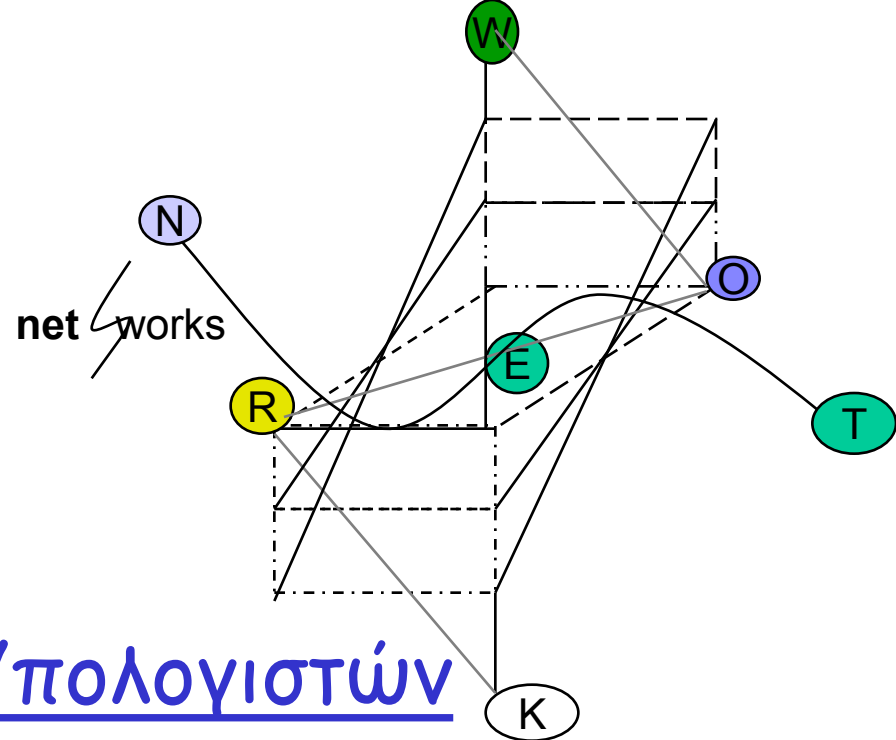




ΗΥ-335 : Δίκτυα Υπολογιστών

TCP

Μαρία Παπαδοπούλη

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

TCP

Στόχος μας είναι η κατανόηση των:

Θεμελιωδών αρχών του TCP:

- Αξιόπιστη μεταφορά δεδομένων
- Έλεγχος ροής
- Έλεγχος συμφόρησης

Διαδικτυακά πρωτόκολλα επιπέδου μεταφοράς

Αναξιόπιστη, χωρίς εγγύηση στη σειρά παράδοσης των πακέτων: UDP

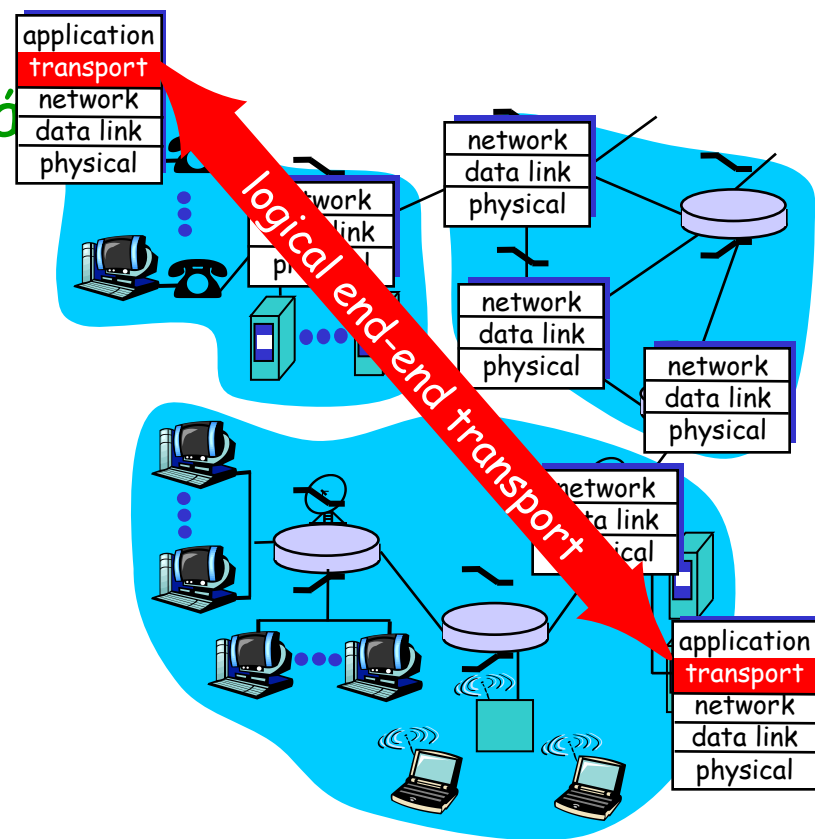
- Δε βελτιώνει τον "best-effort" χαρακτήρα του IP.

Αξιόπιστη, με εγγύηση στη σειρά παράδοσης των πακέτων: TCP

- Έλεγχος συμφόρησης
- Έλεγχος ροής
- Εγκαθίδρυση σύνδεσης

Μη διαθέσιμες υπηρεσίες :

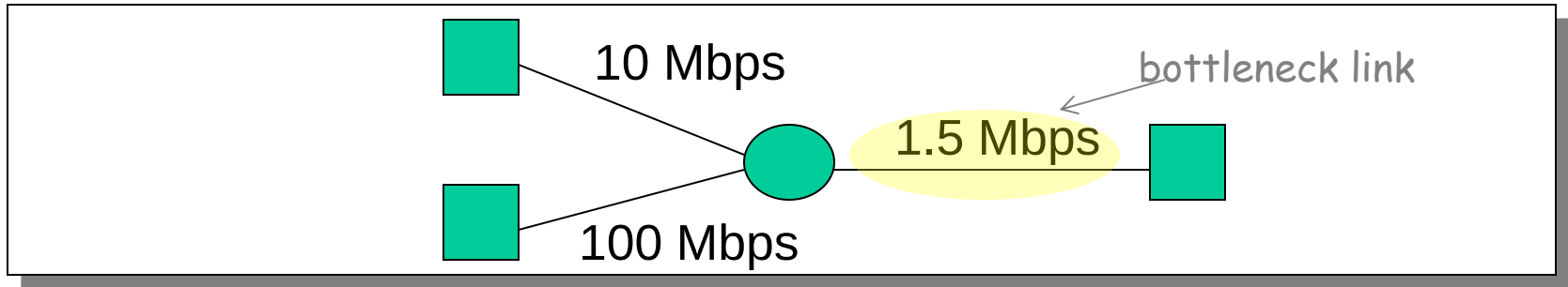
- Εγγυήσεις καθυστέρησης
- Εγγυήσεις bandwidth



TCP

- ☞ Το TCP socket χαρακτηρίζεται από 4 πεδία:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- ☞ Ο παραλήπτης χρησιμοποιεί και τα 4 πεδία για να προωθήσει το segment στο κατάλληλο socket
- Ένας server host μπορεί να υποστηρίξει πολλαπλά ταυτόχρονα ("παράλληλα στον χρόνο") TCP sockets:
 - ☞ κάθε socket χαρακτηρίζεται από τη δική του 4-άδα
- Οι Web servers έχουν διαφορετικά sockets για κάθε client που συνδέεται
 - non-persistent HTTP έχουν διαφορετικά sockets για κάθε αίτημα

Συμφόρηση (congestion)



Τα sources ανταγωνίζονται για τους πόρους του δικτύου, αλλά

- δεν έχουν γνώση των πόρων του δικτύου (state of resource)
- δεν ξέρουν την ύπαρξη η μία της άλλης

Με αποτέλεσμα:

- Πακέτα να χάνονται (λόγω buffer overflow στους δρομολογητές)
- Μεγάλες καθυστερήσεις (αναμονή στις ουρές των buffers στους δρομολογητές)
- throughput μικρότερο από το bottleneck link (1.5Mbps για την παραπάνω τοπολογία) → κατάρρευση λόγω συμφόρησης

Κατάρρευση λόγω συμφόρησης (Congestion Collapse)

Ορισμός: Αύξηση στο φόρτο δικτύου οδηγεί σε μείωση της χρήσιμης δουλειάς που γίνεται

❖ Πολλές πιθανές αιτίες

- “Περιττές” (άκαιρες) επαναμεταδόσεις πακέτων βρίσκονται ακόμα σε εξέλιξη
 - Κλασική κατάρρευση λόγω συμφόρησης
 - Πώς μπορεί να συμβεί αυτό με τη διατήρηση των πακέτων
Λύση: καλύτεροι timers και TCP έλεγχος συμφόρησης
- Μη παραδοθέντα πακέτα
 - Τα πακέτα καταναλώνουν πόρους και γίνονται drop κάπου αλλού στο δίκτυο
Λύση: έλεγχος συμφόρησης για ΌΛΗ την κίνηση

TCP: Επισκόπηση

RFCs: 793, 1122, 1323, 2018, 2581

Σημείο-προς-σημείο:

- Ένας αποστολέας, ένας παραλήπτης (σε αντίθεση με το multicasting)

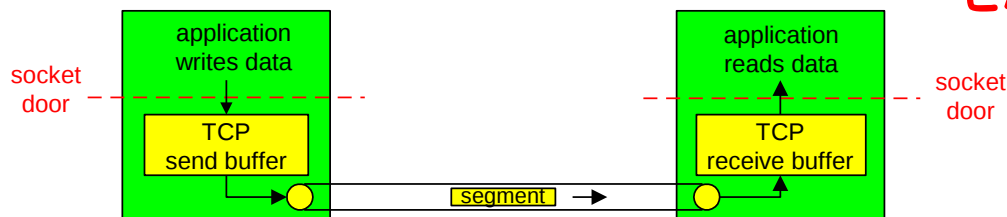
Αξιόπιστο, σε σειρά ροή των *byte*:

- Η εφαρμογή από επάνω θα "παραλάβει" τα πακέτα στη σωστή σειρά

Pipelined:

- Ο TCP έλεγχος συμφόρησης & ροής θέτουν το μέγεθος παραθύρου
- Πολλά πακέτα μπορούν να έχουν σταλθεί παράλληλα και να μην έχουν γίνει ACKed

Buffers αποστολής & παραλαβής



Πλήρως αμφίδρομα δεδομένα:

- Ροή δεδομένων και προς τις δύο κατευθύνσεις στην ίδια σύνδεση
- MSS: maximum segment size (μέγιστο μέγεθος του segment)

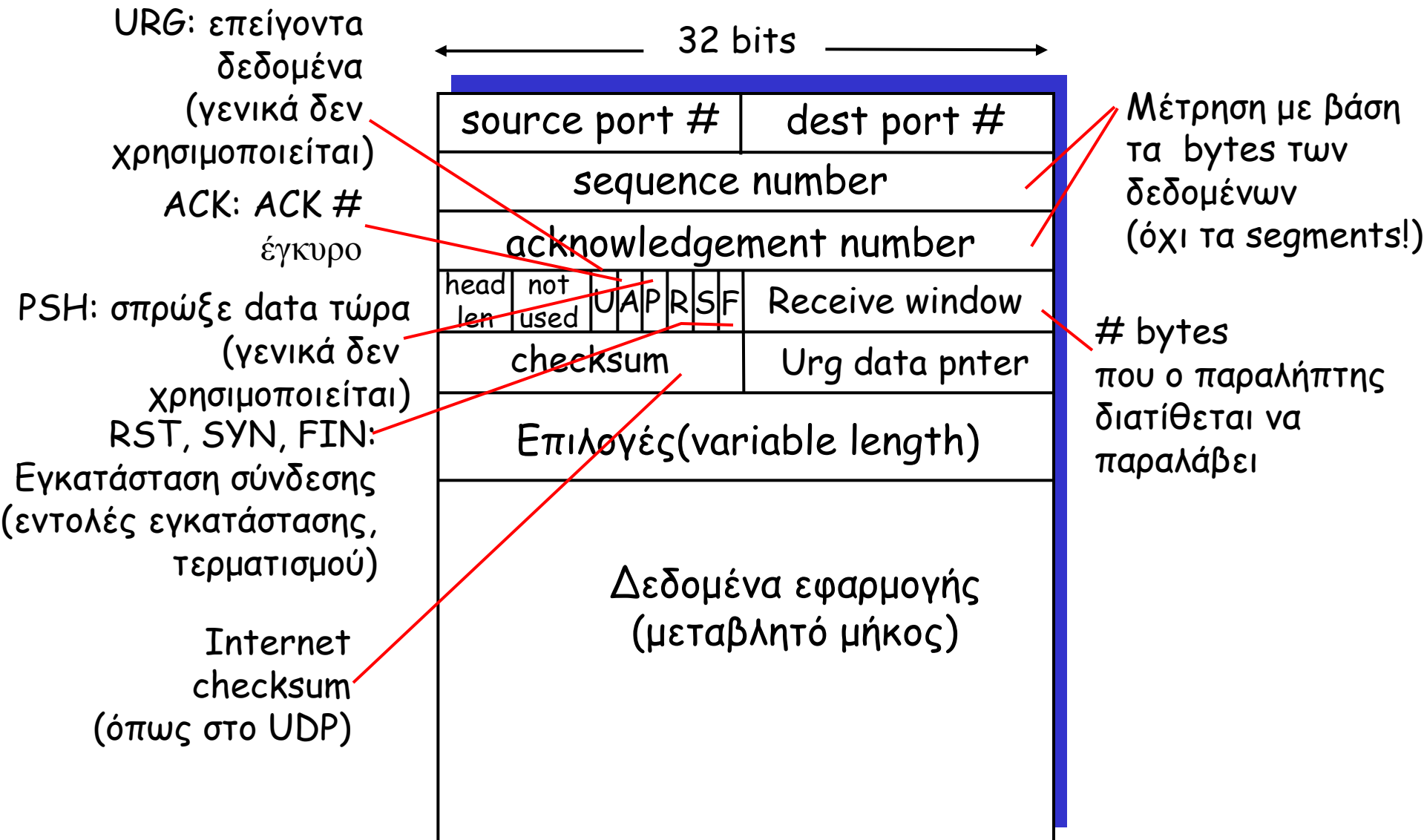
Συνδεδειστροφές:

- χειραψία (ανταλλαγή μηνυμάτων ελέγχου) αρχικοποιούν την κατάσταση του αποστολέα και του παραλήπτη πριν την ανταλλαγή δεδομένων

Ελεγχόμενη ροή:

- Ο αποστολέας δεν θα «κατακλύσει» τον παραλήπτη

Δομή TCP segment



Sequence Number Space

Το TCP βλέπει τα δεδομένα σαν μια ροή δεδομένων, σειρά από bytes

Ο αποστολέας στέλνει τα πακέτα σύμφωνα με τη σειρά αυτή

Ο παραλήπτης θα προσπαθήσει να την "ξαναδημιουργήσει"

☞ Κάθε byte στη ροή των bytes είναι αριθμημένο

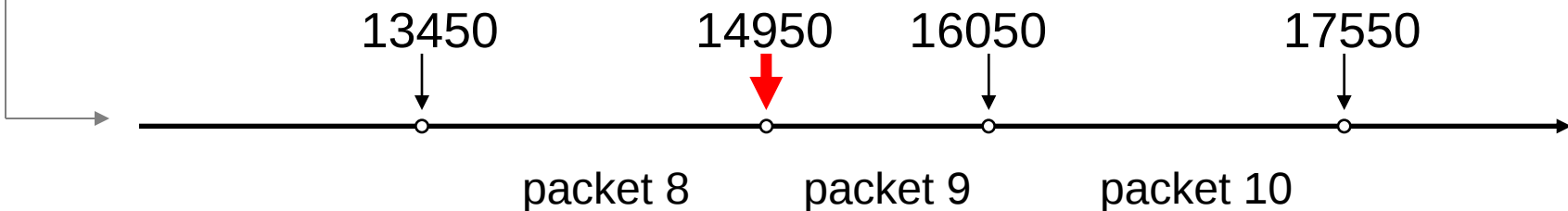
- 32 bit τιμή
- Κάνει wrap around
- Η αρχική τιμή επιλέγεται τη στιγμή εκκίνησης

Το TCP διασπάει τη ροή δεδομένων σε πακέτα

- Το μέγεθος πακέτου περιορίζεται από το μέγιστο μέγεθος segment (MSS)

Κάθε πακέτο έχει ένα **sequence number** (αριθμό σειράς)

Προσδιορίζει που βρίσκεται στη ροή δεδομένων



TCP σύνδεση: χειραψία σε 3 βήματα

Βήμα 1: ο client host **στέλνει το TCP SYN segment** στον server

- Προσδιορίζει τον αρχικό αριθμό σειράς (seq #)
- Καθόλου δεδομένα

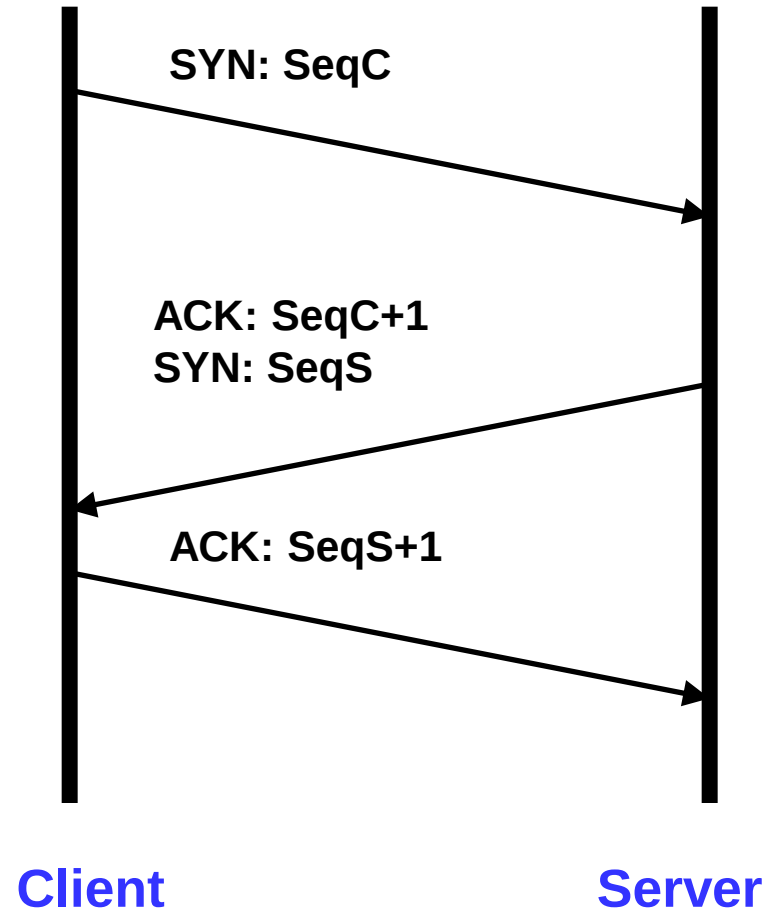
Βήμα 2: ο server host λαμβάνει το SYN, **απαντάει με SYNACK segment**

- Ο server δεσμεύει buffers
- Προσδιορίζει τον αρχικό αριθμό σειράς

Βήμα 3: ο client λαμβάνει SYNACK, **απαντάει με ACK segment**, που μπορεί να περιέχει και δεδομένα

Εγκαθίδρυση σύνδεσης: χειραψία σε 3 βήματα

- Κάθε πλευρά ειδοποιεί την άλλη για τον αρχικό αριθμό σειράς (*seq #*) που θα χρησιμοποιήσει για την αποστολή
 - Γιατί να μην επιλέξουμε απλά το 0;
 - Πρέπει να αποφύγει την επικάλυψη με προηγούμενο πακέτο
 - Θέματα ασφάλειας
- Κάθε πλευρά επιβεβαιώνει τον αριθμό σειράς της άλλης
SYN-ACK: αριθμός σειράς επιβεβαίωσης + 1
- Μπορεί να συνδυάσει το δεύτερο SYN με το πρώτο ACK



Διαχείριση TCP σύνδεσης

Υπενθύμιση:

Οι TCP sender & receiver **εγκαθιδρύουν σύνδεση πριν ανταλλάξουν segments δεδομένων**

αρχικοποίηση μεταβλητών του TCP :

- seq. #s
- **buffers, πληροφορίες ελέγχου ροής** (e.g. RcvWindow)

client: ξεκινάει τη σύνδεση

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

server: αποδέχεται επικοινωνία από τον client

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

Βήμα 1: ο client host στέλνει το TCP SYN segment στον server

- Προσδιορίζει τον αρχικό αριθμό σειράς (seq #)
- **καθόλου δεδομένα**

Βήμα 2: ο server host λαμβάνει το SYN, απαντάει με SYNACK segment

- **Ο server δεσμεύει buffers**
- Προσδιορίζει τον αρχικό αριθμό σειράς

Βήμα 3: ο client λαμβάνει SYNACK, απαντάει με ACK segment, που μπορεί να περιέχει και δεδομένα

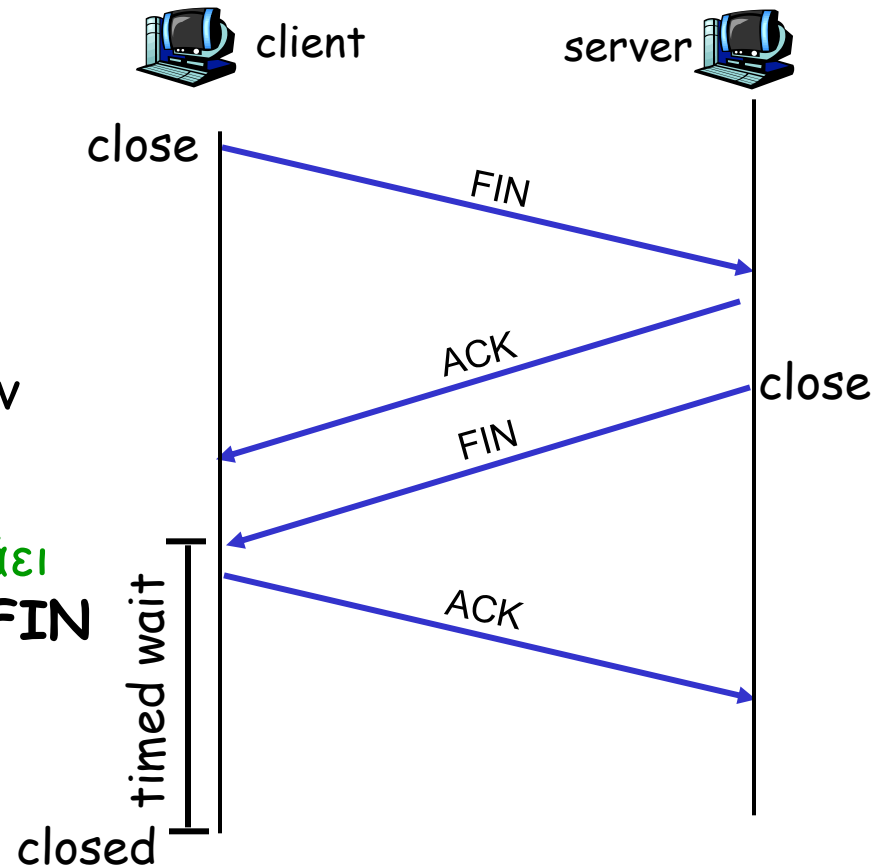
Διαχείριση TCP σύνδεσης(συνέχεια)

Κλείσιμο μίας σύνδεσης:

Ο client κλείνει το socket:
`clientSocket.close()` ;

Βήμα 1: το τερματικό σύστημα του **client**
στέλνει **TCP FIN** segment ελέγχου στον
server

Βήμα 2: ο **server** λαμβάνει το FIN, απαντάει
με **ACK**. Κλείνει την σύνδεση, στέλνει **FIN**



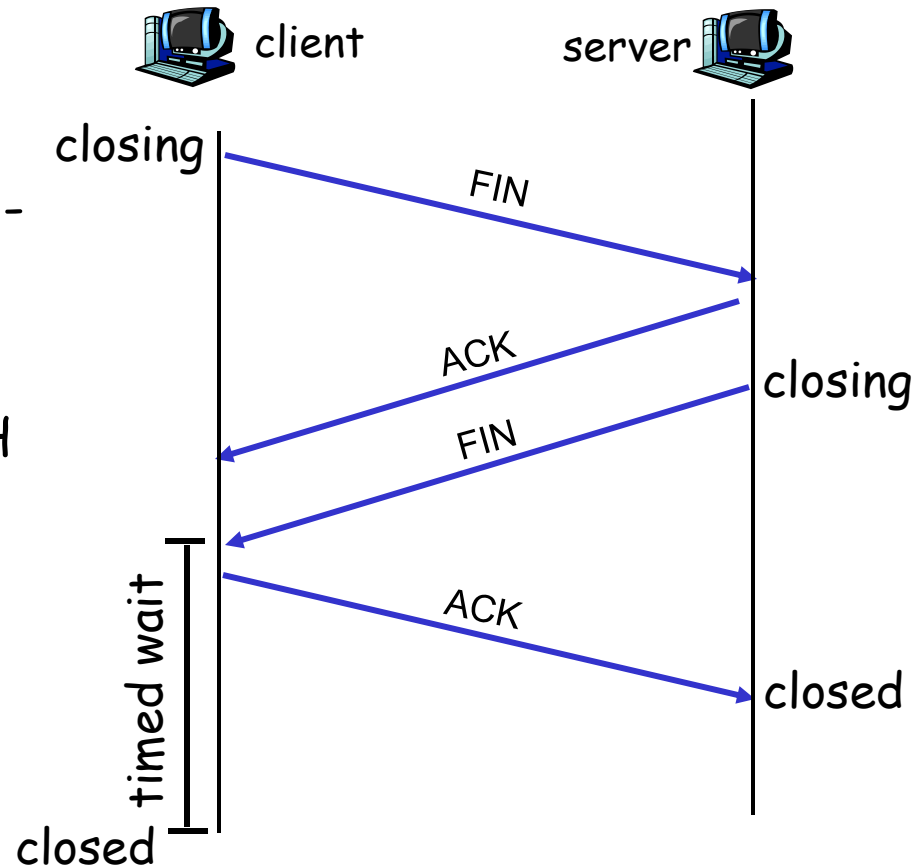
Διαχείριση TCP σύνδεσης(συνέχεια)

Βήμα 3: ο **client** λαμβάνει το FIN, απαντάει με ACK

Μπαίνει σε χρονισμένη αναμονή -
θα απαντήσει με ACK στα FINs που λαμβάνει

Βήμα 4: ο **server**, λαμβάνει το ACK. Η σύνδεση έκλεισε.

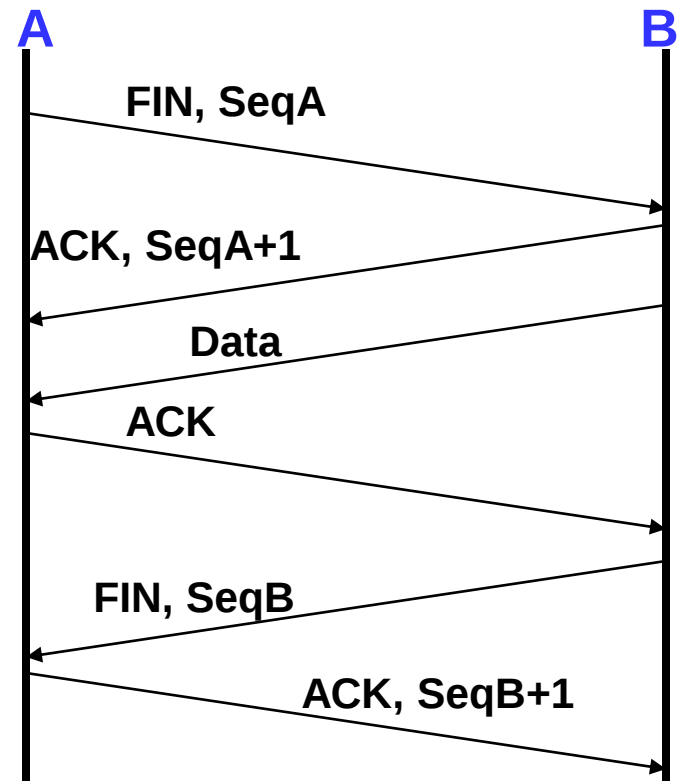
Σημείωση: με μικρές μετατροπές, μπορεί να γίνει διαχείριση ταυτόχρονων FINs.



Κλείσιμο σύνδεσης

Οποιαδήποτε πλευρά μπορεί να ξεκινήσει το κλείσιμο της σύνδεσης

- Στέλνει FIN σήμα
- "Δε θα στείλω άλλα δεδομένα"
- Η άλλη πλευρά μπορεί να συνεχίσει να στέλνει δεδομένα
 - «Ημι-ανοιχτή» σύνδεση
 - Πρέπει να συνεχίσει να επιβεβαιώνει
- Επιβεβαίωση του FIN
 - Επιβεβαίωση με sequence number + 1



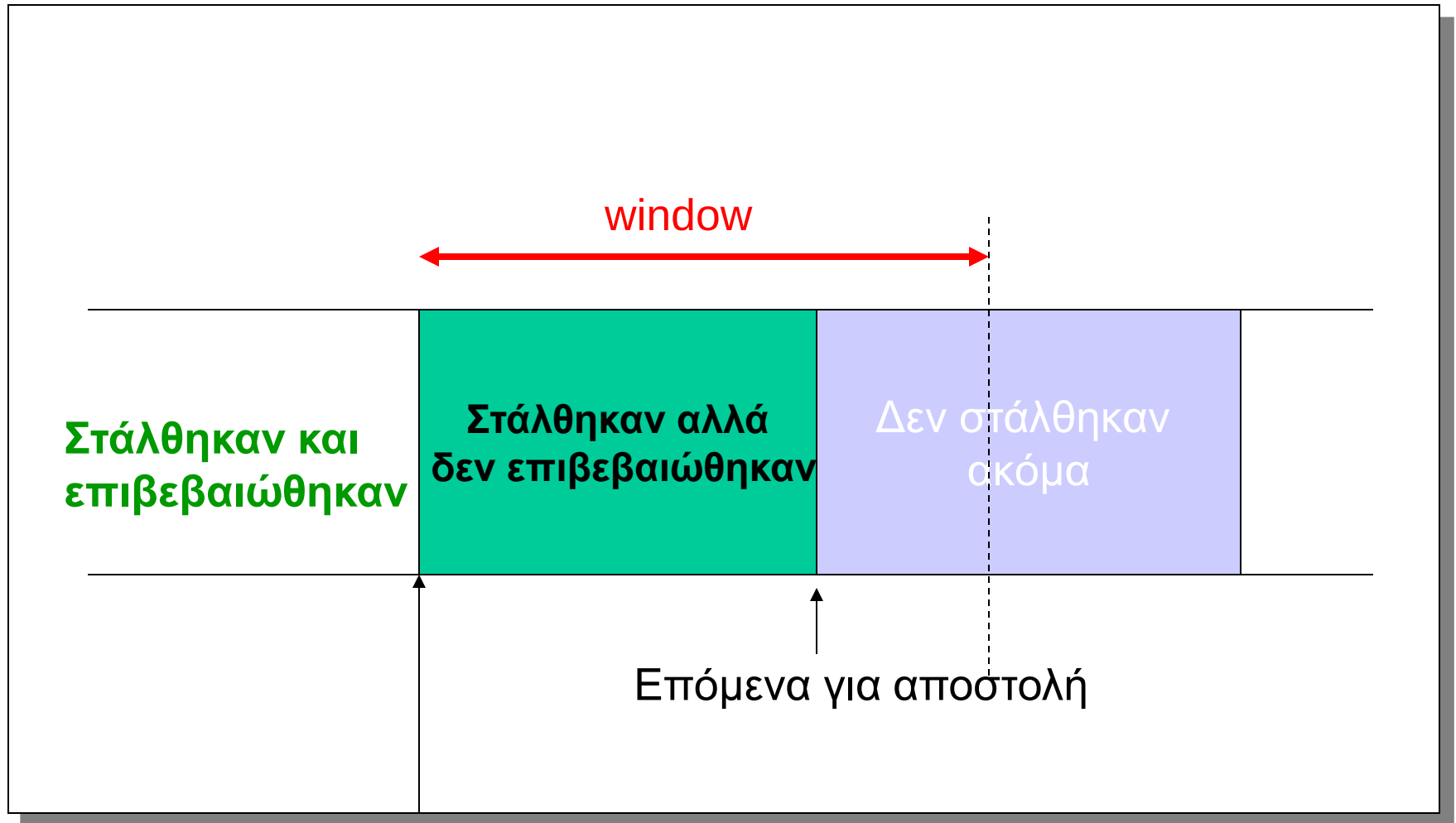
Υπολογισμός του Round-trip time & του timeout

- Μηχανισμός που καθορίζει πόσο να περιμένει ο αποστολέας μέχρι να ξαναστείλει το πακέτο
 - Ο timer (εάν ήδη δεν "τρέχει" για κάποιο άλλο segment) ξεκινά όταν το segment "παραδίδεται" στο IP επίπεδο
 - Όταν ο timer λήξει, το segment ξαναστέλνεται και το TCP ξεκινά ξανά τον timer
- ☞ Το TCP του sender διατηρεί πληροφορία για το **παλιότερο unacknowledged byte**

Έλεγχος ροής του TCP

- ☞ Το TCP είναι ένα πρωτόκολλο κυλιόμενου παραθύρου (**sliding window**)
 - Για μέγεθος παραθύρου **n** , μπορεί να στείλει έως και **n bytes** χωρίς να λάβει επιβεβαίωση
 - Όταν τα δεδομένα επιβεβαιωθούν τότε το παράθυρο μετακινείται προς τα **μπρος**
- Κάθε πακέτο δημοσιεύει ένα μέγεθος παραθύρου
Προσδιορίζει τον **αριθμό των bytes** για τα οποία έχει χώρο
- Το original TCP στέλνει πάντα ολόκληρο το παράθυρο
Ο έλεγχος συμφόρησης τώρα το περιορίζει αυτό

Έλεγχος ροής με παράθυρο: αποστέλλουσα πλευρά



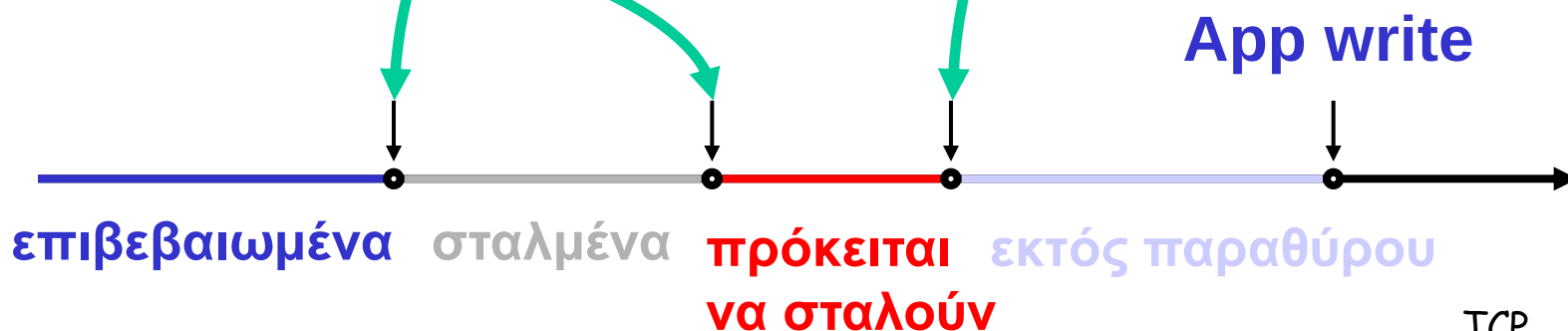
Έλεγχος ροής με παράθυρο: αποστέλουσα πλευρά

Packet Sent

Source Port	Dest. Port
Sequence Number	
Acknowledgment	
HL/Flags	Window
D. Checksum	Urgent Pointer
Options...	

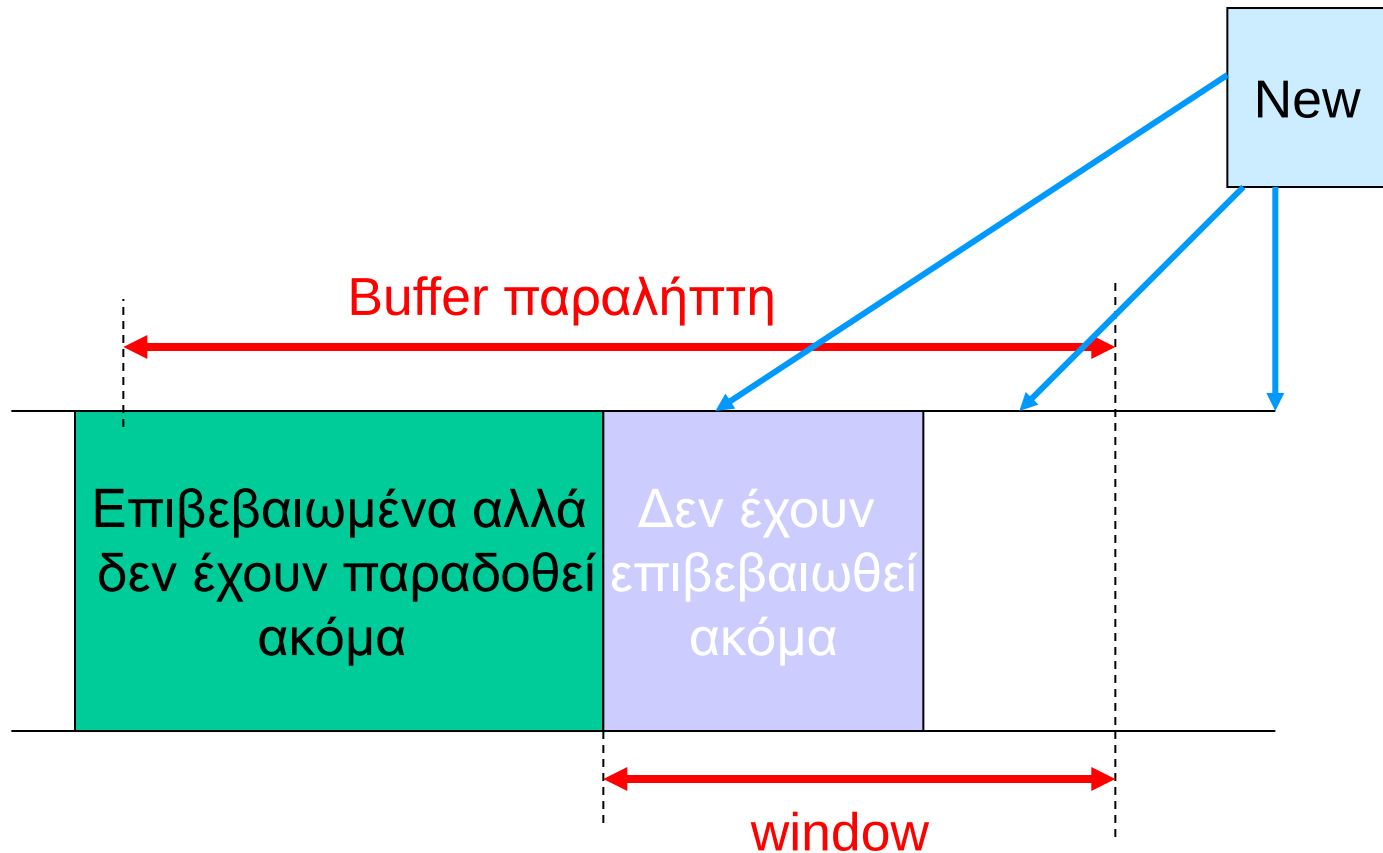
Packet Received

Source Port	Dest. Port
Sequence Number	
Acknowledgment	
HL/Flags	Window
D. Checksum	Urgent Pointer
Options...	



Έλεγχος ροής με παράθυρο: λαμβάνουσα πλευρά

Τι πρέπει να κάνει ο παραλήπτης;



Ερωτήσεις για τον έλεγχο ροής

What happens if window is 0?

- ☞ Receiver updates window when application reads data
- What if this update is lost?

TCP Persist state

- ☞ Sender **periodically sends 1 byte packets**
- Receiver responds with ACK even if it can't store the packet
- Σε κάποια στιγμή το **Receive Window** του Receiver θα είναι $\neq 0$, θα φανεί στο ACK και ο Sender **θα ξέρει ότι μπορεί να στείλει δεδομένα**

Performance Considerations

- The window size can be controlled by receiving application
 - Can change the socket buffer size from a *default* (e.g. 8Kbytes) to a *maximum* value (e.g. 64 Kbytes)
- The window size field in the TCP header limits the window that the receiver can advertise

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

☞ TCP only acknowledges bytes up to the first missing byte in the stream

◆ Q: how receiver handles **out-of-order segments**

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

Ο παραλήπτης έχει τις παρακάτω δύο γενικές επιλογές

- αμέσως "πετά" τα segments που έφτασαν με λάθος σειρά, ή
- "κρατά" τα segments που ήρθα με λάθος σειρά και περιμένει τα λάβει πακέτα με τα bytes που "χάθηκαν/δεν έφτασαν" να καλύψουν τα κενά

TCP seq. #'s and ACKs

Seq. #'s:

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



Host A

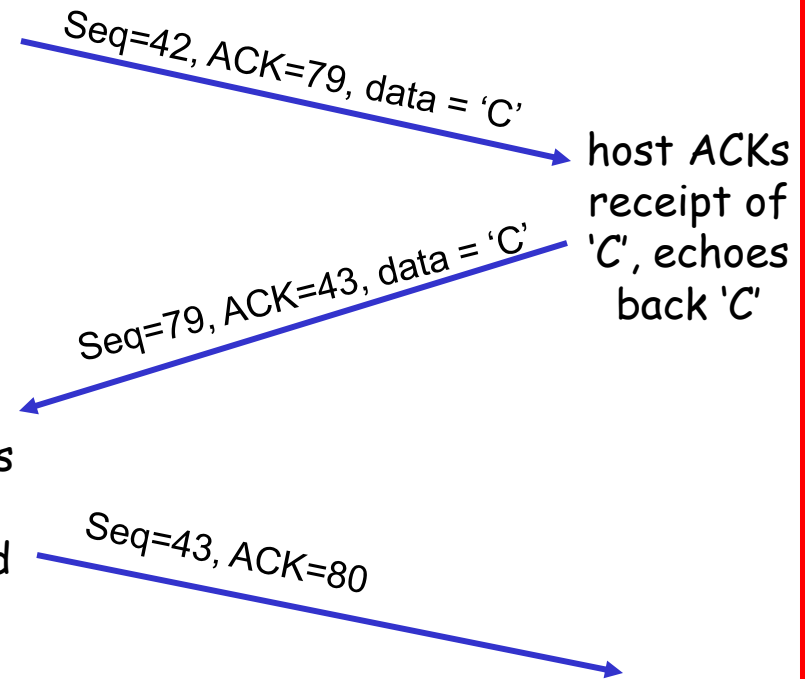


Host B

ACKs:

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

User
types
'C'



host ACKs receipt of 'C', echoes back 'C'

host ACKs receipt of echoed 'C'

time

simple telnet scenario

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

Ο παραλήπτης:

- αμέσως "πετά" τα segments που έφτασαν με λάθος σειρά, ή
- "κρατά" τα segments που ήρθα με λάθος σειρά και περιμένει τα λάβει πακέτα με τα bytes που "χάθηκαν/δεν έφτασαν" να καλύψουν τα κενά

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
 - ☞ but **RTT varies**
- 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**
 - ☞ Αγνοεί segments που έχουν φτάσει με retransmissions
- **SampleRTT** will vary, want estimated RTT "smoother"
average several recent measurements, not just current **SampleRTT**

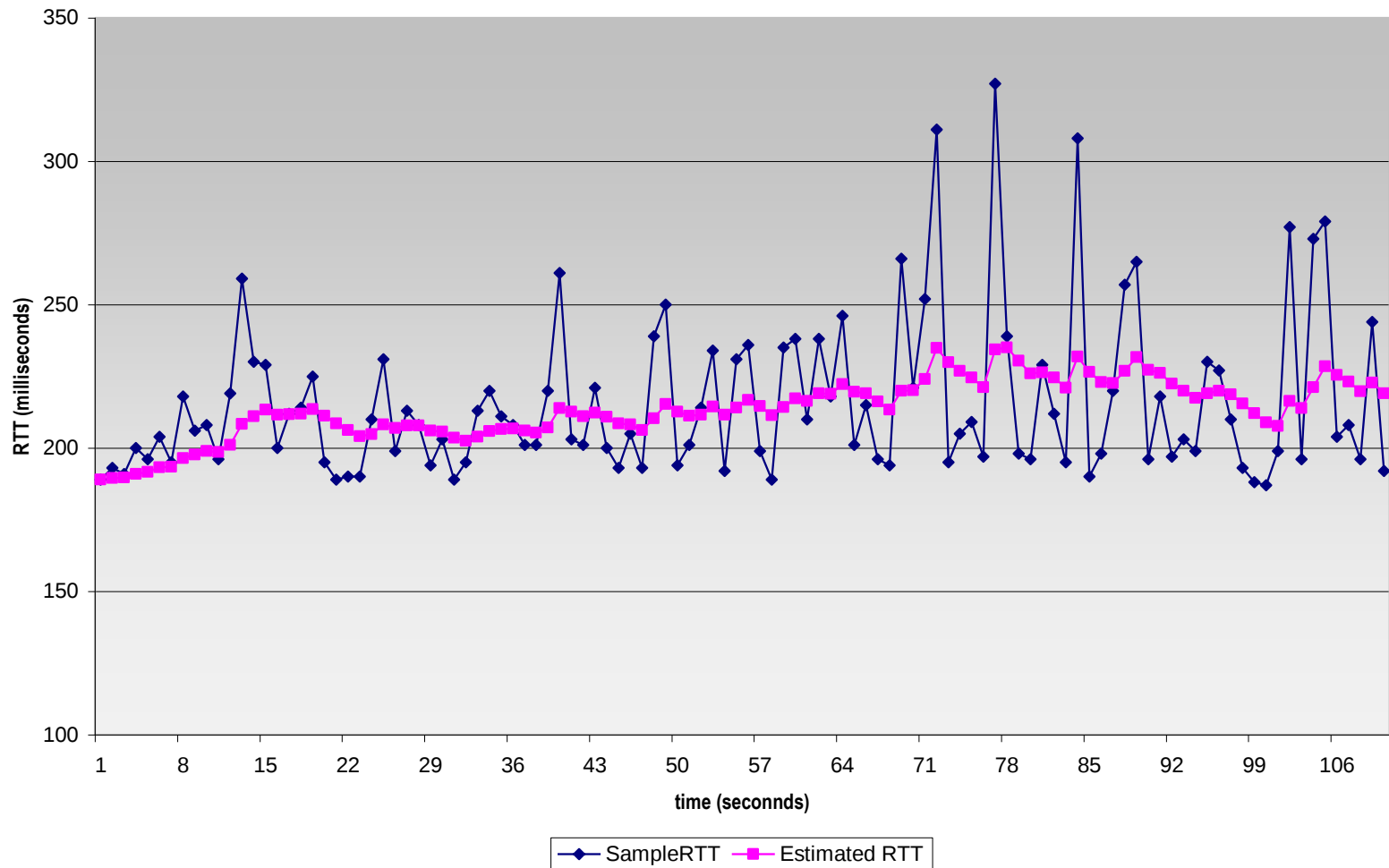
TCP Round Trip Time and Timeout

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

Example RTT estimation:

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr



TCP Round Trip Time and Timeout

Setting the timeout

- `EstimatedRTT` plus "safety margin"
large variation in `EstimatedRTT` -> larger safety margin
- first estimate of how much `SampleRTT` deviates from `EstimatedRTT`:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

Then set timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
 - Pipelined segments
 - Cumulative acks
 - TCP uses single retransmission timer
- **Retransmissions are triggered by:**
 - timeout events
 - duplicate acks
 - Initially consider simplified TCP sender:
 - ignore duplicate acks
 - ignore flow control, congestion control

TCP sender events:

Data received from application:

- Δημιουργεί segment με seq #
seq # is byte-stream αριθμός του πρώτου data byte στο segment
- Αρχίζει τον timer, εάν δεν "τρέχει" ήδη

Ο timer κάθε φορά αντιστοιχεί στο παλιότερο unpacked segment

 Υπάρχει μονάχα ένας timer για το κάθε TCP flow στο host

□ expiration interval: `TimeoutInterval`
Timer expired: Timeout event:

- Ξαναστέλνει το segment not-yet-acked-with-the-smallest-sequence-number
- Ξανα-αρχινά το timer

Ack received:

Εάν acknowledges παλιότερα unpacked segments

- Ενημέρωσε τους buffers/παραμέτρους για το τι είναι γνωστό να έχει γίνει acked
- Αρχισε το timer εάν υπάρχουν outstanding segments

NextSeqNum = InitialSeqNum (sequence # του επόμενου byte που θα πρέπει να σταλεί)
SendBase = InitialSeqNum (segment # με τον μικρότερο αριθμό που έχει στείλει αλλά δεν έχει γίνει ACKed ακόμη)

```
loop (forever) {  
    switch(event)
```

```
    event: data received from application above  
        create TCP segment with sequence number NextSeqNum  
        if (timer currently not running)  
            start timer  
        pass segment to IP  
        NextSeqNum = NextSeqNum + length(data)
```

```
    event: timer timeout  
        retransmit not-yet-acknowledged segment with  
        smallest sequence number  
        start timer
```

```
    event: ACK received, with ACK field value y  
        if (y > SendBase) {  
            SendBase = y  
            if (there are currently not-yet-acknowledged segments)  
                start timer  
        }
```

```
} /* end of loop forever */
```

TCP sender (simplified)

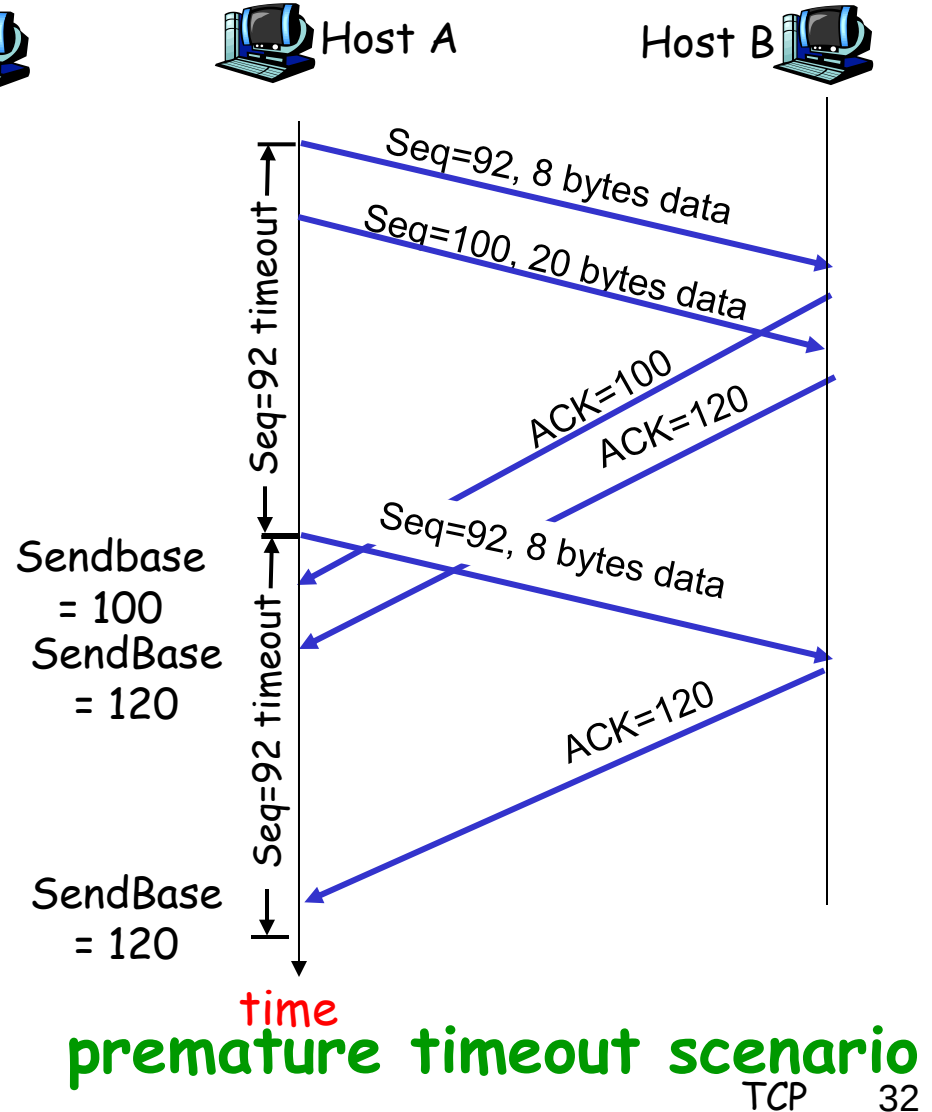
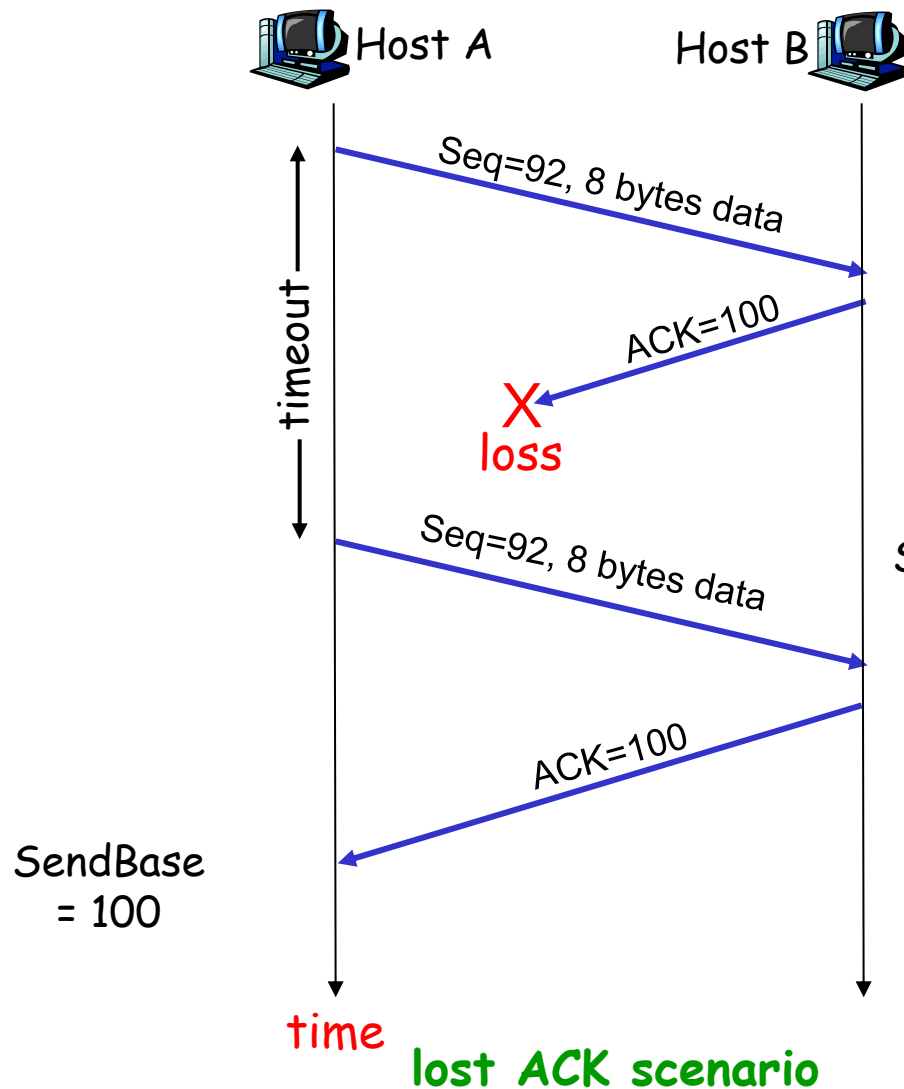
Comment:

- **SendBase-1: last cumulatively ack'ed byte**

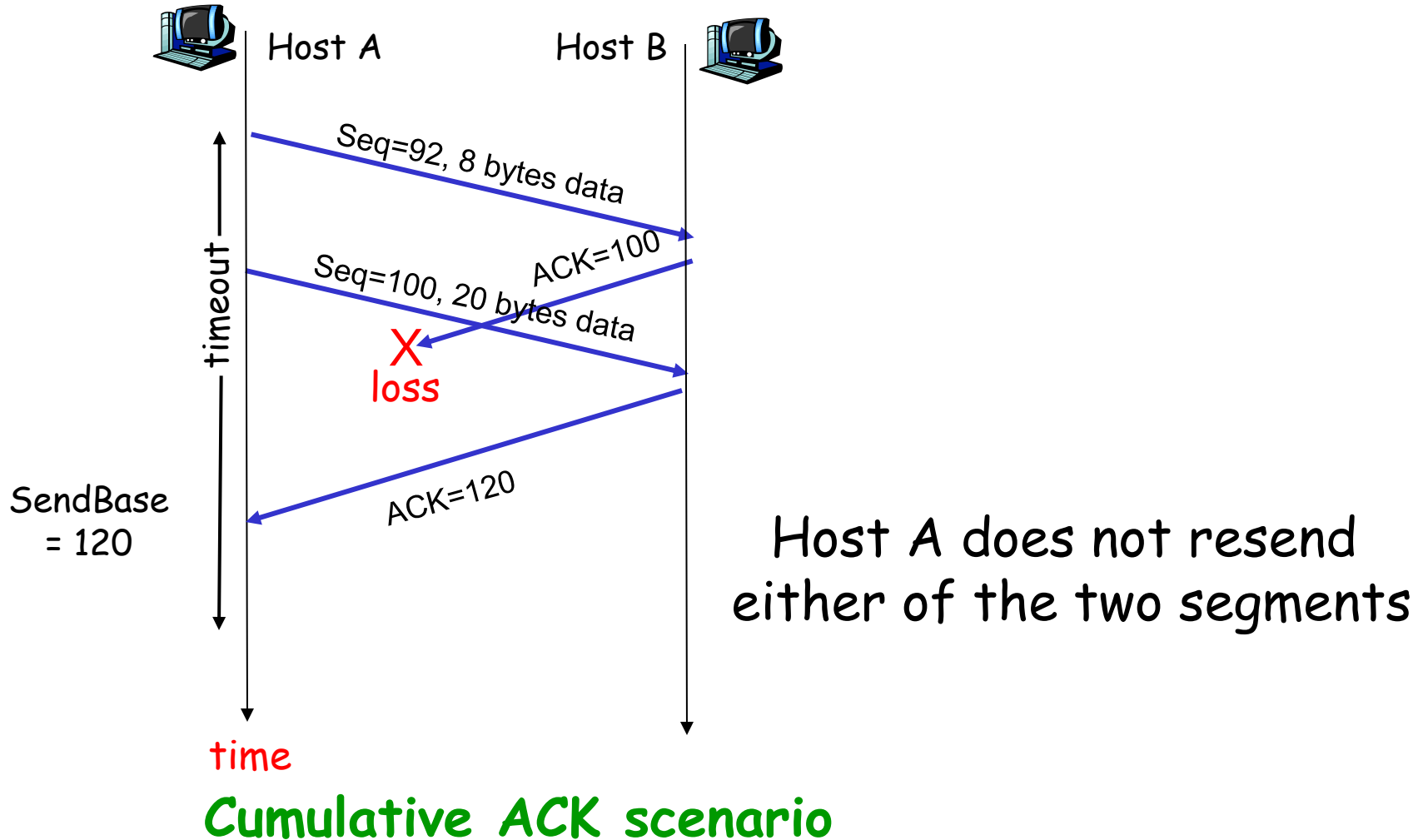
Example:

- SendBase-1 = 71;
y = 73, so the rcvr wants 73+ ;
y > SendBase, so that new data is acked

TCP: retransmission scenarios



TCP retransmission scenarios (more)



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	<u>Delayed ACK</u> . 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	<u>Immediately</u> send <u>single cumulative ACK</u> , ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . <i>Gap detected</i>	<u>Immediately</u> send <u>duplicate ACK</u> , indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	<u>Immediately</u> send ACK , provided that segment starts at lower end of gap

Να θυμάστε

- TCP acks είναι **cumulative!**
- ☞ Ένα segment που έχει ληφθεί σωστά **αλλά σε λάθος σειρά δεν** γίνεται ACKed από τον παραλήπτη
- TCP sender διατηρεί μονάχα την πληροφορία:
 - segment # με τον μικρότερο αριθμό που έχει στείλει αλλά δεν έχει γίνει ACKed ακόμη (**sendBase**), καθώς και
 - sequence # του επόμενου byte που θα πρέπει να σταλεί (**NextSeqNum**)

Παρατήρηση σχετικά με τα Timeout Διαστήματα

- Τα timer expiration συμβαίνουν συνήθως εξαιτίας της συμφόρησης του δικτύου
- Πολλά πακέτα που φτάνουν σε ζεύξεις στο μονοπάτι μεταξύ του sender & receiver χάνονται ή έχουν μεγάλες καθυστερήσεις στις ουρές των δρομολογητών λόγω συμφόρησης
-  Εάν οι senders συνεχίσουν να ξαναστέλνουν τα πακέτα "σταθερά", η συμφόρηση μπορεί να χειροτερεύσει
- Με την εκθετική αύξηση με κάθε retransmission του sender, ο TCP sender προσπαθεί "ευγενικά" **να περιμένει όλο και μεγαλύτερο διάστημα**

Διπλασιάζοντας το Timeout Διάστημα



Κάθε φορά που το TCP *retransmits* **διπλασιάζει** το επόμενο timeout διάστημα αντί να θέτει την τιμή απο τον υπολογισμό των `lastEstimatedRTT` & `DevRTT`

- έχουμε δηλαδή **εκθετική αύξηση του timer** μετά από κάθε retransmission



Οταν ένα από τα παρακάτω γεγονόταν συμβούν:

- **Νεα δεδομένα προωθούνται** απο την εφαρμογή για τον σχηματισμό segment
- **Παραλαβή ενός ACK**

Ο timer παίρνει τιμή βάσει των **`lastEstimatedRTT` και `DevRTT`**

✓ Οι περισσότερες TCP υλοποιήσεις το υποστηρίζουν

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

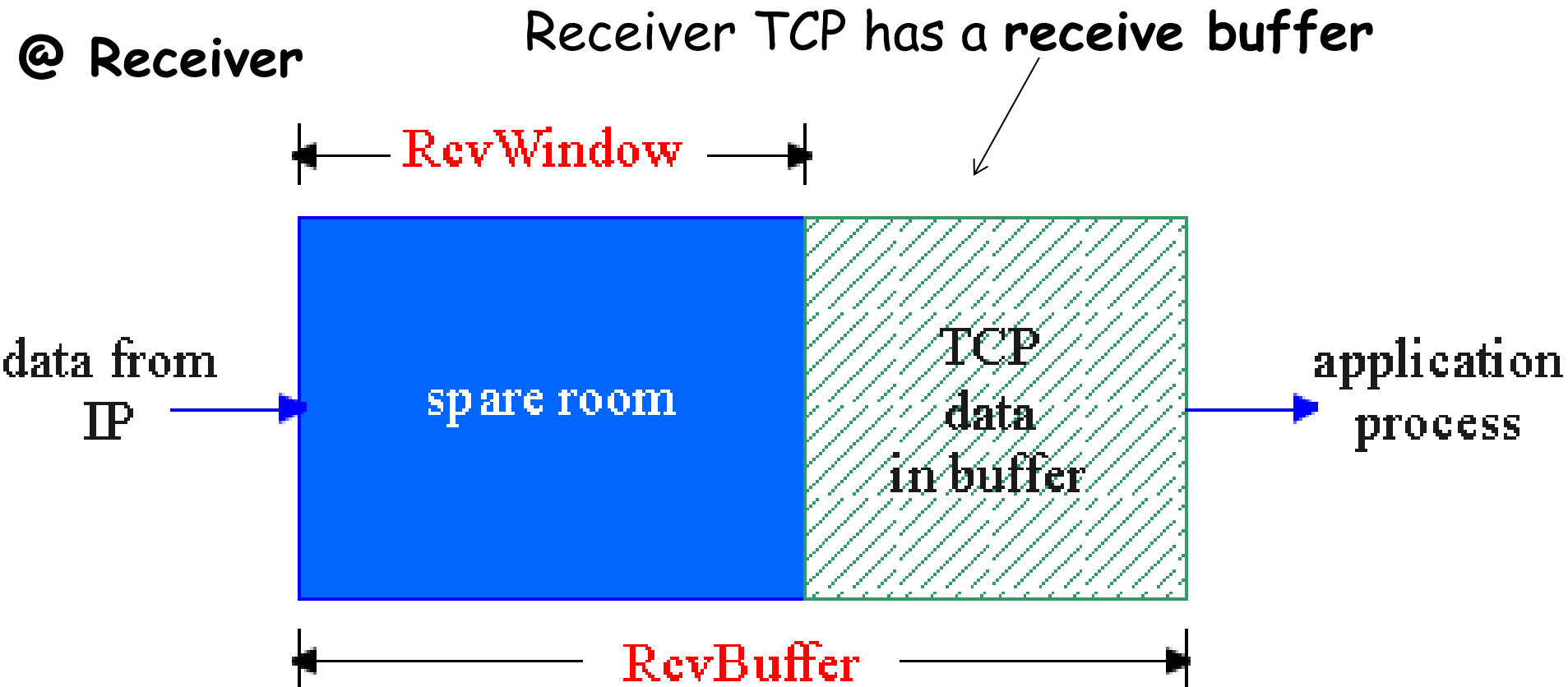
fast retransmit

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - **flow control**
 - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP Flow control (1/3)

- **Receiver advertises spare room** (ReceiveWindow in segments)
- Sender limits unACKed data to Receive Window
 - guarantees receive buffer doesn't overflow

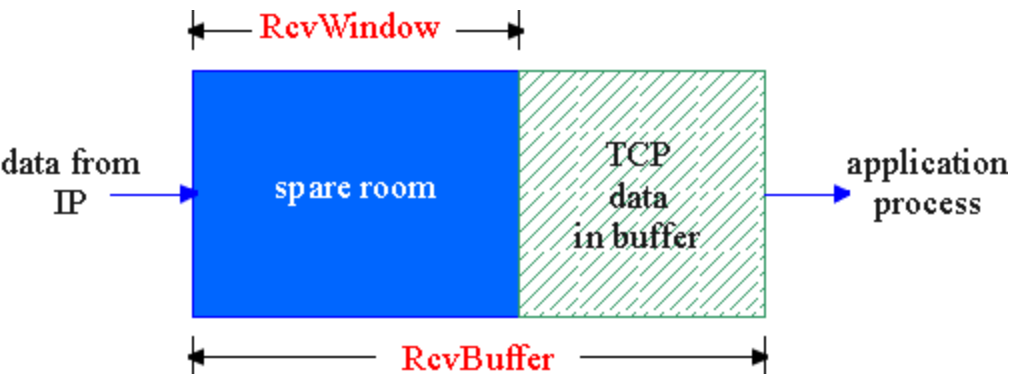


TCP Flow Control (2/3)

Receiver TCP has a receive buffer:

flow control

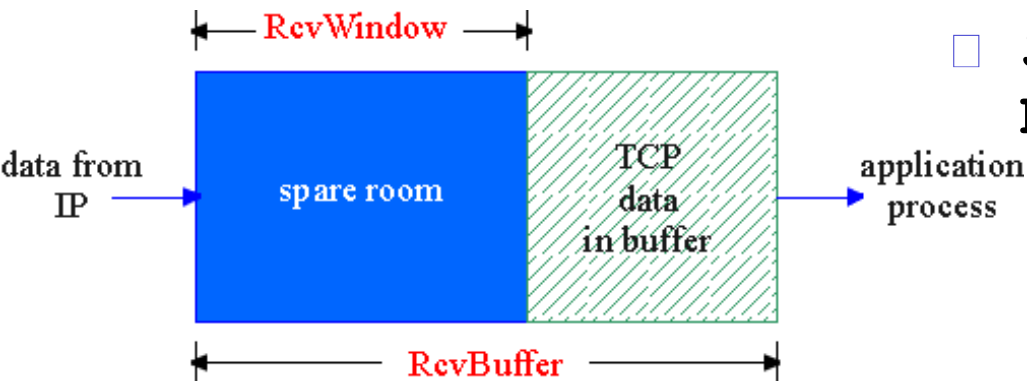
sender won't overflow
receiver's buffer by
transmitting too much, too fast



- speed-matching service:
*matching the **send rate** to the receiving application's drain rate*

👉 application process may be slow at reading from buffer

TCP Flow control (3/3)



- Receiver advertises spare room by including value of **Receive Window** in segments
- Sender limits unACKed data to **Receive Window**
- guarantees receive buffer doesn't overflow

(Suppose TCP receiver discards out-of-order segments)

□ spare room in buffer

= **RcvWindow**

= **RcvBuffer - [LastByteRcvd - LastByteRead]**

Buffer @ Receiver becomes full

- The sender will be notified with the RcvWindow value (=0) in the ACK messages
- The receiver will keep processing the received segments and forward them to the application ...

Eventually there will be space at its buffer ...

◇ ? How the sender will be informed when there is empty space at the receiver buffer?

- TCP requires the sender to keep sending messages of 1Byte when the receiver's *RcvWindow=0*
- These segments will be ACKed and eventually they will contain nonzero RcvWindow value!

Βασικά για τον Έλεγχο Συμφόρησης

Συμφόρηση:

informally: "*too many sources sending too much data too fast for **network** to handle*"

 Διαφορετικό από τον έλεγχο ροής (flow control) !!!

Εκδηλώνεται με:

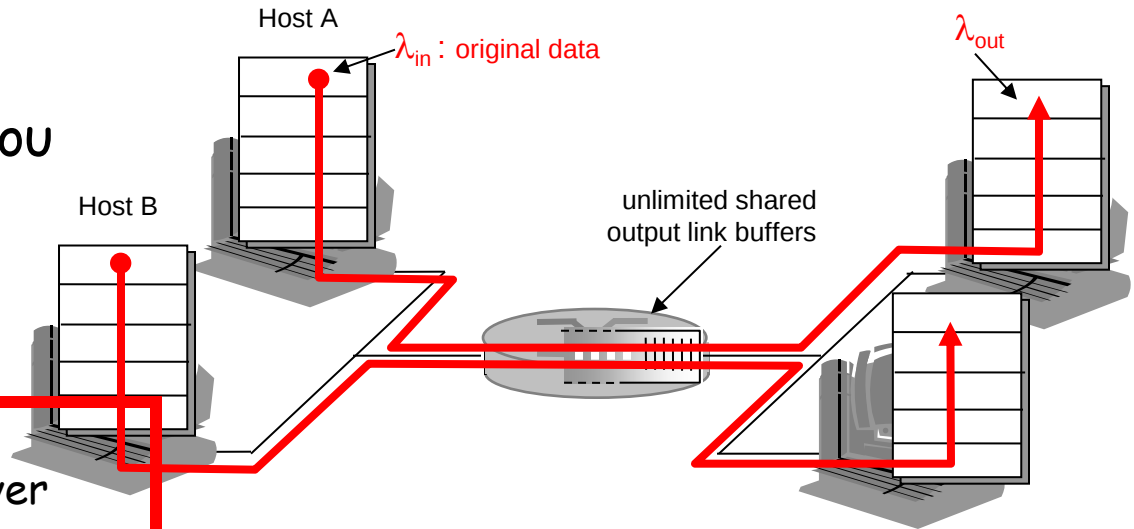
- lost packets (buffer overflow at routers)
- long delays (queuing in router buffers)



a top-10 problem!

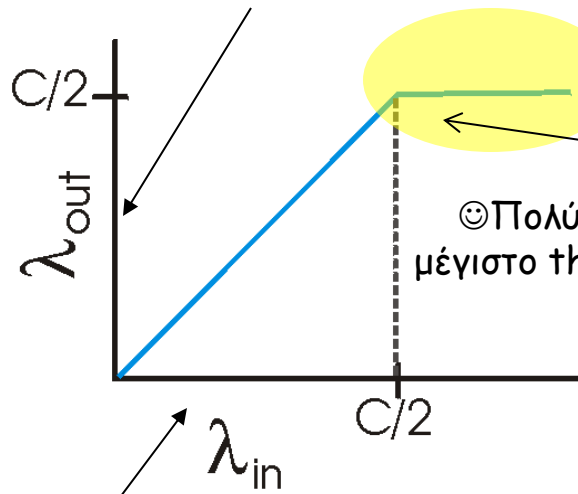
Causes/costs of congestion: scenario 1

- 2 senders, 2 receivers
- 1 router με buffer απείρου μεγέθους
- no retransmission

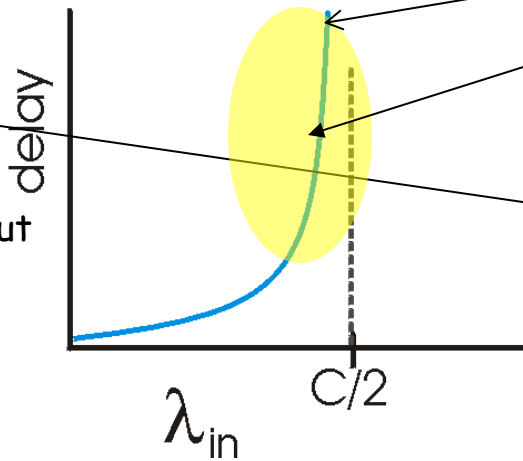


Per-connection throughput:
Number of bytes/second @ receiver

💣 Αλλά αυτό είναι πρόβλημα!



☺ Πολύ ωραία!
μέγιστο throughput

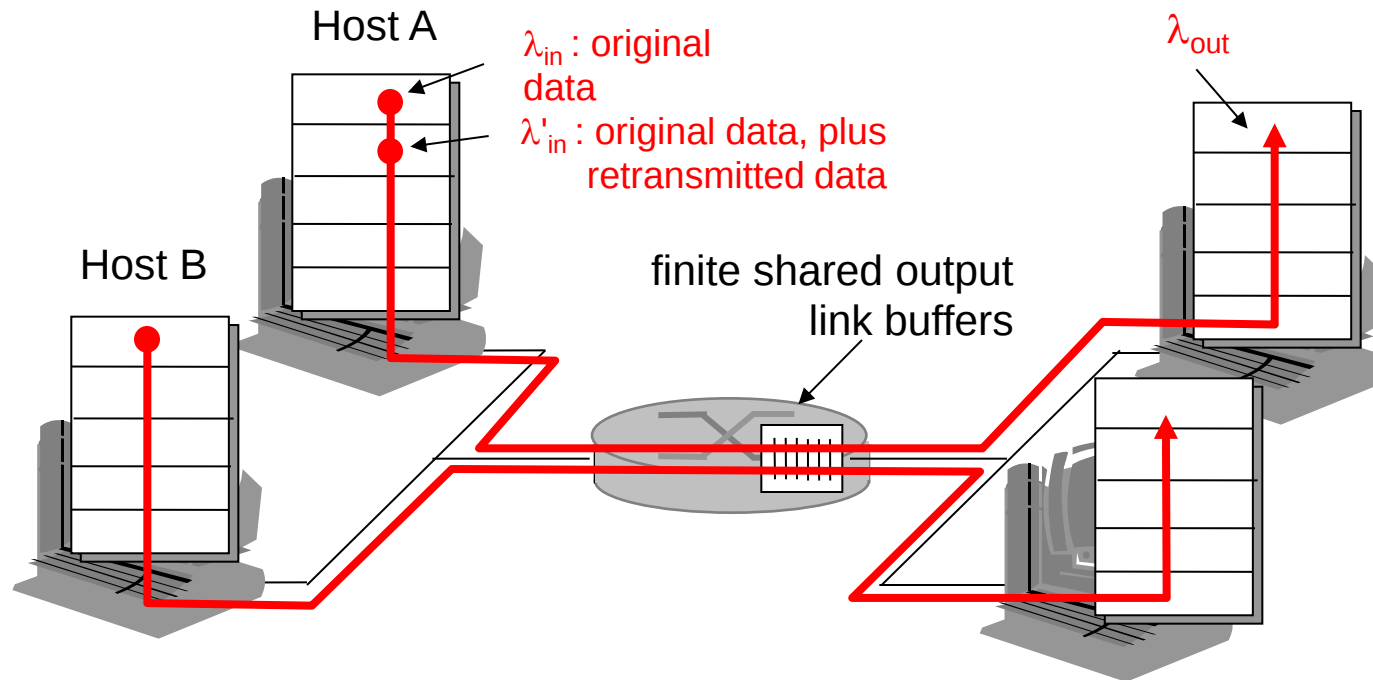


- **large queuing delays** when congested!!!!
- **maximum achievable throughput!!!**

Connection sending rate

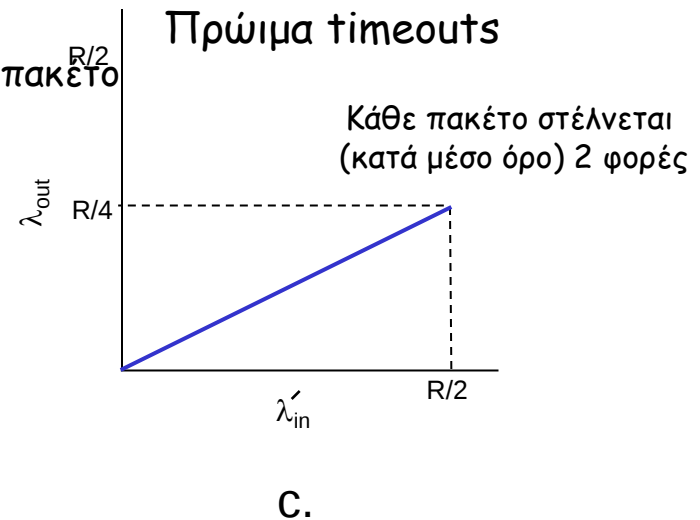
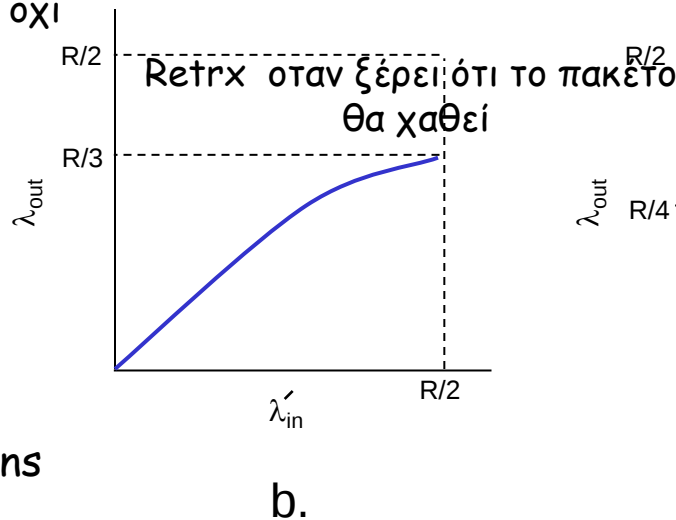
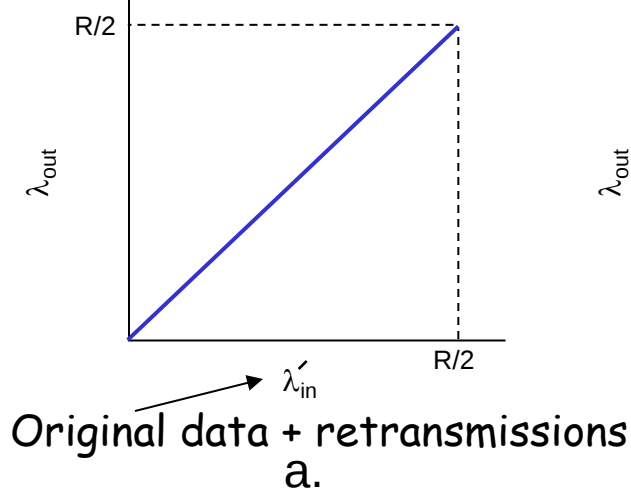
Causes/costs of congestion: scenario 2

- 1 router, με περιορισμένου μεγέθους buffer
- sender retransmission of lost packet



Causes/costs of congestion: scenario 2

Μαντεύει αν ο buffer είναι γεμάτος ή όχι



☞ "costs" of congestion:

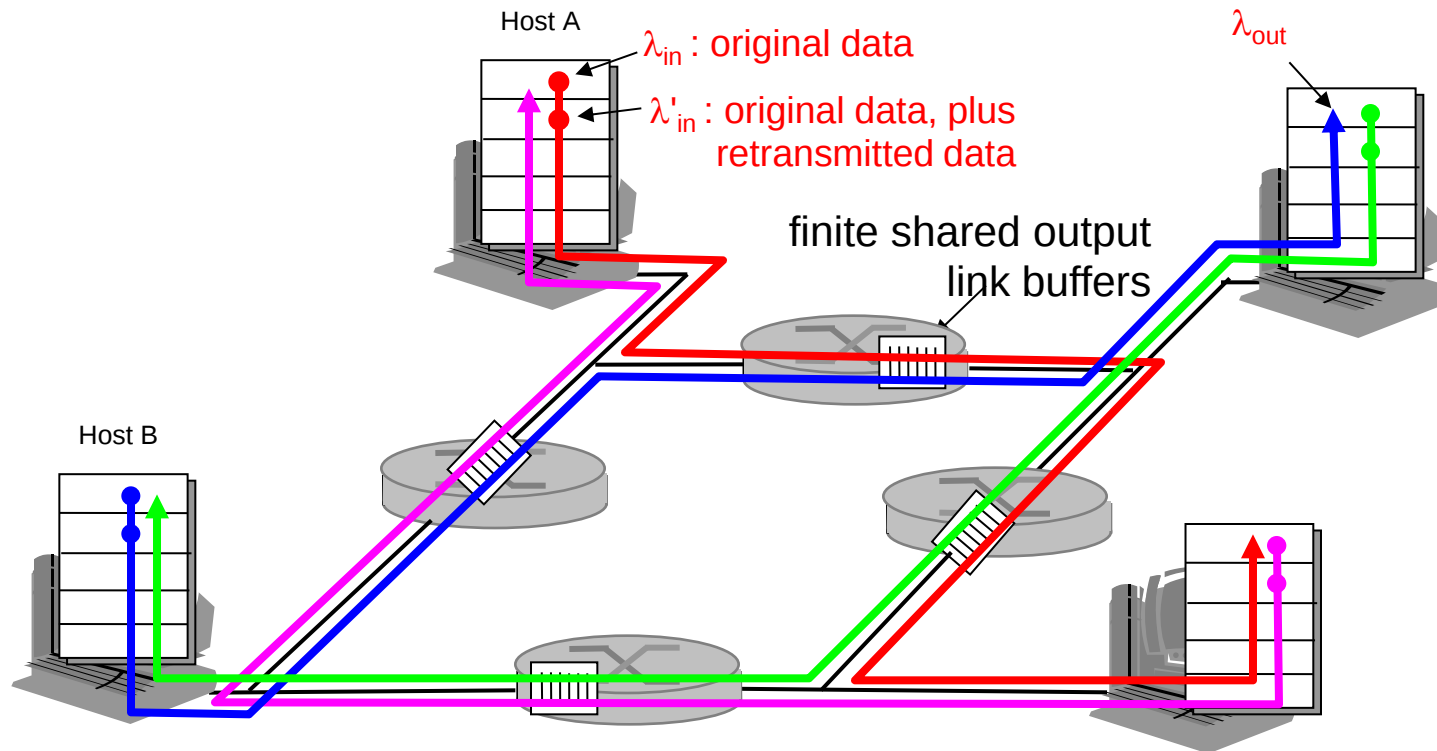
☹ more work (retransmissions) for given "goodput"

☹ **unnneeded retransmissions:** link carries multiple copies of packet

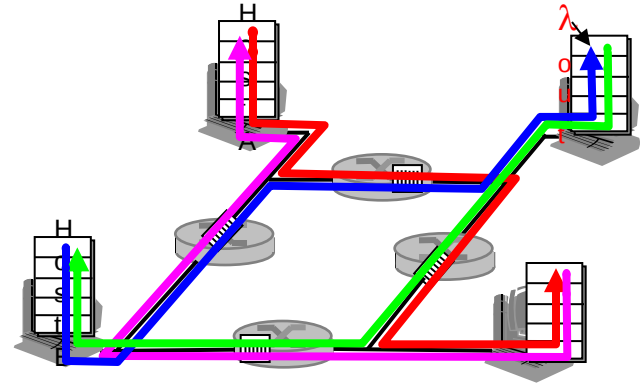
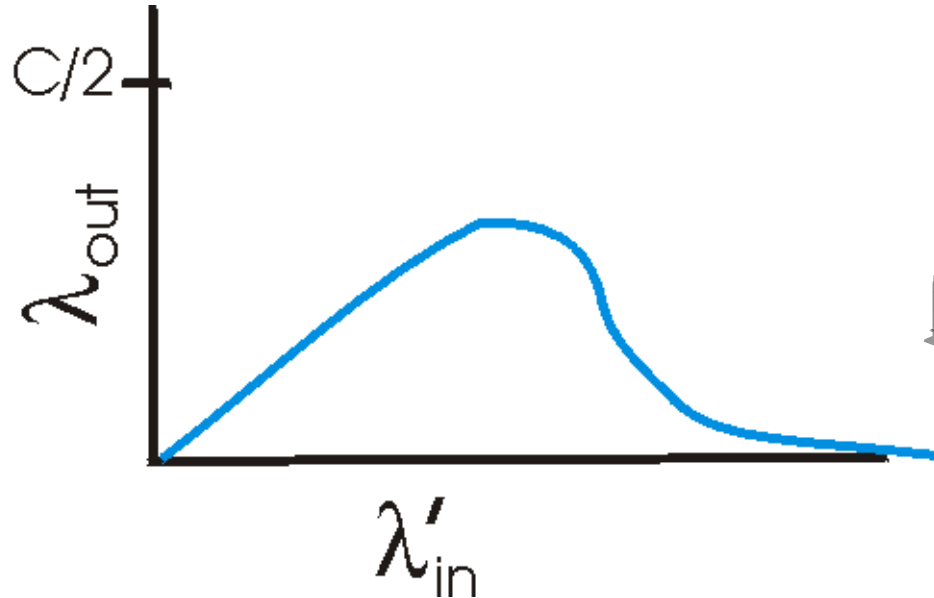
Causes/costs of congestion: scenario 3

- 4 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 "transmission capacity used for that packet at each of the upstream routers to forward that packet to the point at which it is dropped was wasted!

Γενικοί τρόποι αντιμετώπισης συμφόρησης

Two broad approaches towards congestion control:

👉 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

end-to-end: τα 2 hosts που συμμετέχουν παρακολουθούν και ρυθμίζουν το ρυθμό κίνησης τους

👉 End-to-end congestion control:

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

IP does not provide explicit feedback to the end systems

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
 - segment structure
 - reliable data transfer
 - flow control
 - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP congestion control: έλεγχος στο sending rate

🔊 Ο αποστολέας:

- Μειώνει το *sending rate* με το να ελαττώνει το *congestion window* όταν ένα *loss event* εμφανίζεται
- Και αυξάνει το *sending rate*, όταν η συμφόρηση μειώνεται

timeout ή *3 όμοια ACKs*

◇ But how much should a sender reduce its congestion window ?

$\text{LastByteSent} - \text{LastByteAcked} \leq \min \{ \text{Congestion Window}, \text{Receive Window} \}$

συμφόρηση

flow control
TCP

Observation: Large Congestion Windows → Large Number of non-ACKed packets trx → Large Sending Rate

3 κεντρικά σημεία του TCP congestion control

1. Additive increase, multiplicative decrease:

Increase transmission rate (window size), probing for usable bandwidth, until loss occurs

- **additive increase:** increase Congestion Window by *1 MSS* every RTT until loss detected
- **multiplicative decrease:** cut Congestion Window in *half* after loss

1. Slow start

2. Reaction to timeout events

time

TCP congestion control: additive increase, multiplicative decrease (AIMD)

Approach: increase transmission rate (window size),
probing for usable bandwidth **until loss occurs**

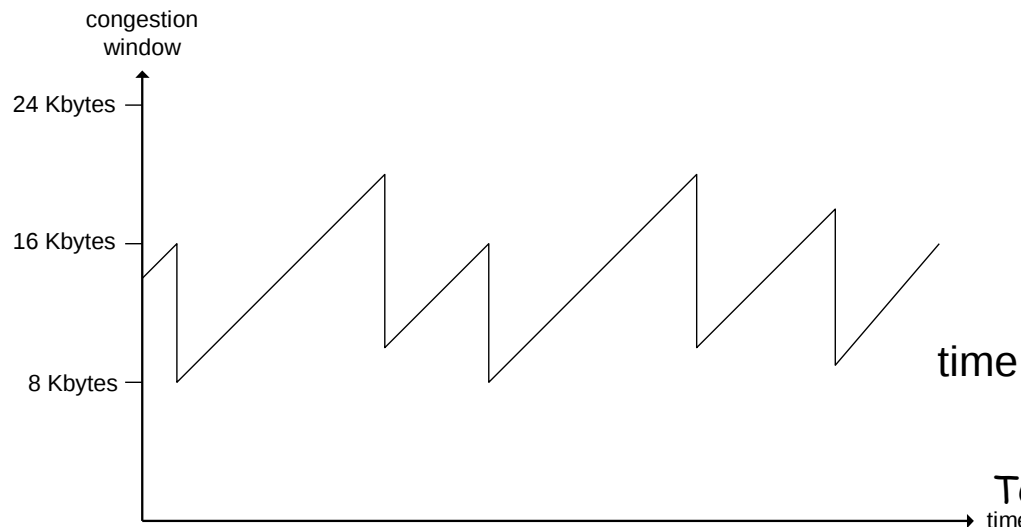
additive increase: increase Congestion Window
by 1 MSS every RTT **until loss detected**

- **multiplicative decrease:** cut Congestion Window in
half **after loss**

Λέγεται και congestion avoidance

Με την γραμμική αύξηση
"διστακτικά" ελέγχει κατά πόσο
η συμφόρηση έχει ελαττωθεί

congestion window size



Saw tooth
behavior: probing for
bandwidth

Προσθετική αύξηση της AIMD

Πώς ακριβώς λειτουργεί η προσθετική αύξηση:

Κάθε φορά που ο TCP sender στέλνει με *επιτυχία πακέτα* (δηλαδή όλα τα πακέτα που στέλνονται κατά τη διάρκεια του τελευταίου RTT έχουν επιβεβαιωθεί με πακέτα ACK) με συνολικό μέγεθος ίσο με το CongWin,

προσθέτει το ισοδύναμο του **ενός πακέτου** στο CongWin

 Προσέξτε ότι στην πράξη το TCP αυξάνει ελαφρώς το μέγεθος του CongWin *με την άφιξη κάθε ACK*.

Congestion Avoidance

- Η φάση της γραμμικής αύξησης του TCP congestion control protocol λέγεται και φάση αποφυγής συμφόρησης

AIMD: παρατηρήσεις

? Γιατί έχουμε γραμμική αύξηση και όχι εκθετική αύξηση?

- Ο sender είναι πρόθυμος να μειώσει το μέγεθος του παραθύρου συμφόρησης με αρκετά μεγαλύτερο ρυθμό απ' ότι είναι πρόθυμος να το αυξήσει
- Έχει αποδειχτεί ότι η μέθοδο AIMD αποτελεί απαραίτητη συνθήκη έτσι ώστε οι μηχανισμοί ελέγχου συμφόρησης να είναι σταθεροί.
- Ένας διαισθητικός λόγος για την **επιθετική μείωση** και τη **συντηρητική αύξηση** του μεγέθους του παραθύρου είναι ότι οι **συνέπειες ενός παραθύρου με μεγάλο μέγεθος είναι πολύ χειρότερες** από τις αντίστοιχες ενός παραθύρου με μικρό μέγεθος.
- πχ, όταν το μέγεθος του παραθύρου είναι πολύ μεγάλο, τα πακέτα που απορρίπτονται **θα αναμεταδοθούν προκαλώντας ακόμα χειρότερη συμφόρηση.**

Επομένως η έξοδος από αυτή την κατάσταση έχει ιδιαίτερη σημασία.

TCP Congestion Control: παρατηρήσεις

- sender limits transmission:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin}$$

(ignoring flow control here)

Roughly:

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- Congestion Window is **dynamic**, function of perceived network congestion

How does sender perceive congestion?

- loss event = timeout or 3 duplicate acks
- TCP sender reduces rate (Congestion Window) after loss event

three mechanisms:

- AIMD
- slow start
- conservative after timeout events

Observation: Large Congestion Windows → Large Sending Rates
in the slides, sometimes we talk about increase/decrease of the rate and
other times about increase/decrease of CongWindow

TCP Slow Start (1/3)

- **When connection begins:** $\text{CongestionWindow} = 1 \text{ MSS}$
congestion window is very small (hence the name "slow start")
Example: $\text{MSS} = 500 \text{ bytes}$ & $\text{RTT} = 200 \text{ msec}$
initial rate = 20 kbps

note: Διαθέσιμο bandwidth μπορεί να είναι $\gg \text{MSS}/\text{RTT}$

☺ Οπότε γρήγορα θέλουμε να το αυξήσουμε ώστε να φτάσουμε το bandwidth που μπορούμε να πετύχουμε

When connection begins:

increase rate exponentially fast until first loss event

✓ Η αύξηση ($\text{CongestionWindow}++$) γίνεται κάθε φορά που λαμβάνεται 1 ACK ...

◇ ? why the rate increases exponentially fast?

Example of TCP Slow start (2/3)

1. TCP sends the first segment and waits for the ACK
2. If this segment is acked before a loss event,
 - the TCP sender increases the congestion window by 1 MSS, and
 - sends out two maximum-sized segments.
3. If these segments are acked before loss events,
 - the sender increase the congestion window by 1 MSS *for each of the ACK segments*,
(giving a congestion window of 4 MSS), and
 - sends out 4 maximum sized segments.

The value of CongWindow **effectively doubles** every RTT during the slow-start phase.

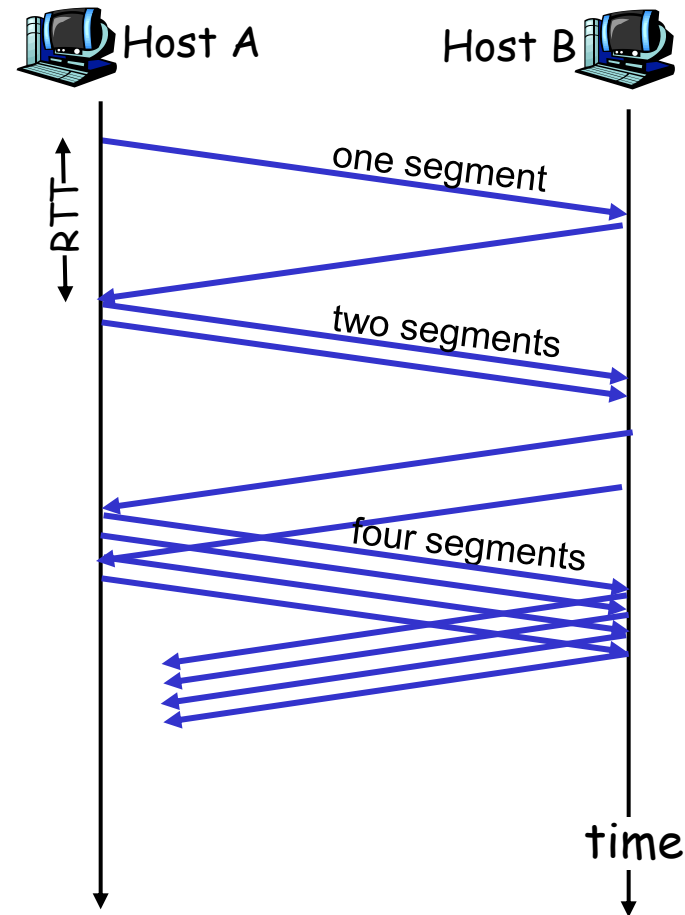
TCP Slow Start (3/3)

When connection begins,
increase rate exponentially **until**
first loss event:

double CongWin every RTT

☞ done by incrementing CongWin
for every ACK received

Summary: initial rate is **slow but**
ramps up exponentially fast



Refinement: inferring loss

☞ Διαφοροποιείται η αντίδραση εάν είναι *timeout* ή *3 όμοια ACKs*

Philosophy:

3 dup ACKs (duplicate acknowledgements) indicate network capable of delivering *some* segments

☞ *timeout* indicates a *more alarming* congestion scenario

Αντίδραση σε γεγονότα συμφόρησης



Διαφοροποιείται η αντίδραση εάν είναι **timeout** ή 3-όμοια-ACKs

Εάν είναι 3-όμοια-ACKs:

- Congestion window *διαιρείται δια 2* και μετά αυξάνεται *γραμμικά*

Εάν όμως συμβεί **timeout**:

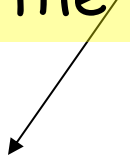


Ο sender μπαίνει *σε slow-start φάση!!!*

- Congestion window = 1MSS
- μετά *αυξάνεται εκθετικά* μέχρι να φτάσει το μισό της τιμής που είχε πριν το timeout,
και μετά από αυτό το σημείο
- *αυξάνεται γραμμικά* (όπως θα γινόταν μετά από ένα 3-όμοια-ACKs γεγονός)

TCP Congestion Control

Determines the window size at which the *slow start will end* and the *congestion avoidance will begin*



When $\text{CongWin} \leq \text{Threshold}$:

sender in *slow-start* phase, window grows *exponentially*

When $\text{CongWin} > \text{Threshold}$:

sender is in *congestion-avoidance* phase, window grows *linearly*

When *triple duplicate ACK* occurs:

- $\text{Threshold} = \text{CongWin} / 2$
- $\text{CongWin} = \text{Threshold}$

When *timeout* occurs:

- $\text{Threshold} = \text{CongWin} / 2$
- $\text{CongWin} = 1 \text{ MSS}$

Quiz 8

Περιγράψετε την *slow start* φάση του TCP.
Πότε μπαίνει σε αυτή ; Πώς αλλάζει το *Congestion Window*? Πόσο είναι το *data rate* του TCP sender?