Lecture 6A

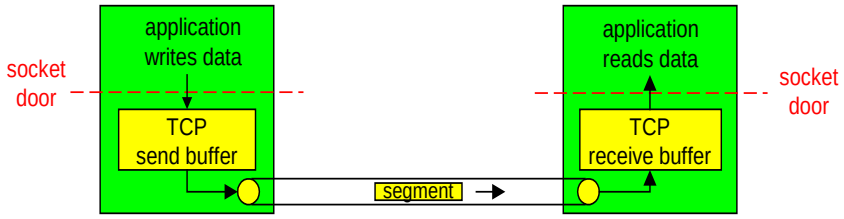
TCP and Flow / Congestion Control

TCP Problems

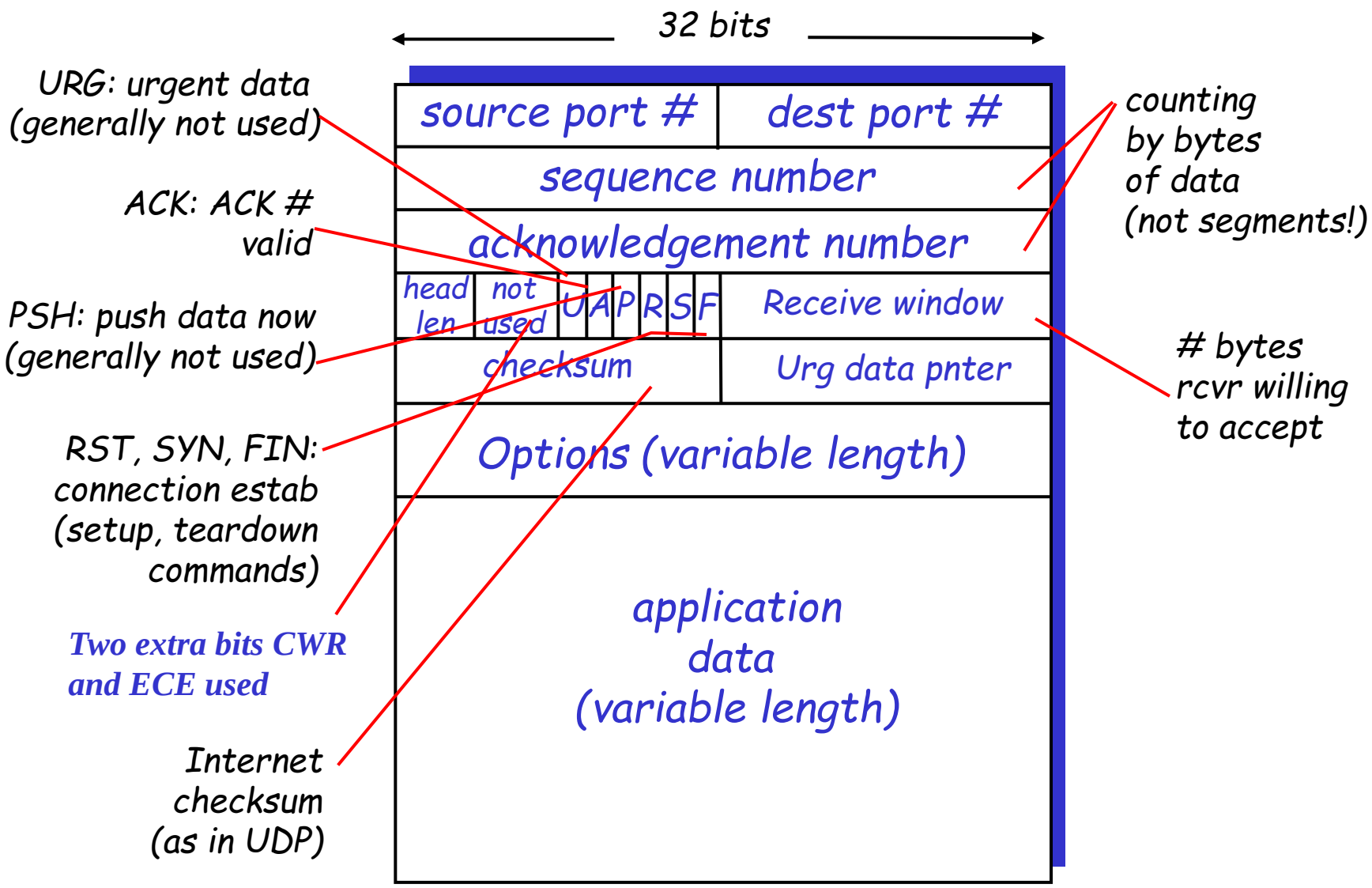
- The problem is that the network can delay, reorder, and lose packets
  - Time-out/retransmission could introduce duplicates of data, acknowledgement, connect, close packets
- Worst case scenario: consider this bank transaction example
  - (a) setup connection
  - (b) transfer $100
  - (c) close connection
  - all messages are delayed and replayed.

TCP Basic Service

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

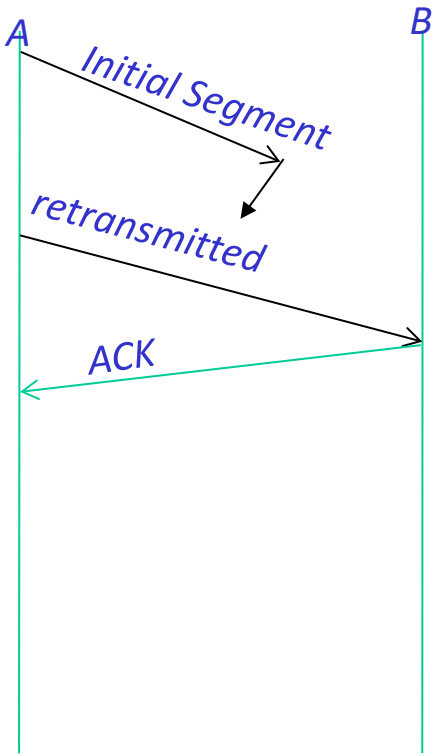


TCP segment structure

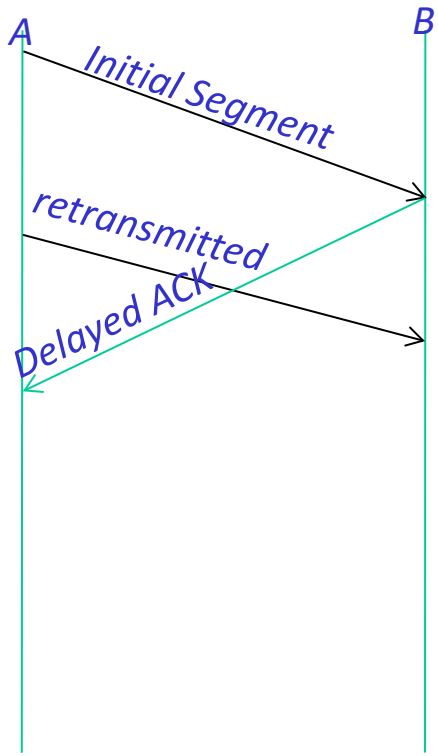


Retransmission Ambiguity in TCP

- A segment is retransmitted if an ack is not received for that segment within the retransmission timeout (RTO) time.
- Unfortunately, we cannot distinguish by the ack whether or not it is for the initial segment or the retransmitted one!



Note that the ACK in both cases will be exactly the same. This is a fundamental problem with the TCP retransmission scheme



TCP Round Trip Time and Timeout

Q: how to set TCP retransmission timeout (RTO) value?

- longer than RTT
  - note: RTT will vary
- 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 (Karn's algorithm)
  - ignore retransmissions, cumulatively ACKed segments
  - Only count single transmission ACKs
- **SampleRTT** will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout (cont)

$$EstimatedRTT' = (1-x)*EstimatedRTT + x*SampleRTT$$

Exponential weighted moving average
influence of given sample decreases exponentially fast
typical value of x: 0.1

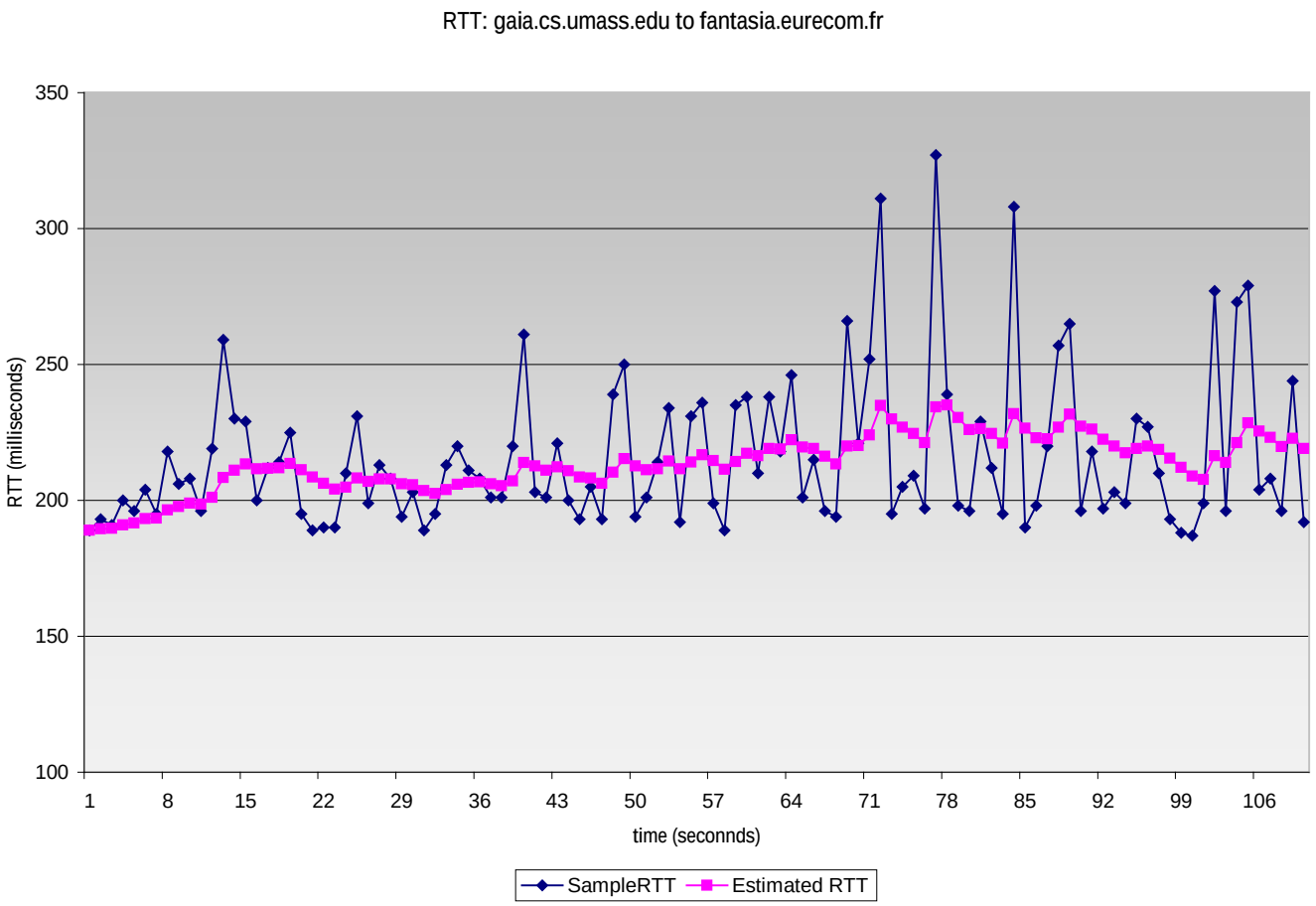
Setting the timeout

- Need **EstimatedRTT** plus "safety margin"
- Timeout is doubled every time a segment is timed out!
- When no timeout, since large variation in **EstimatedRTT** implies that we need larger safety margin, we do the following:

$$Timeout' = EstimatedRTT + 4*Deviation'$$

$$Deviation' = (1-x)*Deviation + x*|SampleRTT-EstimatedRTT|$$

TCP Timeout Example



TCP Connection Management: closing a connection

Modified three-way handshake:

client closes socket:

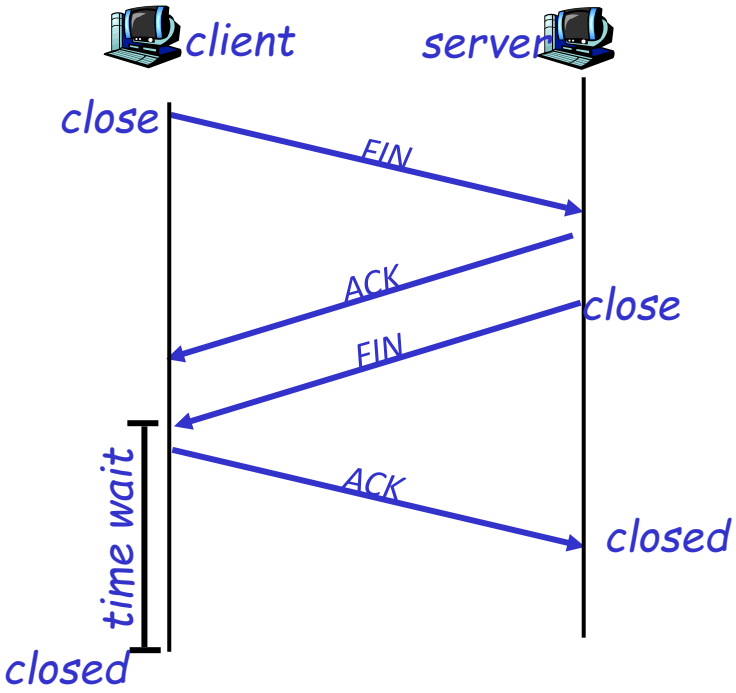
Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Sends FIN.

Step 3: client receives FIN, replies with ACK.

Enters "time wait" - will respond with ACK to received FINs

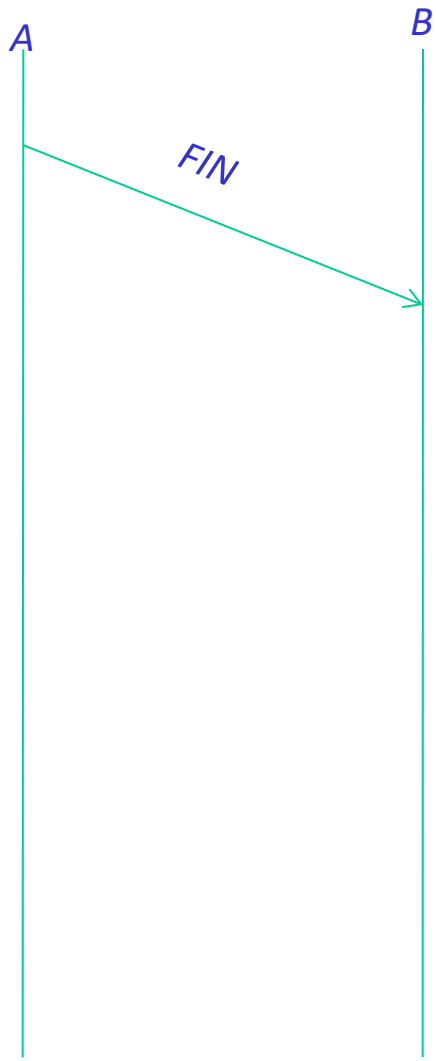
Step 4: server, receives ACK. Connection closed.



- *Socket programming interface*
  - *close() vs shutdown()*

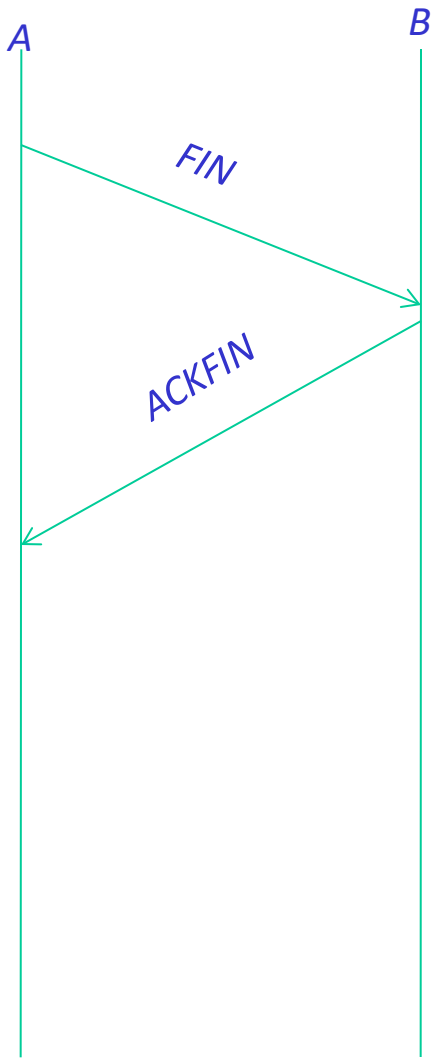
Why does TCP close the connection this way?

- Saying goodbye is difficult!
- Consider A sending file to B. When A gets the last ACK for data, A sends FIN.
- Can A quit at this time? No, because B may not know that A knows that B got all data (in case FIN is lost). So, B might keep sending the last data ACK.



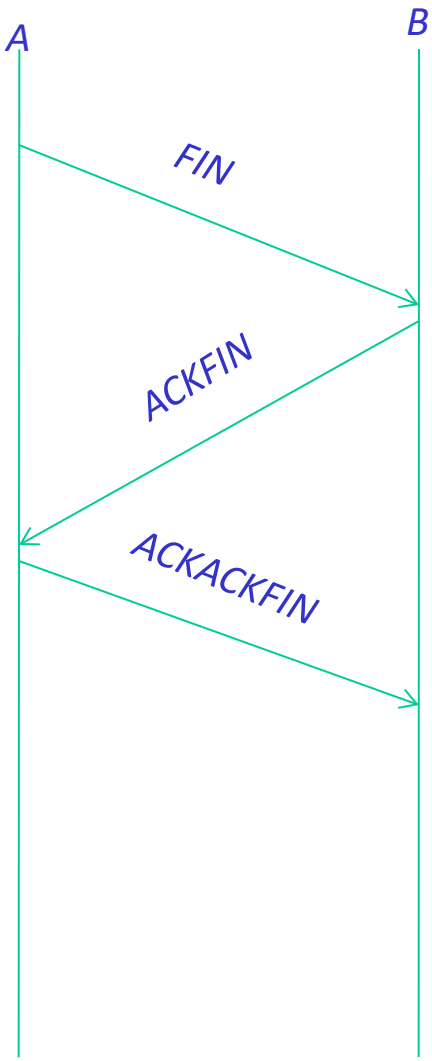
Closing continued

- B ACKs this FIN by sending ACKFIN.
- Can B quit at this time? No, because B does not know whether ACKFIN got through or not.
- If not, A does not know that B knows that A knows that B got all the data. If A is not sure about this, A might keep on sending FIN.



Closing continued

- So A sends ACKACKFIN. Can A quit at this time? Still no, because … (too long, you get it).
- The point is, if you have reason to send the last ACK, you have just as good reason to send last + 1 ACK, because the only way to make sure that the last ACK is received is to receive the ACK for that ACK, and you have to send an ACK to ACK that ACK because the other side is waiting for it.
- Conclusion: No protocol can make sure of graceful close of connection. So have to use timeout. If we do use timeout, better use it earlier than later.



TCP connection closed

- Now, A enters the TIMEWAIT state because it is not sure its ACKACKFIN will be received or not. If not, it assumes that B will retransmit ACKFIN. If it does not receive ACKFIN for TIMEWAIT, it assumes that its ACKACKFIN got through and quit. However, it could happen that all the ACKACKFINs were lost. It could also happen that All B's retransmit of ACKFIN were lost. So there is a (very slight) chance that B did not receive the final ACKACKFIN.
- The approach adopted by TCP at least makes sure that A is sure that B receives FIN (A must receive ACKFIN.)
- So TCP is still reliable for data transfer, because both sides know that the data has been transferred correctly in order.

TCP flow/congestion control

- *Sometimes sender shouldn't send a packet whenever its ready*
  - *Flow control - Receiver not ready (e.g., buffers full)*
  - *React to congestion*
    - *Many unACK'ed packets, may mean long end-end delays, congested networks*
    - *Network itself may provide sender with congestion indication*
  - *Avoid congestion*
    - *Sender transmits smoothly to avoid temporary network overloads*

TCP Flow Control

flow control

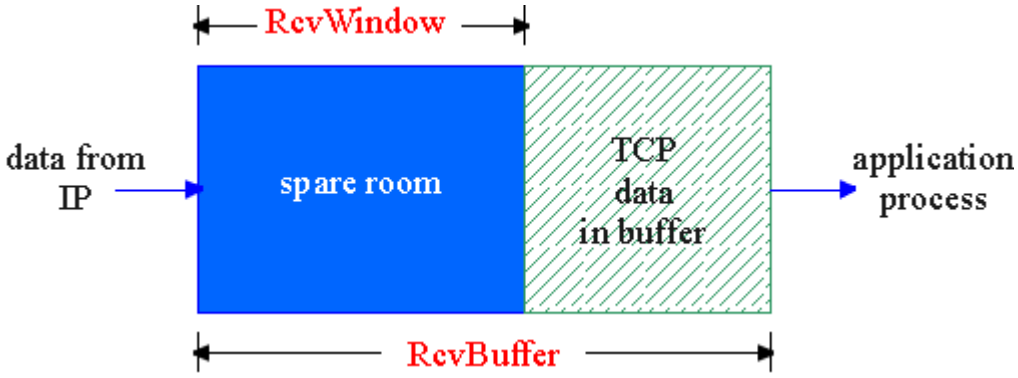
sender won't overrun receiver's buffers by transmitting too much, too fast

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- ***RcvWindow field** in TCP segment*

***RcvBuffer** = size of TCP Receive Buffer*

***RcvWindow** = amount of spare room in Buffer*



receiver buffering

*sender: keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow***

TCP Congestion Control – send window

- To react to congestion, TCP has only one knob – the size of the send window
  - Reduce or increase the size of the send window
- The size of the send window is determined by two things:
  - The size of the receiver window the receiver indicated in the TCP segment
  - The sender's perception about the level of congestion in the network

TCP Congestion Control - approach

- Idea
  - Each source determines network capacity for itself
  - Uses implicit feedback, adaptive congestion window
  - ACKs paced transmission (self-clocking)
- Challenge
  - Determining the available capacity in the first place
  - Adjusting to changes in the available capacity to achieve both efficiency and fairness

Some details of congestion control

- Under TCP, congestion control is done by end systems (doing flow control) that attempt to determine the network capacity without any explicit feedback from the network. *Any packet loss is assumed to be an indication of congestion* since transmission errors are very rare in wired networks. A congestion window is maintained and adapted to the changes in the available capacity. This congestion window limits the number of bytes allowed in transit.

- Since TCP does not have explicit feedback from the network, it is difficult to determine the available capacity precisely. The challenge faced by TCP is how to determine the available capacity in the first place when the connection is just established and also how to adapt to dynamically changing capacity.

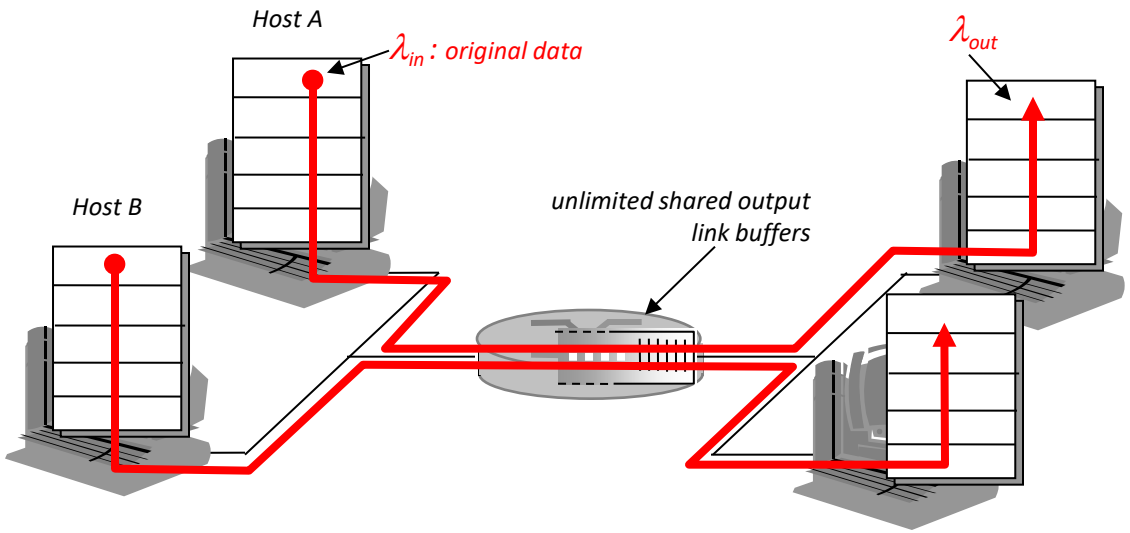
- It should be noted that all things described here about TCP are done per each connection.

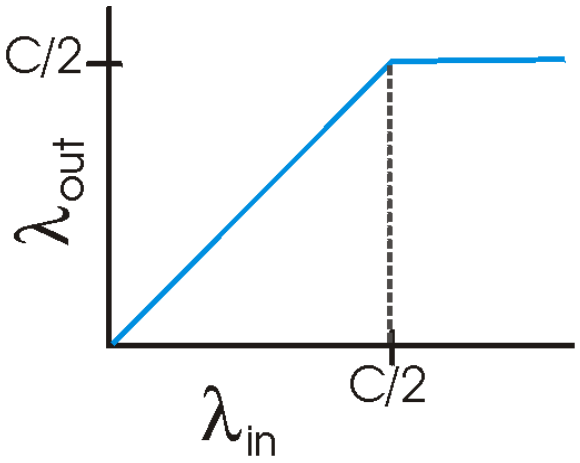
What is Congestion?

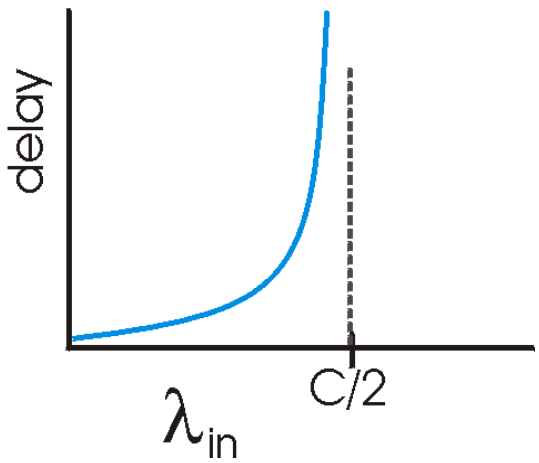
- Informally: “too many sources sending too much data too fast for network to handle”
- Different from flow control, caused by the network not by the receiver
- How does the sender know whether there is congestion? Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queuing in router buffers)

Causes/costs of congestion

- two senders, two receivers
- Bottleneck is channel capacity C across middle links
- one router, infinite buffers
- no retransmission



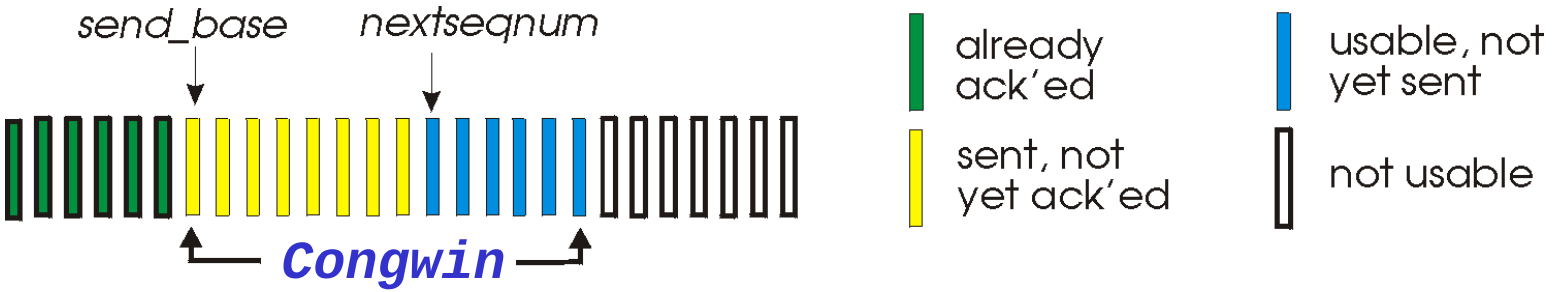




- large delays when congested
- maximum achievable throughput

TCP Congestion Control

- Window-based, implicit, end-end control
- Transmission rate limited by congestion window size, **`Congwin`**, over segments:



w segments, each with MSS (maximum segment size) in bytes sent in one RTT:

$$\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

TCP Congestion Control

- "probing" for usable bandwidth:
  - ideally: transmit as fast as possible (**`Congwin`** as large as possible) without loss
  - *increase* **`Congwin`** until loss (congestion)
  - loss: *decrease* **`Congwin`**, then begin probing (increasing) again

- two "phases"
  - slow start
  - congestion avoidance
- important variables:
  - **`Congwin`**
  - **`threshold:`** defines threshold between slow start phase and congestion avoidance phase

TCP Slowstart

Slowstart algorithm

initialize: Congwin = 1
for (each segment ACKed)
Congwin++
until (loss event OR
CongWin > threshold)

- exponential increase (per RTT) in window size (not so slow!)
- loss event: timeout (Tahoe TCP)

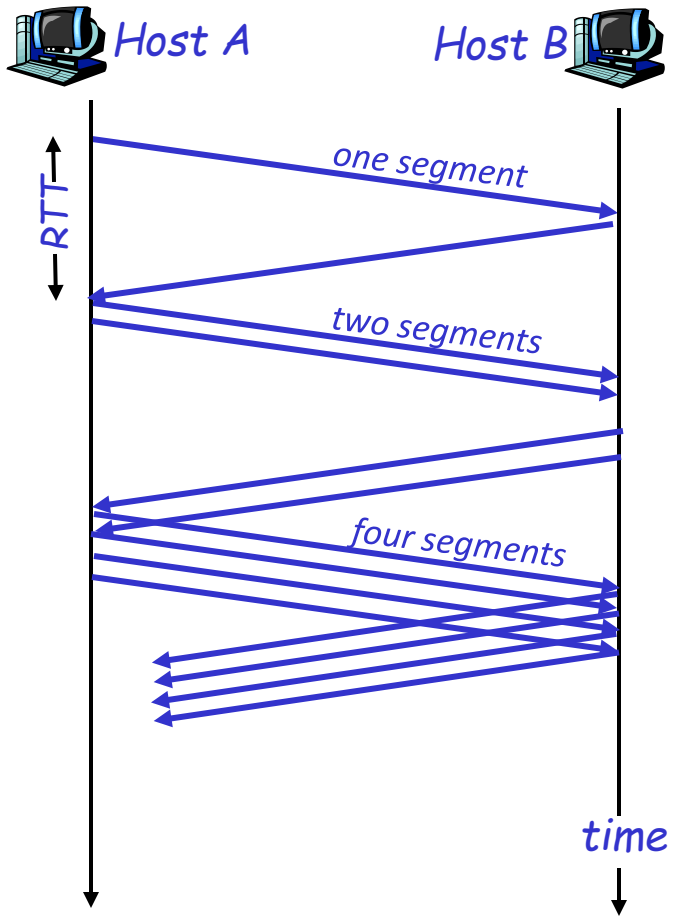


Illustration of slow start: Congwin starts with 1 and after a round trip time it increases to 2, then 4 and so on. This exponential growth continues till there was any loss or the threshold is reached. After reaching threshold, we enter congestion avoidance phase. Loss may be detected either through a timeout or duplicate acks.

Why Slow Start?

- Objective
  - Determine the available capacity in the first place
- Idea
  - Begin with congestion window = 1 pkt
  - Double congestion window each RTT
    - Increment by 1 packet for each ack
- Exponential growth but slower than one blast
- Used when
  - First starting connection
  - Connection goes dead waiting for a timeout

Note: slow start is essentially used to figure the available capacity in the network when the connection is established. This is where slow start is used. CongWin grows exponentially in the slow start phase.
Why call this slow start? This approach is considered slower than sending all the initial data for a connection in one blast.

After Slowstart and During Congestion Avoidance

- A connection stays in slow start phase only till the threshold is reached. It then enters congestion avoidance phase where CongWin is increased by 1 for every round trip time.

- Essentially, CongWin is increased exponentially during slow start phase and linearly during congestion avoidance phase.

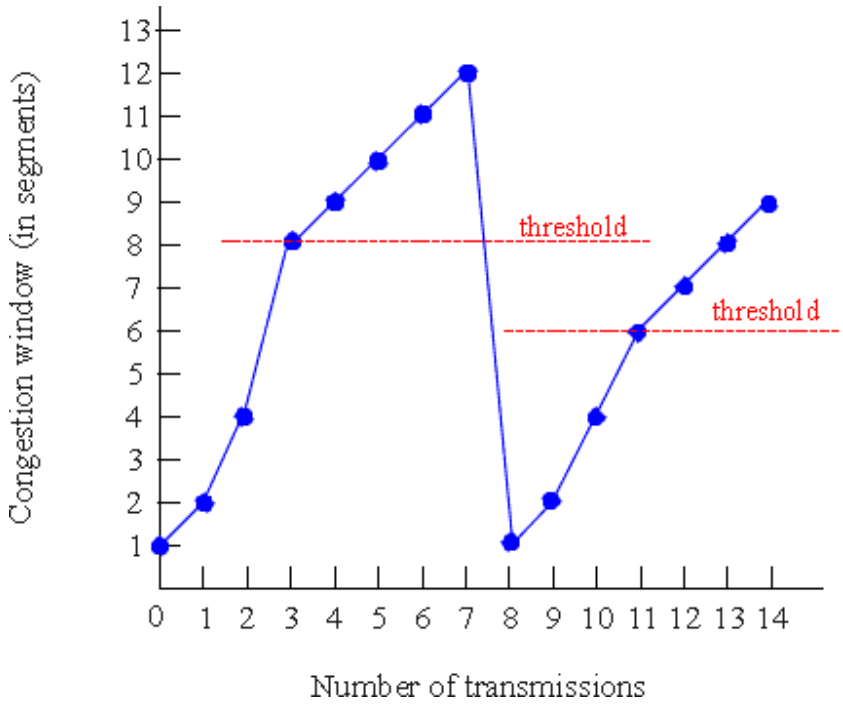
- These phases are illustrated in the graph in the next slide.
  - Initially, CongWin is set to 1. Between the round trip times 0 and 3, the connection is slow start phase and CongWin is increase to 8.
  - At this point the threshold is reached and then on CongWin is increased by 1 for every RTT. This CongWin reaches the value of 12.

- All this is when there were no losses. What happens when there is a loss? Drastic action is taken.
  - The threshold is reset to current CongWin / 2. The CongWin is set to 1.
  - we are back in slow start phase. In the graph there was a loss when CongWin was 12. So the threshold is set to 6, CongWin is set to1 and we are back in the slow start phase.

TCP Congestion Avoidance

Congestion avoidance

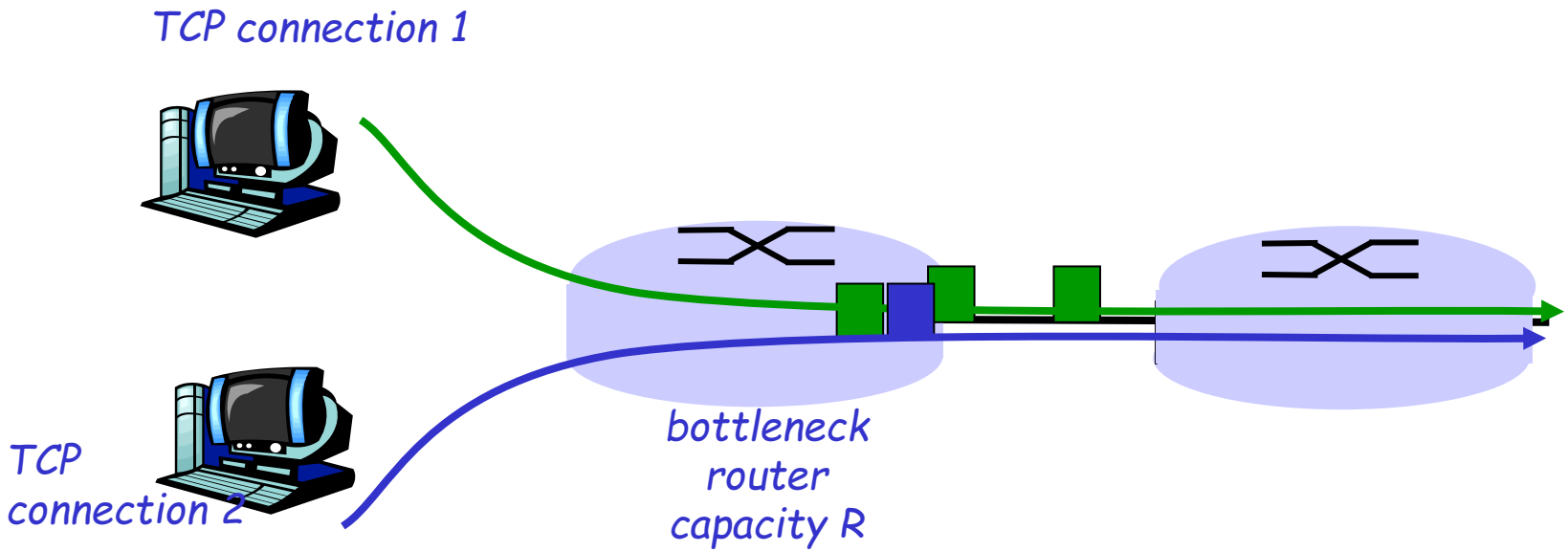
```
/* slowstart is over        */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
     Congwin++
  }
threshold = Congwin/2
Congwin = 1
perform slowstart
```



The algorithm and graph show how TCP adapts its congestion window based on packet successes/losses. This pattern of continually increasing and decreasing CongWin continues thru the lifetime of a connection and looks like a saw tooth pattern.

TCP Fairness

Fairness goal: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity

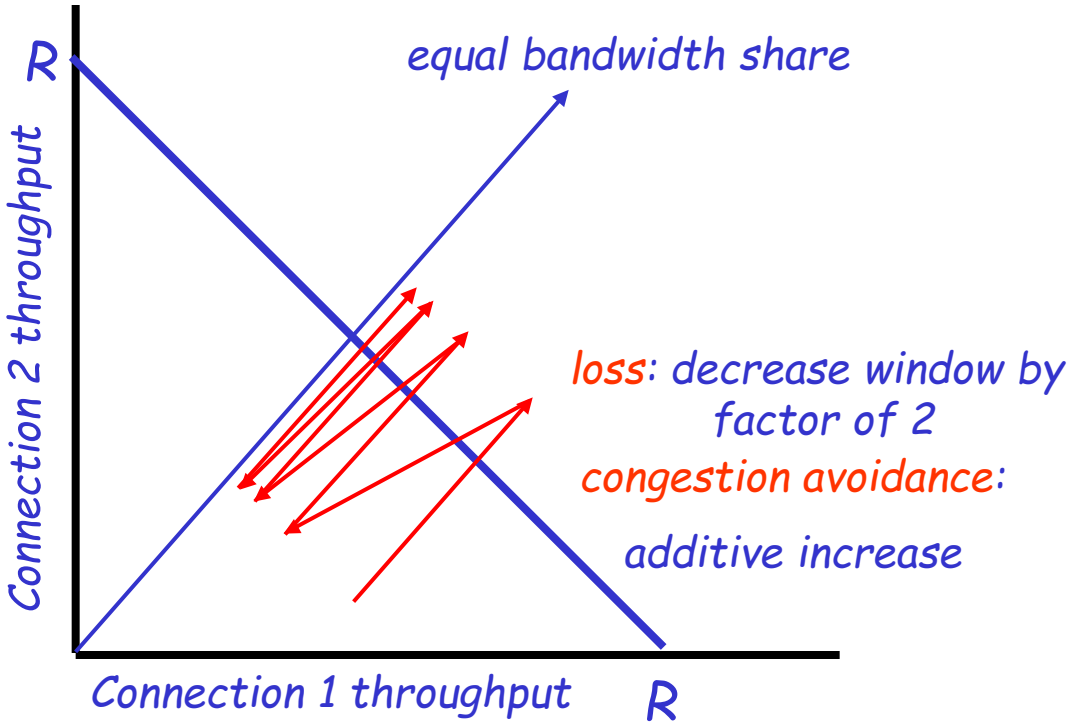


The AIMD (additive increase, multiplicative decrease) approach used by TCP is meant to ensure fairness and stability. Suppose a bottleneck link with capacity R is shared by two connections. Fairness criteria says that both connections should get roughly equal share.

Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughput increases
- multiplicative decrease decreases throughput proportionally
- Red line shows behavior of connection 1



However, TCP is not perfectly fair. It biases towards flows with small RTT.

Graph Details

- The graph shows connection 1's throughput on the x-axis and connection 2's throughput on the y-axis. The sum of their throughputs has to be below R. Crossing that line means loss. The middle arrow shows the line of equal bandwidth share. If you stay on this line, each connection is getting its fair share. If the sum is close to R, that means we are getting the best throughput.

- In this illustration, connection 1 starts at an arbitrary point (lowest right) and linearly increases its congestion window. Once it senses loss, it decreases the window by 2 (moves to the left, up). This way with linear increases and multiplicative decreases the connection 1's throughput approaches the middle line ensuring fairness. Similarly for connection 2 if they have the same rtt.

Details in Updating cwnd and ssthresh (RFC 2581)

- Terms
  - SMSS: sender maximum segment size
  - FlightSize: number of bytes sent not acked
- In slow start, per ACK (for new data)

  cwnd += SMSS
- In congestion avoidance, per (non-duplicate) ACK,

  cwnd += SMSS*SMSS/cwnd
- When timer expires,

  ssthresh = max (FlightSize / 2, 2*SMSS)

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	*TCP Receiver action*
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	*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 single cumulative ACK, ACKing both in-order segments*
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	*Immediately send duplicate ACK, indicating seq. # of next expected byte*
Arrival of segment that partially or completely fills gap	*Immediate send ACK*

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.

- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
        if (y > SendBase) {
            SendBase = y
            if (there are currently not-yet-acknowledged segments)
                start timer
          }
        else {
            increment count of dup ACKs received for y
            if (count of dup ACKs received for y = 3) {
                resend segment with sequence number y
            }
```

a duplicate ACK for already ACKed segment

fast retransmit

Fast Recovery

- Fast retransmit means that we do not have to go into the normal recovery mode – no need to do a real slow start.
- Set cwnd to be a larger value.

Fast Retransmit and Fast Recovery in Details

- The fast retransmit and fast recovery algorithms are usually implemented together as follows.
  1. When the third duplicate ACK is received, set ssthresh to no more than *max (FlightSize / 2, 2*SMSS)*.
  2. Retransmit the lost segment and set cwnd to ssthresh plus 3*SMSS. This artificially "inflates" the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.
  3. For each additional duplicate ACK received, increment cwnd by SMSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.
  4. Transmit a segment, if allowed by the new value of cwnd and the receiver's advertised window.
  5. When the next ACK arrives that acknowledges new data, set cwnd to ssthresh (the value set in step 1). This is termed "deflating" the window.
     - This ACK should be the acknowledgment elicited by the retransmission from step 1, one RTT after the retransmission (though it may arrive sooner in the presence of significant out- of-order delivery of data segments at the receiver). Additionally, this ACK should acknowledge all the intermediate segments sent between the lost segment and the receipt of the third duplicate ACK, if none of these were lost.

- http://www.faqs.org/rfcs/rfc2581.html