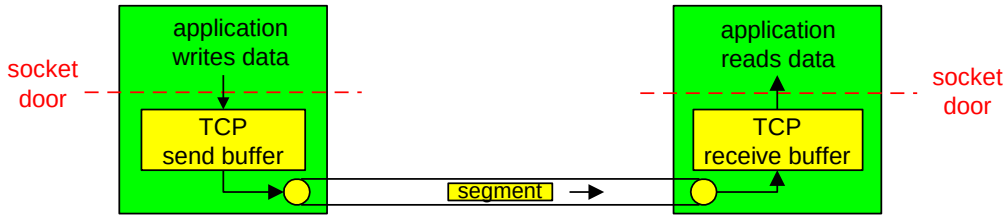
TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*

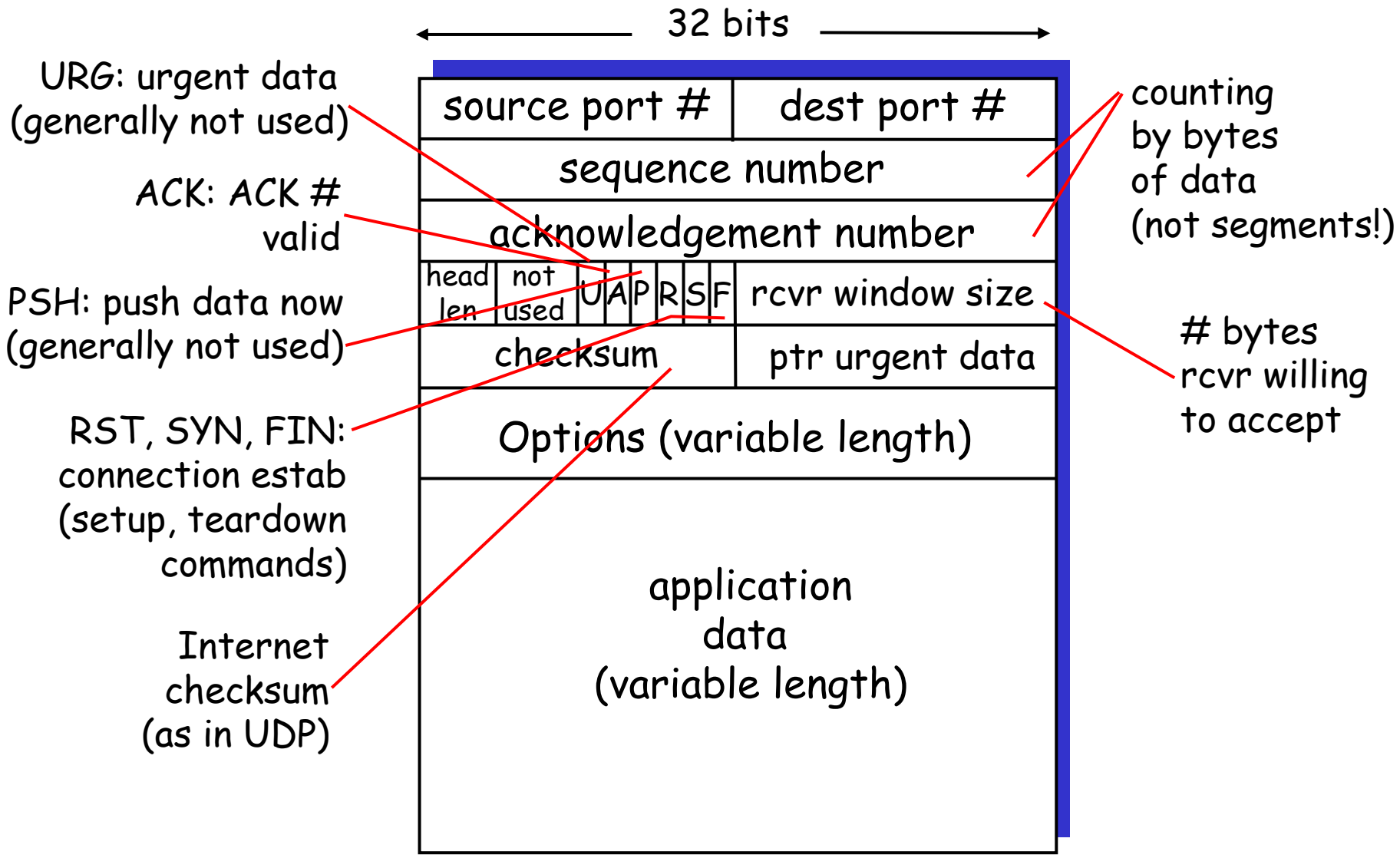


- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

Γενικά χαρακτηριστικά και λειτουργίες του πρωτοκόλλου TCP

- Reliable Delivery (Αξιόπιστη Μετάδοση)
- Error Detection (Ανίχνευση Λαθών)
- Error Correction (Διόρθωση Λαθών)
- Full-duplex (Δικατευθυντήρια/ Αμφίδρομη Μετάδοση)
- Flow Control (Έλεγχος Ροής)
- Sequence numbers
- Cumulative Acknowledgement
- Point-to-Point σύνδεση (ένας αποστολέας – ένας παραλήπτης)
- Multiplexing/demultiplexing (μέσω των port numbers – βασικό!!!)
- Connection-oriented (με three-way handshake σήματα)
- Congestion Control (έλεγχος συμφόρησης)

TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

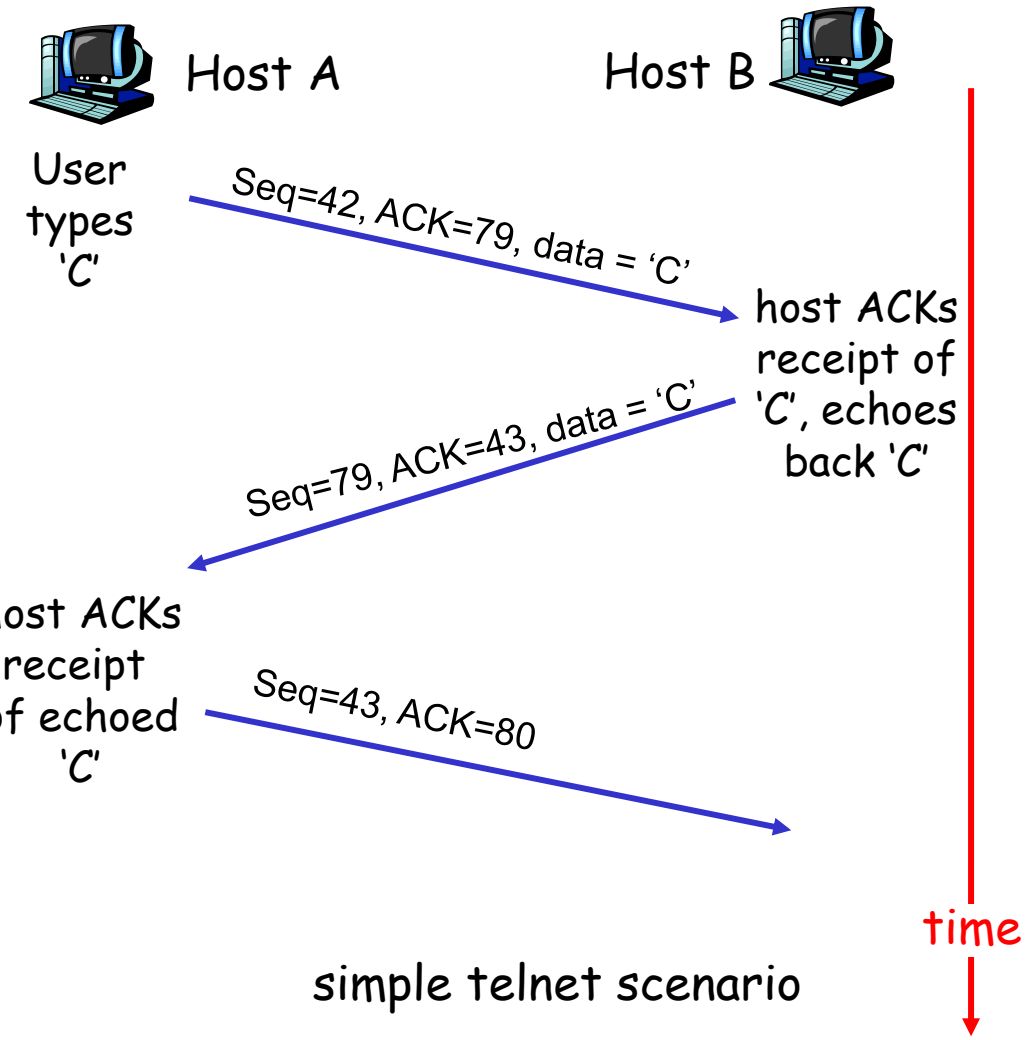
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

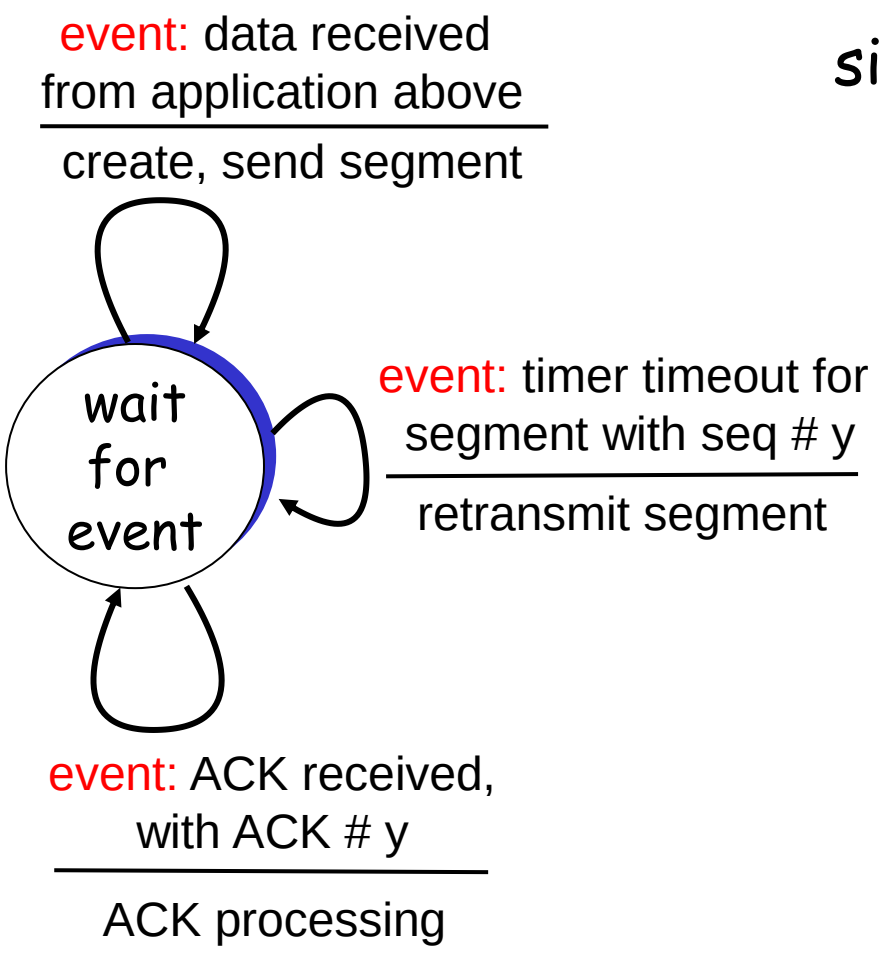
Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

TCP: reliable data transfer



simplified sender, assuming

- one way data transfer
- no flow, congestion control

TCP: reliable data transfer

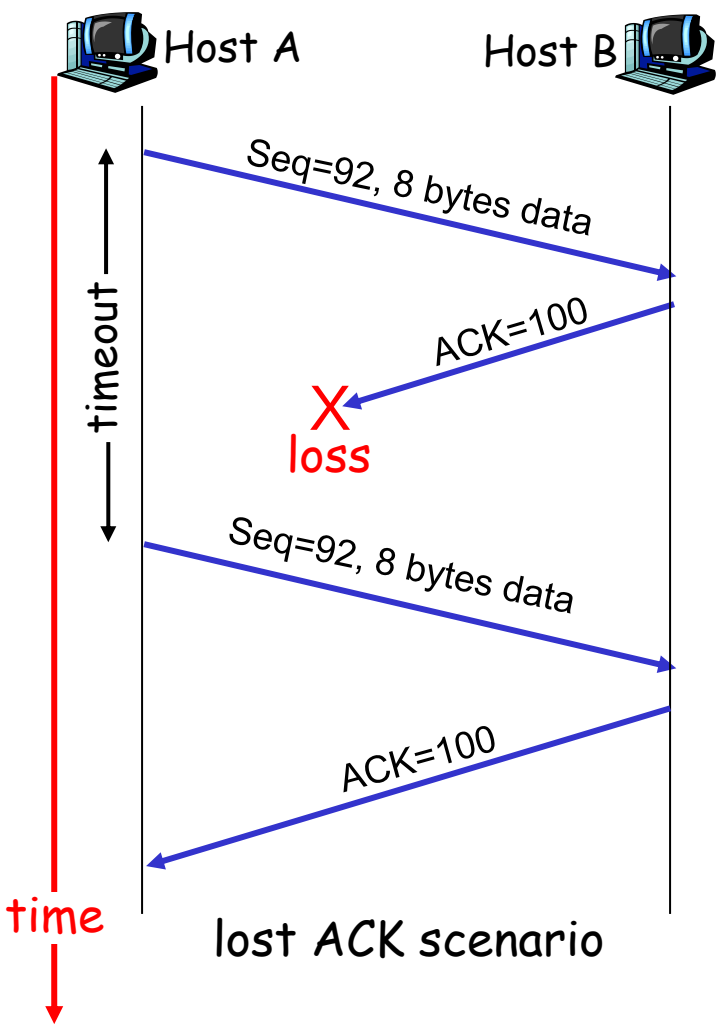
Simplified TCP sender

```
00  sendbase = initial_sequence number
01   nextseqnum = initial_sequence number
02
03    loop (forever) {
04      switch(event)
05      event: data received from application above
06           create TCP segment with sequence number nextseqnum
07           start timer for segment nextseqnum
08           pass segment to IP
09           nextseqnum = nextseqnum + length(data)
10       event: timer timeout for segment with sequence number y
11           retransmit segment with sequence number y
12           compute new timeout interval for segment y
13           restart timer for sequence number y
14       event: ACK received, with ACK field value of y
15           if (y > sendbase) { /* cumulative ACK of all data up to y */
16               cancel all timers for segments with sequence numbers < y
17                sendbase = y
18                }
19           else { /* a duplicate ACK for already ACKed segment */
20                increment number of duplicate ACKs received for y
21                if (number of duplicate ACKS received for y == 3) {
22                   /* TCP fast retransmit */
23                   resend segment with sequence number y
24                   restart timer for segment y
25               }
26      }  /* end of loop forever */
```

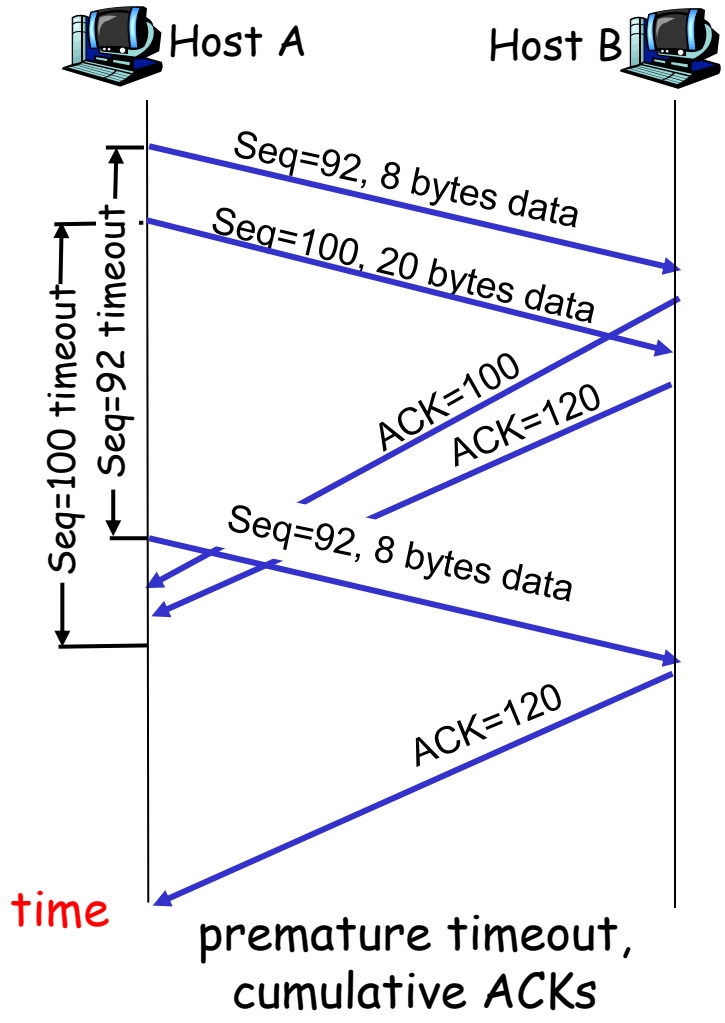
TCP ACK generation [RFC 1122, RFC 2581]

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

TCP: retransmission scenarios

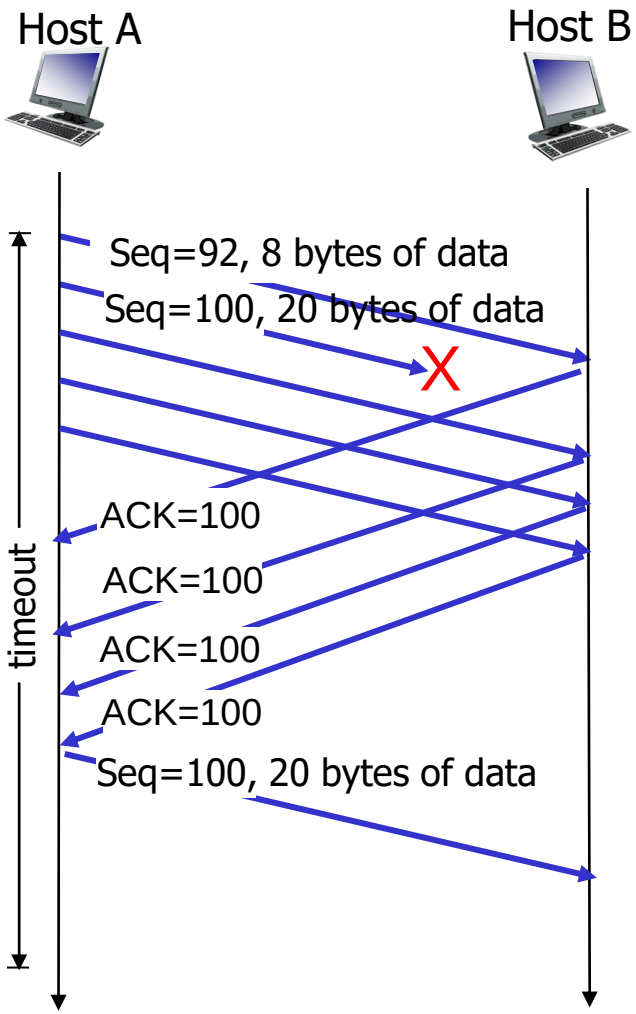


lost ACK scenario



premature timeout, cumulative ACKs

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

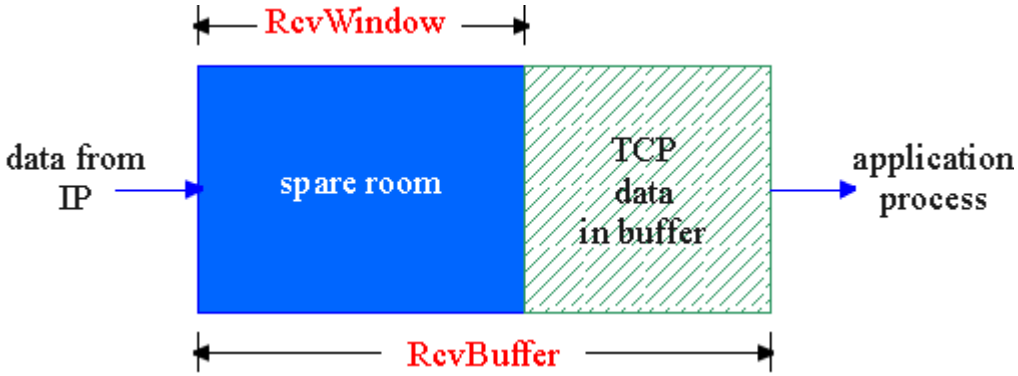
TCP Flow Control

flow control

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

RcvBuffer = size or TCP Receive Buffer

RcvWindow = amount of spare room in Buffer



receiver buffering

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

- **RcvWindow field** in TCP segment

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

TCP Round Trip Time and Timeout

Q: how to set TCP timeout 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
  - ignore retransmissions, cumulatively ACKed segments
- **SampleRTT** will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current **SampleRTT**

TCP Round Trip Time and Timeout

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

- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.1 (or x = 0.125)

Setting the timeout

- **EstimtedRTT** plus "safety margin"
- large variation in **EstimatedRTT** -> larger safety margin

Timeout = EstimatedRTT + 4*Deviation

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

typical value of x: 0.25

TCP Connection Management (setup)

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **RcvWindow**)
- *client*: connection initiator

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

- *server*: contacted by client

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

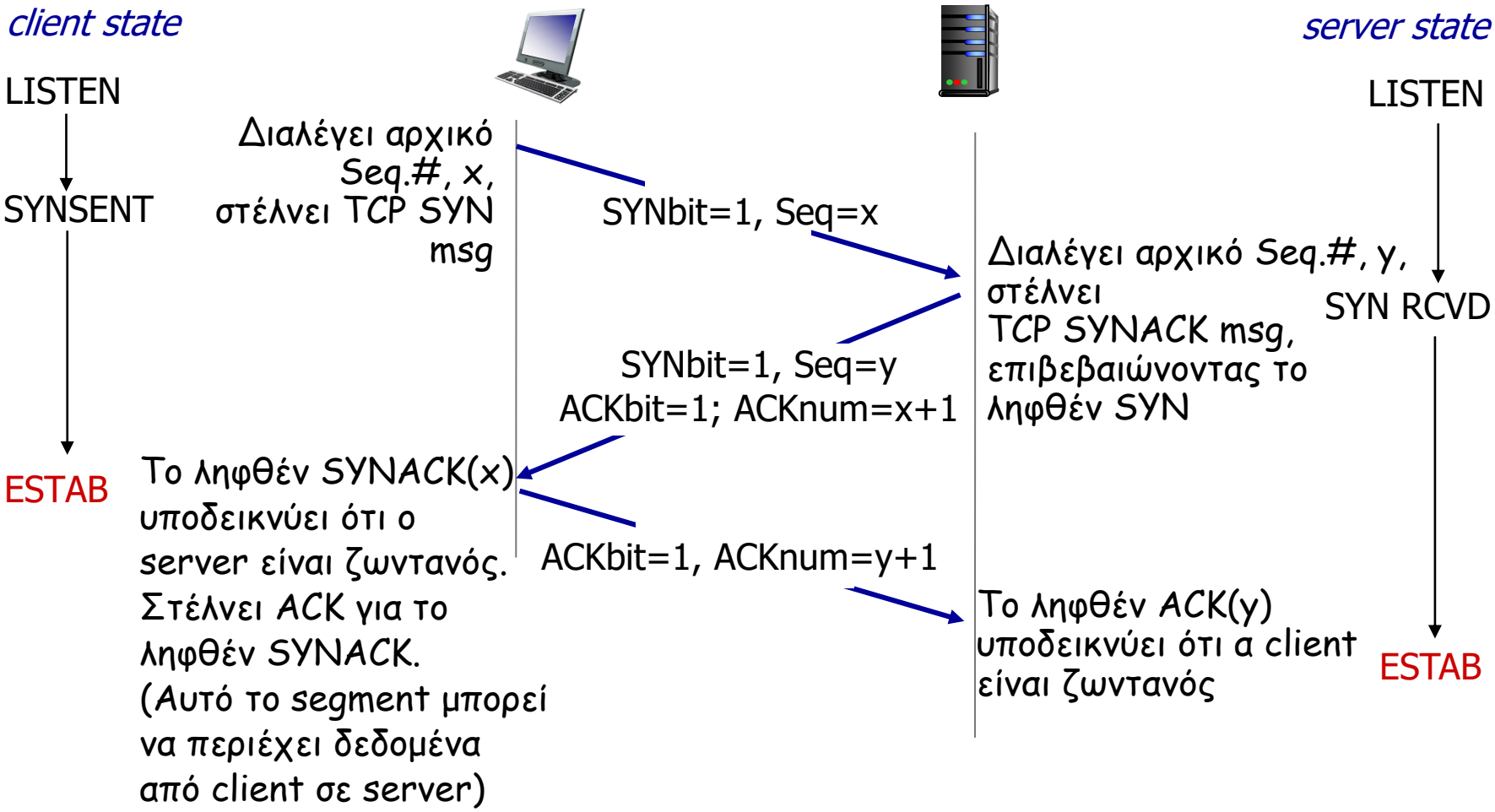
Three way handshake:

Step 1: client end system sends TCP SYN control segment to server
- specifies initial seq #

Step 2: server end system receives SYN, replies with SYNACK control segment
- ACKs received SYN
- allocates buffers
- specifies server-> receiver initial seq. #

Step 3: επιβεβαίωση από client end system

Διαδικασία εγκατάστασης διασύνδεσης TCP



TCP Connection Management (teardown)

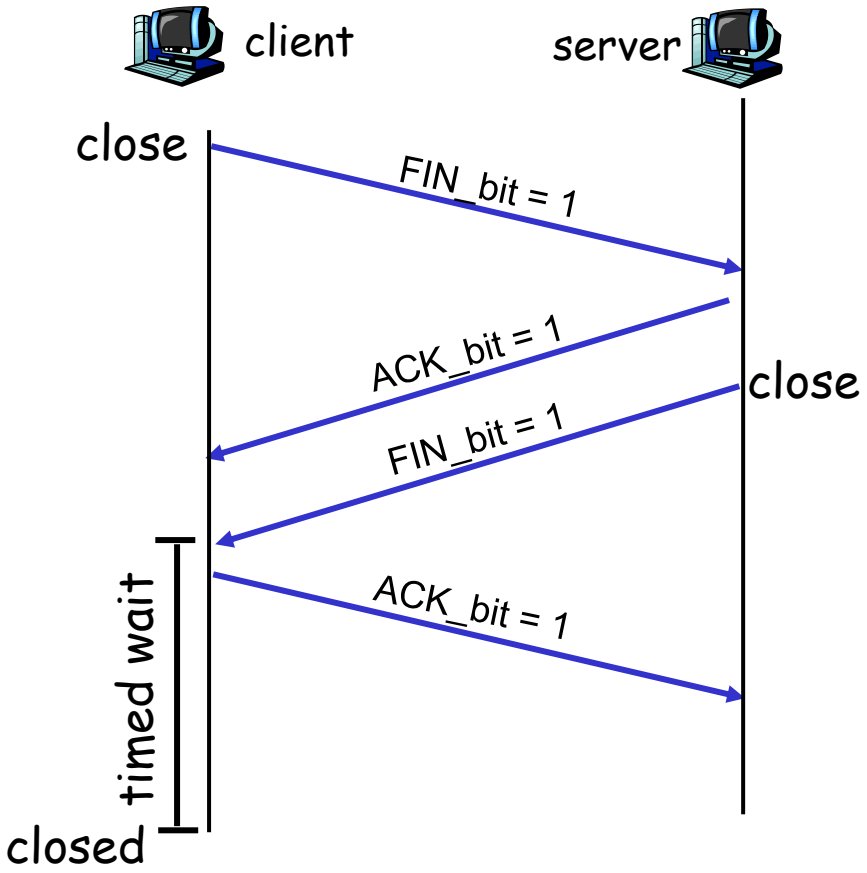
Closing a connection:

client closes socket:

```
clientSocket.close();
```

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

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



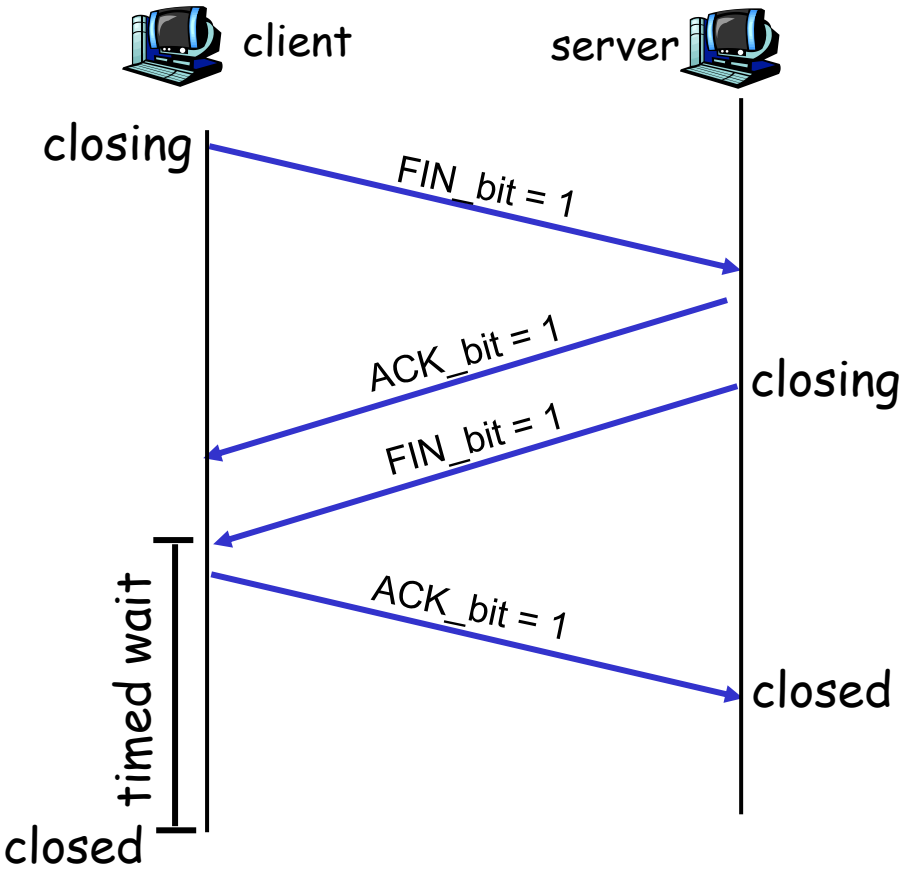
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

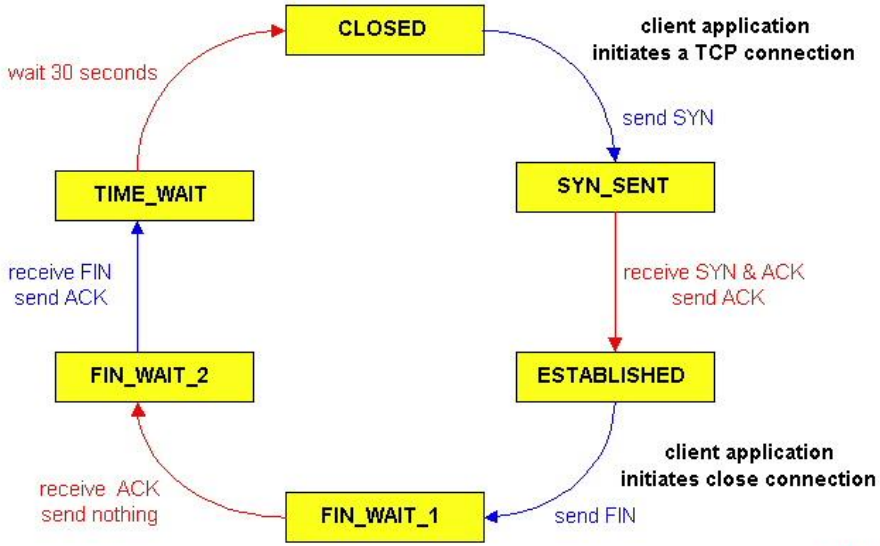
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handly simultaneous FINs.

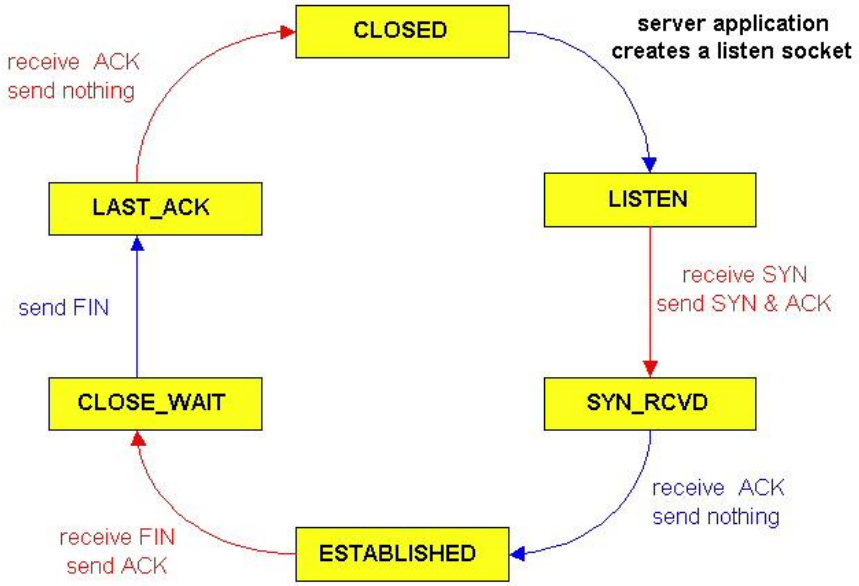


TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle



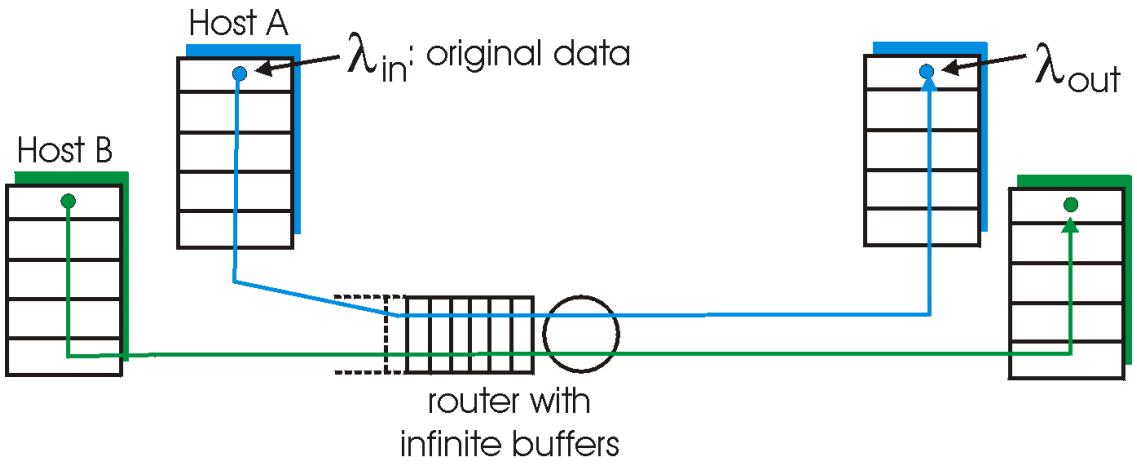
Principles of Congestion Control

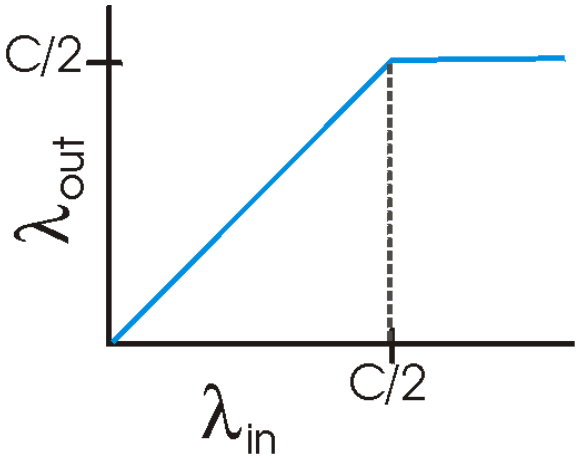
Congestion:

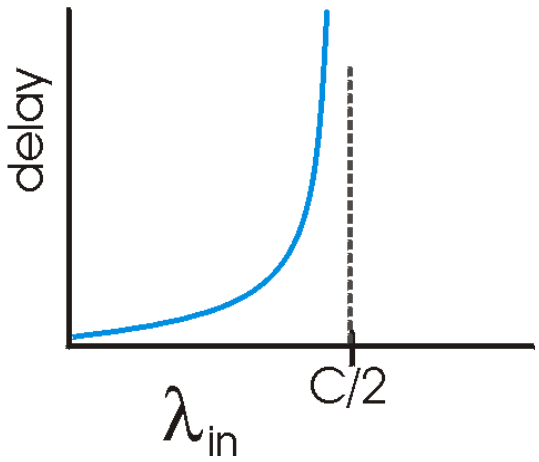
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission



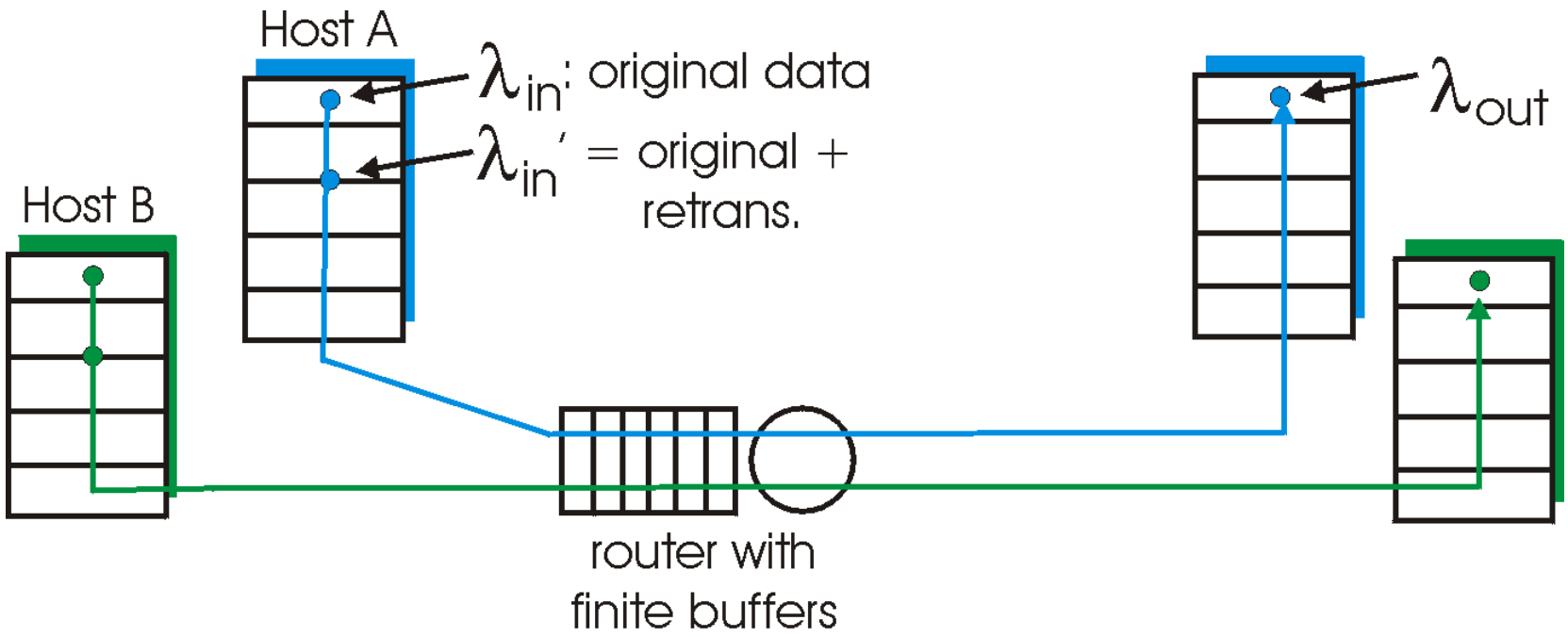




- large delays when congested
- maximum achievable throughput

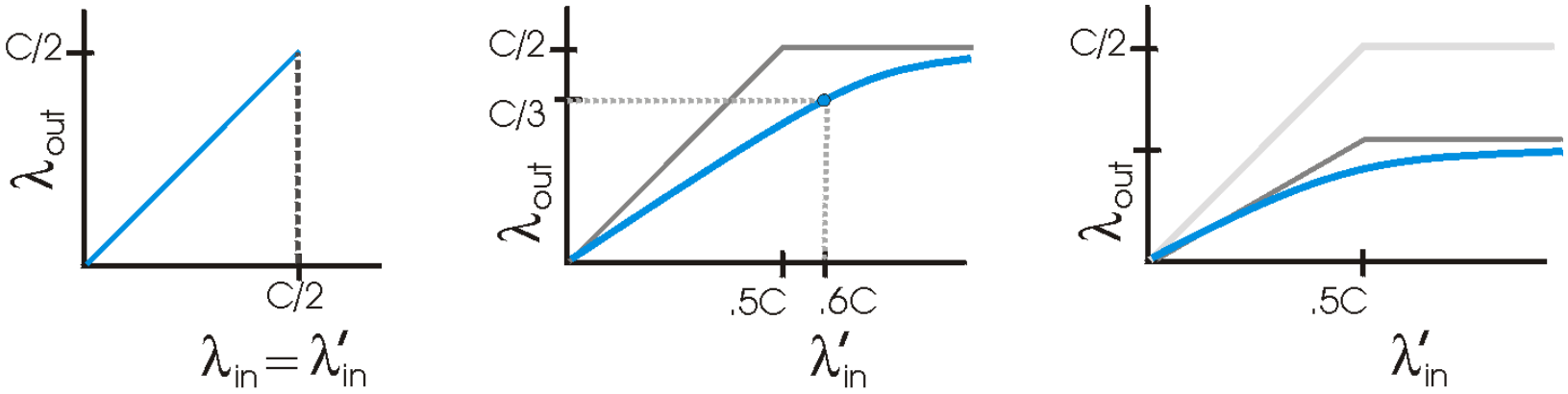
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet



Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



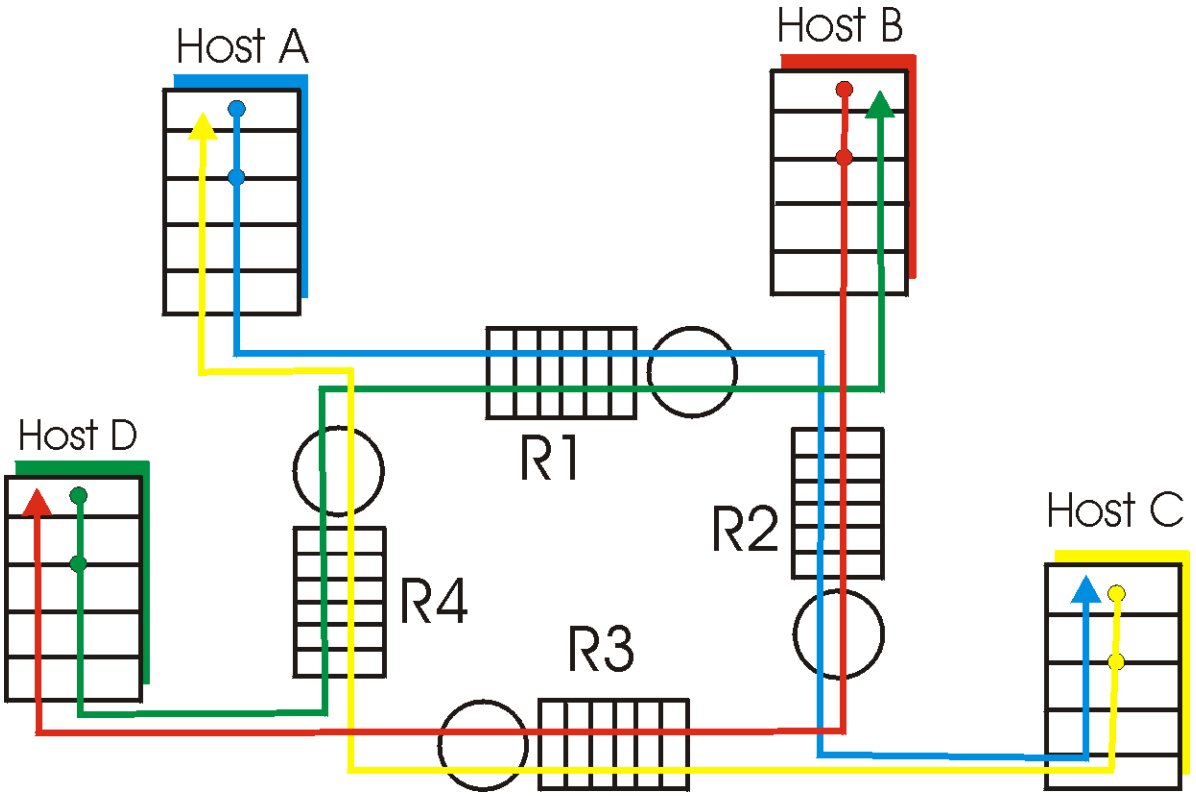
"costs" of congestion:

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

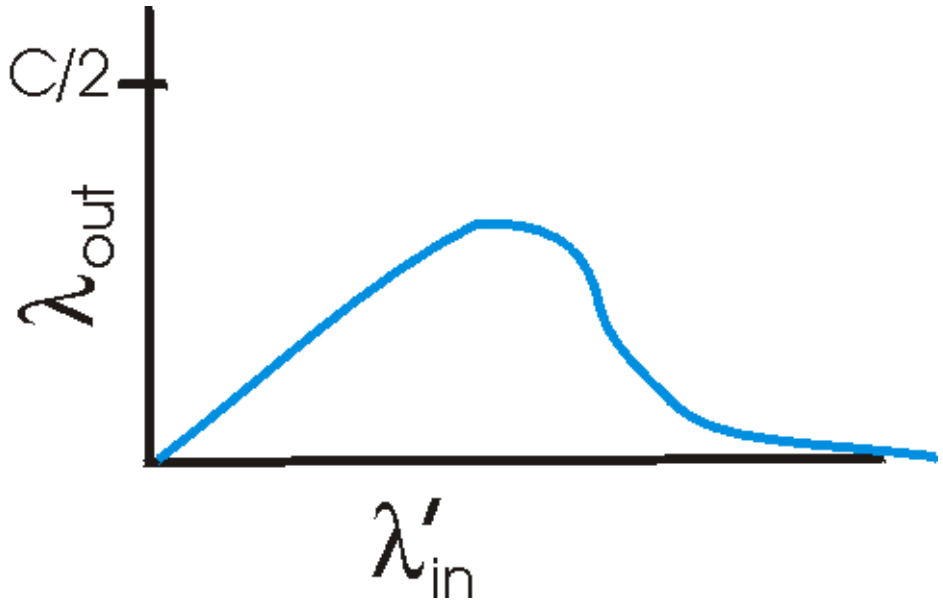
Causes/costs of congestion: scenario 3

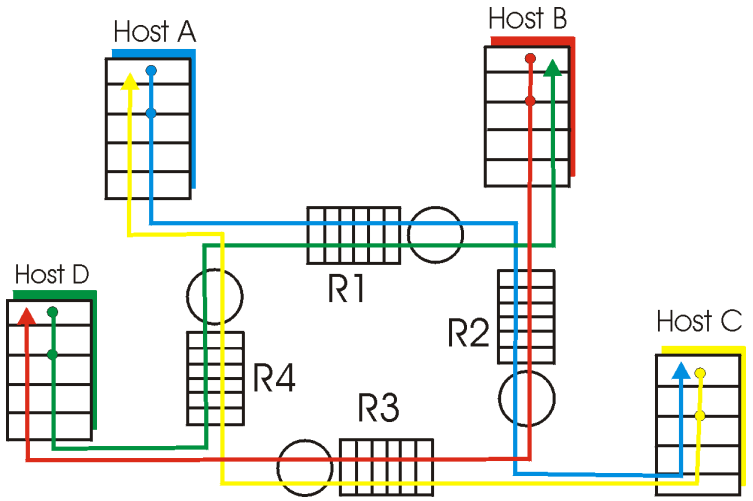
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as λ_{in} and λ'_{in} increase ?



Causes/costs of congestion: scenario 3





Another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

Case study: ATM ABR congestion control

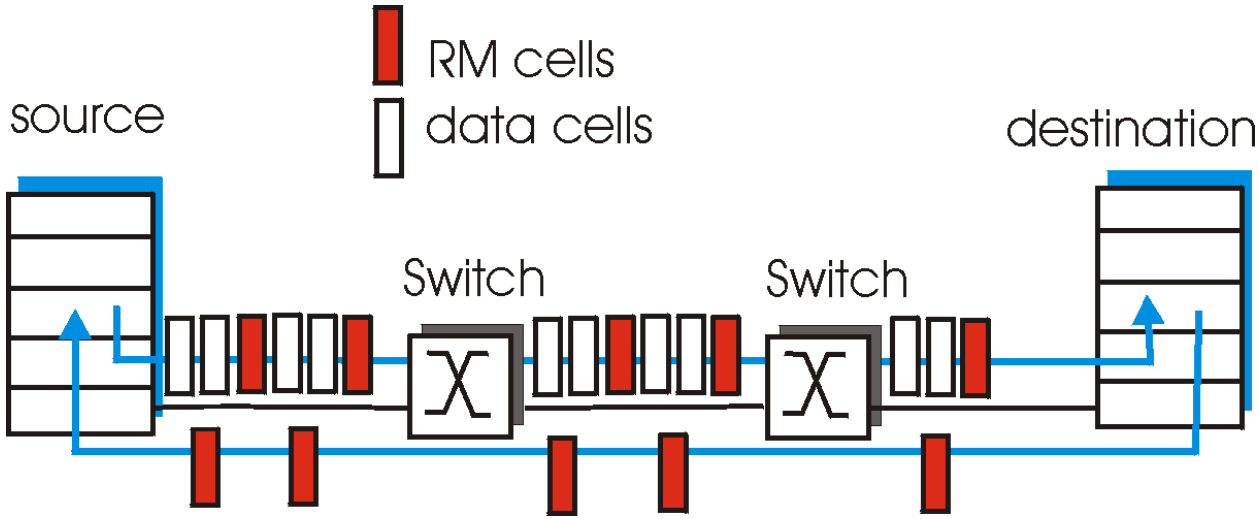
ABR: available bit rate:

- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:

- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("*network-assisted*")
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact

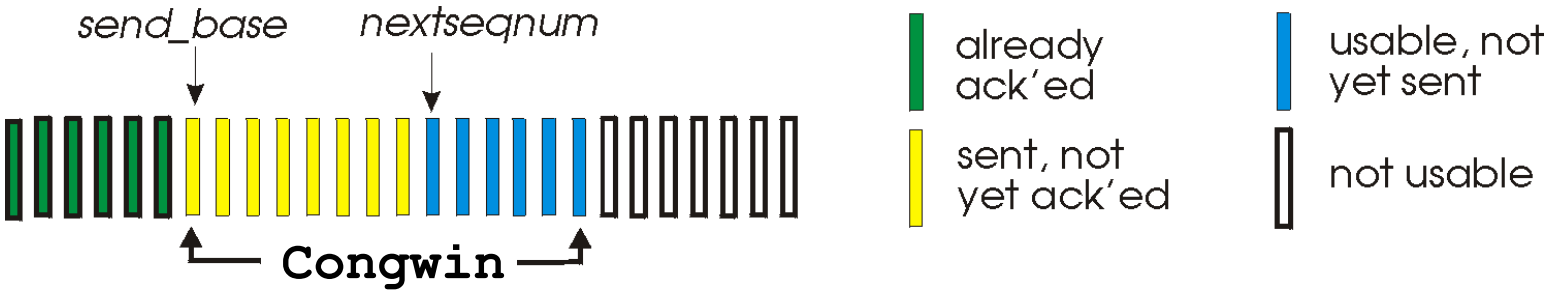
Case study: ATM ABR congestion control



- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender' send rate thus minimum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

TCP Congestion Control

- end-end control (no network assistance)
- transmission rate limited by congestion window size, `Congwin`, over segments:



- w segments, each with MSS bytes sent in one RTT:

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

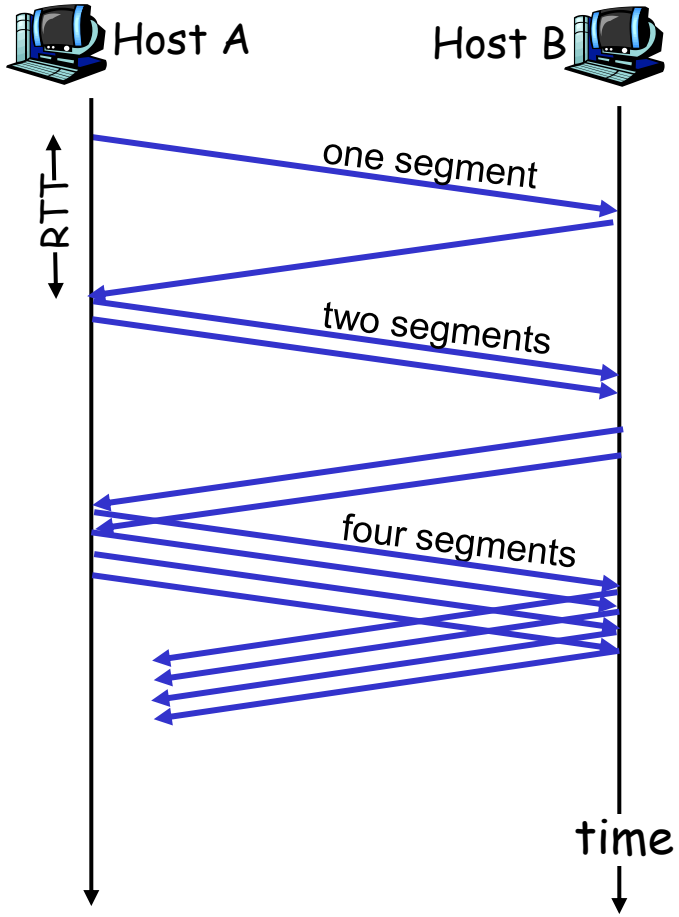
TCP congestion control:

- "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 two slow start phase, congestion control phase

TCP Slowstart

Slowstart algorithm

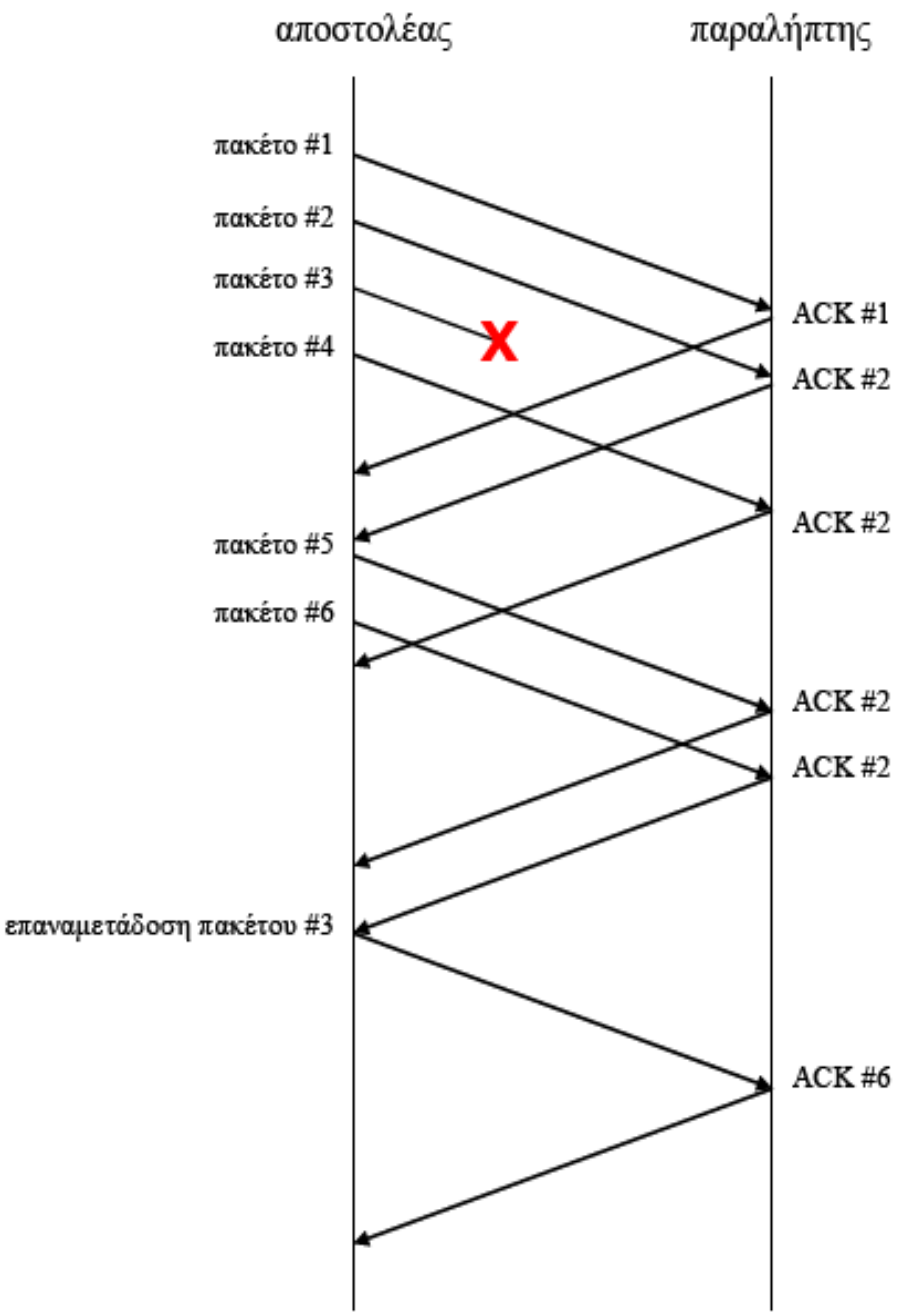
initialize: Congwin = 1
for (each segment ACKed)
    Congwin++
until (loss event OR
    CongWin > threshold)



- exponential increase (per RTT) in window size (not so slow!)
- loss event: timeout (Tahoe TCP) and/or or three duplicate ACKs (Reno TCP)

Γρήγορη επαναμετάδοση

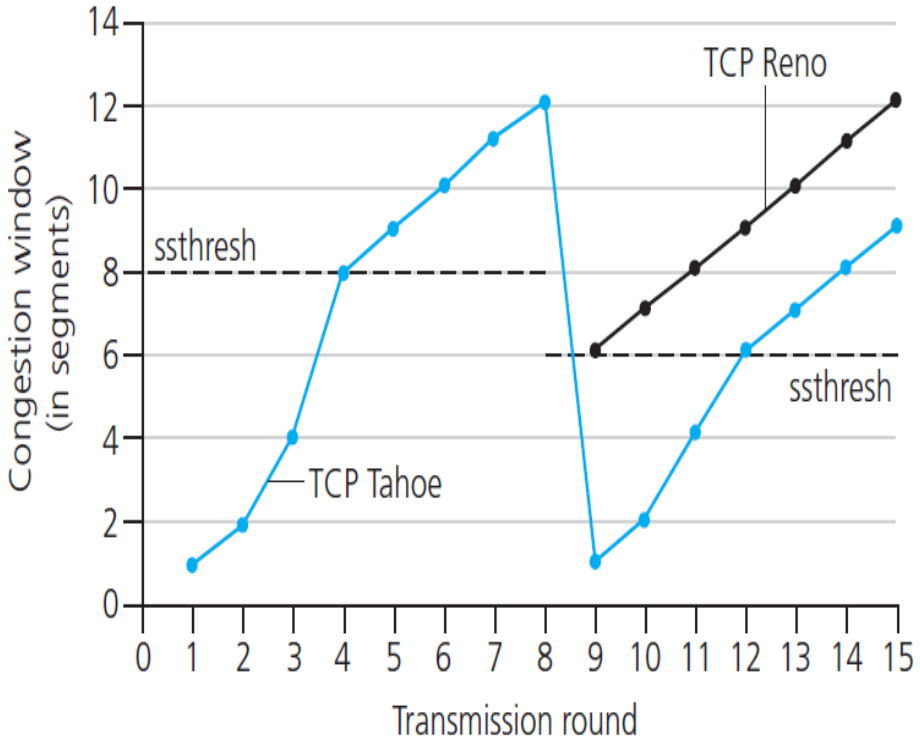
- Διπλότυπα ACKs
  - Τρία διπλότυπα μέχρι επαναμετάδοση
- Σωρευτική επιβεβαίωση αριθμού μηνυμάτων
  - Και όχι ενός μόνο



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 [1]
```



1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs

TCP New Reno

- Όταν βρισκόμαστε σε κατάσταση **slow-start,** σε κάθε λήψη νέου ACK θέτουμε cwnd=cwnd+1 (οπότε προκύπτει διπλασιασμός της τιμής του cwnd σε κάθε RTT - Round Trip Time, που αφορά στην επιτυχή μετάδοση ολοκλήρου του τρέχοντος παραθύρου cwnd - γύρου μετάδοσης).
- Αν cwnd ≥ Threshold, μεταβαίνουμε στην κατάσταση **congestion avoidance**, κατά την οποία σε κάθε λήψη νέου ACK, το cwnd αυξάνεται κατά 1/cwnd (οπότε σε κάθε RTT το cwnd αυξάνεται κατά 1). Η αύξηση αυτή του cwnd δείχνει ότι μπορεί το cwnd να είναι (για χρόνο< RTT) μη ακέραιος αριθμός.
- Αν η κατάσταση του TCP Reno είναι **slow-start** ή **congestion avoidance** και συμβεί timeout (λήξη χρόνου αναμονής ACK), μεταβαίνουμε σε κατάσταση **slow-start,** θέτοντας Threshold = cwnd/2 και cwnd = 1.
- Αν η κατάσταση είναι **slow-start** ή **congestion avoidance** και συμβεί λήψη διπλότυπου (duplicate - πανομοιότυπου) ACK τρεις συνεχόμενες φορές, θεωρούμε ότι εχάθη το segment, το οποίο και μεταδίδομε ξανά, θέτοντας Threshold = cwnd/2 και cwnd = Threshold+3. Ακολούθως, μεταβαίνουμε σε κατάσταση **fast retransmission / fast recovery**, στην οποία αν λάβουμε διπλότυπο ACK, θέτουμε cwnd=cwnd+1 και μεταδίδουμε segment αν επιτρέπεται από το μέγεθος του cwnd. (Αν cwnd/2 δεν είναι ακέραιος, στρογγυλοποιείται, συνήθως προς τα πάνω).
- Ευρισκόμενοι σε κατάσταση **fast recovery,** αν λάβουμε **νέο ACK** (όχι διπλότυπο), μεταβαίνουμε σε κατάσταση **congestion avoidance** θέτοντας **cwnd=Threshold**, ενώ αν συμβεί **timeout**, μεταβαίνουμε στην κατάσταση **slow-start,** θέτοντας **cwnd=1**.

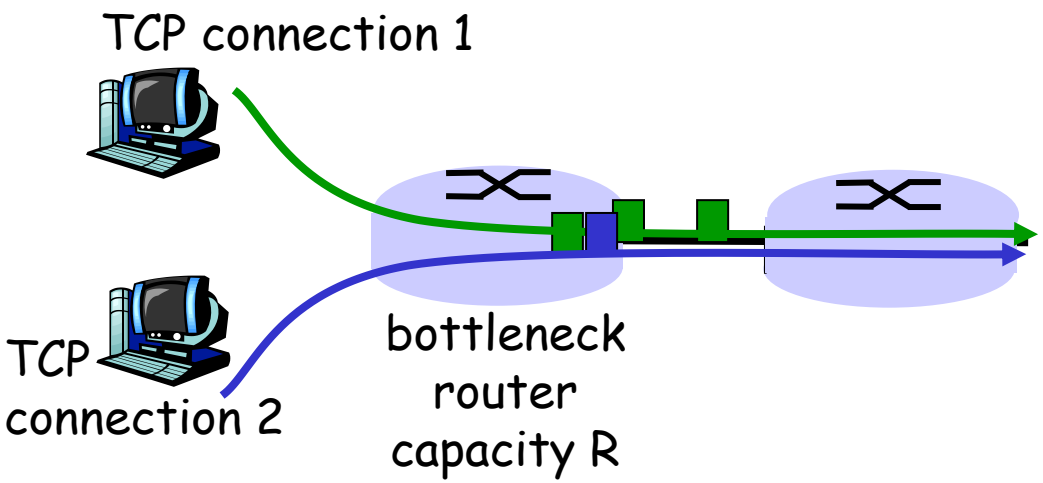
AIMD

TCP congestion avoidance:

- AIMD: *additive increase, multiplicative decrease*
  - increase window by 1 per RTT
  - decrease window by factor of 2 on loss event

TCP Fairness

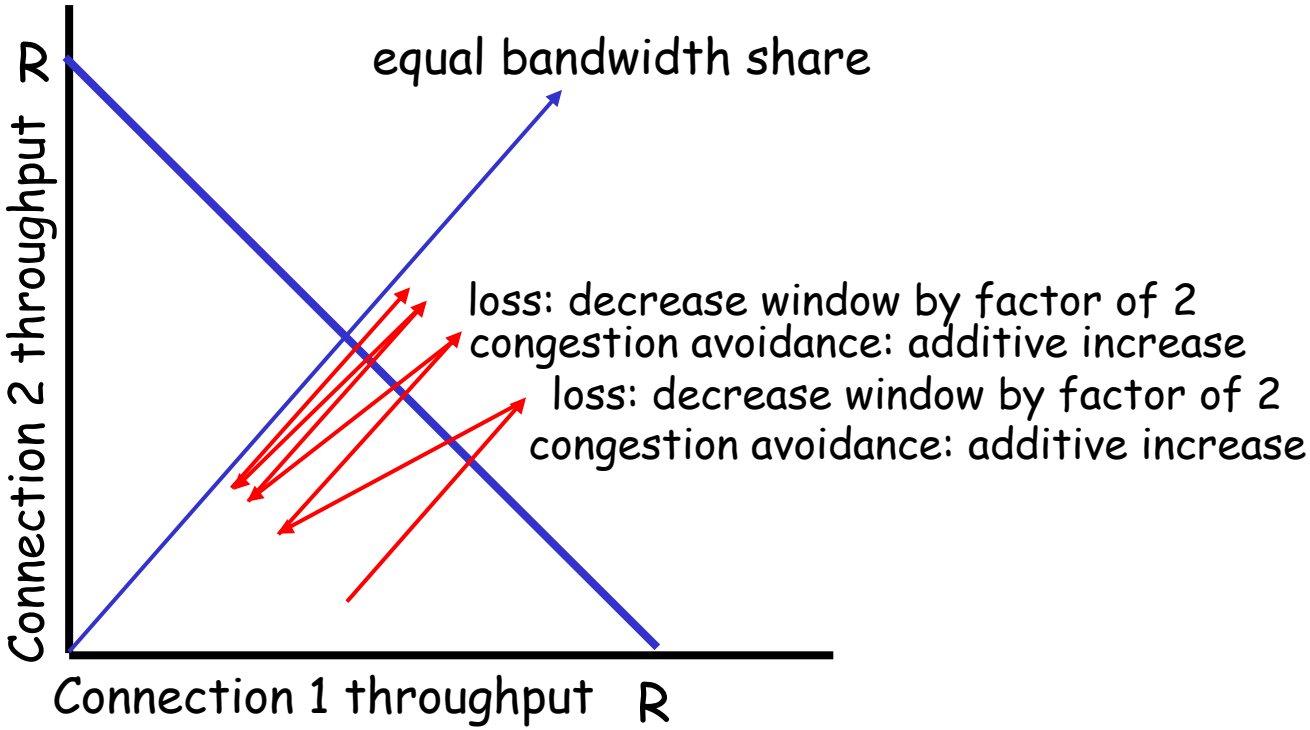
Fairness goal: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



TCP latency modeling

Q: How long does it take to receive an object from a Web server after sending a request?

- TCP connection establishment
- data transfer delay

Notation, assumptions:

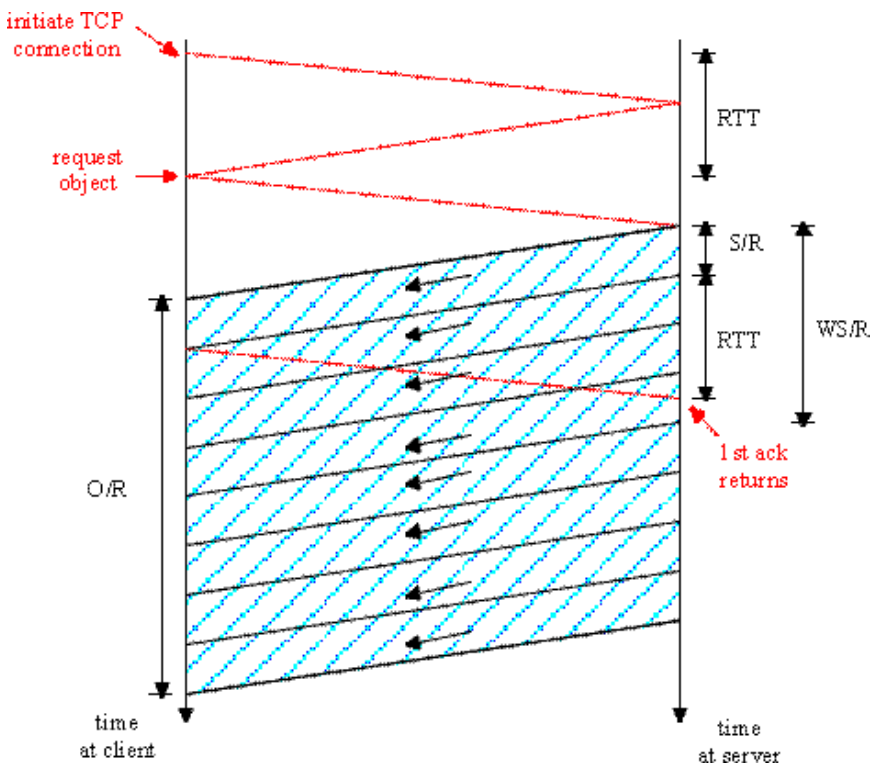
- Assume one link between client and server of rate R
- Assume: fixed congestion window, W segments
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

Two cases to consider:

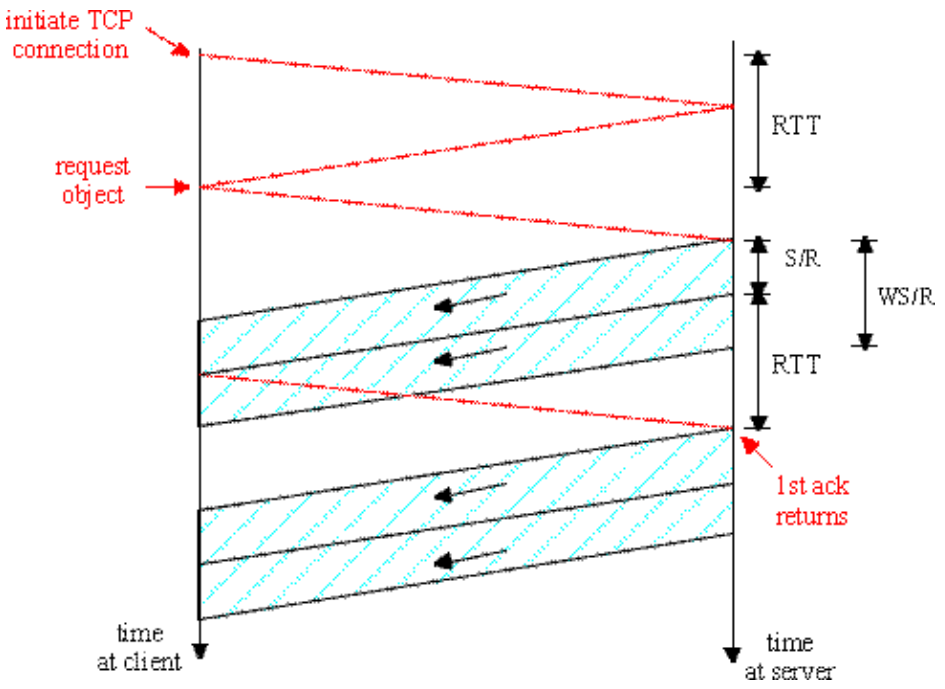
- WS/R > RTT + S/R: ACK for first segment in window returns before window's worth of data sent
- WS/R < RTT + S/R: wait for ACK after sending window's worth of data sent

TCP latency Modeling

K:= O/WS



Case 1: latency = 2RTT + O/R



Case 2: latency = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]

TCP Latency Modeling: Slow Start

- Now suppose window grows according to slow start.
- Will show that the latency of one object of size O is:

$$Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

where *P* is the number of times TCP stalls at server:

$$P = \min\{Q, K-1\}$$

- where Q is the number of times the server would stall if the object were of infinite size.

- and K is the number of windows that cover the object.

TCP Latency Modeling: Slow Start (cont.)

Example:

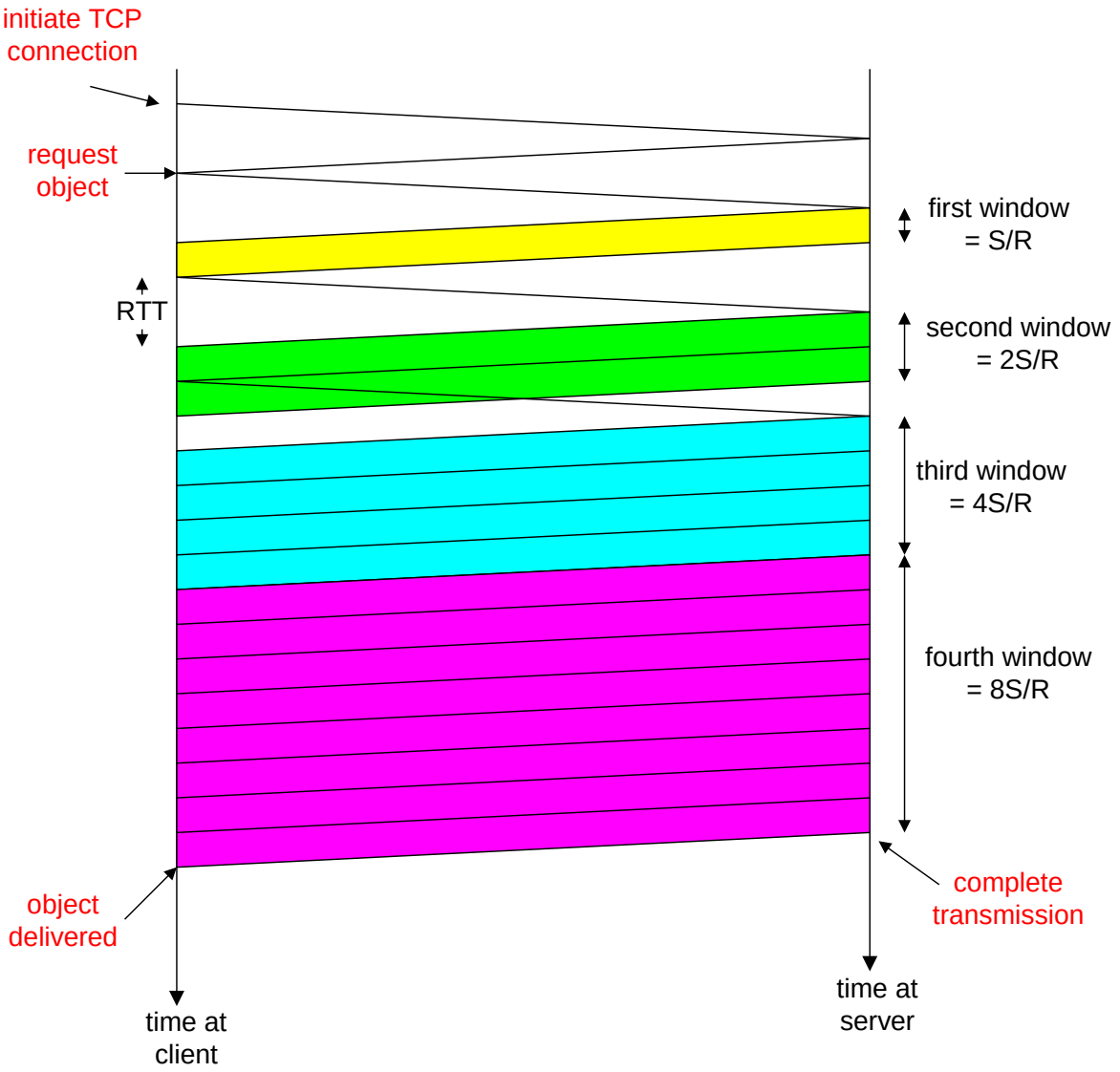
O/S = 15 segments

K = 4 windows

Q = 2

P = min{K-1,Q} = 2

Server stalls P=2 times.



TCP Latency Modeling: Slow Start (cont.)

$$\frac{S}{R} + RTT = \text{time from when server starts to send segment}$$

until server receives acknowledgement

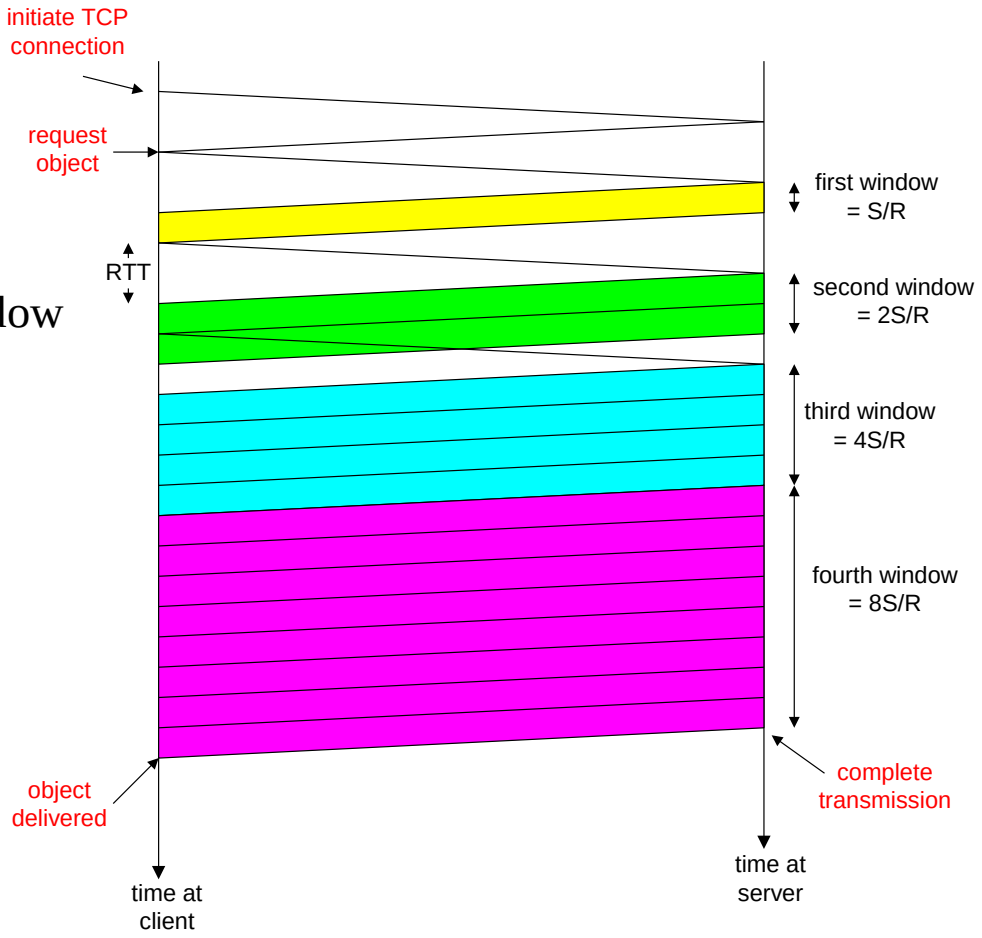
$$2^{k-1}\frac{S}{R} = \text{time to transmit the kth window}$$

$$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} = \text{stall time after the } k\text{th window}$$

$$\text{latency} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} stallTime_p$$

$$= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]$$

$$= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^{P} - 1)\frac{S}{R}$$



Παράδειγμα 1 TCP Tahoe - TCP Reno

Στο διάγραμμα του παραθύρου μετάδοσης που φαίνεται στο σχήμα που ακολουθεί:

α) Πόσοι Γύροι Μετάδοσης περιλαμβάνει η φάση "slow start"; και ποιο είναι το Threshold;

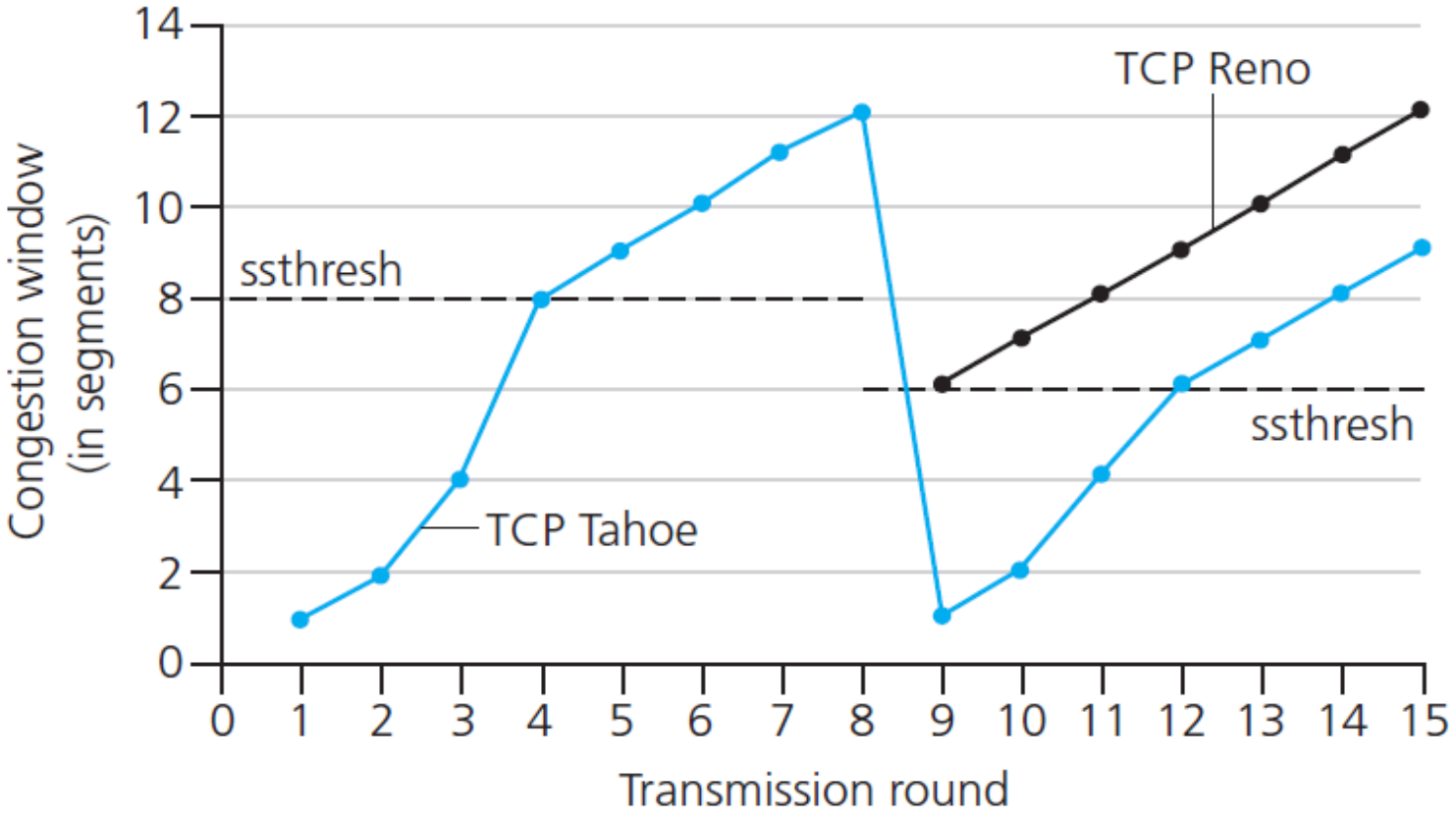
β) Ποιοι Γύροι Μετάδοσης ανήκουν στην φάση "Congestion Avoidance" του TCP Tahoe;

γ) Τι συνέβη μετά τον 8ο γύρο μετάδοσης;

δ) Αν είχαμε το πρωτόκολλο TCP New Reno, αντί του TCP Reno, τι θα άλλαζε στο διάγραμμα;

ε) Ποιο είναι το Throughput (διεκπεραιωτική ικανότητα) της μετάδοσης του TCP Tahoe και του TCP Reno, μέχρι και τον 14 Γύρο Μετάδοσης; Υποθέστε ότι κάθε segment είναι 10 bytes, και ότι 1 Γύρος Μετάδοσης = 1 sec, Timeout = 2 sec.

Παράδειγμα 1 TCP Tahoe - TCP Reno (συνέχεια)



Απαντήσεις Παραδείγματος 1

α) Η φάση «slow start» περιλαμβάνει τους γύρους **1 έως 4**, όπου Threshold = ssthresh = **8**, καθώς και τους γύρους **9 έως 12**, όπου Threshold = ssthresh = **6**.

β) Στην φάση "Congestion Avoidance" του TCP Tahoe ανήκουν οι γύροι μετάδοσης από **4 μέχρι 8**, και από **12 μέχρι 15**.

γ) Μετά τον 8ο γύρο μετάδοσης συνέβη λήψη τριών διπλότυπων πακέτων, οπότε το μεν TCP Tahoe εισέρχεται σε φάση Slow Start, το δε TCP Reno, σε φάση fast retransmission – fast recovery.

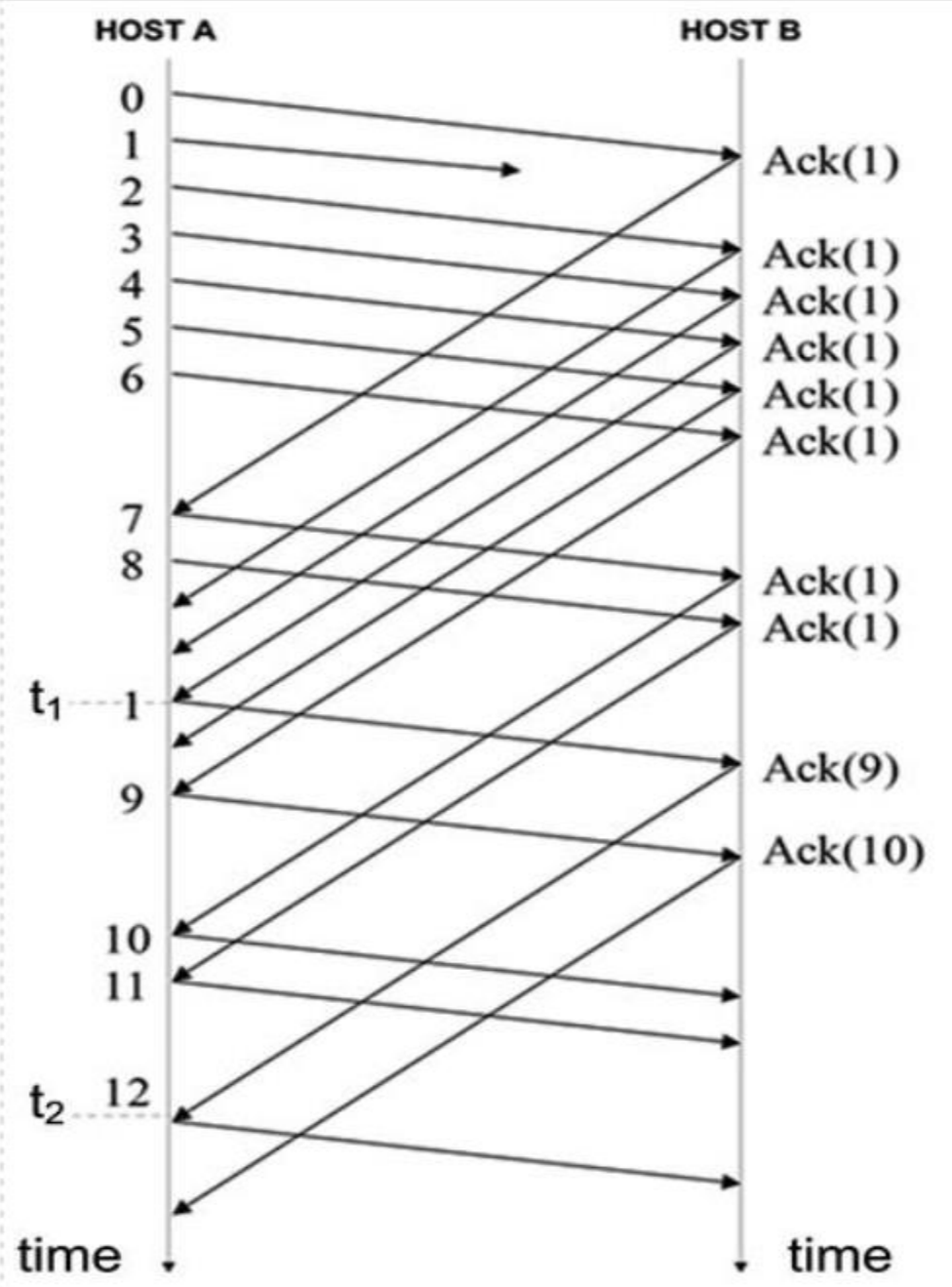
δ) Αν είχαμε το πρωτόκολλο TCP New Reno, αντί του TCP Reno, στον γύρο μετάδοσης 9, στο διάγραμμα, θα είχαμε congestion window **9** (αντί 6).

ε) TCP Tahoe: Μέχρι και τον 14 Γύρο Μετάδοσης έχουν μεταδοθεί 85 segments δηλ. 850 bytes εντός 14 s, οπότε **Throughput = 60,7 bytes/s**

TCP Reno: Μέχρι και τον 14 Γύρο Μετάδοσης έχουν μεταδοθεί 108 segments δηλ. 1080 bytes εντός 14 s, οπότε **Throughput = 77,14 bytes/s**

Παράδειγμα 2

Στο σχήμα φαίνεται η διαδοχή των TCP μηνυμάτων μεταξύ του HOST A και του HOST B, αμέσως μετά από την εγκατάσταση μιας σύνδεσης TCP (φάση slow-start, όπου το πρώτο cwnd δεν είναι μονάδα), για το πρωτόκολλο **TCP New Reno**. Οι αριθμοί 0, 1, 2, κλπ. στον άξονα χρόνου του HOST A, δείχνουν την τιμή του πεδίου Sequence Number του segment που στέλνεται προς τον HOST B. Κάθε φορά στέλνεται 1 byte, γι' αυτό και ο Sequence Number αυξάνει κατά 1 (χάριν παραδείγματος 1 MSS = 1 byte). Στον άξονα χρόνου του HOST B, δείχνεται η τιμή του πεδίου Acknowledgement του segment που στέλνει ως απάντηση ο HOST B προς τον HOST A.



Σημειωτέον ότι δεν δείχνεται πουθενά στο σχήμα ότι συνέβη timeout κατά την λήψη ACK.

Παράδειγμα 2 (συνέχεια)

1	Ποια είναι η αρχική τιμή του παραθύρου μετάδοσης (cwnd);
2	Με βάση το διάγραμμα του σχήματος, ποια είναι η τελευταία τιμή του cwnd και του Threshold;
3	Πότε ακριβώς η τιμή του cwnd θα γίνει 8 και πότε 9; (Αν η ίδια τιμή 8 ή 9 τεθεί στο cwnd περισσότερο από μία φορές, να εντοπισθεί πότε γίνεται αυτό.)
4	Ποια είναι η τιμή του cwnd και του Threshold τις χρονικές στιγμές t_1 (δηλ. κατά την επαναμετάδοση του segment με Sequence Number 1), και t_2 (δηλ. μετά την λήψη από τον HOST B segment με Acknowledgement Number 9);
5	Πότε το πρωτόκολλο TCP New Reno εξέρχεται από την αρχική κατάσταση slow-start; Σε ποια κατάσταση εισέρχεται τότε; Ποια είναι η τελική κατάσταση του πρωτοκόλλου και πότε εισέρχεται σ΄αυτή;

Λύση Έλεγχος συμφόρησης με το πρωτόκολλο TCP New Reno

- Η αρχική τιμή του παραθύρου μετάδοσης (cwnd) είναι **7**, αφού μετά από επτά μεταδόσεις ξαναρχίζει μετάδοση μετά από λήψη ACK.
- Μόλις λάβει το πρώτο ACK (**Ack(1)**) από τον HOST B, ο HOST A θα βάλει το cwnd = 8. Με cwnd = 8 θα κάνει δύο μεταδόσεις ακόμη όπως δείχνει το σχήμα, μεταδίδοντας segments με seq. #7 και seq. #8. Ακολούθως πρέπει να περιμένει ACK (για τα segments #1, #2, #3, #4, #5, #6, #7, #8). Λαμβάνει τρία ACK διπλότυπα (βάσει του σχήματος) και εισέρχεται στην φάση Fast Retransmit (Επαναμεταδίδει το seq. #1), θέτοντας το Threshold = 8/2 = 4 και το cwnd = Threshold + 3, δηλ. cwnd = 7, για να αρχίσει η φάση Fast Recovery. Στην φάση Fast Recovery, ο HOST A θεωρεί ότι όλα βαίνουν καλώς και κάθε φορά που θα λάβει ACK (διπλότυπο) αυξάνει το cwnd κατά 1. Έτσι μετά την επαναμετάδοση του seq. #1 μόλις λάβει το **Ack(1)** το cwnd θα γίνει από 7, cwnd = 8. Εν συνεχεία λαμβάνει κι άλλο Ack(1) οπότε cwnd = 9.
- Το πρωτόκολλο εξέρχεται από την κατάσταση Slow Start μόλις λάβει το τρίτο διπλότυπο Ack (δηλ. τέταρτο κατά σειράν Ack(1)), οπότε περνά στην Fast Retransmit και Fast Recovery. Βγαίνει από την Fast Recovery μόλις λάβει το πρώτο μη διπλότυπο ACK, δηλ. το Ack(9), οπότε εισέρχεται στην κατάσταση (φάση) Congestion Avoidance, η οποία είναι η τελική κατάσταση του πρωτοκόλλου στο σχήμα αυτό.
- Κατά την επαναμετάδοση του seq. #1, την χρονική στιγμή t_1, έχουμε Threshold = 4 και cwnd = 7 (TCP New Reno, Fast Retransmit). Την χρονική στιγμή t_2, δηλ. όταν το TCP New Reno εξέρχεται από την κατάσταση Fast Recovery, έχουμε Threshold = 4 και cwnd = Threshold = 4.
- Στο τέλος του διαγράμματος, το TCP New Reno βρίσκεται στην φάση Congestion Avoidance (αποφυγής συμφόρησης), οπότε η τελευταία τιμή του Threshold είναι 4 (όπως έχει εξηγηθεί στις προηγούμενες ερωτήσεις). Η τελευταία τιμή του cwnd = 4,25 που τίθεται μόλις ο HOST A έλαβε το Ack(10). Σημειωτέον ότι στην φάση της αποφυγής συμφόρησης (Congestion Avoidance) το παράθυρο συμφόρησης cwnd αυξάνει κατά 1/cwnd.

Chapter 3: Summary

- principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

- leaving the network "edge" (application transport layer)
- into the network "core"