of a best-effort network
- Three important kinds of guarantees
  - Reliability
  - Ordered delivery
  - Resource sharing in the network core

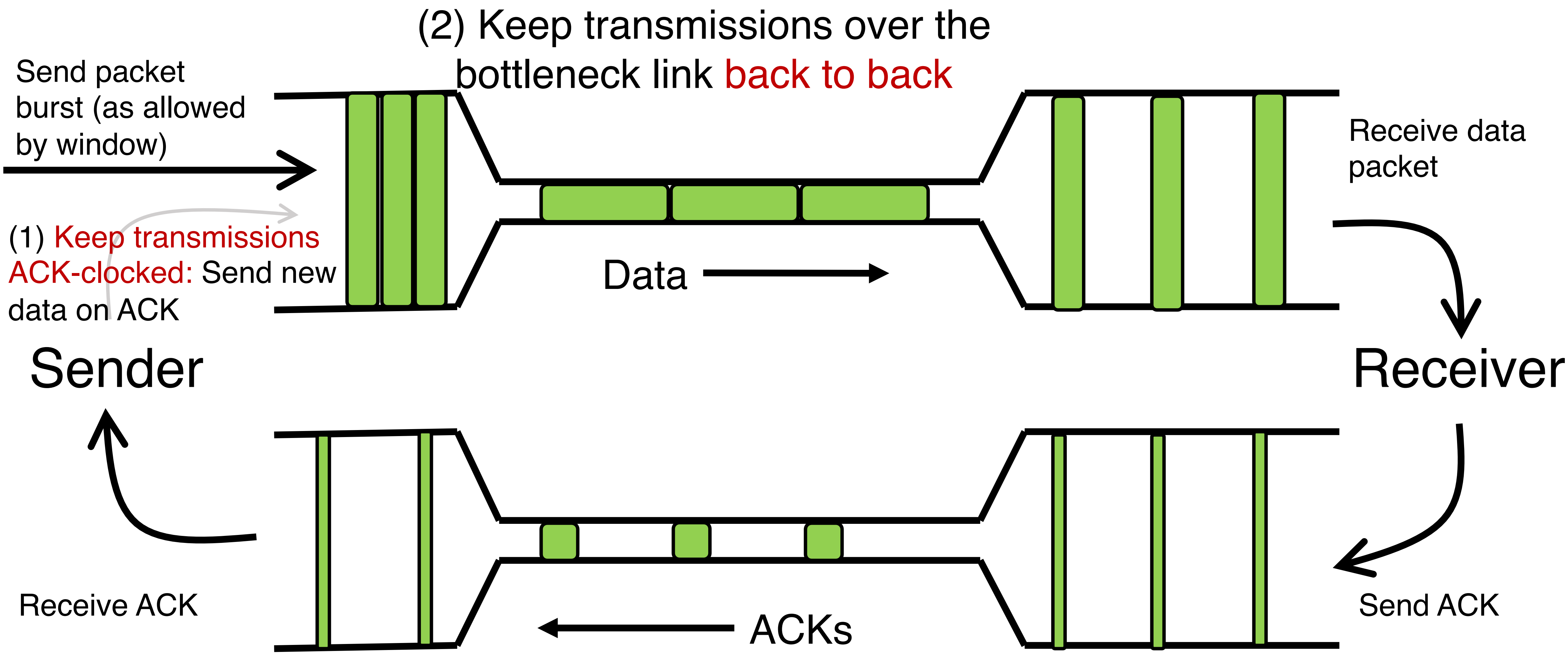
Transmission Control Protocol (TCP)

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



Goal of steady state operation



Steady state `cwnd` for a single flow

- Suppose the bottleneck link has rate C
- Suppose the propagation round-trip delay (propRTT) between sender and receiver is T
- Ignore transmission delays for this example;
- Assume steady state: highest sending rate with no bottleneck congestion

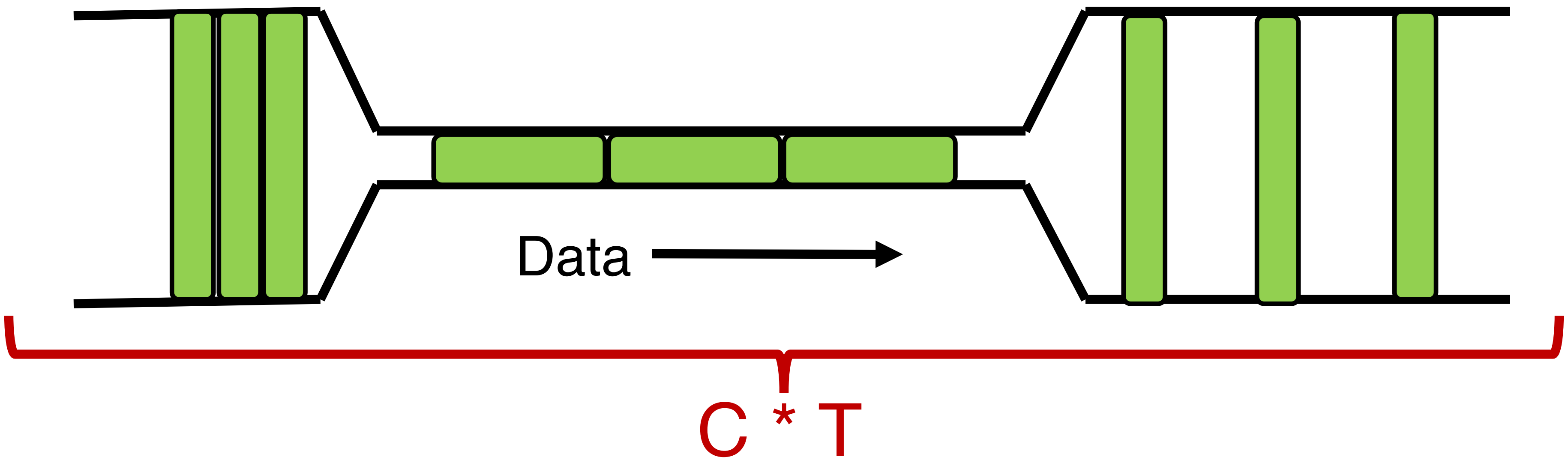
- Q: how much data is in flight over a single RTT?

- C * T data i.e., amount of data unACKed at any point in time
- ACKs take time T to arrive (without any queueing). In the meantime, sender is transmitting at rate C

The Bandwidth-Delay Product

- C * T = bandwidth-delay product:
  - The amount of data in flight for a sender transmitting at the ideal rate during the ideal round-trip delay of a packet

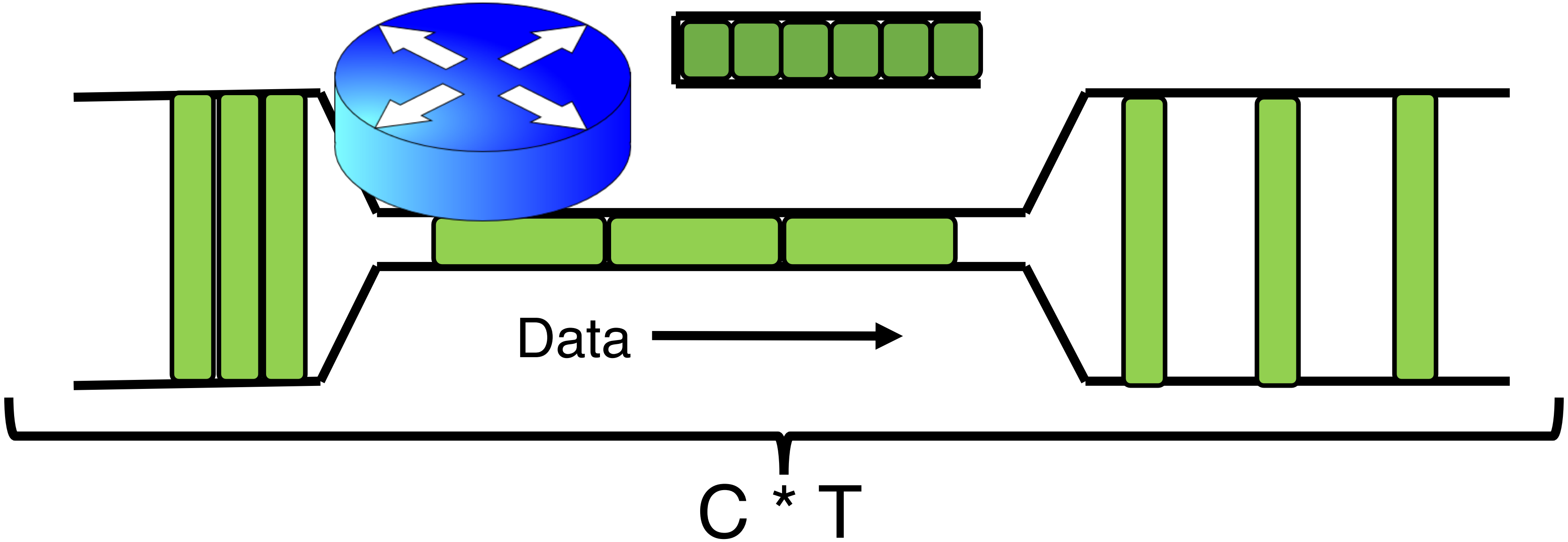
- Note: this is just the amount of data “on the pipe”



The Bandwidth-Delay Product

- Q: What happens if cwnd > C * T?
  - i.e., where are the rest of the in-flight packets?

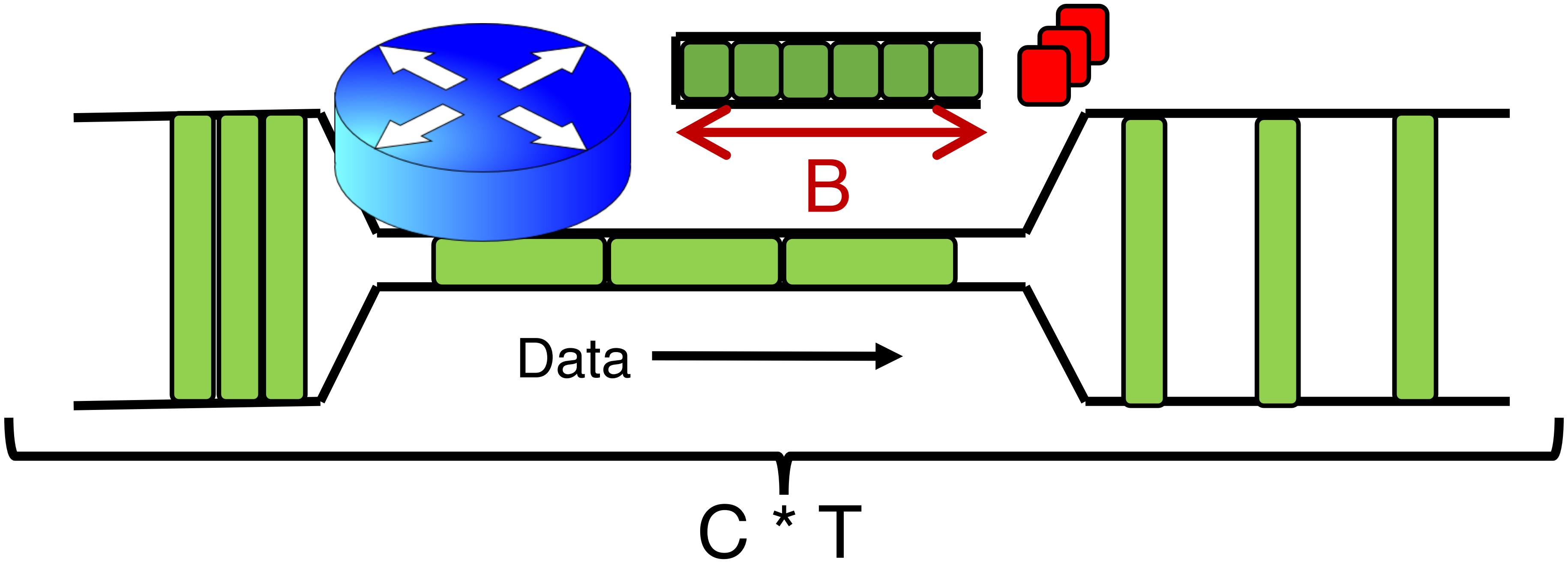
- A: Waiting at the bottleneck router queues



Router buffers and the max `cwnd`

- Router buffer memory is finite: queues can only be so long
  - If the router buffer size is B, there is at most B data waiting in the queue

- If cwnd increases beyond C * T + B, data is dropped!



Summary

- Bandwidth-Delay Product (BDP) governs the window size of a single flow at steady state
- The bottleneck router buffer size governs how much the `cwnd` can exceed the BDP before packet drops occur

CS 352
Detecting & Reacting to Losses

CS 352, Lecture 13.2

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

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

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?

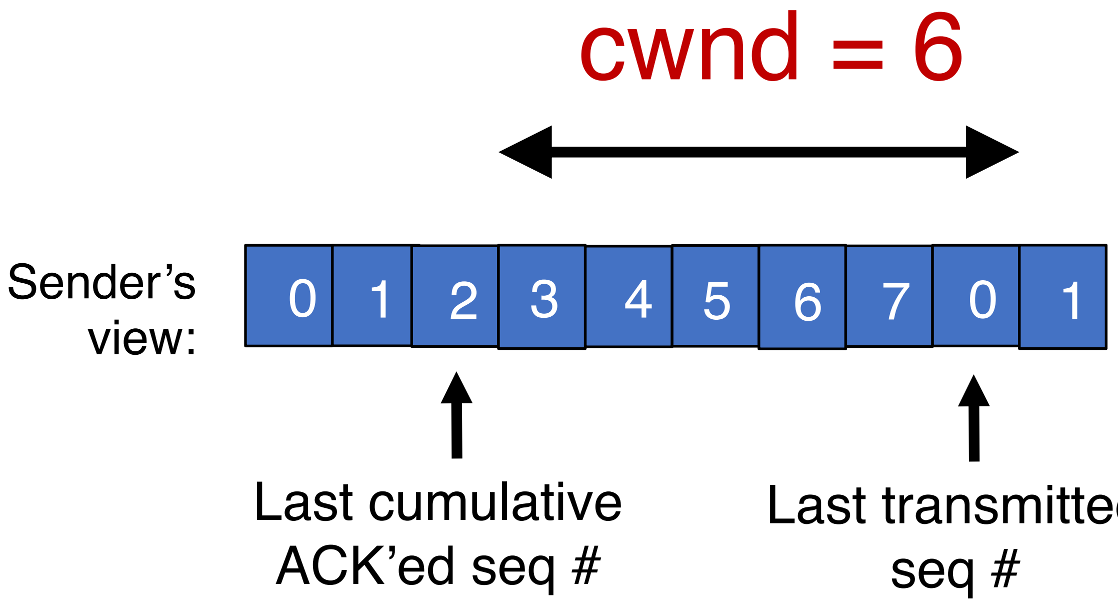
Can we detect loss earlier than RTO?

- 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

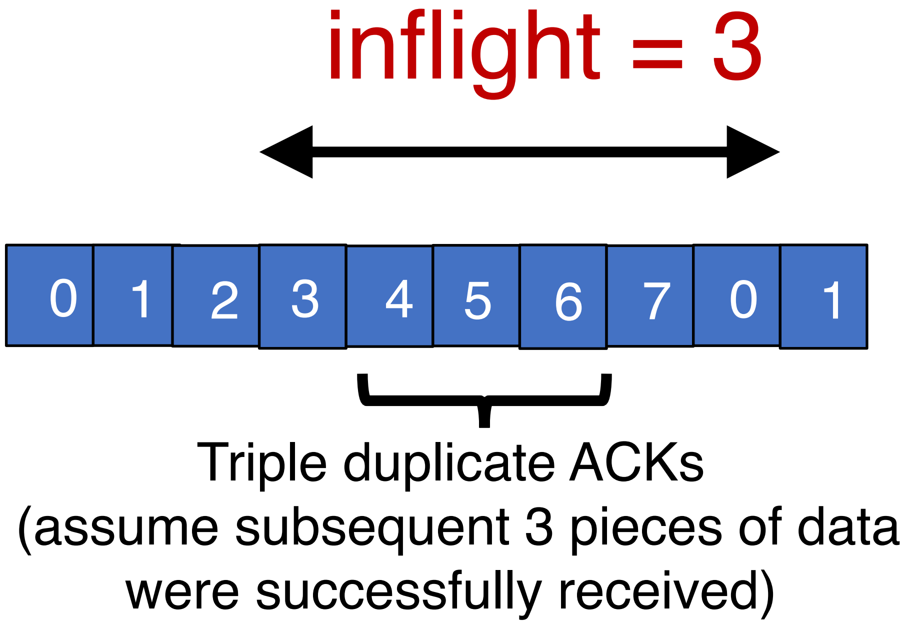
Fast Retransmit & Fast Recovery

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

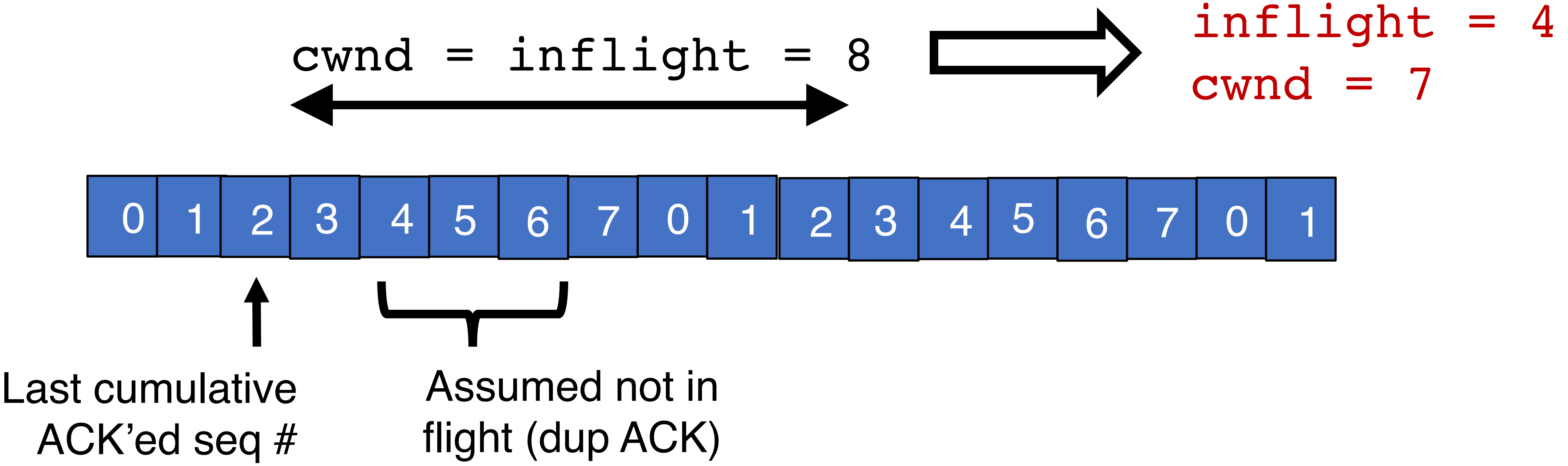
TCP fast retransmit (RFC 2581)

- (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

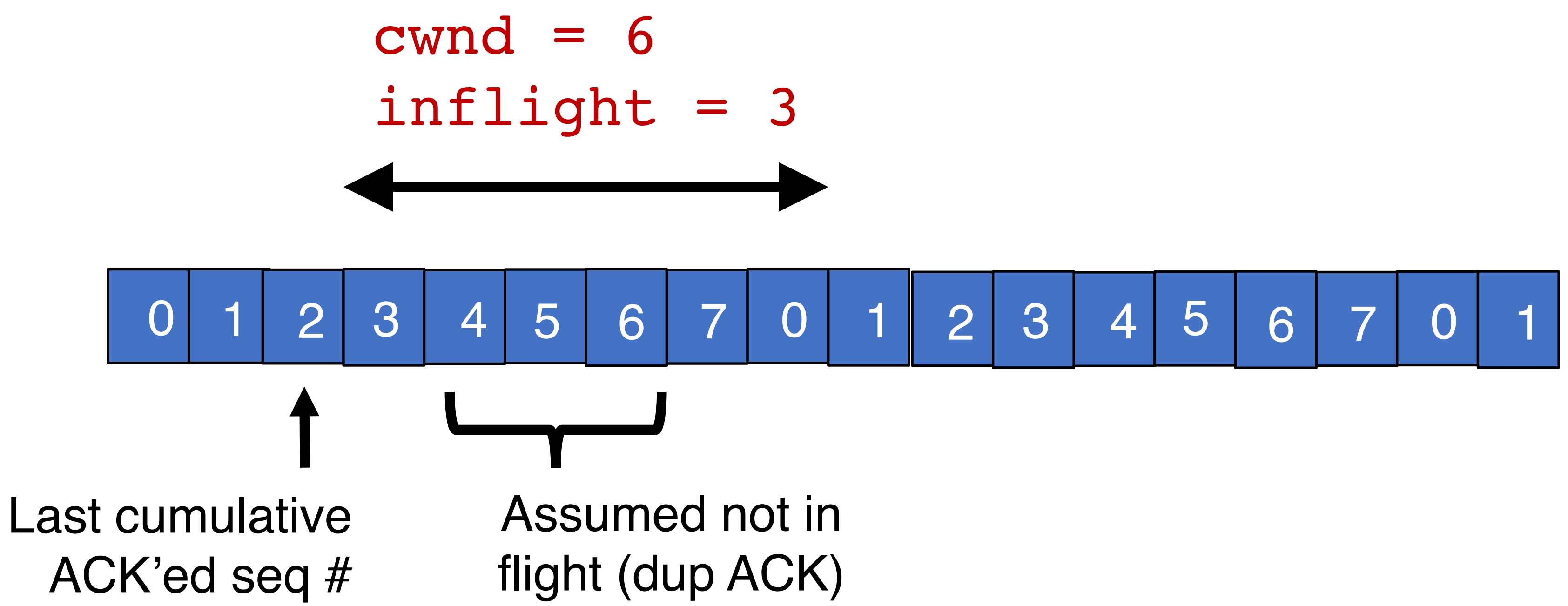
TCP fast recovery (RFC 2581)

- 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)

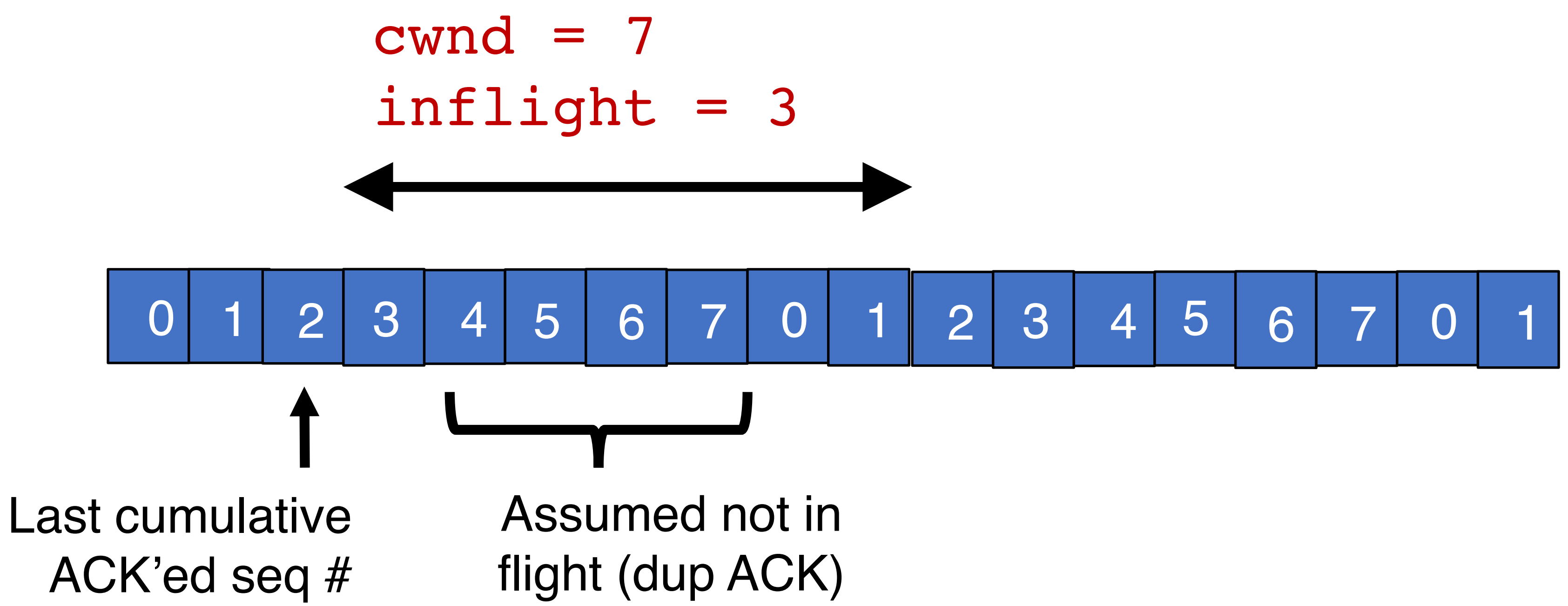
TCP fast recovery (RFC 2581)

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



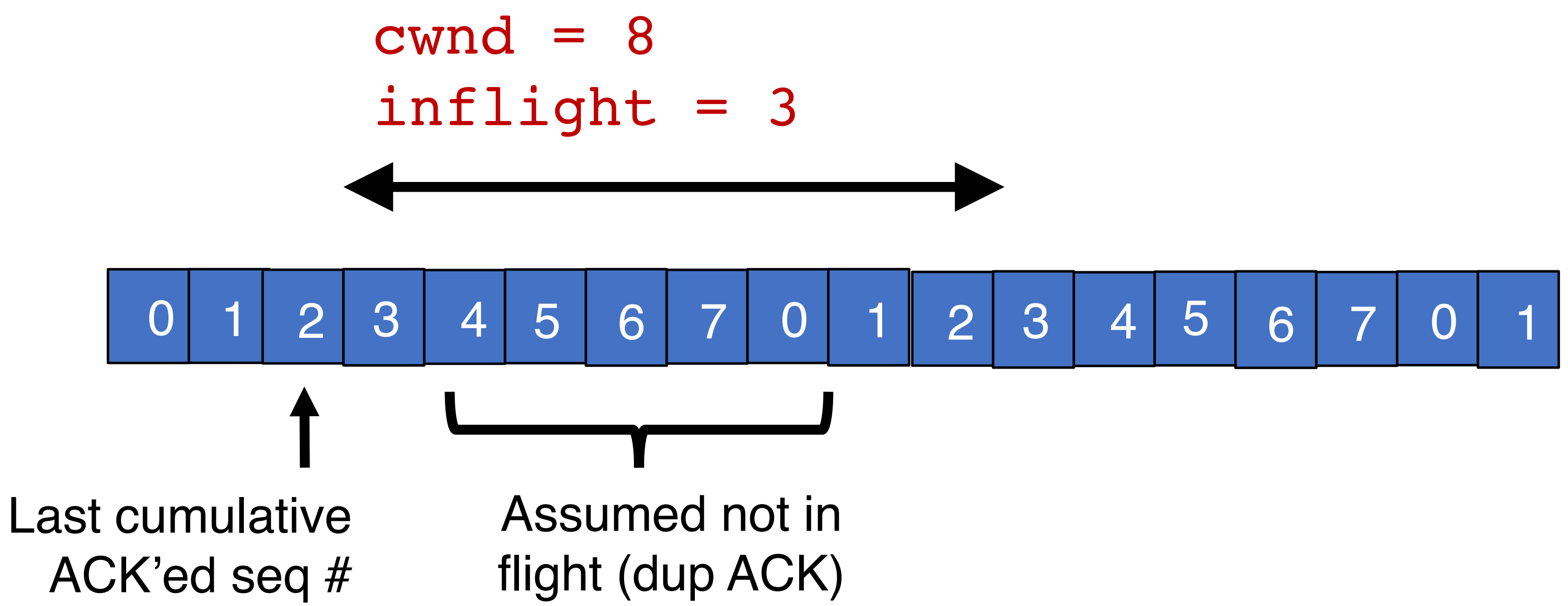
TCP fast recovery (RFC 2581)

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



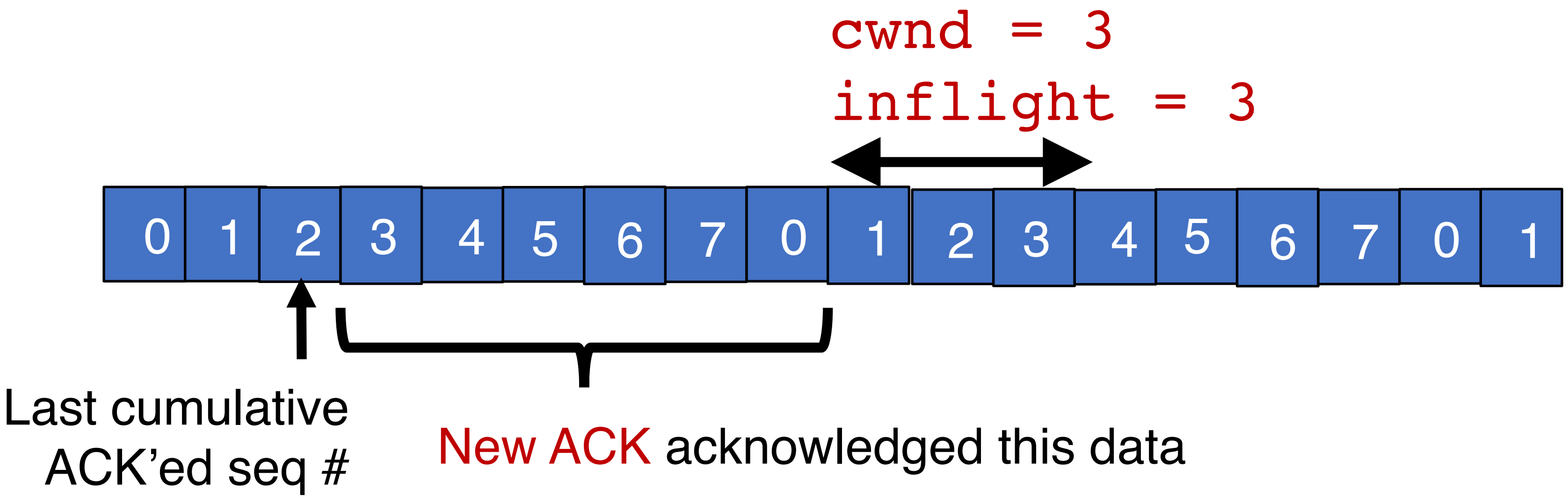
TCP fast recovery (RFC 2581)

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



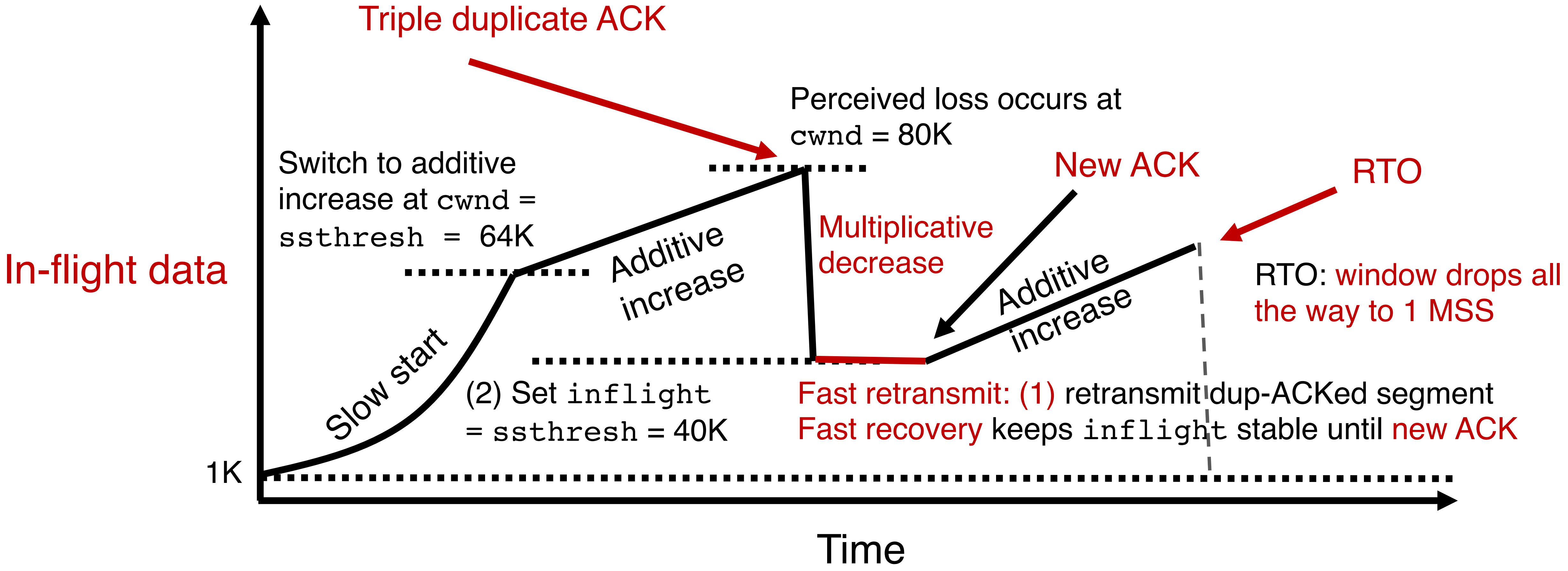
TCP fast recovery (RFC 2581)

- 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 13.3

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

Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

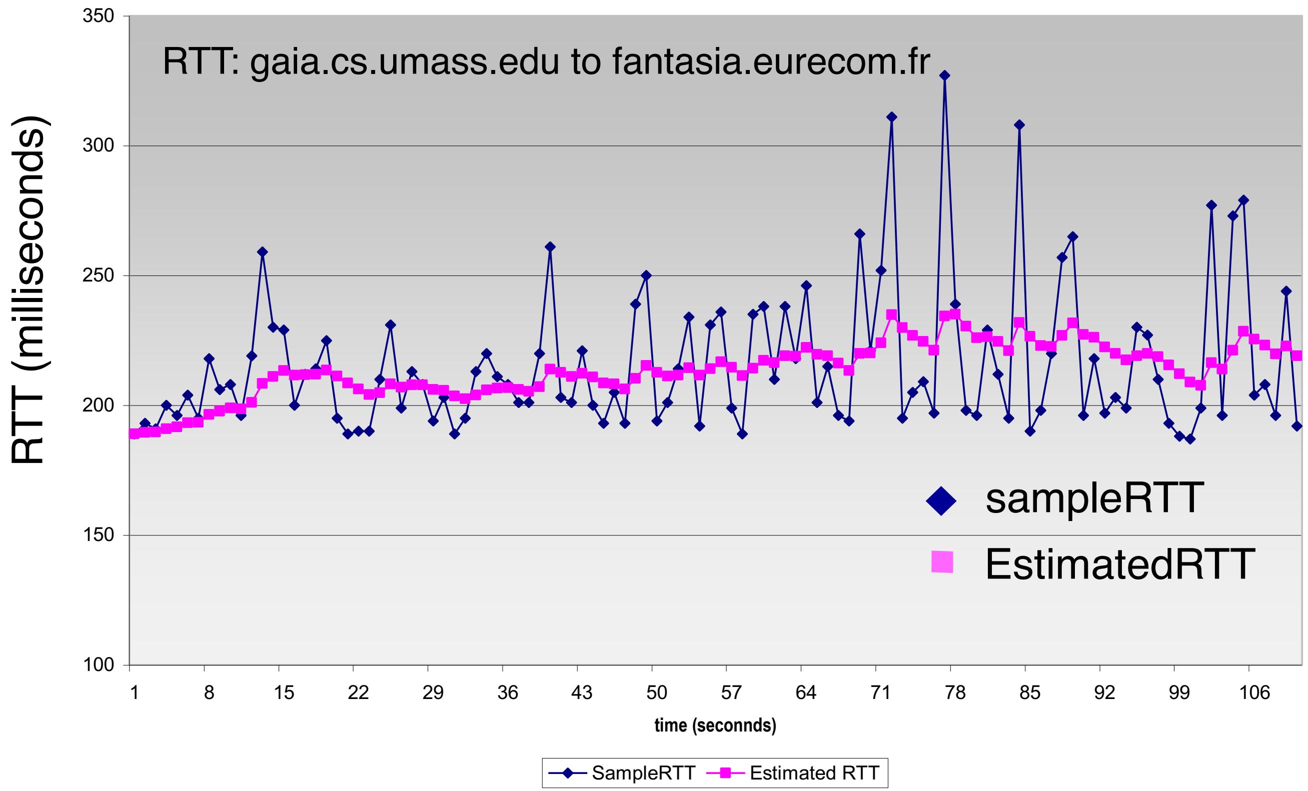
TCP timeout (RTO)

- 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
- Want: RTO must predict the upper bound of RTTs resulting from a successful packet + ACK
- Intuition: somehow use the observed RTT (`sampleRTT`)
  - Can we just directly set the latest RTT as the RTO?
- No. RTT can vary significantly!
  - Intermittent congestion, path changes, signal quality changes on wireless channel, etc.

Estimate an "average" RTT

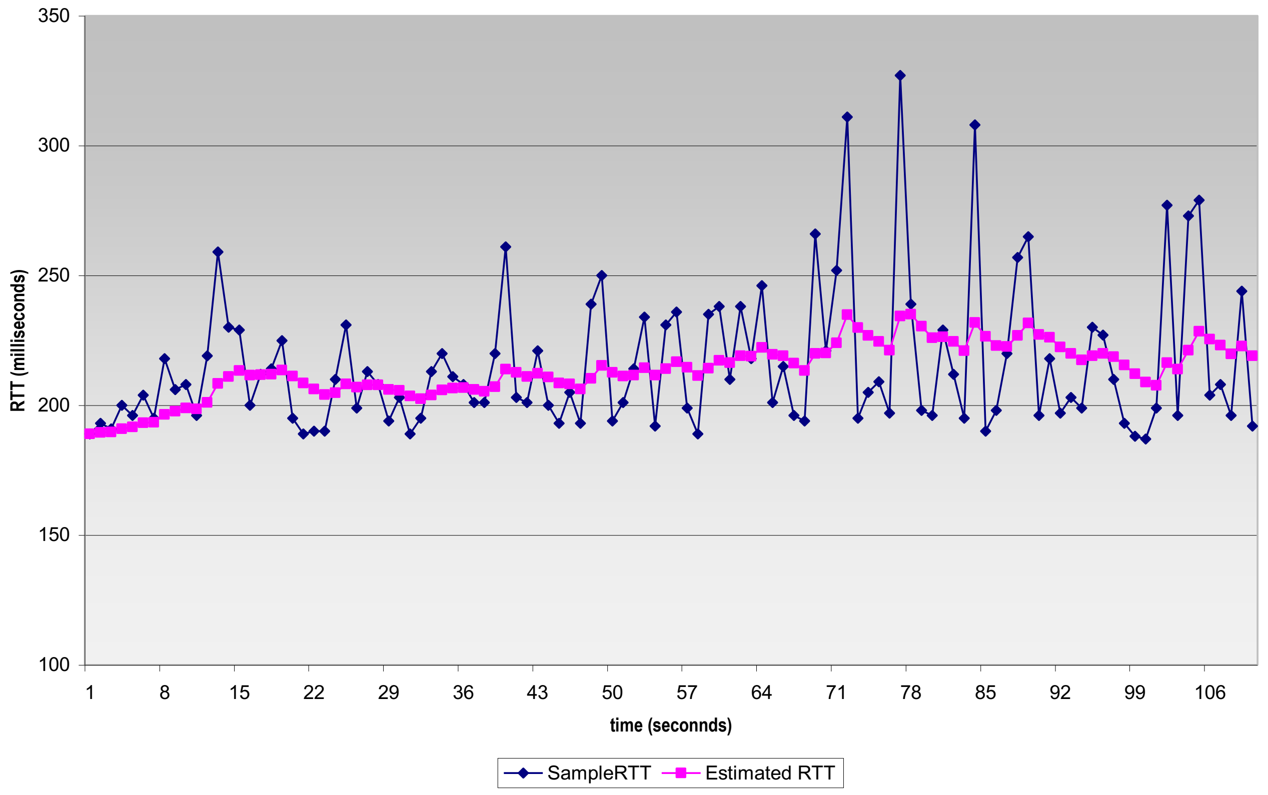
- Exponential weighted moving average (typical alpha = 1/8)

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



Accounting for RTT variance

- RTT samples can have a large variance
- Use a safety margin in the RTO estimate to account for variance



TCP timeout computation

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

```
(typically, β = 0.25)
```

```
RTO = EstimatedRTT + 4*DevRTT
```

average RTT — safety margin

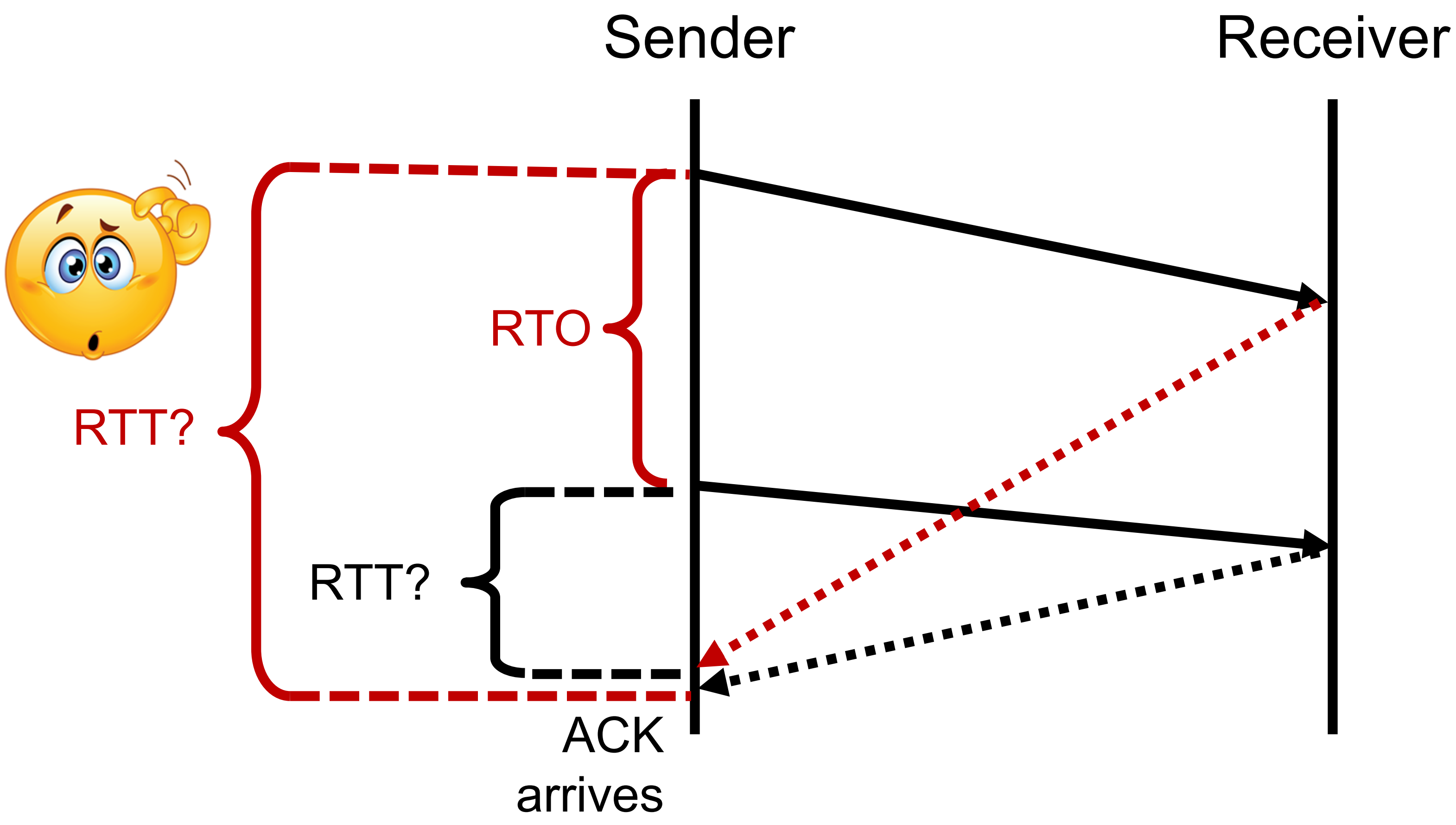
Conceptually, there is an RTO timer for each seq #.

Too many timers?

- Timers are expensive – we don’t want one per sequence #
  - Interrupts, OS data structures, and book-keeping
- The TCP stack maintains just one “real” timer per connection
- When a packet is transmitted, its transmission time is recorded
- The only real timer in the system is the RTO for the first unACK’ed segment
  - Expiration interval: RTO
- If ACK before RTO fires: set timer for next unACK’ed segment, based on recorded transmission time of that segment
- If RTO fires: retransmit the segment, restart RTO timer

Retransmission ambiguity

Real RTT of a retransmitted segment?



Retransmission ambiguity

Aside: problem would go away if packets had a flag to indicate retransmission, or a field to uniquely identify each transmission and its ACK (TCP has neither)

How to estimate RTT/RTO despite retxmit?

- One solution: Never update RTT measurements based on ACKs 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 high-RTT secondary path

- If RTT estimates are not updated, the RTO estimate isn't, and that leads to a host of other problems.
  - Ex: Unnecessary retransmissions since RTOs needlessly expire

Karn's algorithm

- Use back-off as part of the `sampleRTT` computation
- Whenever packet loss (RTO), RTO is increased by a factor
  - Conservatively assume that RTT may have increased since the last unambiguous `RTTsamples` were obtained
- Use this increased RTO as RTO estimate for the next segment
  - Don't use the `estimatedRTT` from stale `sampleRTT`
- Only after an ACK is received for a successful transmission is the RTO timer set to a value obtained from `EstimatedRTT`

Summary

- RTO computation is an important part of TCP's behavior under loss
- TCP uses both an average RTT as well as the variance to obtain a safe prediction of an upper bound of a successful RTT
- Resolve retransmission ambiguity under path changes by avoiding `sampleRTT` measurements and multiplicatively increasing the RTO each time

CS 352
TCP Connection Management

CS 352, Lecture 13.4

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

Srinivas Narayana

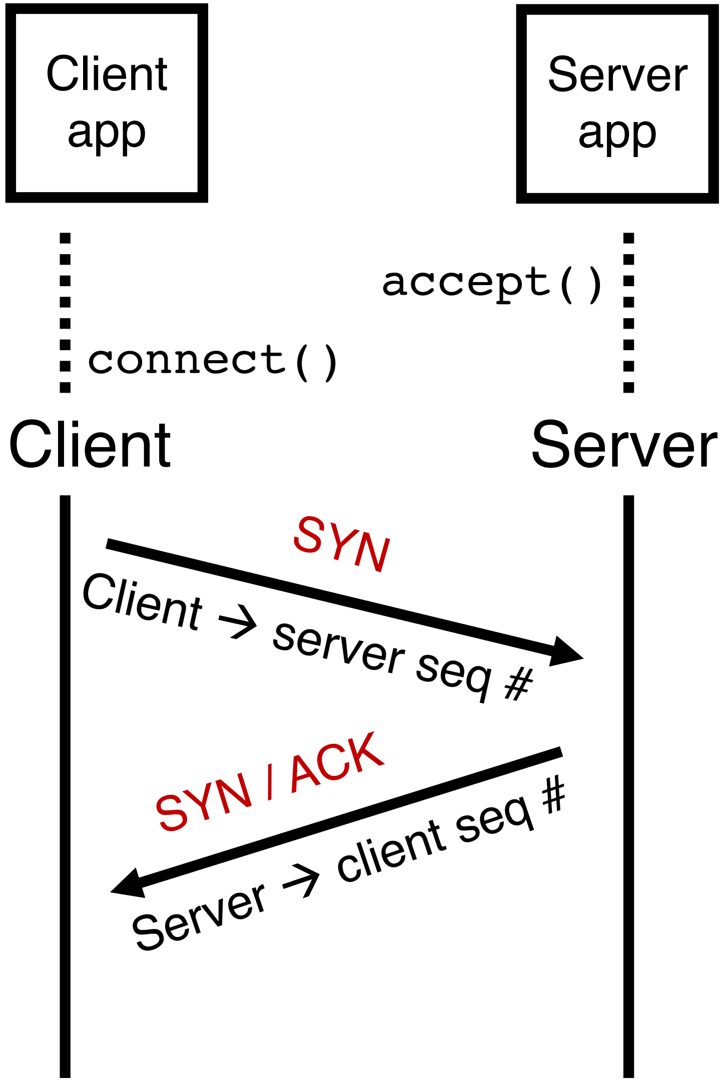
RUTGERS
UNIVERSITY | NEW BRUNSWICK

TCP connections need lots of bookkeeping

- Socket buffer memory
- Entries in connection lookup tables
- Data structures and parameters (e.g., sequence numbers) in the operating system kernel
- These resources can get expensive on machines running many connections, e.g., web servers

Handshake

- Before starting data transmission, TCP client and server perform a handshake and agree on parameters
- TCP is bidirectional: independent set of sequence numbers for each direction
- Sequence numbers start from a random initial value
- Specific TCP flags indicate connection initiation and acceptance



TCP flags in the header

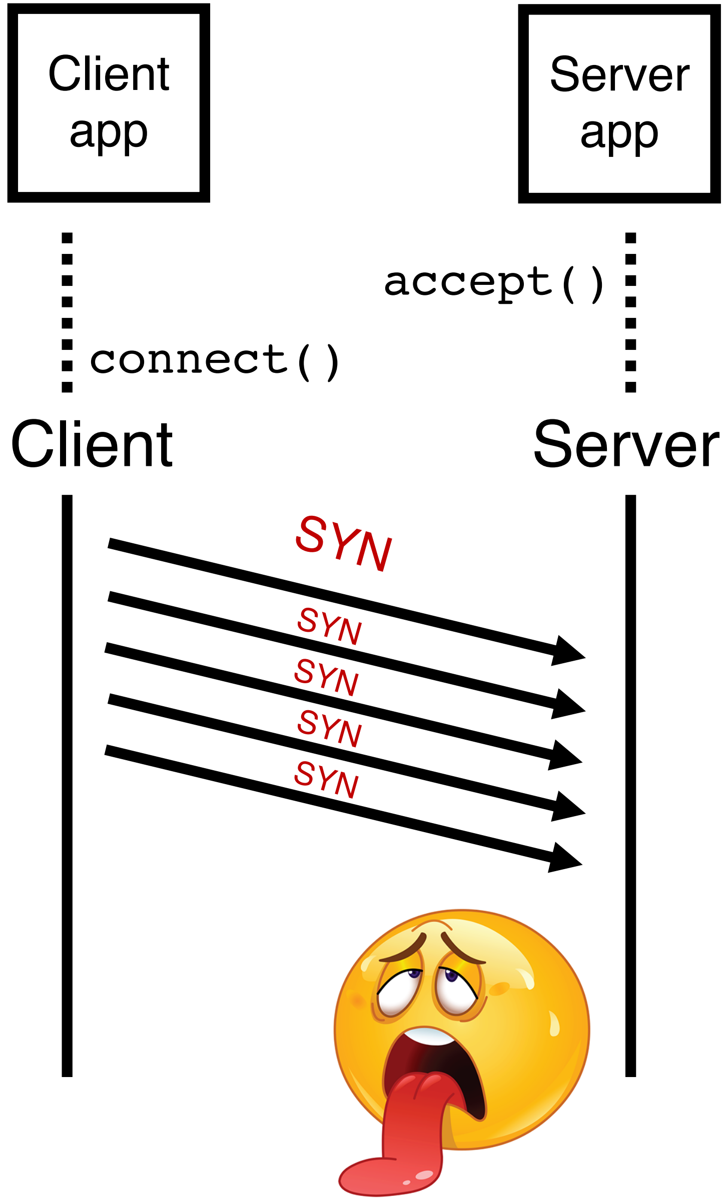
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                        TCP Header Format

      Note that one tick mark represents one bit position.
```

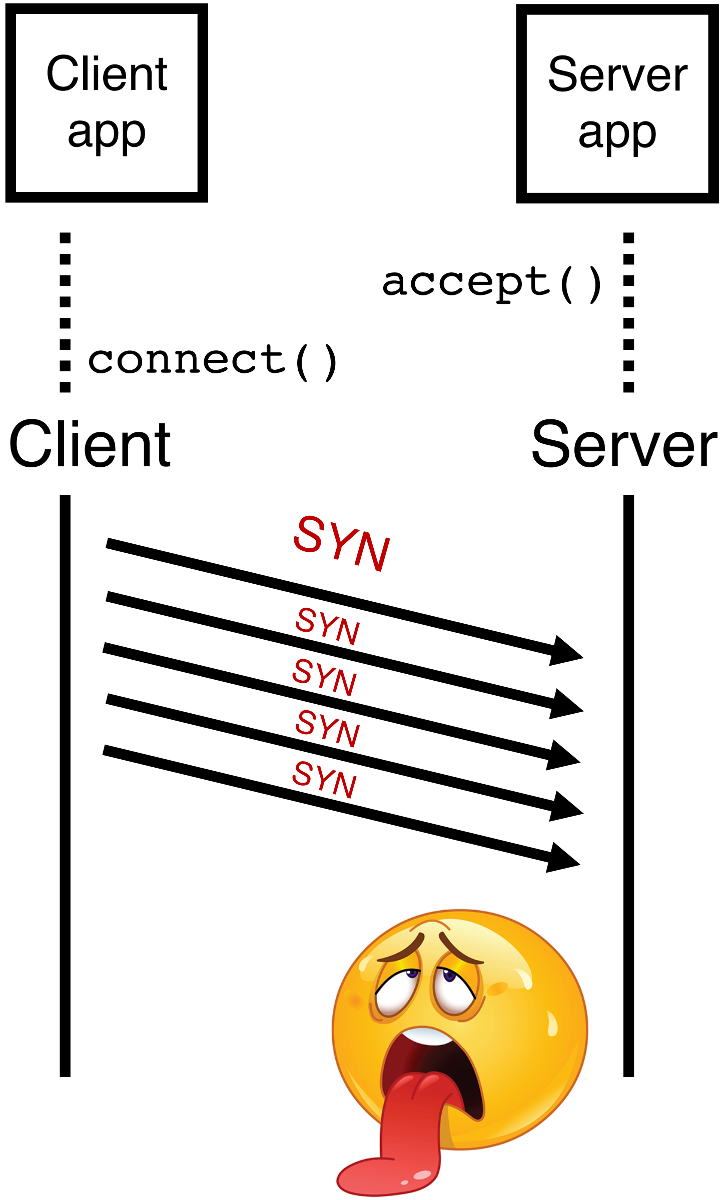
2-way handshake not enough

- Suppose the server receives the first SYN packet and decides to allocate all the resources needed for the connection.
- What happens if a malicious client sends a ton of SYN packets?
- Asymmetric work: client doesn't need to allocate any resources of its own
  - Just have to send a well-crafted packet
- However, server's resources exhausted!
- SYN flood attack: a form of denial of service



Consequences

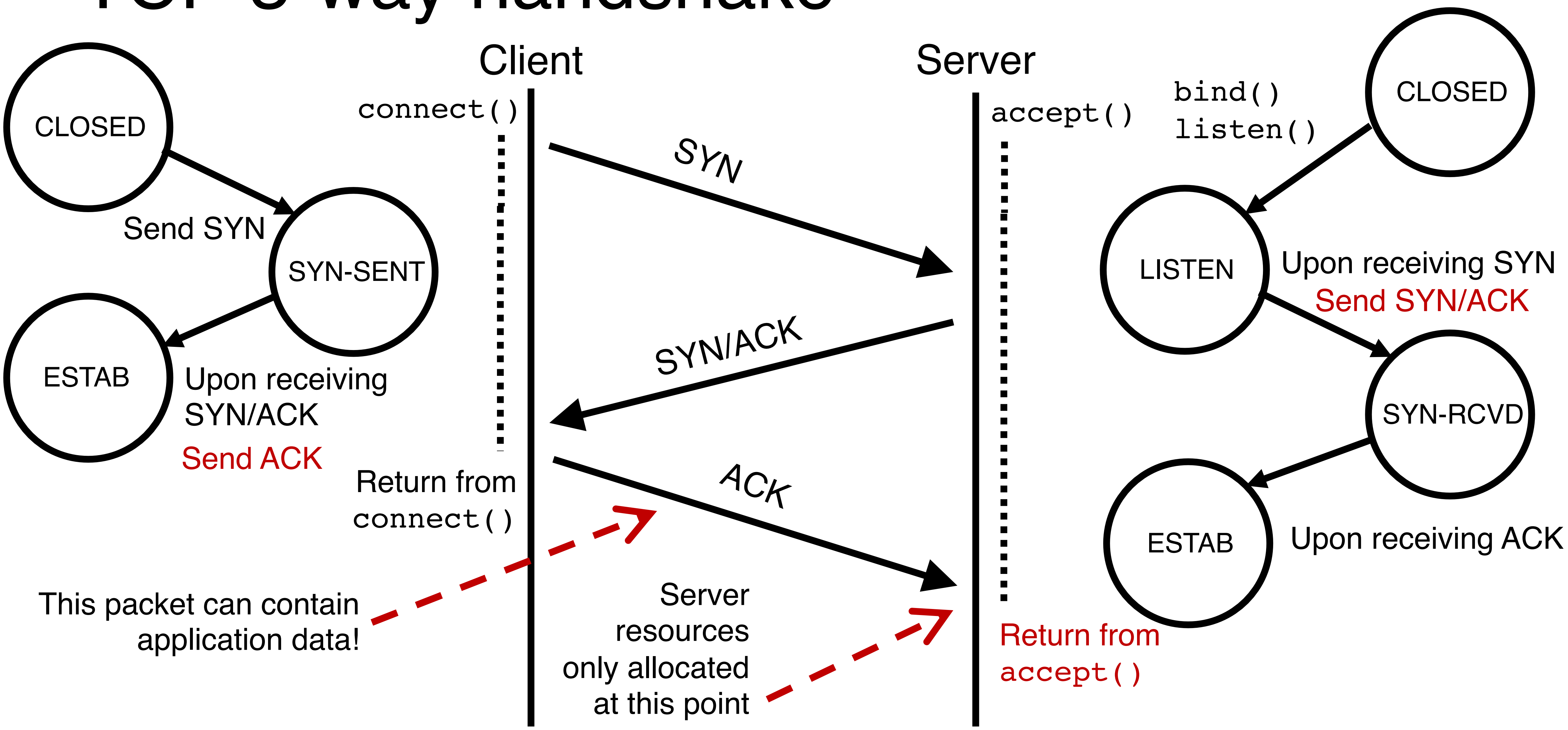
- The server should not allocate resources upon receiving the first client message (SYN)
- The server cannot carry any application data in SYN/ACK
  - Server hasn't yet allocated all necessary resources
- Client cannot send any data in the SYN packet
- Recall: HTTP requires an RTT for the handshake before sending HTTP request



Mitigating the denial of service problem

- Key idea: Make the client do more work before allocating server resources
- The client should send at least one more packet, responding to the data in the server's SYN/ACK, before the server decides to call the connection established
  - That is, before all required server resources like buffers are allocated
- Result: 3-way handshake
- Per-connection finite state machine tracks this process

TCP 3-way handshake



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
- However, 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

Summary of TCP connection management

- TCP connections have associated resources: managing them requires book-keeping the establishment of a connection carefully
- Simple 2-way handshakes suffer from denial of service vulnerability
  - Moral: don’t allocate resources on the first client message
- 3-way handshake mitigates this issue by making client work harder
  - Client must send ACK to server’s SYN/ACK before server can handle data
  - The cost: increased time before sending application data from client