CS-340 Introduction to Computer Networking

Lecture 6: TCP

Steve Tarzia

Many diagrams & slides are adapted from those by J.F Kurose and K.W. Ross
Many TCP flow diagrams from Stevens' "TCP/IP Illustrated Vol. 1" 1st ed.

Last Lecture

- Apps can send individual packets w/ *UDP*; delivery is not guaranteed.
  - Adds a *port number* and *checksum* to packets.

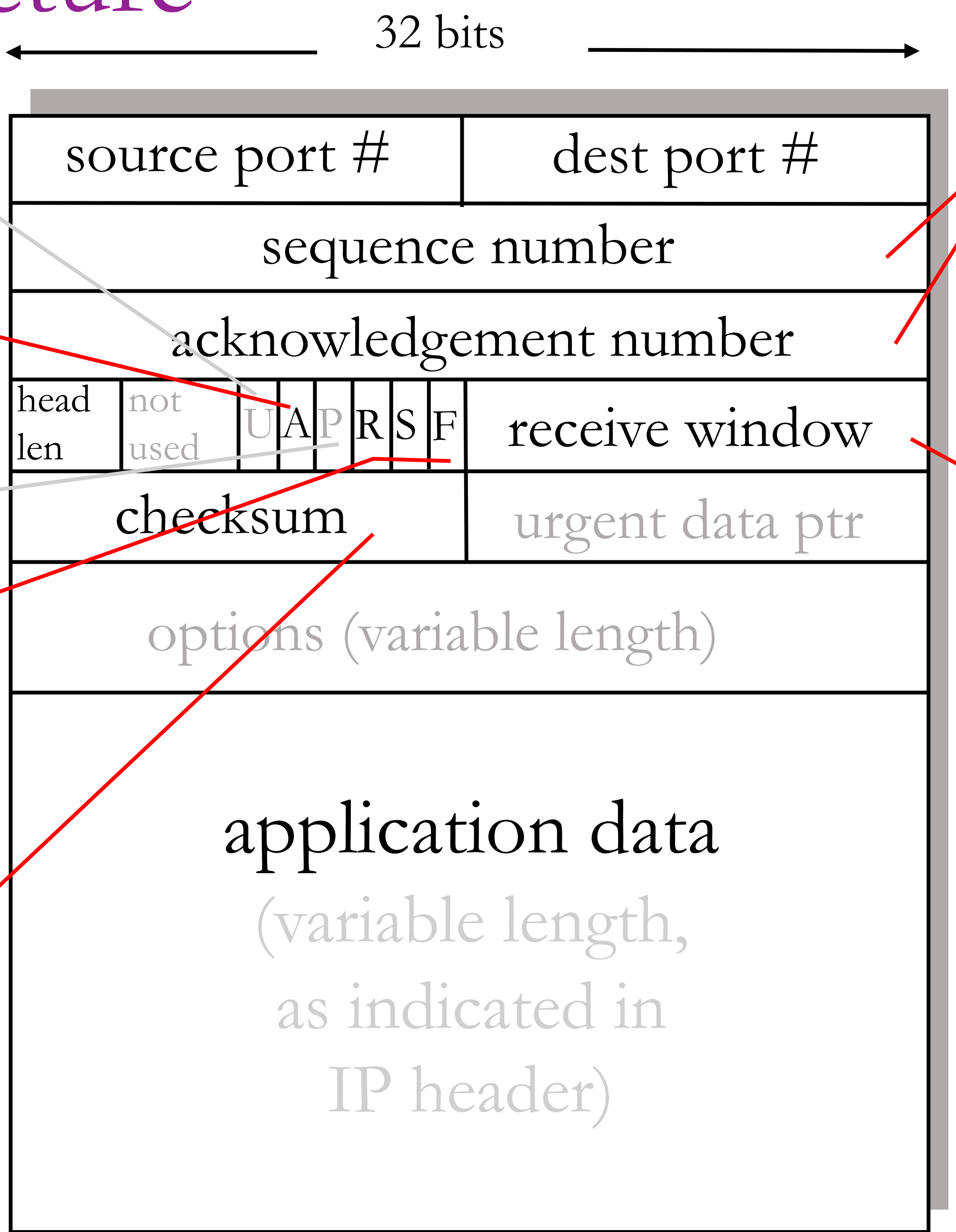
But most apps want reliable, stream-oriented transport *(eg., TCP):*

- Delivery confirmation & ordering is possible by sending *ACKs*
  - After a *timeout*, resend packet that was not ACK'ed.
- *Pipelining* packets allow much better use of link capacity.
  - *Parallelizes* ACK'ed communication
  - *Window size* determines the number of allowed in-flight packets
- *Go Back N* is a simple pipelining protocol that uses *cumulative ACKs*.
- *Selective Repeat* adds buffering to the receiver to avoid unnecessary repetition.

TCP is ***practical*** reliable transport

- Has evolved from 1970s through today.
- Uses positive ACKS. Combines ideas from *go-back-N* and *selective repeat*.
- Also manages connection **pacing** (flow & congestion control)
- Unlike UDP, TCP requires that two hosts setup a ***connection*** before exchanging data. Why?
  - Exchange ***initial sequence numbers*** for both directions of the connection.
- Choose a *random* initial sequence number for two reasons:
  - So new packets are not confused with retransmission from prior connection.
  - So an attacker cannot easily inject fake packets in the data stream.

TCP packet structure



URG: urgent data (generally not used)

ACK: ACK # is valid

PSH: push data now (generally not used)

RST, SYN, FIN: connection establishment (setup, teardown commands)

checksum (as in UDP)

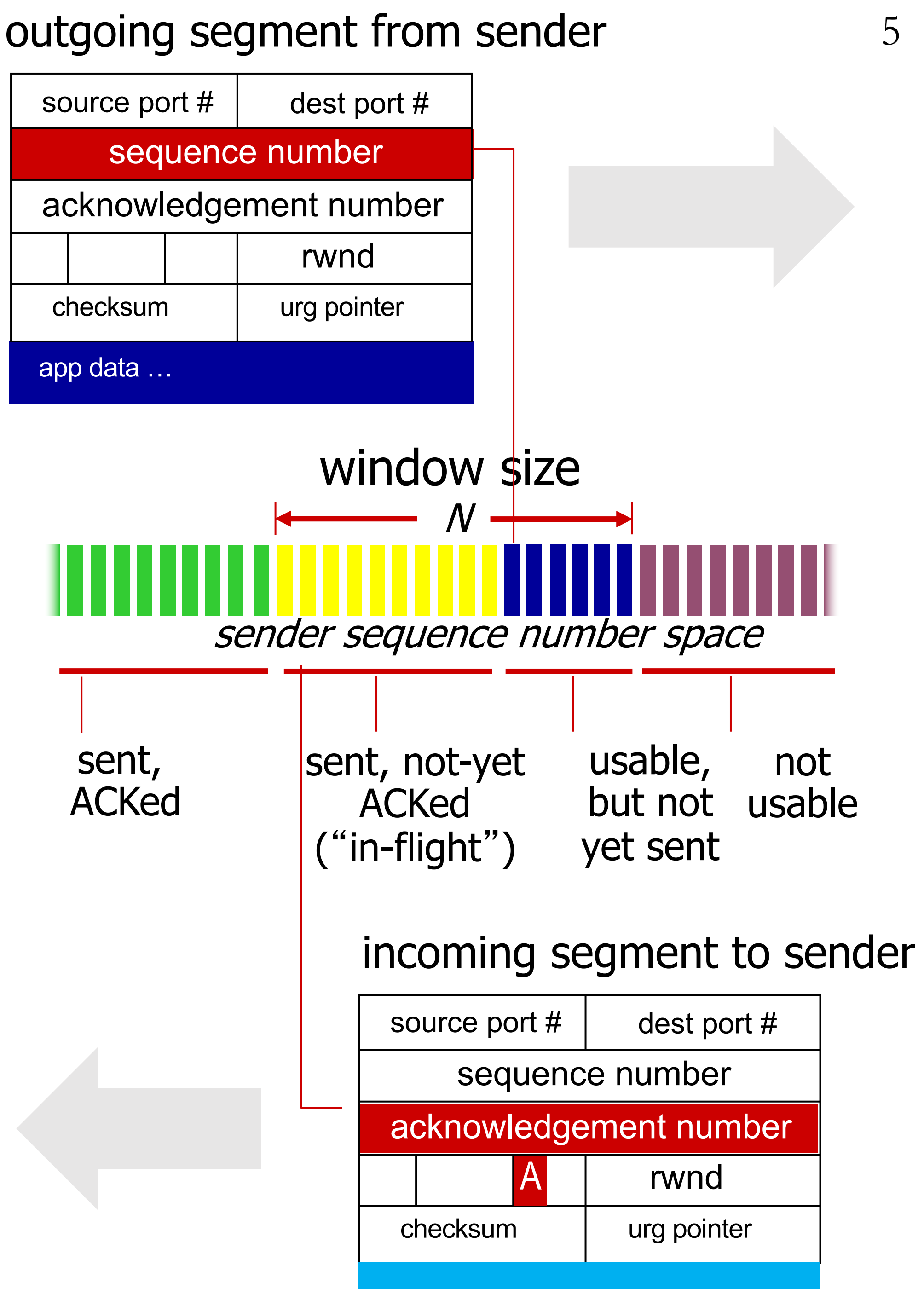
counting by **bytes** of data (not segments!)

bytes receiver is willing to accept

Any TCP packet can carry an ACK number, so ACKs can be "piggy backed" on data flowing in the opposite direction.

TCP seq #s and ACKs

- Sequence Numbers:
  - Indicate the offset in the byte stream of the segment's first byte
- **Cumulative** ACKs:
  - Send next expected sequence number (like Go Back N)
- Receiver may drop out-of-order segments (like GBN), or buffer them for later reassembly (like Selective Repeat).

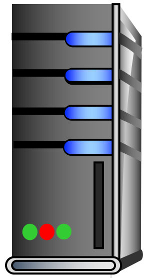


Simple TCP example *(after the handshake)*

Seq = index of data being sent.
ACK = index of data it expects to receive next

Web Browser
Seq_{init}=42

Web Server
Seq_{init}=79



Visits page, sending 100-byte long HTTP request

Seq=42, ACK=79, data = ‘`GET /index.htm HTTP/1.1\r\nHost:…`’

Server ACKs request, and sends back 1200 bytes of HTML

Seq=79, ACK=142, data = ‘`HTTP/1.1 200 OK…`’

Client ACKs HTML body. Does not have any other data to send.

Seq=142, ACK=1279, data = ‘’

Server has already ACK’ed 142+0, so don’t send an ACK.

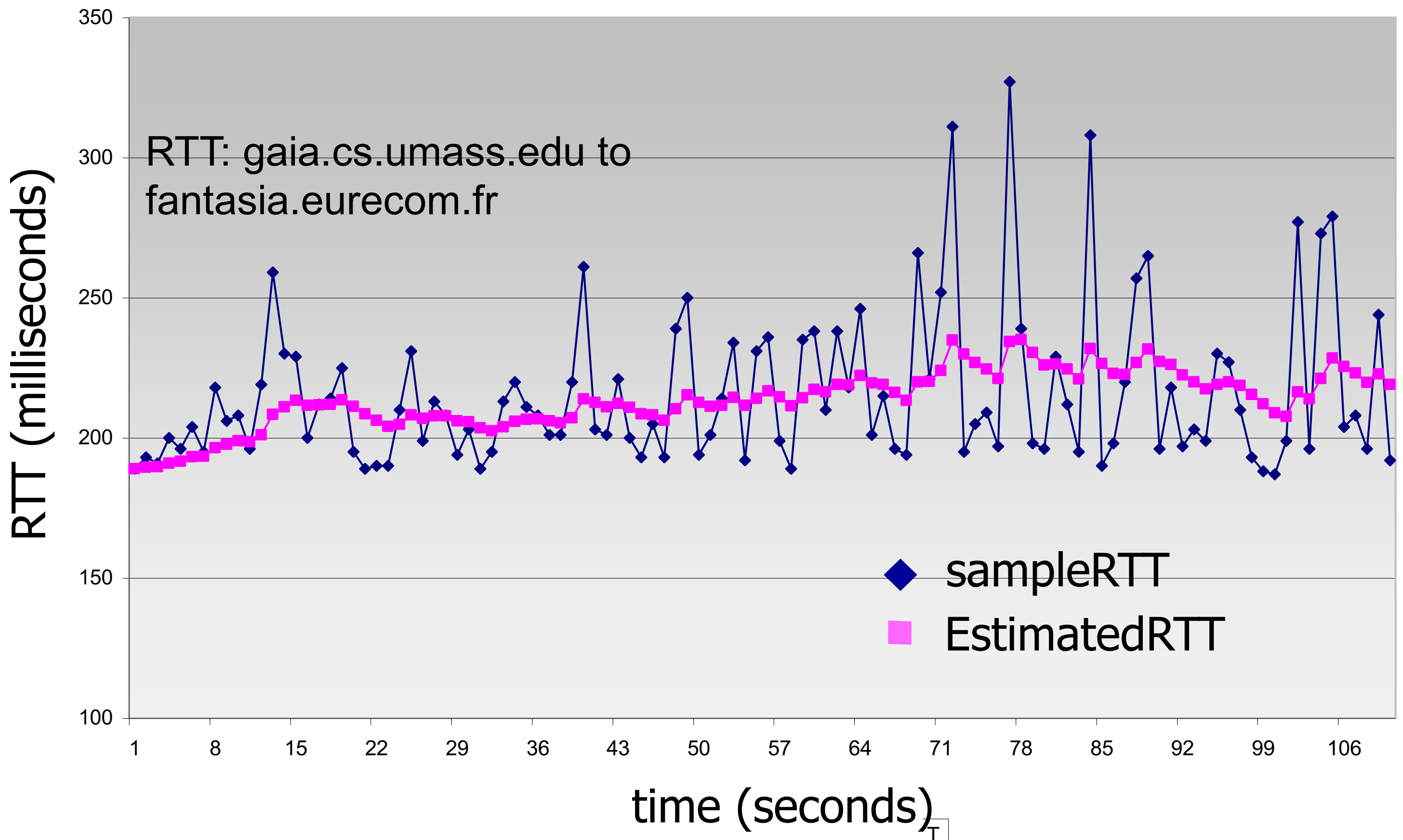
Timeouts are an important parameter

- TCP keeps **one timer**, for oldest un-ACK'ed segment
  - Retransmit that *one segment* when timer expires. Why just one?
  - ACK received → start timer for next-lowest un-ACK'ed segment, if any.
- Timer must be set carefully:
  - Too long → waste time waiting before a necessary retransmit.
  - Too short → send duplicate packets unnecessarily.
- What is the ideal value of the timer?
  - In other words, how much time do we expect to elapse before getting ACK?
  - **Answer**: just slightly longer than expected round-trip time (RTT).
- Thus, TCP keeps track of recent RTTs by constantly measuring delay between every transmission and its ACK.

Exponentially-weighted moving average RTT

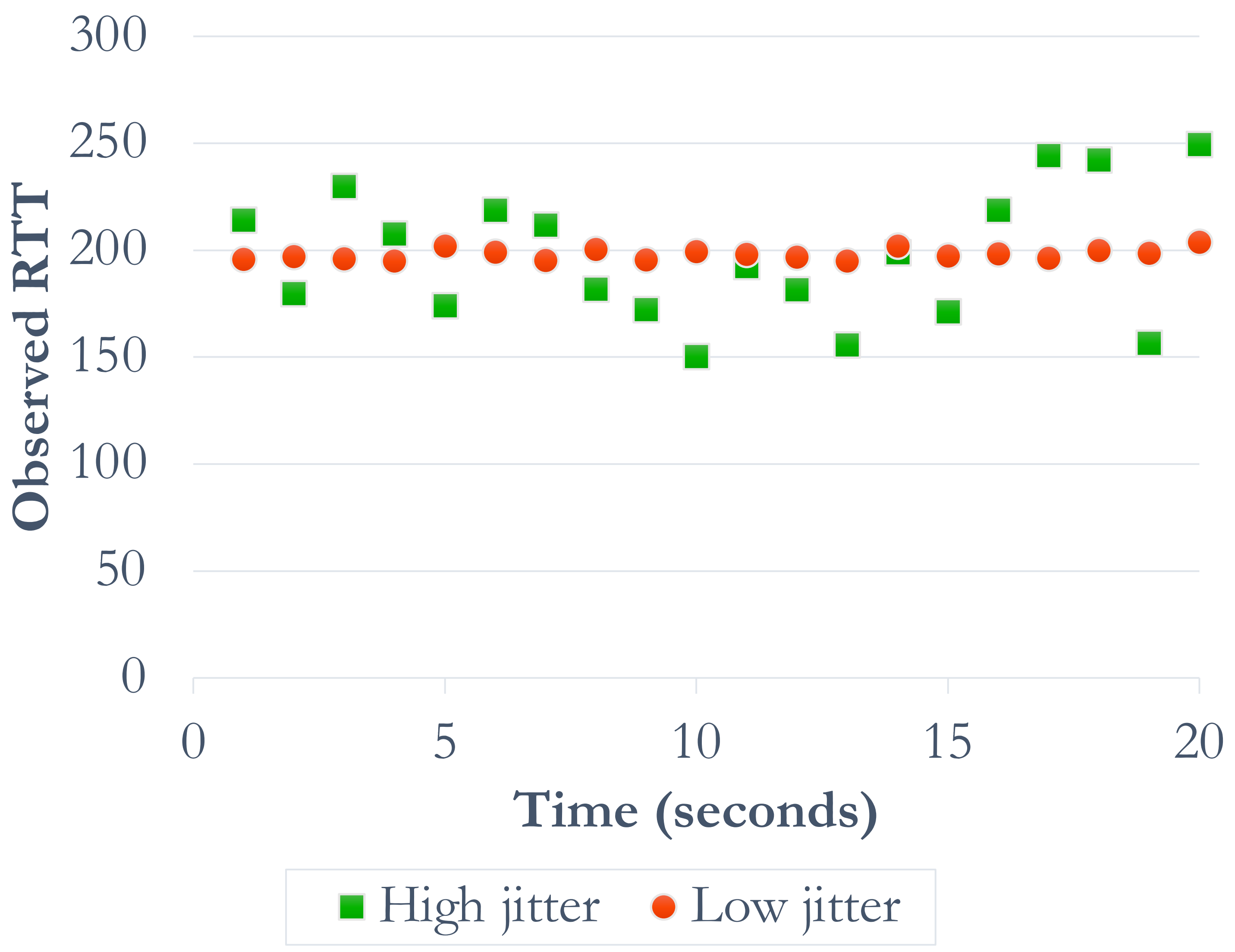
`EstimatedRTT = (1-α)*EstimatedRTT + α*SampleRTT`

- Every time a new SampleRTT is observed, update the EWMA RTT.
- Typically, α = 0.125
- Gives us a "smoothed" average of recent RTT.
- Then set timeout > EstimatedRTT
- But how much greater?



RTT variance *(jitter)* also affects timeout choice

- Square points show traffic with *high* variance in RTT (high jitter)
  - Should choose timer significantly > *EstimatedRTT*
- Circle points show traffic with *low* variance in RTT (low jitter)
  - Can choose timer just slightly > *EstimatedRTT*



Final RTT estimation

- Also track an exponentially-weighted moving average of RTT deviation (jitter):

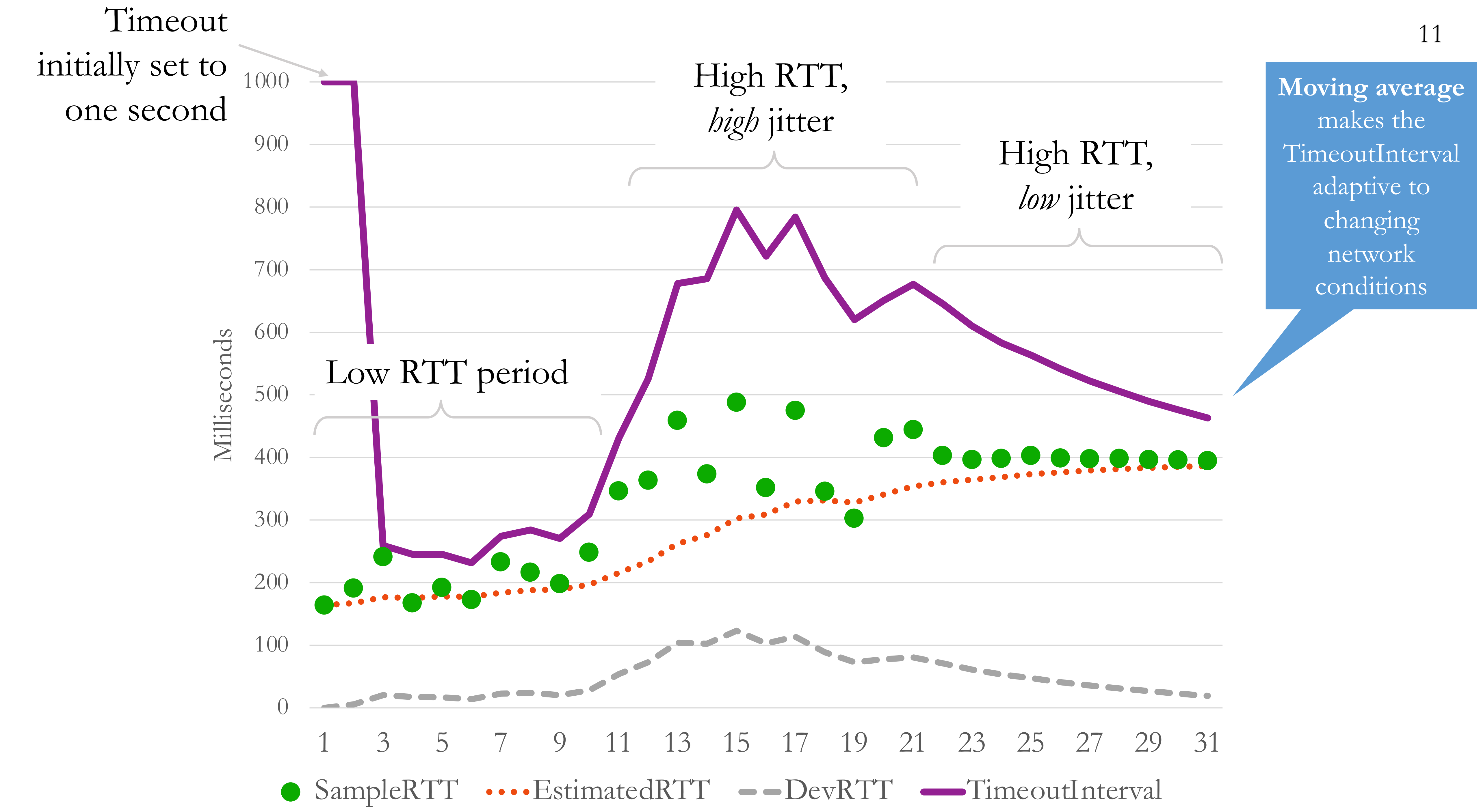
```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

Typically **β**=0.25

- Add a multiple of DevRTT as a "safety margin" above EstimatedRTT:

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

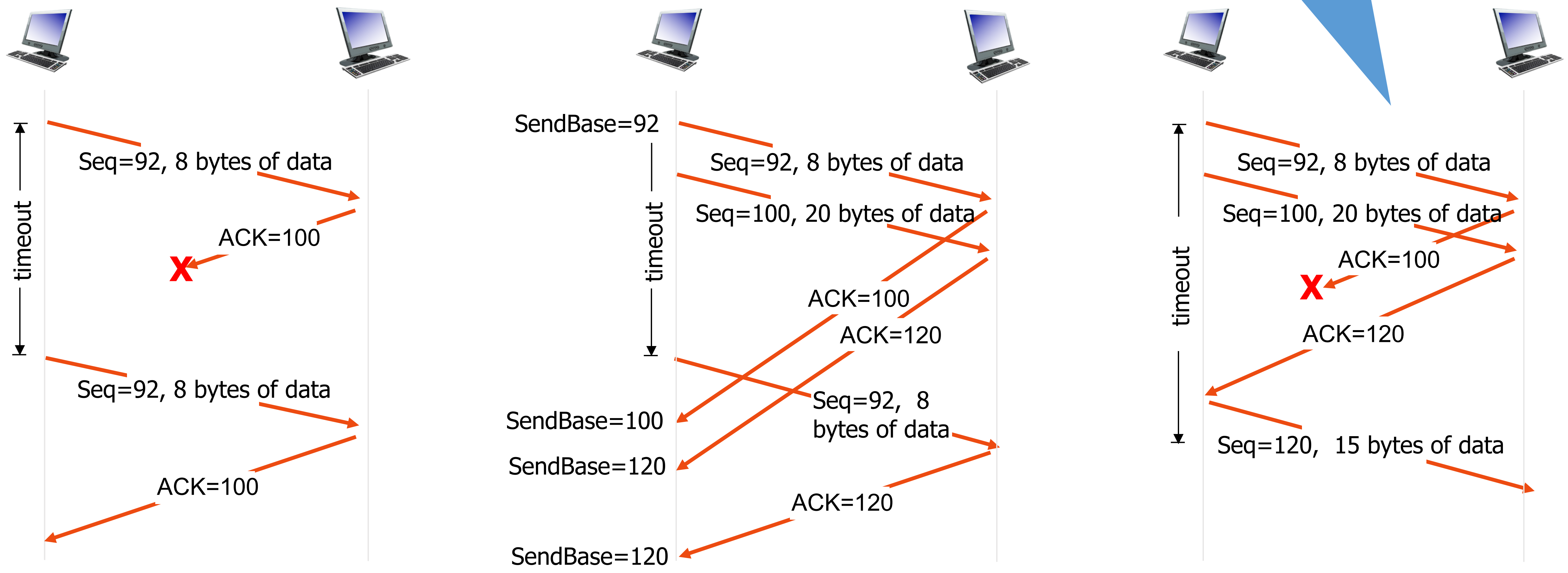
- Initially set Timeout to one second, until we have some measurements.


Timeout initially set to one second
High RTT, *high* jitter
High RTT, *low* jitter
Low RTT period
Moving average makes the TimeoutInterval adaptive to changing network conditions
Milliseconds
1000
900
800
700
600
500
400
300
200
100
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
SampleRTT
EstimatedRTT
DevRTT
TimeoutInterval

Intermission

TCP retransmission scenarios

What if the second ACK is dropped instead of the first?



Lost ACK

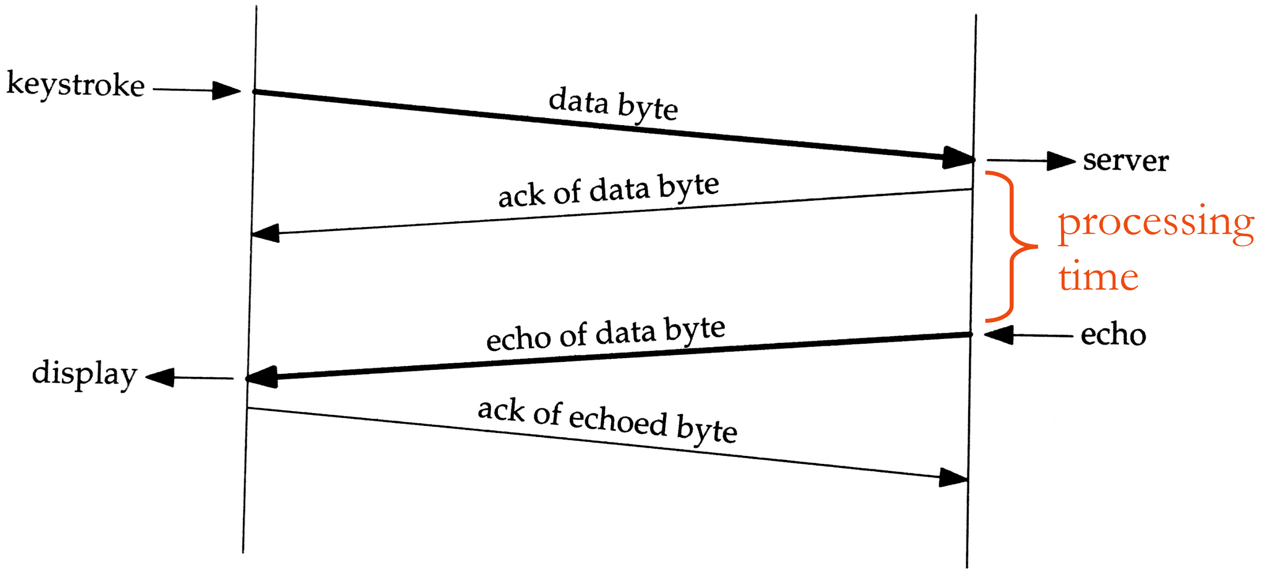
Premature timeout

Lost ACK, but cumulative ACK prevents retransmission

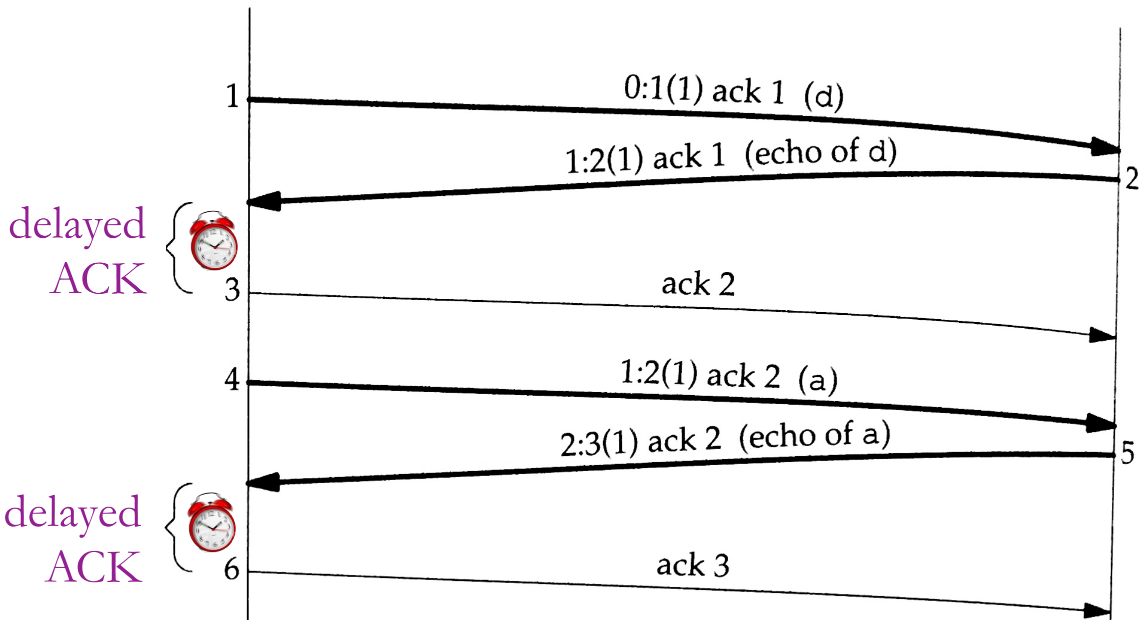
Delayed ACKs

- TCP recommends that receiver wait before sending an ACK (RFC 1122).
- This allows the TCP's ACK response (and receive window update) to be piggy-backed on an *application-layer response*.
- Send ACK only after 500ms or with next data in other direction.
- Eg., an "echo" app that repeats back the data received:

Eager ACKs



With delayed ACKs

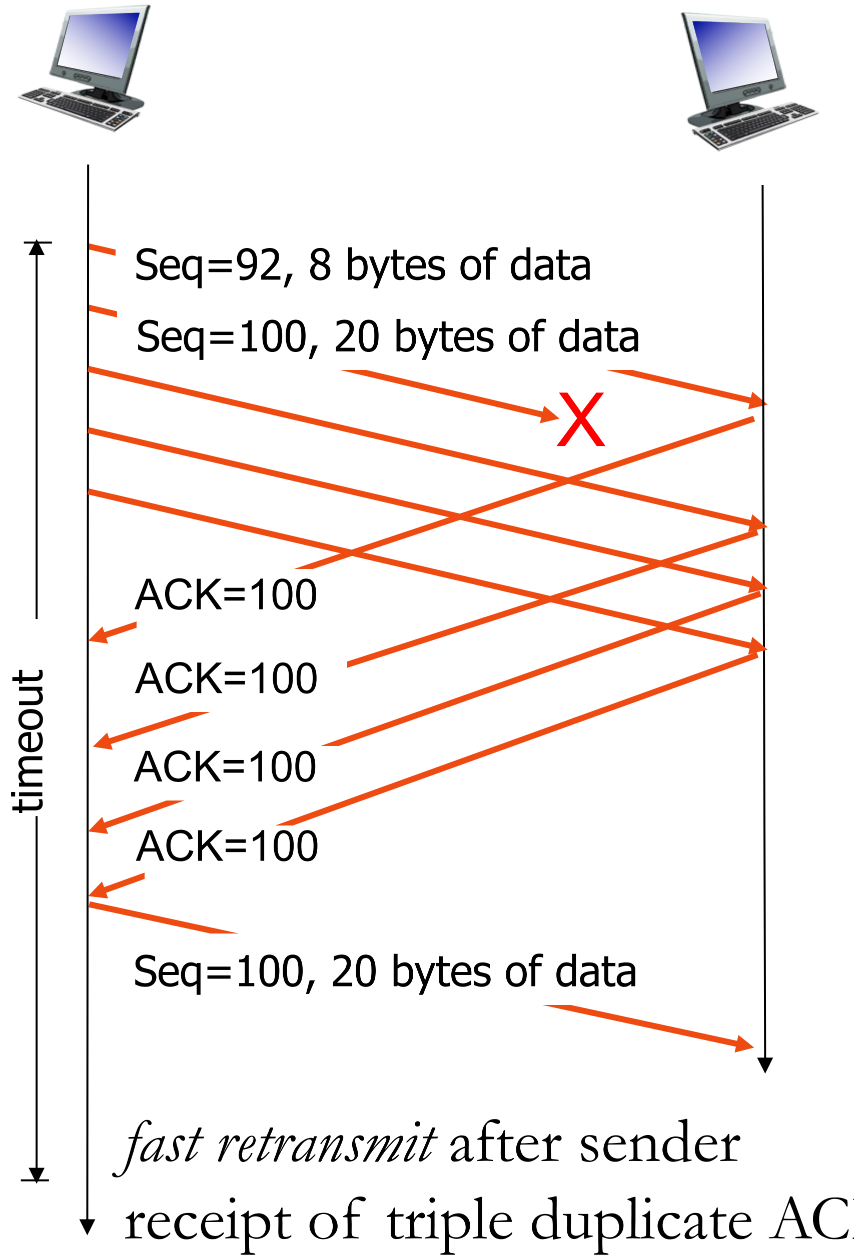


TCP ACK generation (RFC 1122, 2581)

Event at Receiver	TCP action taken
• Arrival of in-order segment with expected seq #. All data up to expected seq # already ACK'ed.	***Delayed ACK***. Wait up to 500ms for next segment. If no next segment, send ACK.
• Arrival of in-order segment with expected seq #. One other segment has ACK pending.	Immediately send a single ***cumulative ACK***, ACK'ing both in-order segments.
• Arrival of *out-of-order* segment (with higher-than-expect seq #). In other words, a gap was detected.	Immediately send ***duplicate ACK,*** indicating seq. # of next expected byte.
• Arrival of segment that partially or completely fills gap.	Immediately send ACK if segment starts at beginning of gap.

TCP *fast retransmit*

- With using cumulative ACKs, duplicate ACKs suggest packet loss.
    - Receiver will always set ACK # to the index of the next byte expected (the gap).
- On *triple duplicate ACK,* instead of the sender waiting for timer to expire, TCP *fast retransmit* immediately re-sends lowest un-ACK'ed segment.



fast retransmit after sender receipt of triple duplicate ACK

Triple DUP ACK

- Why does TCP wait for **three** duplicate ACKS before performing a fast retransmit? Why not after one?

- RFC 2001:

  "Since TCP does not know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost."

TCP has characteristics of both GBN and SR:

Go Back N

- Only *one timer* is kept, but →
- Send *cumulative ACKs*, but →
- *Duplicate ACK* for early segment.

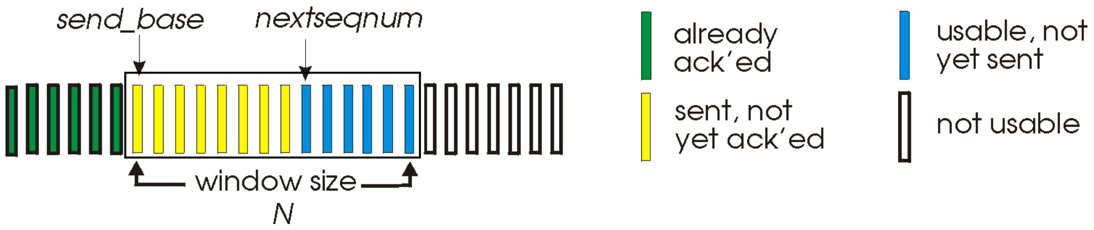
Selective Repeat

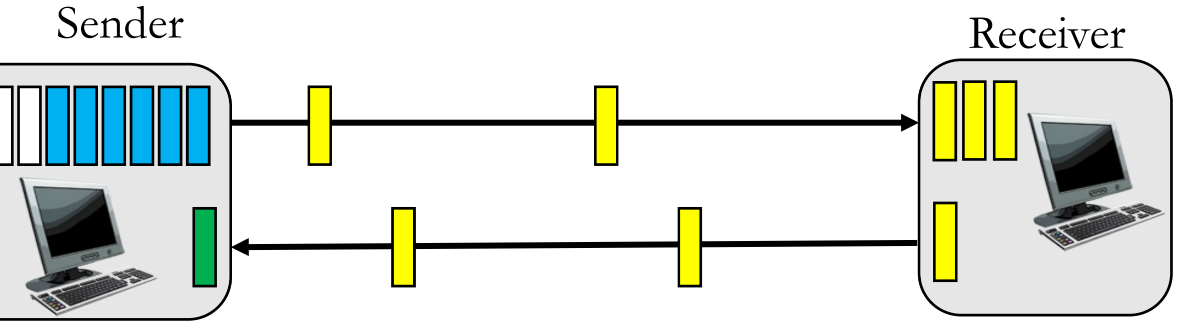
- *Re-send just one segment* on timeout.
- Receiver may *save out-of-order* segments for later reassembly.

Plus some new features:

- Guidelines for setting timeout interval, based on observations
- Delayed ACKs.
- Triple duplicate ACK triggers a retransmit.
- Connection setup with 3-way handshake, and teardown.
- Window size changes to implement flow & congestion control

TCP window → flow and congestion control

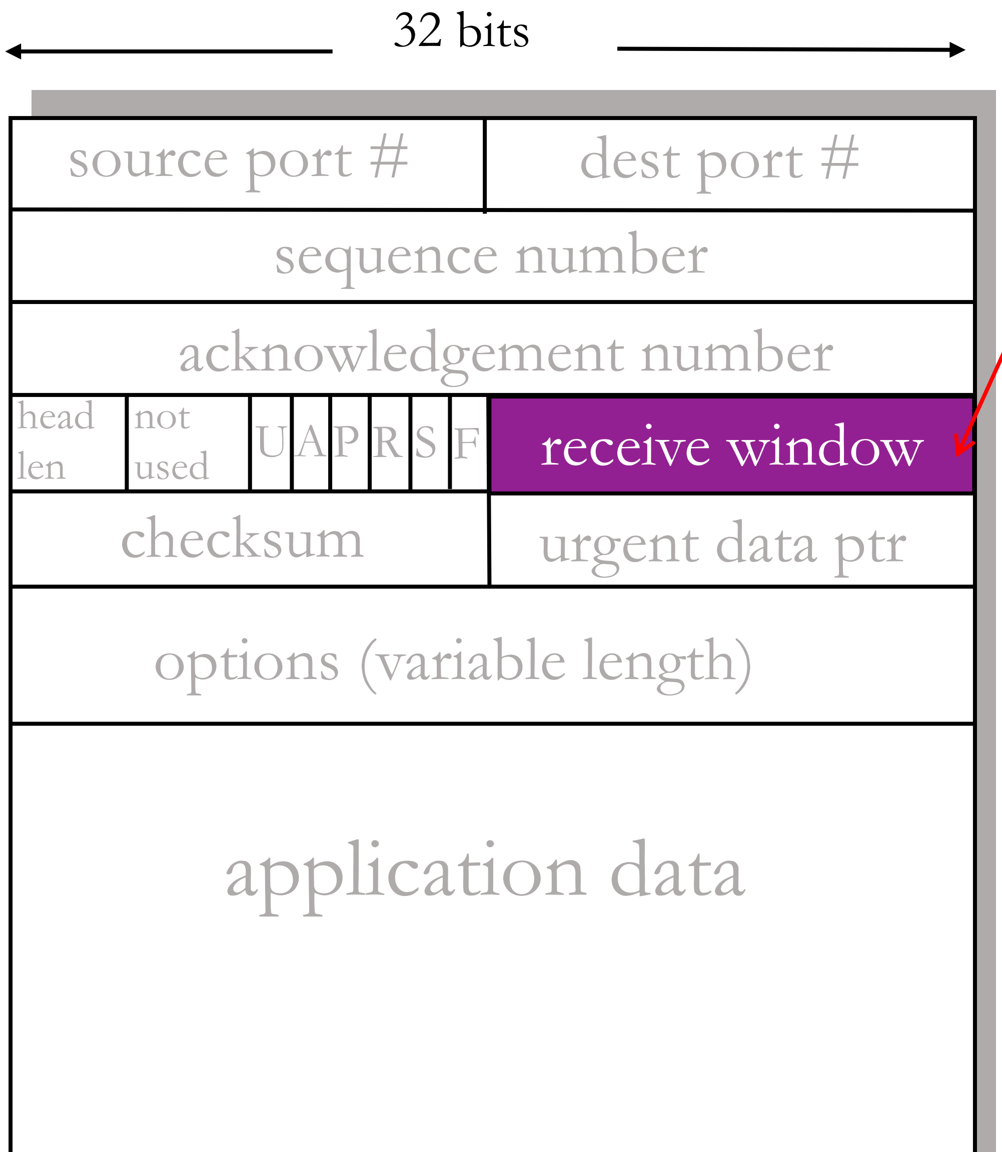




segments ≤ N

- Recall that window size limits the maximum # of in-flight segments.
- Peak throughput is proportional to window size (divided by RTT).
  - Hosts control windows, not RTT.
- Control sender's window size to *prevent packet loss*, by preventing:
  - Overflow of receiver's receive buffer (*flow control*).
  - Overflow of routers' packet queues (*congestion control*).

TCP *flow control* – *to avoid overwhelming the receiver*



- In **receive window**, host tells how many bytes of new data it can receive.
- Sender simply tracks # un-ACK'ed bytes and keeps this ≤ receive window.
  - A simple and effective solution is possible because we can directly observe the receive buffer and report its status.
- ***Congestion control*** requires a more complex solution because it involves many routers along the path, and many flows (connections) across each router.
  - We must *infer* network congestion.

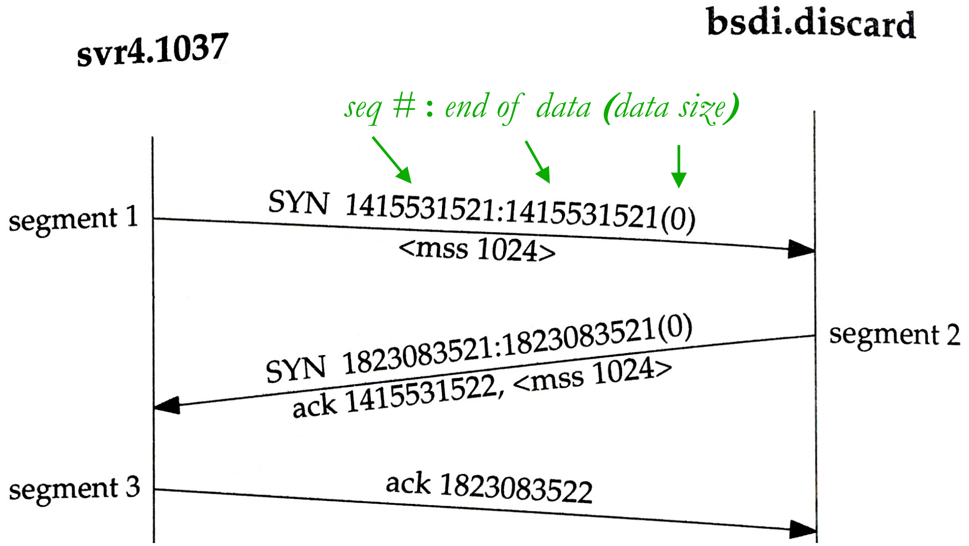
TCP connection setup

- Before starting data exchange, hosts must agree on a few parameters:
  - **Initial sequence numbers** (in both direction)
  - **Receive window size** (for flow control)
- Recall: choose a random initial sequence number for two reasons:
  - So new packets are not confused with retransmission from prior connection.
  - So an attacker cannot easily inject fake packets in the data stream.
- *Three-way handshake* sets up the connection
  1. **SYN:** Initiator sends its parameters (init. seq #, window size, *etc.*).
  2. **SYN-ACK:** Listener sends ACK including its own parameters.
  3. **ACK:** Initiator ACKs (and may include first segment of data).

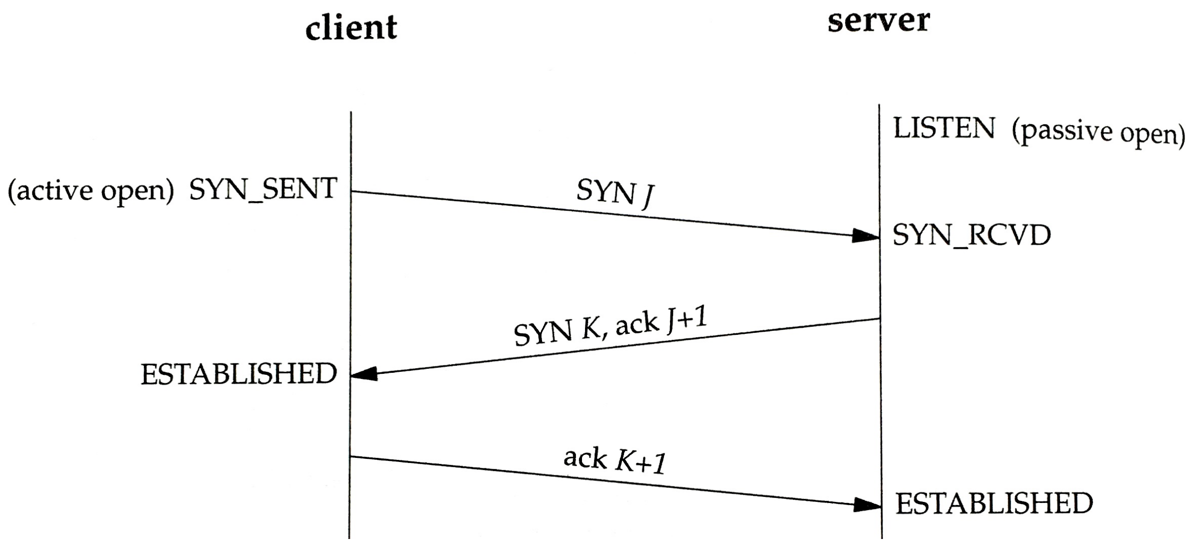
  Above ACKs use initial sequence number + 1

3-way handshake, from "TCP/IP Illustrated" reference book

An example:



In general:



May open a TCP socket:

- **Actively** (we specify the connection partner, and a SYN is sent)
- **Passively** (just **listen** for a SYN from unknown host)

Usually call the active initiator the *client*, and the passive listener the *server*.

TCP connection close

- Each side of the connection sends **FIN** to say it's finished sending.
  - Waits for an ACK.
  - Connection may be *half closed* if only one side is done sending.

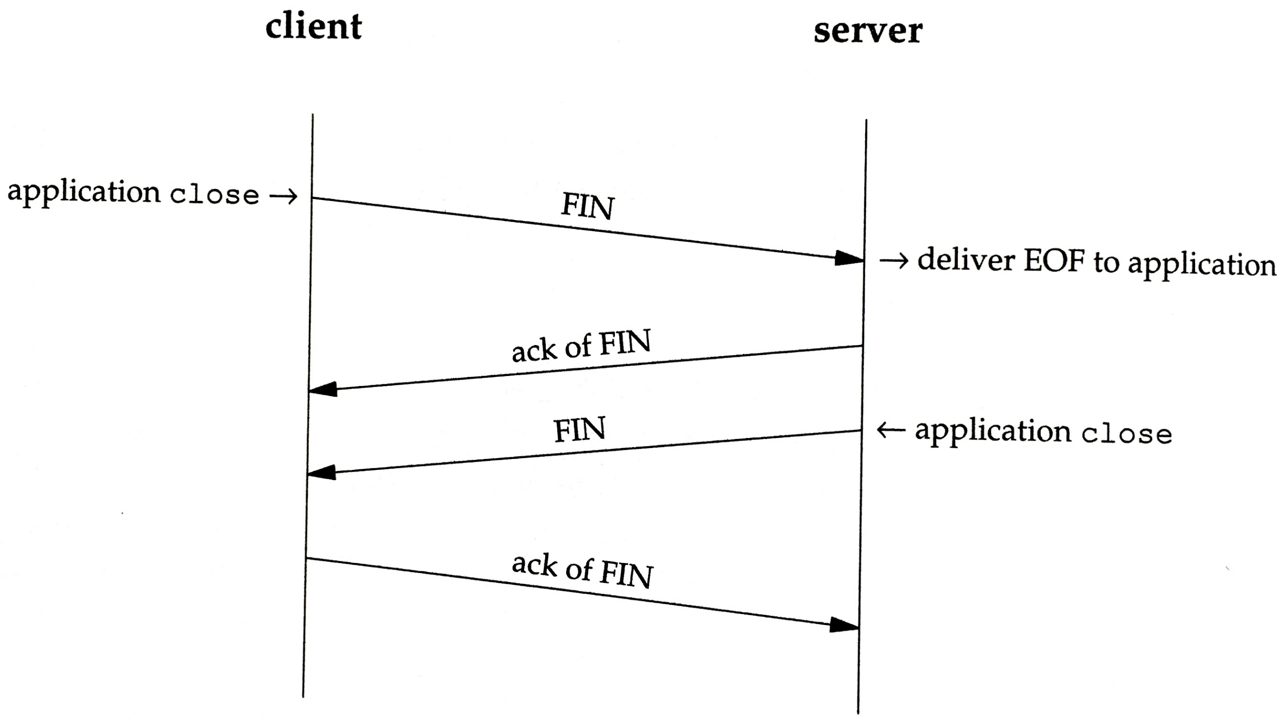


Figure 18.4 Normal exchange of segments during connection termination.

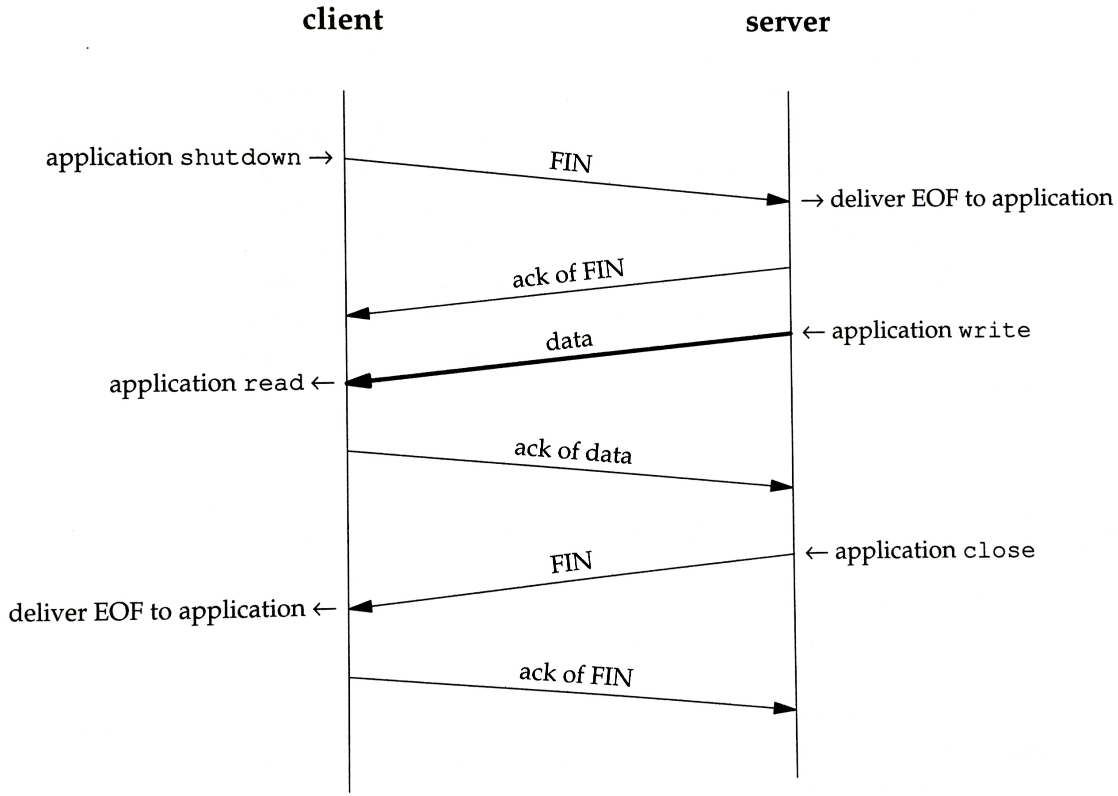


Figure 18.10 Example of TCP's half-close.

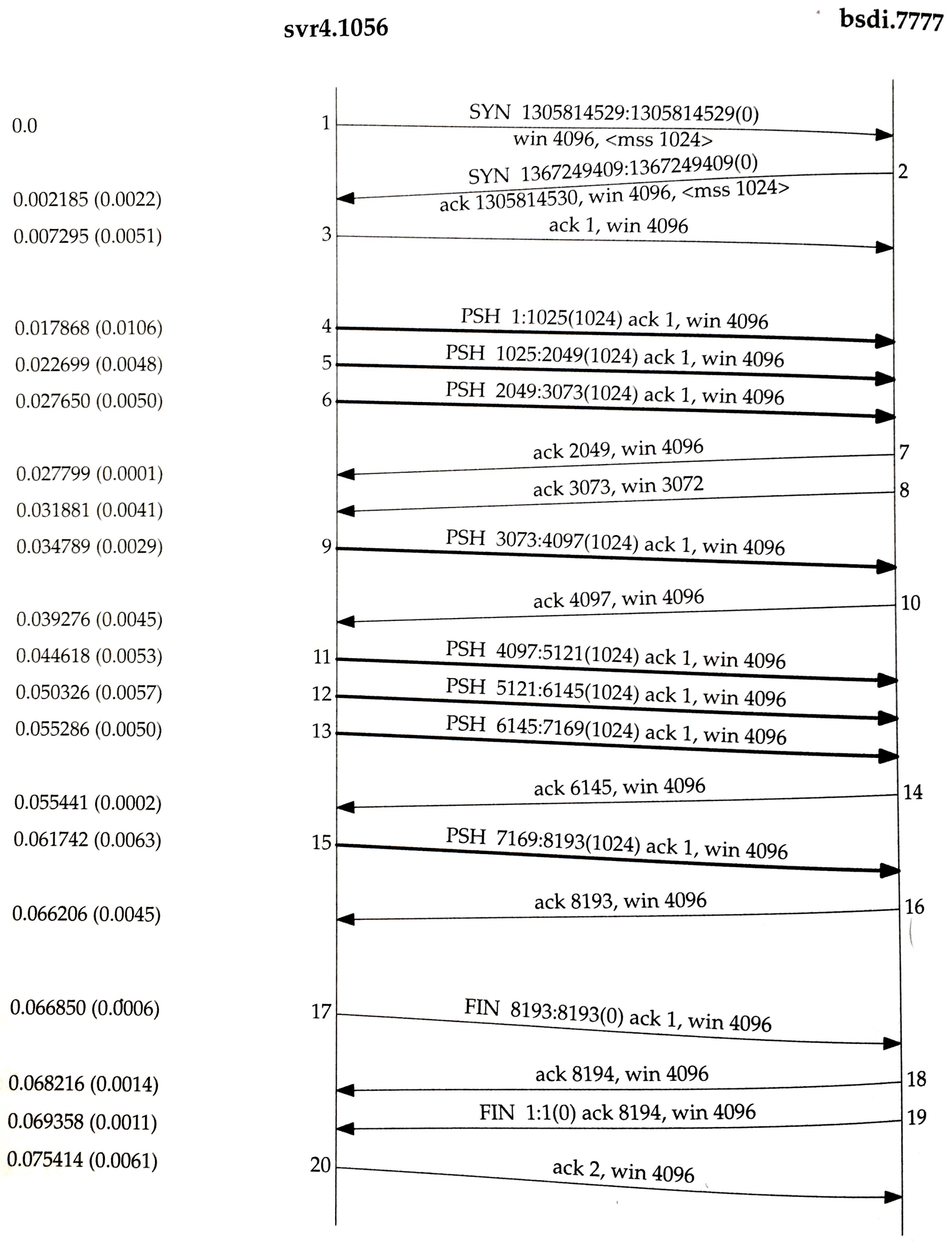


Figure 20.1 Transfer of 8192 bytes from `svr4` to `bsdi`.

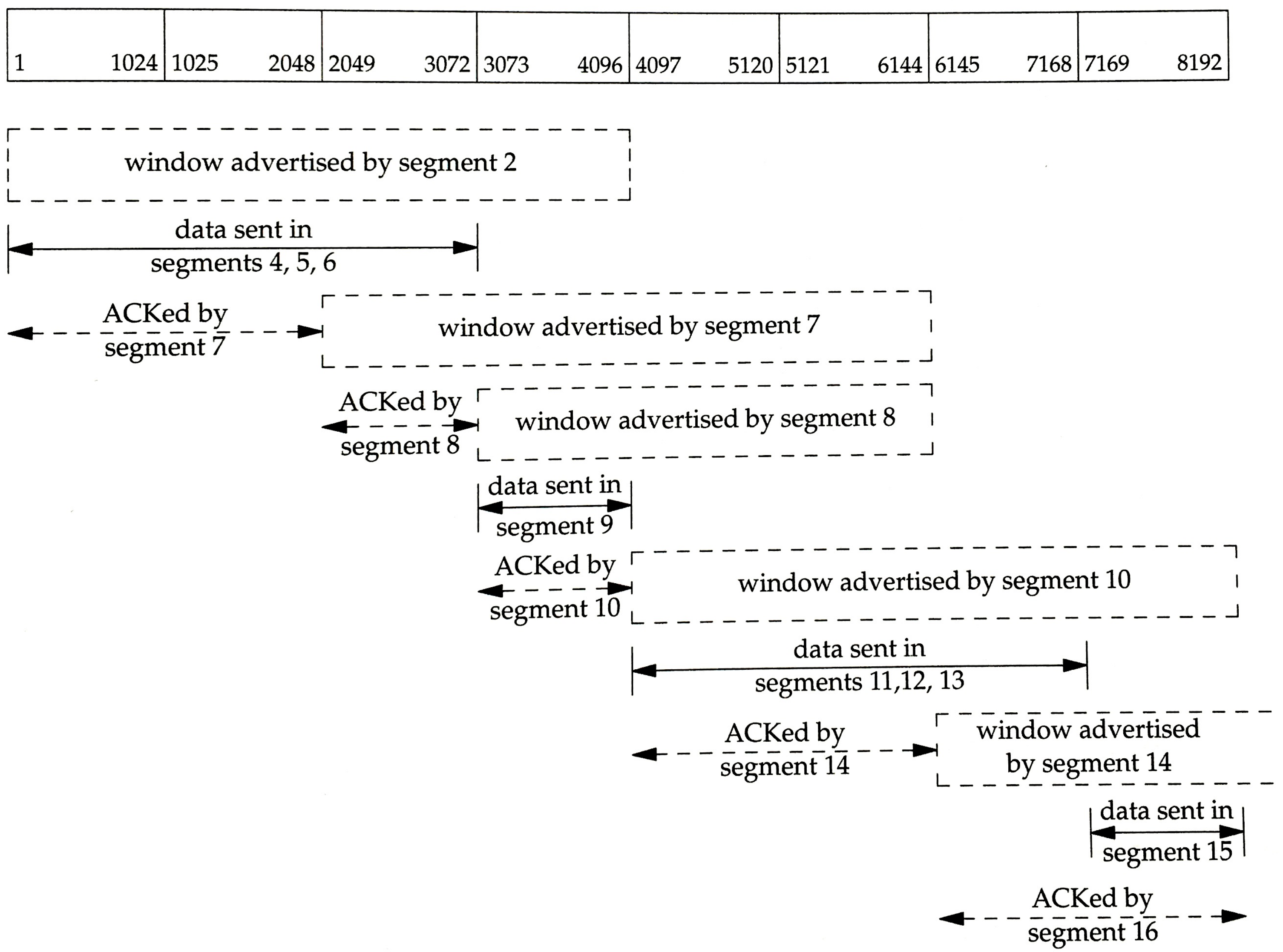


Figure 20.6 Sliding window protocol for Figure 20.1.

Protocol must also handle unusual timings

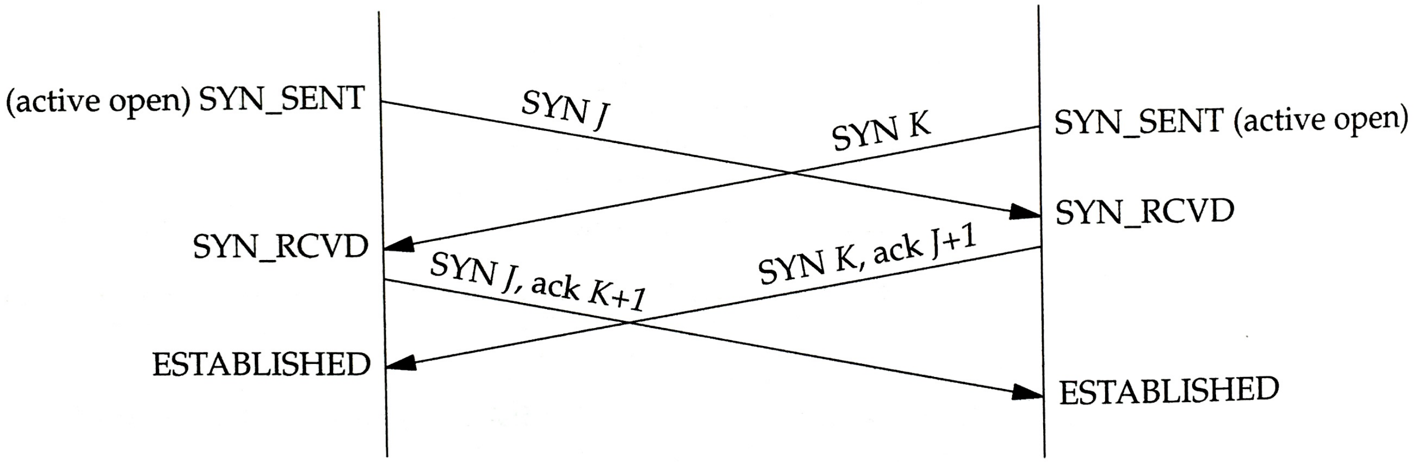


Figure 18.17 Segments exchanged during simultaneous open.

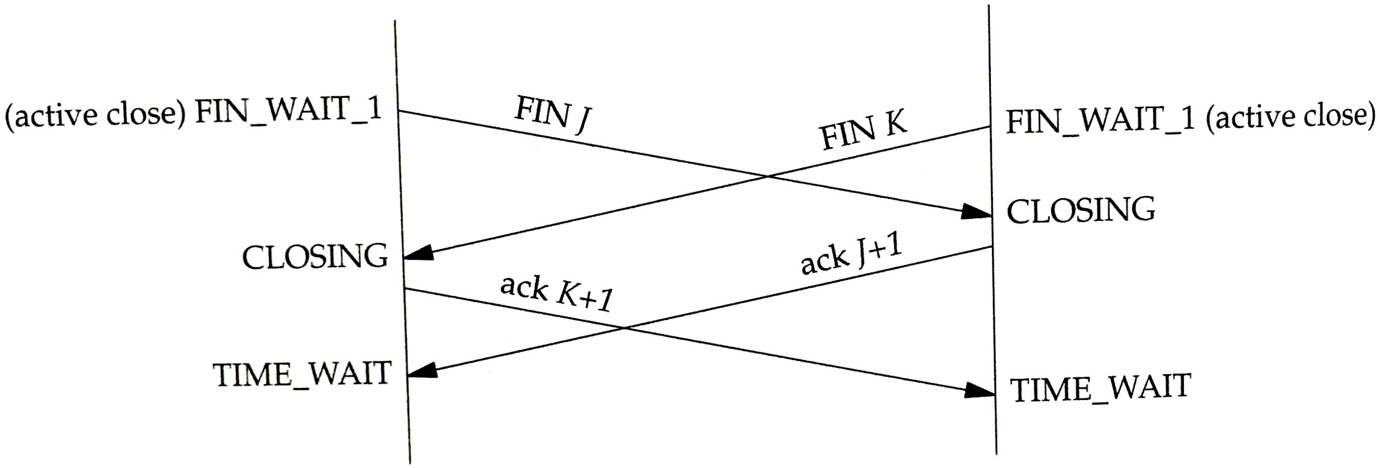
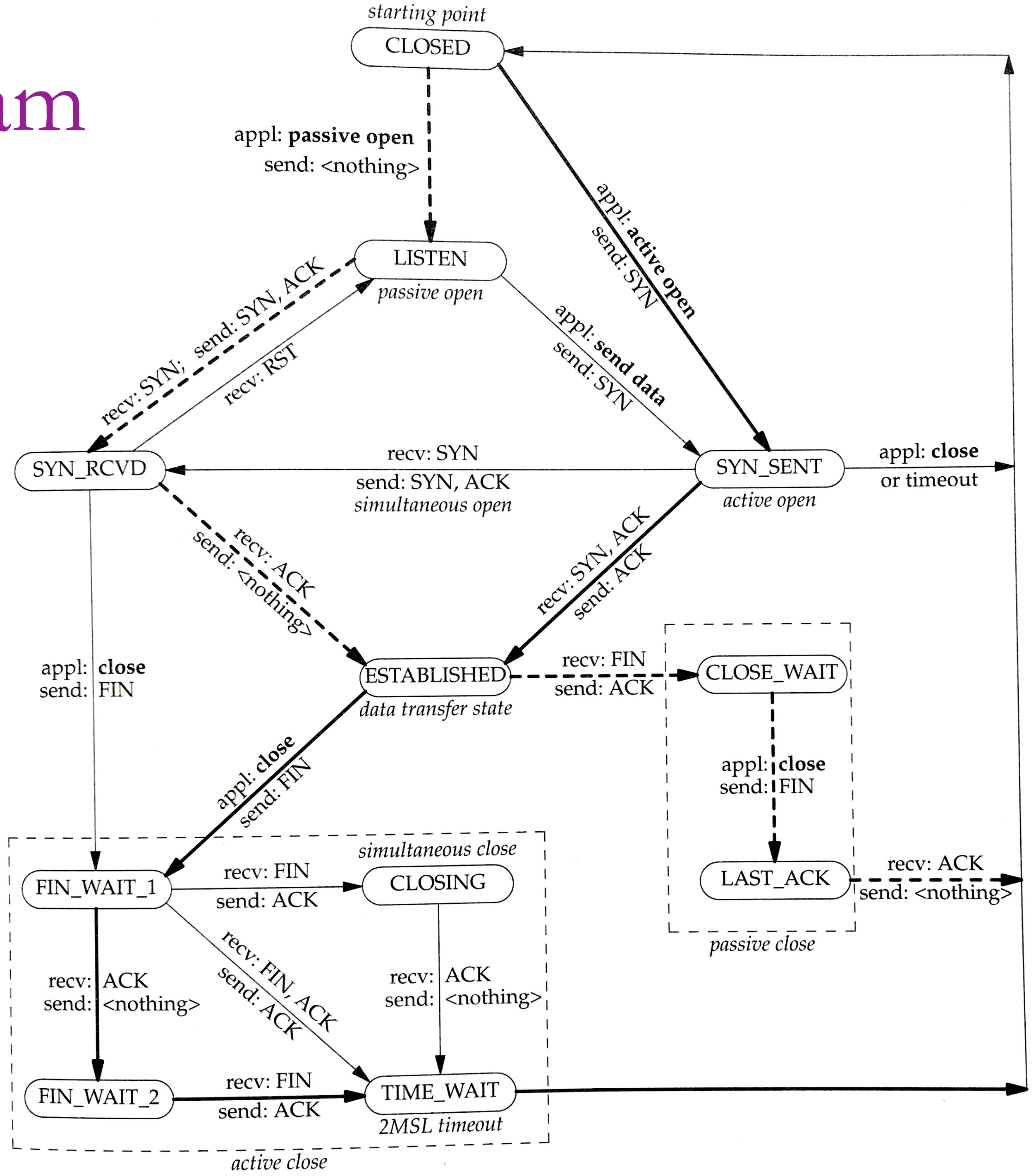


Figure 18.19 Segments exchanged during simultaneous close.

TCP state transition diagram



indicate normal transitions for client
indicate normal transitions for server
appl: *indicate state transitions taken when application issues operation*
recv: *indicate state transitions taken when segment received*
send: *indicate what is sent for this transition*

Figure 18.12 TCP state transition diagram.

Recap

- **TCP** implements a combination of Go Back N and Selective Repeat.
- ACK timeout can be appropriately set with Exponentially-Weighted Moving Average (**EWMA**) of recent RTT and recent **jitter**.
- ACKs count bytes, not packets, and can be piggybacked on data sent in the reverse direction. ACKs are sometimes delayed for efficiency.
- **Triple duplicate ACK** suggests packet loss → retransmit.
- Connection setup requires a 3-way handshake.
  - Connection close also uses a handshake. Each direction is closed.
- TCP throughput should be regulated so as not to overwhelm:
  - the receiver -- **Flow control** is implemented with explicit Receive Window.
  - the network – **Congestion** control will be discussed next lecture.