ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

# Δίκτυα Υπολογιστών

Μαρία Παπαδοπούλη

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

# Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

# Σημείωμα αδειοδότησης


- Το παρόν υλικό διατίθεται με τους όρους της άδειας χρήσης Creative Commons Αναφορά, Μη Εμπορική Χρήση, Όχι Παράγωγο Έργο 4.0 [1] ή μεταγενέστερη, Διεθνής Έκδοση. Εξαιρούνται τα αυτοτελή έργα τρίτων π.χ. φωτογραφίες, διαγράμματα κ.λ.π., τα οποία εμπεριέχονται σε αυτό και τα οποία αναφέρονται μαζί με τους όρους χρήσης τους στο «Σημείωμα Χρήσης Έργων Τρίτων».

[1] http://creativecommons.org/licenses/by-nc-nd/4.0/

- Ως **Μη Εμπορική** ορίζεται η χρήση:
  - που δεν περιλαμβάνει άμεσο ή έμμεσο οικονομικό όφελος από την χρήση του έργου, για το διανομέα του έργου και αδειοδόχο
  - που δεν περιλαμβάνει οικονομική συναλλαγή ως προϋπόθεση για τη χρήση ή πρόσβαση στο έργο
  - που δεν προσπορίζει στο διανομέα του έργου και αδειοδόχο έμμεσο οικονομικό όφελος (π.χ. διαφημίσεις) από την προβολή του έργου σε διαδικτυακό τόπο

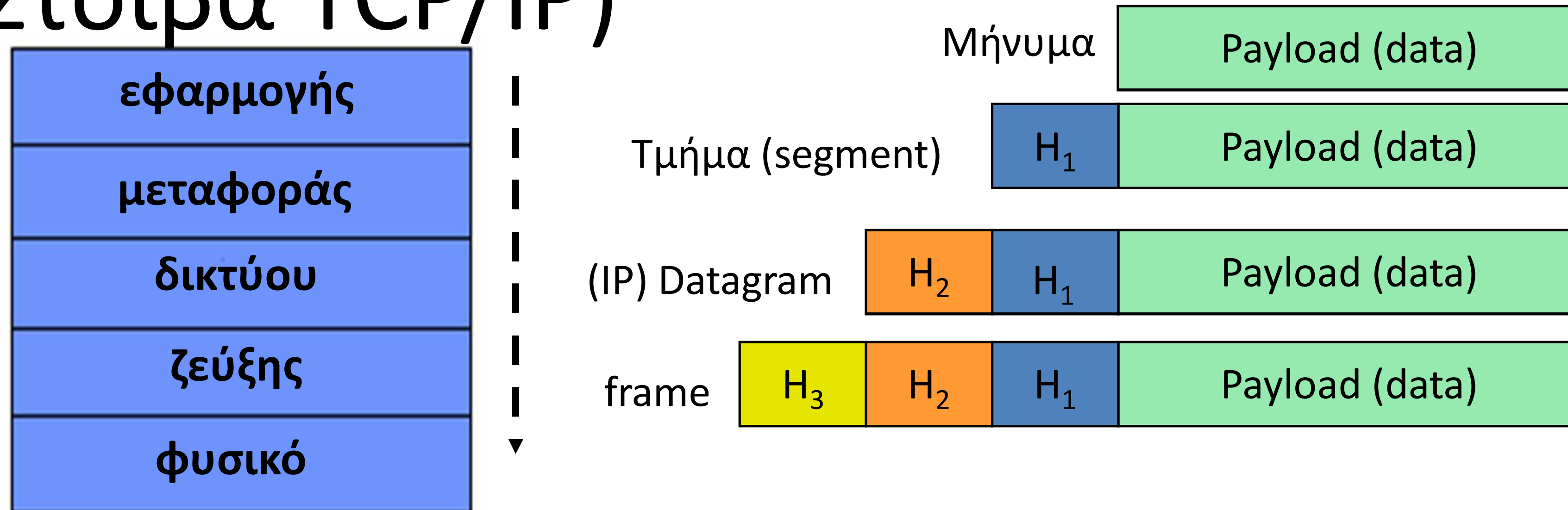
- Ο δικαιούχος μπορεί να παρέχει στον αδειοδόχο ξεχωριστή άδεια να χρησιμοποιεί το έργο για εμπορική χρήση, εφόσον αυτό του ζητηθεί.

# Σκοποί ενότητας

- Revisiting
- layers
- encapsulation
- decapsulation
- packet

Βασισμένο κυρίως στο Κεφ. 5 βιβλίου Kurose

# Μοντέλο επιπέδων Διαδικτύου (Στοίβα TCP/IP)




## Ενθυλάκωση (encapsulation)

@ Καθε επίπεδο:

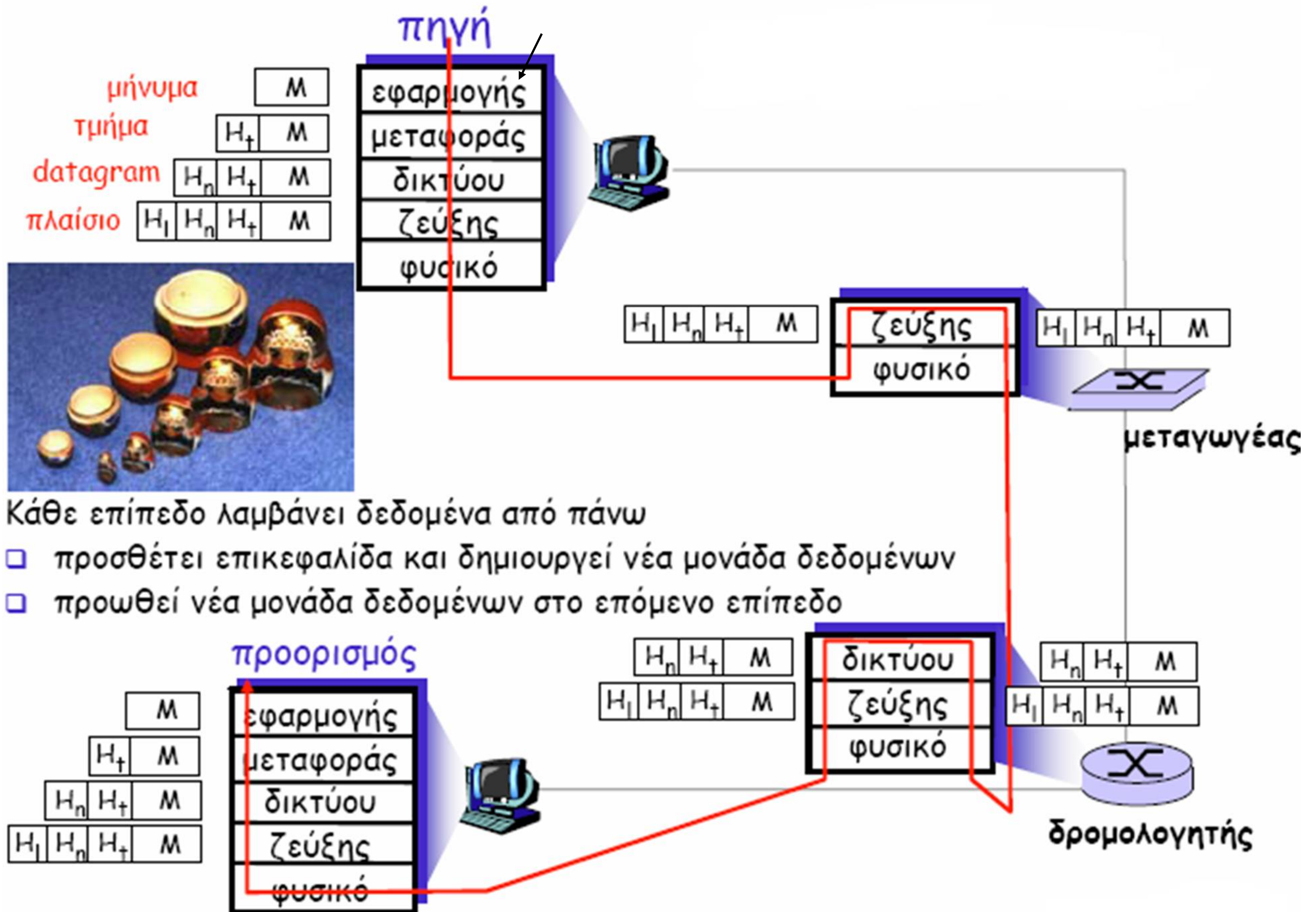
Λαμβάνει δεδομένα από το ανώτερο επίπεδο

Προσθέτει επικεφαλίδα και δημιουργεί νέα μονάδα δεδομένων

Προωθεί την νέα μονάδα στο επόμενο επίπεδο

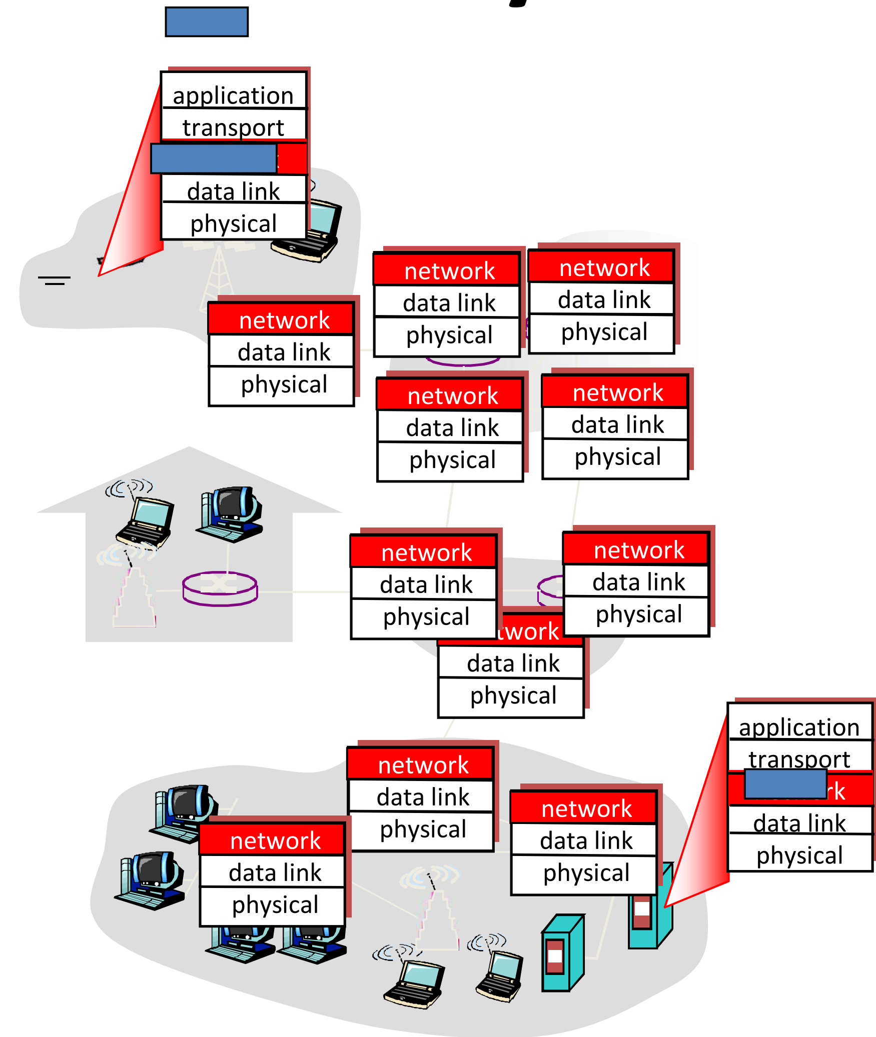
@ the receiver: the “reverse” process (decapsulation)

# Ενθυλάκωση

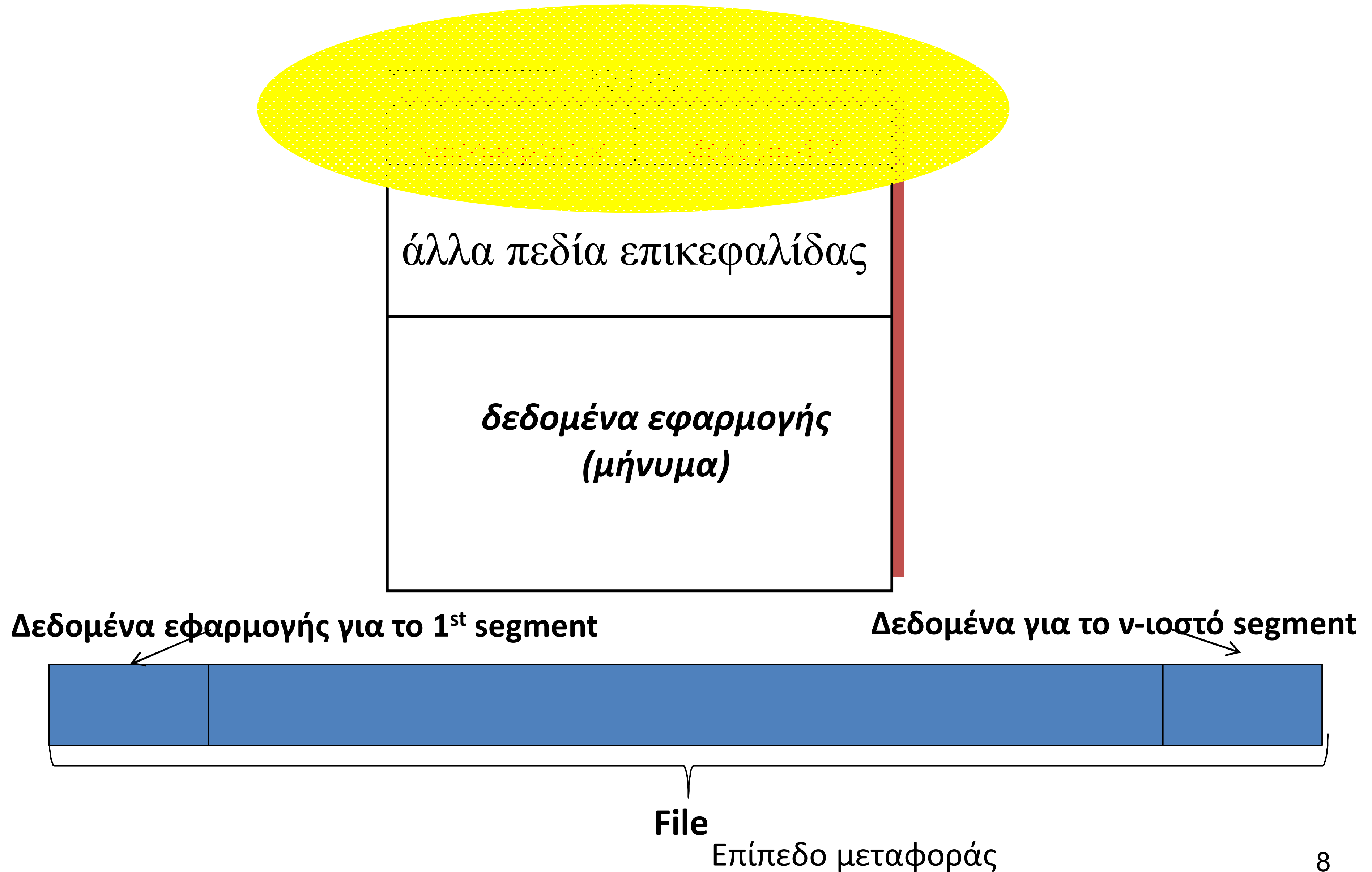


# Brief Review on Network layer

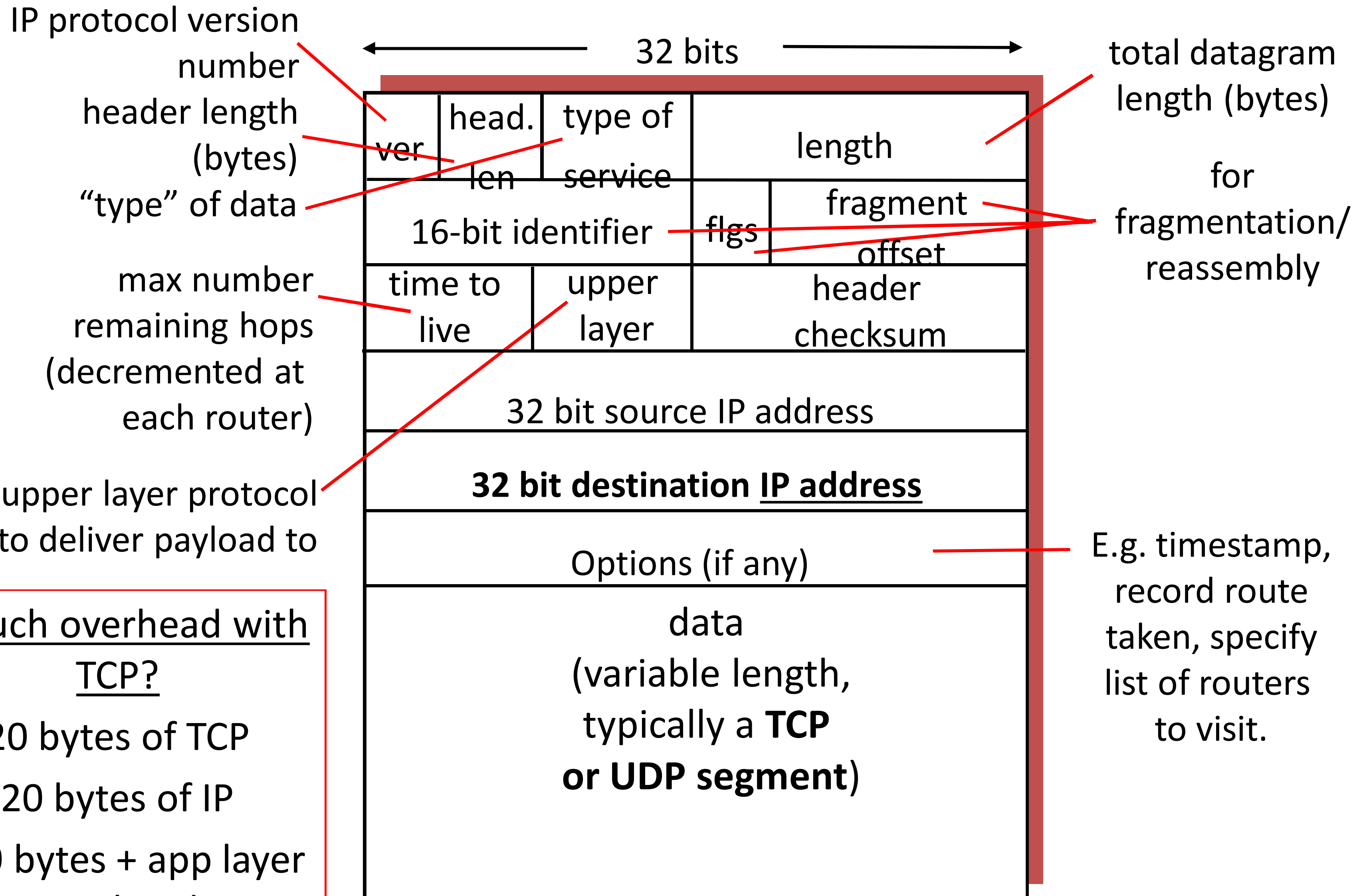
- transport segment from sending to receiving host
- on sending side encapsulates segments into datagrams
- on rcving side, delivers segments to transport layer
- network layer protocols in *every* host, router
- router examines header fields in all IP datagrams passing through it



# TCP/UDP Segment



# Υπενθύμιση: IP datagram format

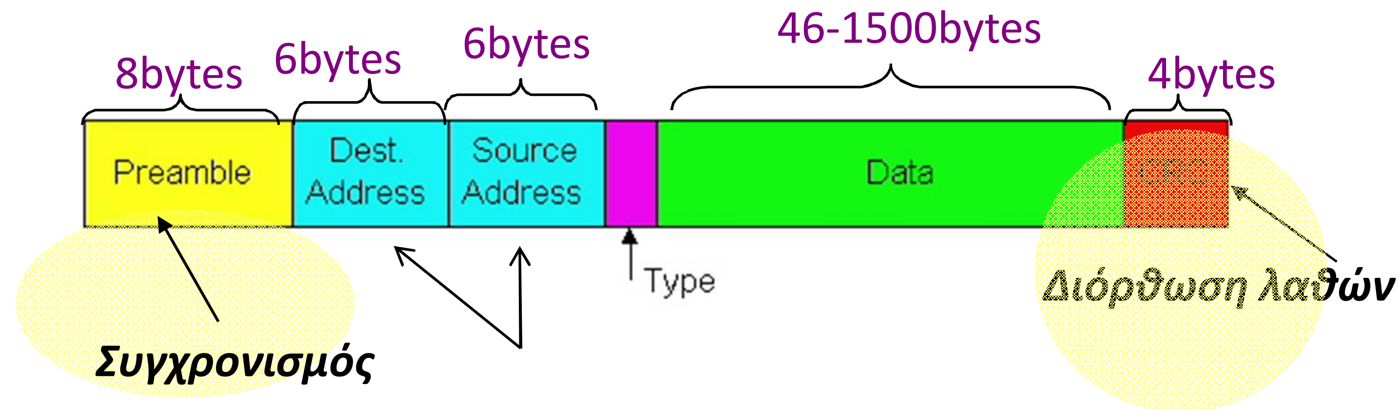



how much overhead with TCP?

- 20 bytes of TCP
- 20 bytes of IP
- = 40 bytes + app layer overhead

# Υπενθύμιση: Ethernet Frame (πλαίσιο)

Ο αποστέλλων ***adapter*** "βάζει" το ***IP δεδομενόγραμμα*** (ή πακέτο κάποιου άλλου πρωτοκόλλου Επιπέδου Δικτύου) στο **Ethernet πλαίσιο**




MAC addresses of neighbors in the link

- Ο adapter "ξέρει" πότε ένα πλαίσιο τελειώνει ***εντοπίζοντας την απουσία ρεύματος***
- Οι Ethernet adapters **μετράνε την *τάση* πριν και κατά τη διάρκεια της μετάδοσης**

Preamble:

- 7 bytes με το μοτίβο *10101010* ακολουθούμενο από ένα byte με το μοτίβο *10101011*
- χρησιμοποιείται για να συγχρονίζει τις τιμές του ρολογιού του παραλήπτη και του αποστολέα

# Κεφ. 3: Επίπεδο μεταφοράς

Στόχος μας είναι η κατανόηση των:

Αρχές πίσω από τις υπηρεσίες του επιπέδου μεταφοράς:

- Πολυπλεξία/αποπολυπλεξία
- ***Αξιόπιστη*** μεταφορά δεδομένων
- ***Έλεγχος ροής*** (flow control)
- ***Έλεγχος συμφόρησης*** (congestion control)

Πρωτόκολλα επιπέδου μεταφοράς στο Internet:

- **UDP**: ασυνδεσιστρεφής μεταφορά
- **TCP**: συνδεσιστρεφής μεταφορά
  & έλεγχος συμφόρησης & ροής

# Υπηρεσίες και πρωτόκολλα επιπέδου μεταφοράς

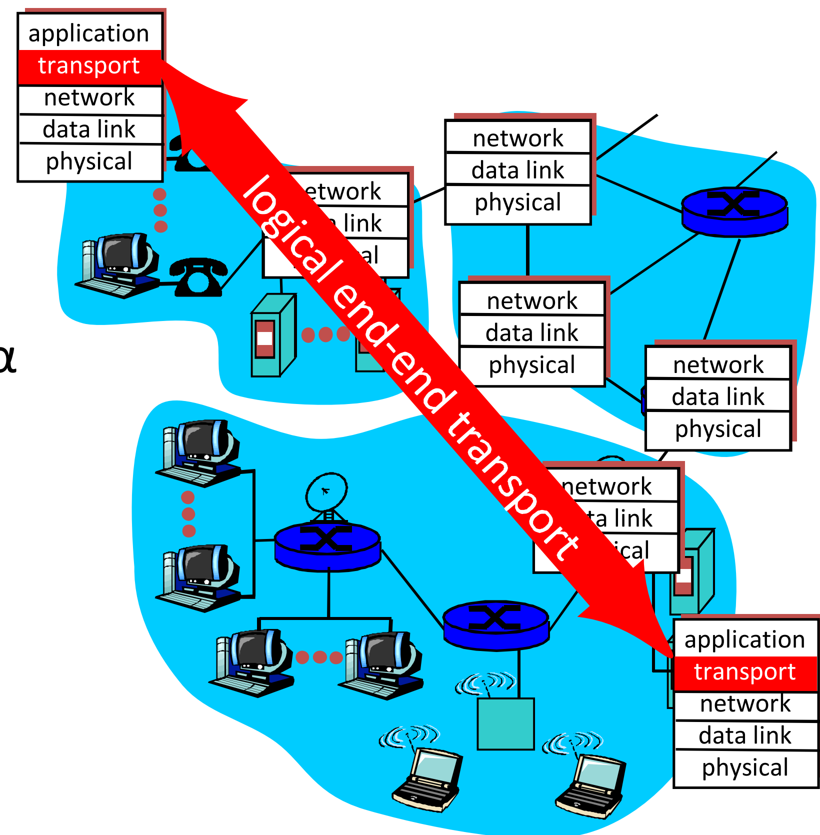
- παρέχουν **επικοινωνία** με τη μορφή “***λογικής***” *σύνδεσης* μεταξύ των διεργασιών που δημιουργούν οι εφαρμογές που τρέχουν σε διαφορετικούς hosts
- Πρωτόκολλα μεταφοράς τρέχουν σε τερματικά συστήματα
  - Αποστέλλουσα πλευρά:

    Χωρίζει τα μηνύματα της εφαρμογής σε ***τμήματα (segments)*** και τα ***προωθεί στο επίπεδο δικτύου***
  - Λαμβάνουσα πλευρά:

    **Επανασυναρμολογεί** τα segments μηνύματα και τα ***προωθεί στο επίπεδο εφαρμογών***

☞ Πάνω από ένα πρωτόκολλα μεταφοράς διαθέσιμα στις εφαρμογές

– Internet: TCP & UDP



# Επίπεδο μεταφοράς vs. δικτύου

*Επίπεδο δικτύου:*

"επικοινωνία" μεταξύ hosts

Το IP πρωτόκολλο ***δεν*** **εγγυάται** ***αξιόπιστη*** **μετάδοση** των πακέτων και **λήψη τους σύμφωνα με τη *σειρά που στάλθηκαν*** *,* ούτε ότι ***δεν*** **θα υπάρξουν λάθη σε bits** των πακέτων

*Επίπεδο μεταφοράς:*

**"λογική" επικοινωνία μεταξύ διεργασιών**

Βασίζεται και επεκτείνει τις υπηρεσίες επιπέδου δικτύου

Ανάλογα με το πρωτόκολλο θα «προσφέρει» κάποιου είδους «εγγυήσεις»

για τη ροή των πακέτων που στέλνονται μεταξύ δύο διεργασιών μέσω δικτύου

# Διαδικτυακά πρωτόκολλα επιπέδου μεταφοράς

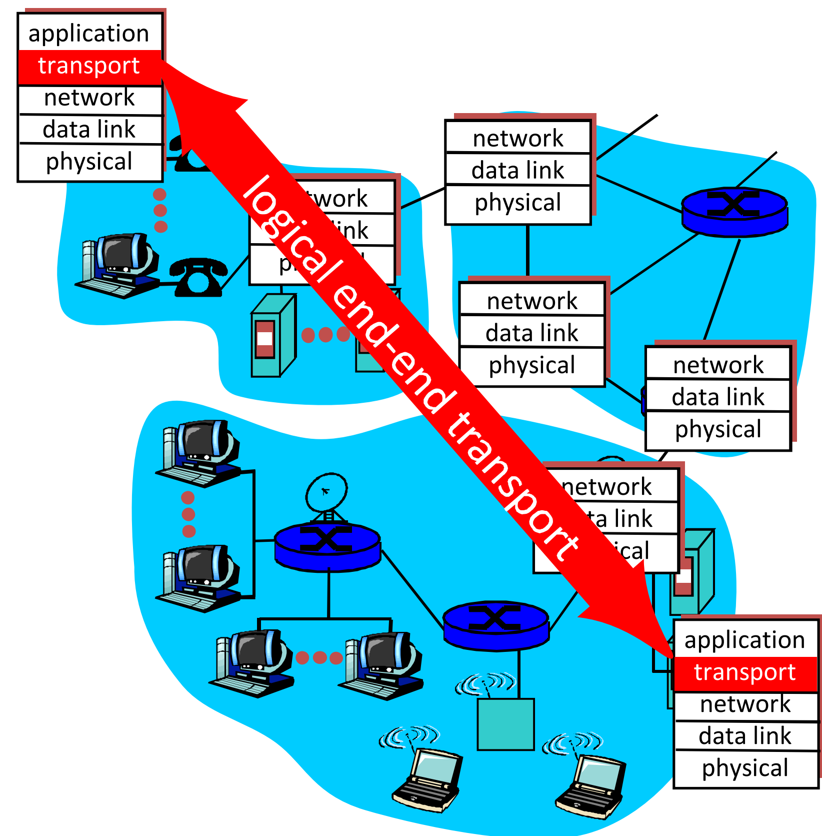
**UDP**: Αναξιόπιστη, χωρίς εγγύηση στη σειρά παράδοσης των πακέτων

- ☹ ***Δεν*** βελτιώνει τον "**best-effort**" χαρακτήρα του IP!!!!

**TCP**: Αξιόπιστη, με εγγύηση στη σειρά παράδοσης των πακέτων

- **Έλεγχος συμφόρησης**
- **Έλεγχος ροής**
- Εγκαθίδρυση σύνδεσης

Προσοχή: Το διαδίκτυο ΔΕΝ δίνει εγγυήσεις καθυστέρησης ή bandwidth



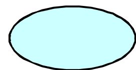
# Πολυπλεξία/αποπολυπλεξία (multiplexing/demultiplexing)

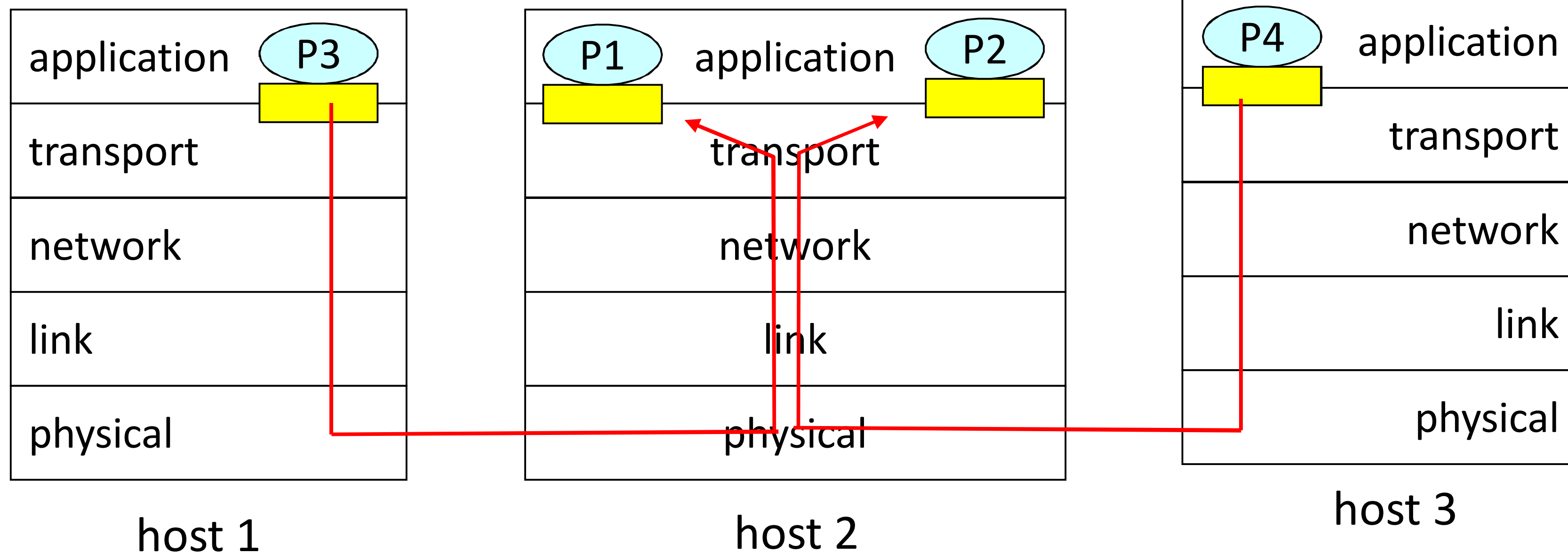
Αποπολυπλεξία στο λαμβάνοντα host:

Μεταφέροντας τα segments που έχουν ληφθεί στο σωστό socket

Πολυπλεξία στον αποστέλλοντα host:

Μαζεύοντας δεδομένα από **πολλαπλά sockets**, ***προσθέτοντας επικεφαλίδα*** (που αργότερα χρησιμοποιείται για demultiplexing)

= socket  = process

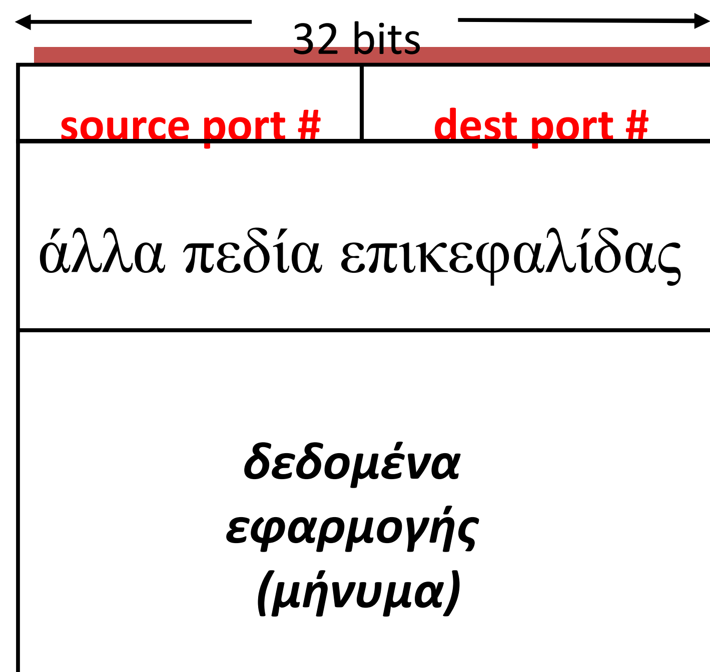


# Πώς δουλεύει η αποπολυπλεξία (demultiplexing)

Το host λαμβάνει IP datagrams

- Κάθε datagram έχει **IP διεύθυνση “πηγής”**, **IP διεύθυνση “προορισμού”**
- ☞ Κάθε datagram “μεταφέρει” **1 segment επιπέδου μεταφοράς**
- **<u>Κάθε segment</u>** έχει ***αριθμό θύρας*** (port) της ***<u>πηγής και προορισμού</u>***

✳ Η συσκευή χρησιμοποιεί ***IP διευθύνσεις & αριθμούς θυρών*** για να **κατευθύνει το segment στο κατάλληλο socket**



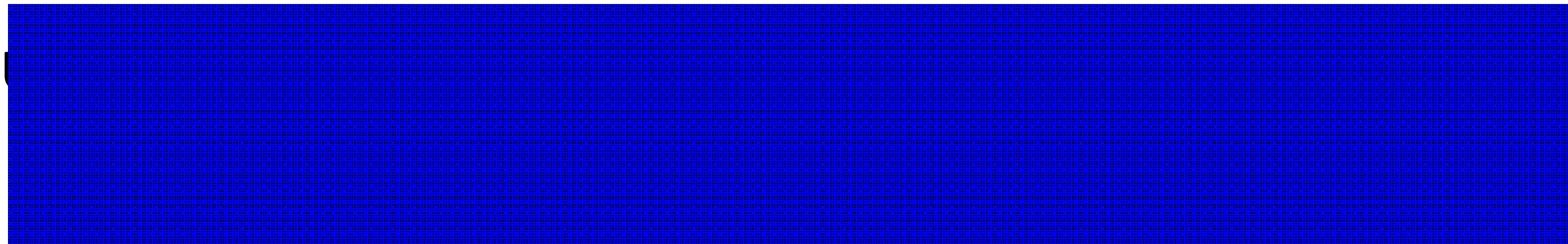

Μορφή **TCP/UDP segment**

Επίπεδο μεταφοράς

# Αποπολυπλεξία *χωρίς "σύνδεση"* (connectionless demultiplexing )

- Δημιουργεί **sockets με αριθμούς θυρών:**

```
DatagramSocket mySocket1 = new DatagramSocket(99111);
DatagramSocket mySocket2 = new DatagramSocket(99222);
```

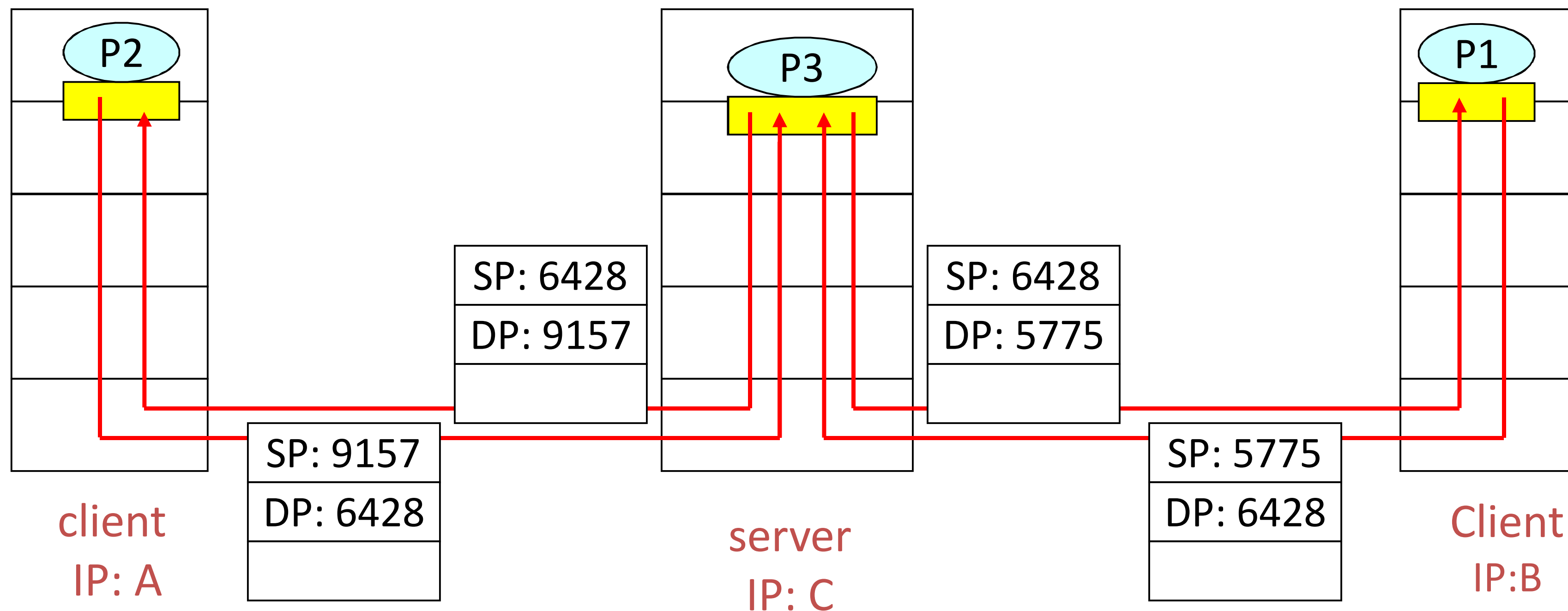


- Όταν ένας host λαμβάνει ένα UDP segment:
  - Ελέγχει τον **<u>αριθμό θύρας *προορισμού*</u>** στο segment
  - Κατευθύνει το UDP segment στο socket με βάση αυτό τον αριθμό θύρας

☞ IP datagrams με ***διαφορετική*** IP **διεύθυνση *πηγής*** ή/και **αριθμό θύρας *πηγής*** κατευθύνονται στο **<u>ίδιο socket</u>**

# Αποπολυπλεξία χωρίς σύνδεση (συνέχεια)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```




Η θύρα πηγής (SP) παρέχει "διεύθυνση επιστροφής"

# Αποπολυπλεξία με σύνδεση (Connection-oriented demultiplexing )

Το TCP socket χαρακτηρίζεται από τα παρακάτω 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***

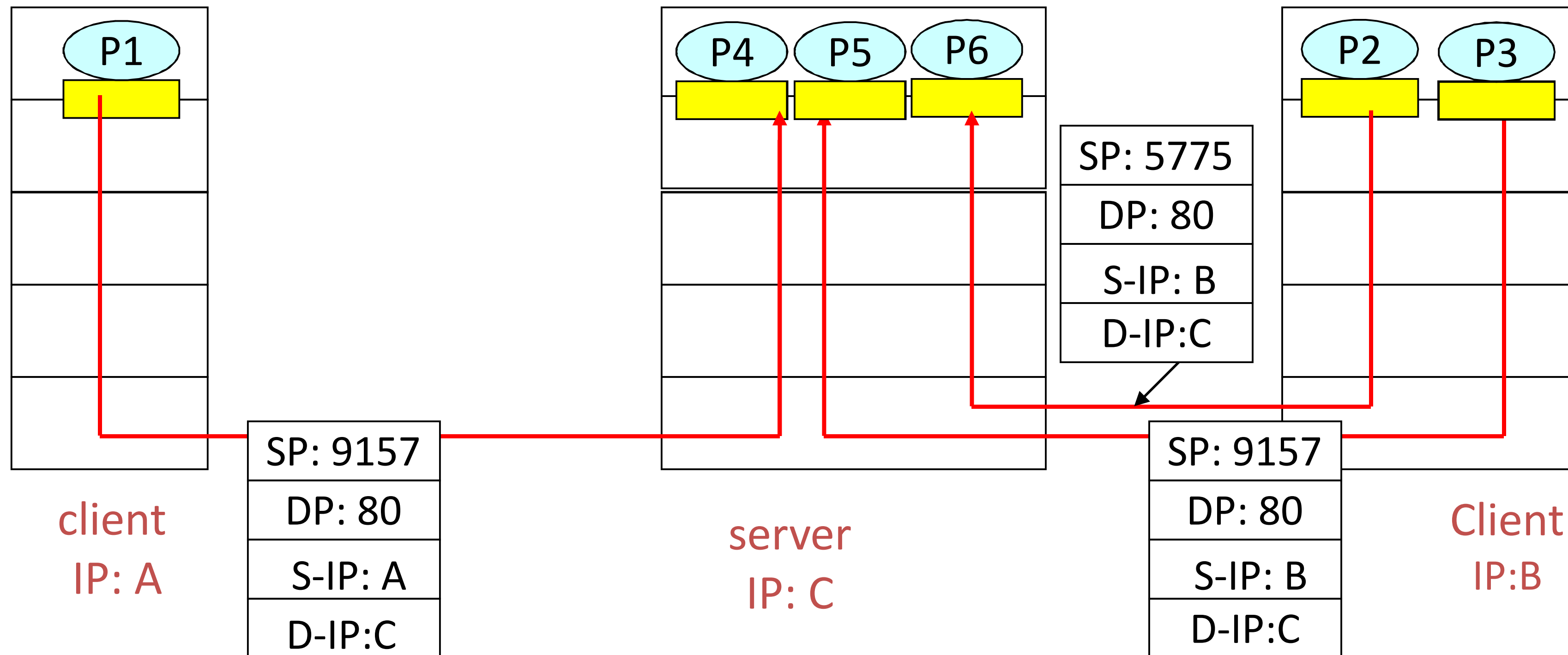
Ένας server host μπορεί να υποστηρίξει ***πολλαπλά*** TCP sockets ταυτόχρονα:

☞ **κάθε socket** χαρακτηρίζεται από μια μοναδική τετράδα πεδίων

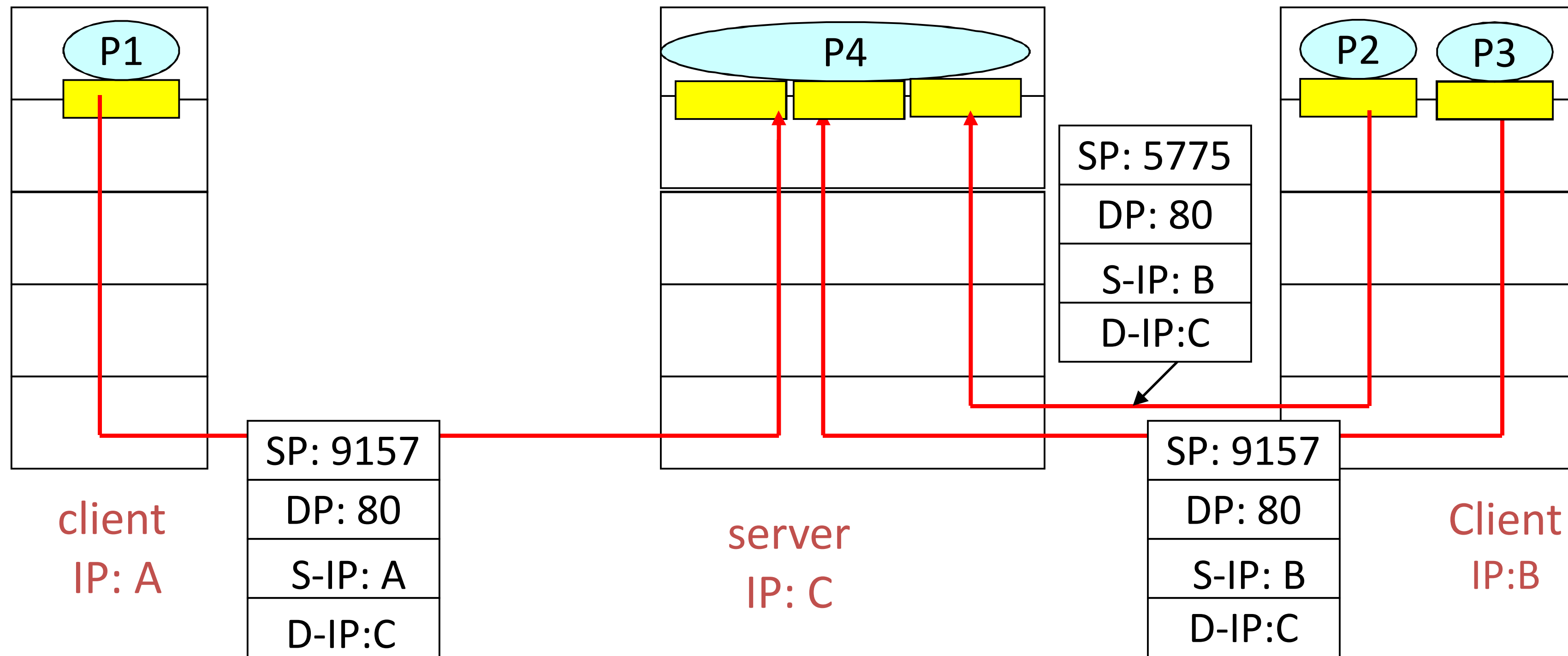
- Web servers έχουν **διαφορετικά sockets** για κάθε client που συνδέεται

  Μάλιστα οι **non-persistent HTTP** έχουν ***διαφορετικά*** sockets για ***<u>κάθε αίτημα</u>***

# Αποπολυπλεξία με σύνδεση (συνέχεια)



# Αποπολυπλεξία με σύνδεση: Threaded Web Server



# UDP: User Datagram Protocol [RFC 768]

- Μινιμαλιστικό πρωτόκολλο μεταφοράς του Διαδικτύου
- Υπηρεσία “καλύτερης δυνατής προσπάθειας” (***best-effort service***),

τα UDP segments μπορεί να:

☹ **χαθούν**

☹ παραληφθούν από την εφαρμογή ***με λάθος σειρά***

☹ ασυνδεσιστρεφές:

☞ ***Δεν* γίνεται χειραψία (handshaking)** μεταξύ UDP sender, receiver

Κάθε UDP segment το χειρίζεται το UDP ανεξάρτητα από τα άλλα

Τι εξυπηρετεί το UDP?

☺ ***Δεν*** χρειάζεται να προηγηθεί εγκατάσταση σύνδεσης (που **προσθέτει καθυστέρηση**)

☺ απλό: ***δεν* διατηρεί “κατάσταση**” στους sender, receiver

☺ **Μικρή** επικεφαλίδα segment

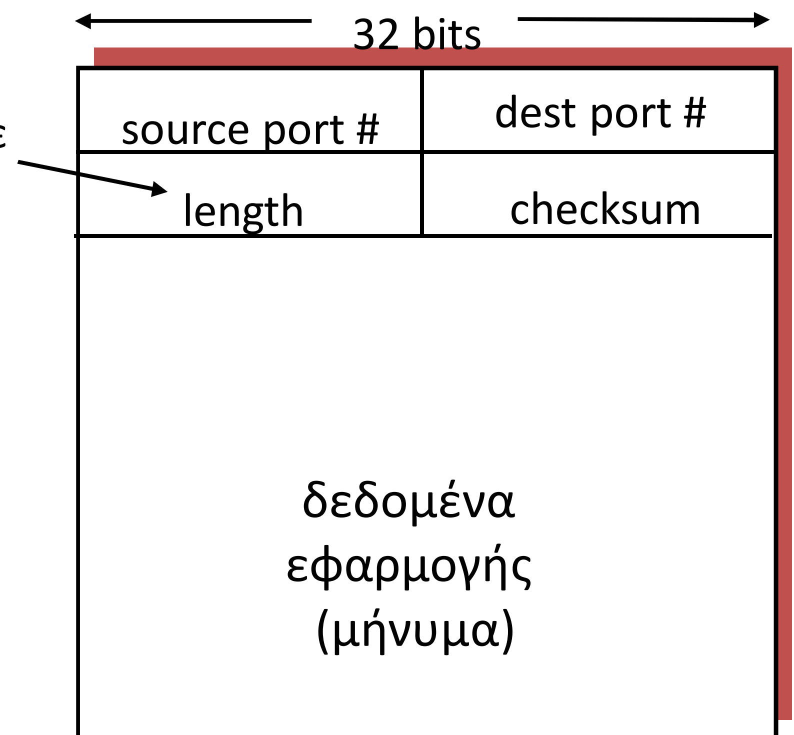
☺ ***Δεν* παρέχει έλεγχο συμφόρησης**: το UDP μπορεί να στείλει δεδομένα όσο γρήγορα μπορεί

# Περισσότερες πληροφορίες για το UDP

Συχνά χρησιμοποιείται για πολυμεσικές εφαρμογές συνεχούς ροής (streaming multimedia apps)

- Ανοχή σε απώλειες
- Ευαισθησία στο ρυθμό

μήκος, σε bytes του UDP segment, μαζί με την επικεφαλίδα




Μορφή UDP segment

Άλλες χρήσεις του UDP

- DNS
- SNMP

Αξιόπιστη μεταφορά πάνω από UDP:

Προστίθεται η αξιοπιστία στο επίπεδο εφαρμογής

☺ Ανάκαμψη από λάθη με βάση την εφαρμογή!

# Domain Name Server (DNS)

- Τρέχει στο ***επίπεδο εφαρμογής***
- Η εφαρμογή ***DNS*** τρέχει σε ένα host. Όταν θέλει να στείλει ένα αίτημα-ερώτηση (query)
  - Φτιάχνει ένα μήνυμα query και το "περνά" στο **UDP**
  - **Δεν γίνεται κάποιο handshake με το UDP που τρέχει στον παραλήπτη** που είναι μία άλλη συσκευή στο δίκτυο
  - Το UDP ***προσθέτει μια επικεφαλίδα στο μήνυμα*** και προωθεί το μήνυμα στο επίπεδο δικτύου
  - Το DNS στον querying host περιμένει την απάντηση στο query που έστειλε
  - Εάν δεν λάβει απάντηση (γιατί είτε το query είτε η απάντηση χάθηκαν) τότε στέλνει το query σε άλλο name server ή ειδοποιεί την εφαρμογή ότι δε έχει λάβει απάντηση

# UDP checksum

<u>Στόχος:</u> ανίχνευση λαθών (π.χ. ανεστραμμένα bits) στο μεταδιδόμενο segment

Αίτια **λαθών**: θόρυβος και παρεμβολές στη σύνδεση (δηλαδή χαμηλό SNR) ή πρόβλημα στο δρομ

Αφού τα χαμηλότερα επίπεδα έχουν ανίχνευση λαθών γιατί γίνεται από το UDP?

***<u>Δεν</u> υπάρχει εγγύηση*** ότι όλες οι συνδέσεις (links) μεταξύ αποστολέα & παραλήπτη **χρησιμοποιούν πρωτόκολλο ανίχνευσης λάθους**

## <u>Αποστολέας:</u>

- Χειρίζεται τα περιεχόμενα του segment ως ακολουθία ακεραίων
- 16-bit
- checksum:
- ***συμπλήρωμα*** ως προς 1 του αθροίσματος των περιεχομένων του segment
- Βάζει την ***τιμή του checksum*** στο πεδίο *checksum* του UDP

## <u>Παραλήπτης:</u>

- Υπολογίζει το checksum του λαμβανόμενου segment
- Ελέγχει εάν η υπολογισθείσα τιμή του checksι ισούται με την τιμή στο πεδίο checksum:
  - NO – ανίχνευση λάθους
  - YES – καμία ανίχνευση σφάλματος

***Αλλά μήπως υπάρχουν λάθη παρά όλα αυτά;***

# Παράδειγμα Internet Checksum

- Σημείωση

  Όταν προσθέτουμε αριθμούς ένα κρατούμενο από το πιο σημαντικό bit πρέπει να προστεθεί στο αποτέλεσμα

Παράδειγμα: πρόσθεση δύο ακεραίων 16-bit

| | | |
|---|---|---|
| | | 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 |
| | | 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 |
| wraparound | 1 | 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 |
| sum | | 1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0 |
| checksum | | 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1 |

# Αρχές αξιόπιστης μεταφοράς δεδομένων

Σημαντικό στα επίπεδα εφαρμογής, μεταφοράς και ζεύξης

top-10 λίστα με **σημαντικά θέματα δικτύου!**

application layer

transport layer

sending process

receiver process

data

data

reliable channel

rdt_send()

data

data

deliver_data()

reliable data transfer protocol (sending side)

reliable data transfer protocol (receiving side)

udt_send()

packet

packet

rdt_rcv()

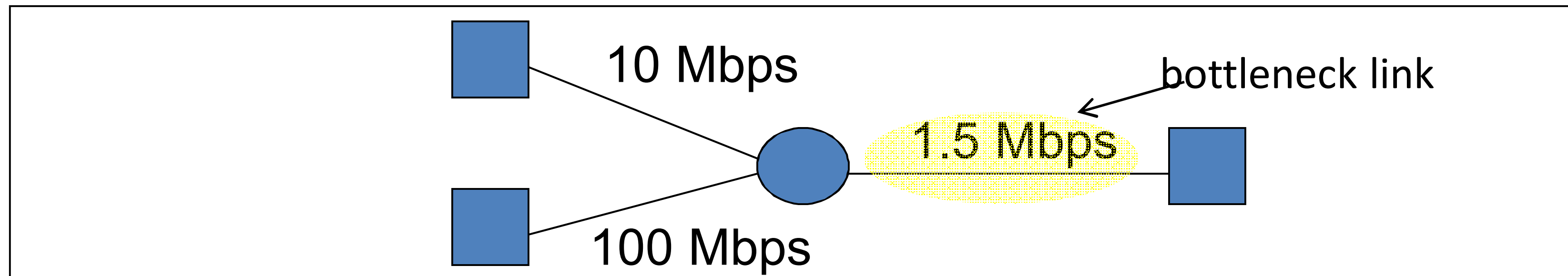
unreliable channel

(a) provided service

(b) service implementation

Τα χαρακτηριστικά του μη αξιόπιστου καναλιού θα προσδιορίσουν την πολυπλοκότητα του πρωτοκόλλου αξιόπιστης μεταφοράς δεδομένων

# Συμφόρηση (congestion)




***Οι χρήστες παράγουν φόρτο κίνησης και ανταγωνίζονται*** για τους πόρους του δικτύου ***αλλά***

- ***δεν* έχουν γνώση των πόρων του δικτύου** (state of resource)
- ***δεν* ξέρουν την ύπαρξη ο ένας του άλλου**

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 (transmission of data)

- Πολλές πιθανές αιτίες
  - ***επαναμεταδόσεις*** πακέτων βρίσκονται ακόμα σε εξέλιξη
    - "κατάρρευση" λόγω συμφόρησης
    - Πώς μπορεί να συμβεί αυτό με τη διατήρηση των πακέτων?

    <u>Λύση:</u> καλύτεροι ***timers*** και ***TCP έλεγχος συμφόρησης***
  - ***Μη παραδοθέντα*** πακέτα
    - Τα πακέτα ***καταναλώνουν πόρους*** και γίνονται ***<u>drop</u>*** κάπου αλλού στο δίκτυο

    Λύση: έλεγχος συμφόρησης για ΌΛΗ την κίνηση

# Προσεγγίσεις Ελέγχου Συμφόρησης

## 1. **End-to-end** congestion control

TCP takes this approach since **IP** does <u>**NOT**</u> provide feedback to the end system regarding network congestion

## 2. **Network**-assisted congestion control

Οι δρομολογητές στέλνουν άμεση πληροφόρηση στον αποστολέα, για παράδειγμα με τους παρακάτω τρόπους:

- a **bit** indicating congestion at a link or
- an explicit control **message**

In ATM, the router can inform the sender <u>*explicitly*</u> of the transmission rate, this router can support on an outgoing link

Στο σημερινό Διαδίκτυο ακολουθείται η end-to-end congestion.

# Γενικές Προσεγγίσεις Ελέγχου Συμφόρησης

1. **End-end** congestion control:
   - **Η συσκευή του χρήστη (end system) <u>δεν</u> παίρνει άμεσο feedback (πληροφορία) από το δίκτυο**
   - **Προβλέπει/συμπεραίνει την συμφόρηση από τις διάφορες μετρήσεις που κάνει η συσκευή του χρήστη (end-system) από τις καθυστερήσεις ή τις απώλειες πακέτων**
   - Approach taken by TCP
   - Αυτό χρησιμοποιείται τώρα στο Διαδίκτυο

2. **Network**-assisted congestion control:
   - **<u>Οι δρομολογητές</u>** πληροφορούν τη συσκευή του χρήστη (end system)
   - Single **bit indicating congestion** (SNA, DECbit, TCP/IP ECN, ATM)
     - ***Explicit*** **rate sender** should send at
     - ☹ Problem: makes routers complicated

# Congestion Control and Avoidance

A mechanism which:

- Uses network resources **efficiently**
- Preserves **fair** network resource allocation
- Prevents or avoids **collapse**

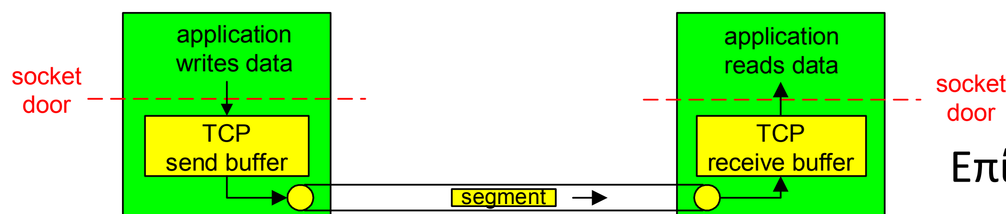
Congestion collapse is not just a theory

Has been frequently observed in many networks

# TCP: Επισκόπηση (1/4) RFCs: 793, 1122, 1323,2018, 2581

## Σημαντικά χαρακτηριστικά του TCP

- **σημείο-προς-σημείο:**
  Ένας αποστολέας, ένας παραλήπτη
  (σε αντίθεση με το multicasting)
- **αξιόπιστο, <u>σε σειρά</u> ροή των *byte:***
  Η **<u>εφαρμογή</u>** από επάνω θα "παραλάβει" τα πακέτα στη σωστή σειρά
- **pipelined:**
  Ο TCP έλεγχος συμφόρησης & ροής θέτουν το μέγεθος παραθύρου
  **<u>Πολλά πακέτα</u>** μπορούν να έχουν σταλθεί ***"ταυτόχρονα"*** 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)
- **"συνδεσιστρεφές":** 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

# TCP seq. #'s and ACKs

Seq. #'s:

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

ACKs:

- seq # of ***next* byte**

**που αναμένεται από την άλλη πλευρά**

- ***Υπάρχουν και τα cumulative* ACK**

Q: πώς χειρίζεται ο παραλήπτης πακέτα που έρχονται σε λανθασμένη σειρά?

A: TCP δεν προσδιορίζει τον τρόπο.

Το αφήνει στον προγραμματιστή που υλοποιεί τη συγκεκριμένη έκδοση

# TCP seq. #'s and ACKs

Seq. #'s:

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

ACKs:

- seq # of ***next* byte**

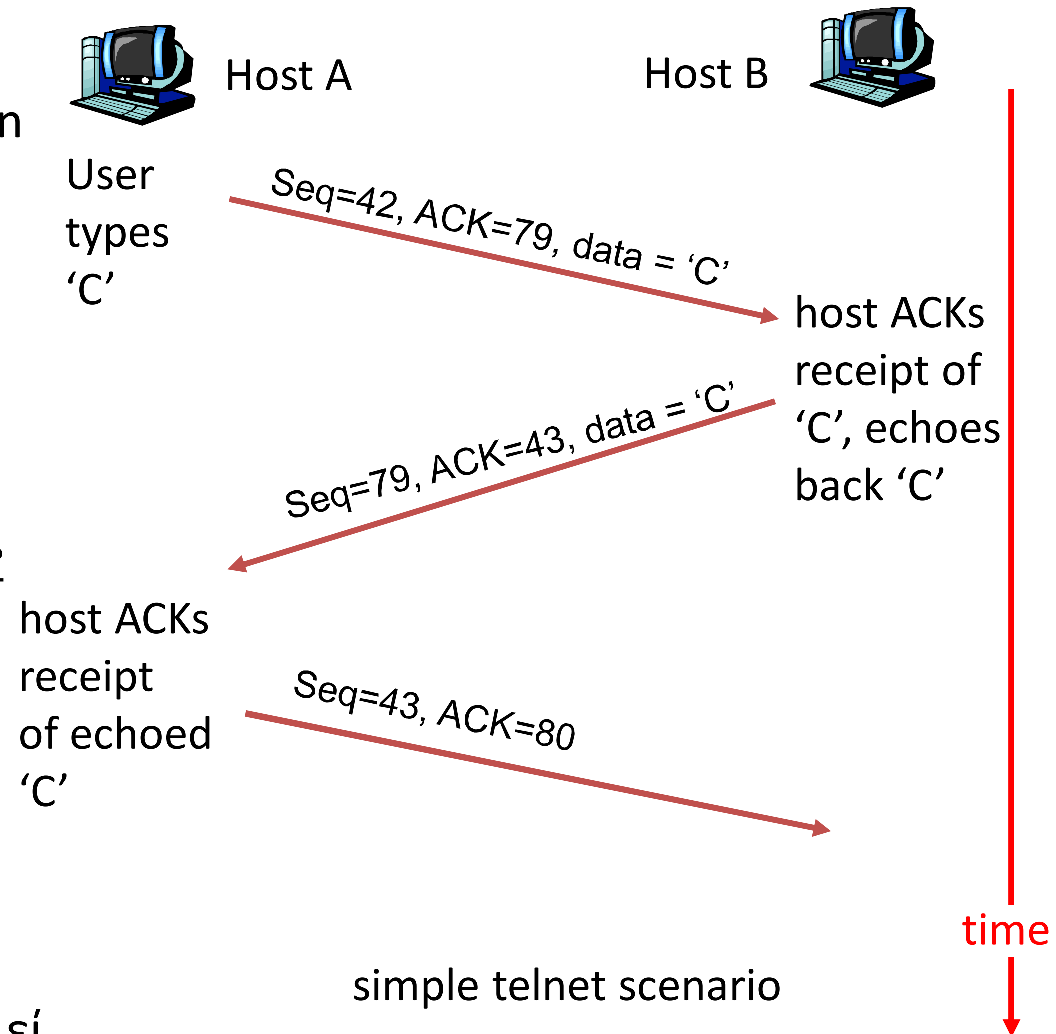
**που αναμένεται από την άλλη πλευρά**

- ***Υπάρχουν και τα cumulative* ACK**

Q: πώς χειρίζεται ο παραλήπτης πακέτα που έρχονται σε λανθασμένη σειρά?

A: TCP δεν προσδιορίζει τον τρόπο. Το αφήνει στον προγραμματιστή που υλοποιεί

τη συγκεκριμένη έκδοση




simple telnet scenario

# TCP: Επισκόπηση (2/4)

ΟιTCP ***έλεγχοι*** ***συμφόρησης*** **&** ***ροής*** θέτουν το μέγεθος παραθύρου

**Πολλά πακέτα** μπορούν να σταλθούν ***"ταυτόχρονα"*** και να **μην** γίνουν ACKed μέσα σε ένα χρονικό διάστημα

***Buffers αποστολής & παραλαβής***

Ελεγχόμενη ροή: αποστολέας δεν θα "κατακλύσει" τον παραλήπτη

***LastByteSent - LastByteAcked*** <= min (***CongWin, RcvWindow***)

**Παράμετροι που πρσδιορίζονται…**

***Ποιά είναι η επίδραση μικρών τιμών στα CongWin*** & ***RcvWindow***?

**Πώς επηρεάζουν τον ρυθμό αποστολής των δεδομένων ?**

# TCP: Επισκόπηση

ΟιTCP **έλεγχοι συμφόρησης & ροής** θέτουν το μέγεθος παραθύρου

**Πολλά πακέτα** μπορούν να έχουν σταλθεί ***"ταυτόχρονα"*** και να μην έχουν γίνει ACKed

Ελεγχόμενη ροή: αποστολέας ***δεν*** θα "κατακλύσει" τον παραλήπτη

Ο αποστολέας ελέγχει το παρακάτω:

***LastByteSent - LastByteAcked*** <= min (***CongWin, RcvWindow***)

**Η ροή αποστολής ελέγχεται από αυτές τις δύο παραμέτρους**

**αναλύει «δικτυακά» δεδομένα, προσπαθεί να συμπεράνει για τις δικτυακές συνθήκες, και αποφασίζει για τις τιμές αυτών των παραμέτρων.**

**Ερώτηση: πώς καθορίζει την κατάσταση του δικτύου?**

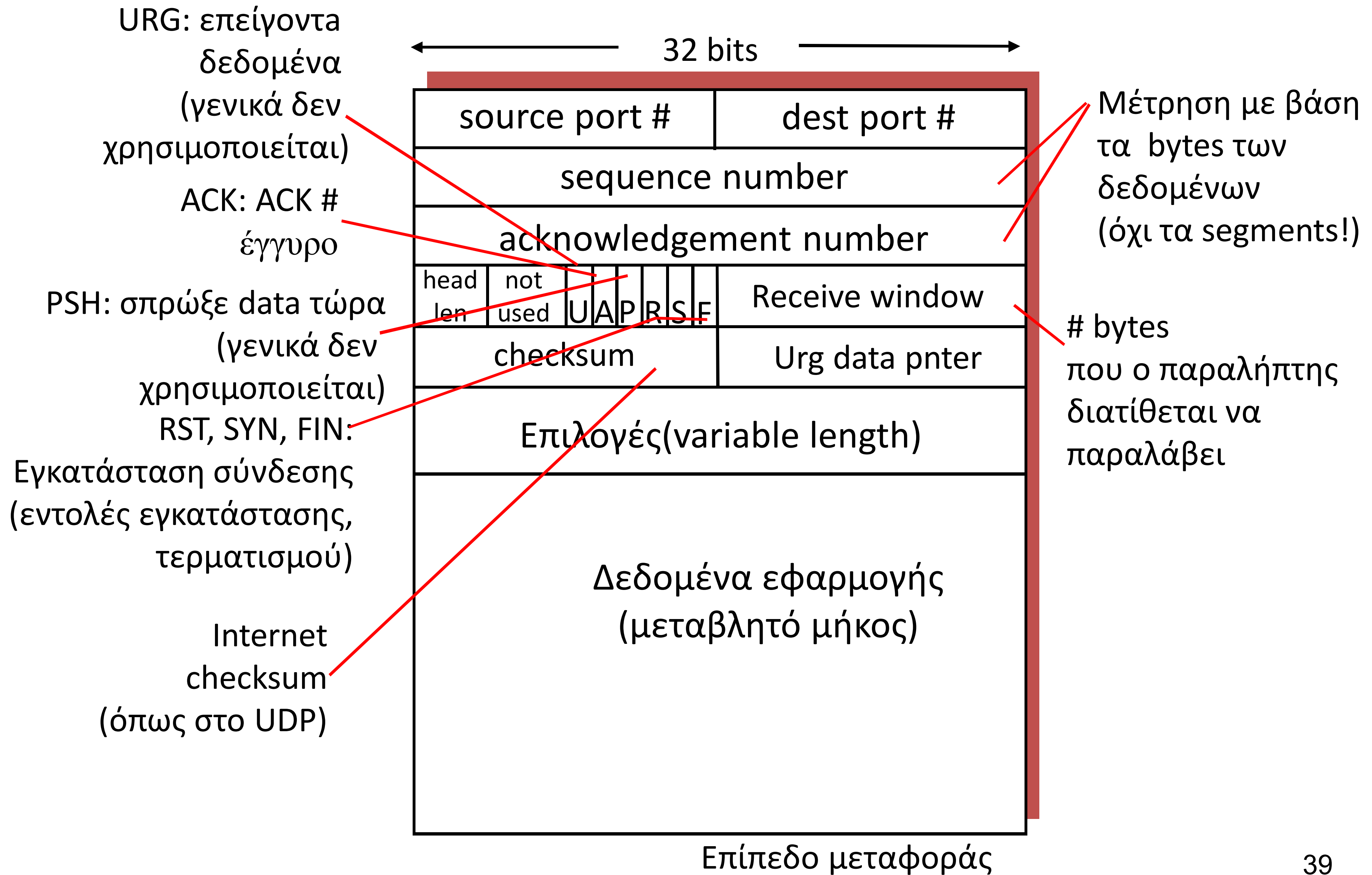
Προσπαθεί να τη **μαντεύσει** μετρώντας τις απώλειες των πακέτων & καθυστερήσεις!!!

# TCP: Επισκόπηση (4/4)

Three important concepts:

- **Additive-increase, multiplicative-decrease**
- **Slow start**
- **Reaction to timeouts**

# Δομή TCP segment



# Sequence Number Space

Κάθε byte στη ροή των bytes είναι αριθμημένο

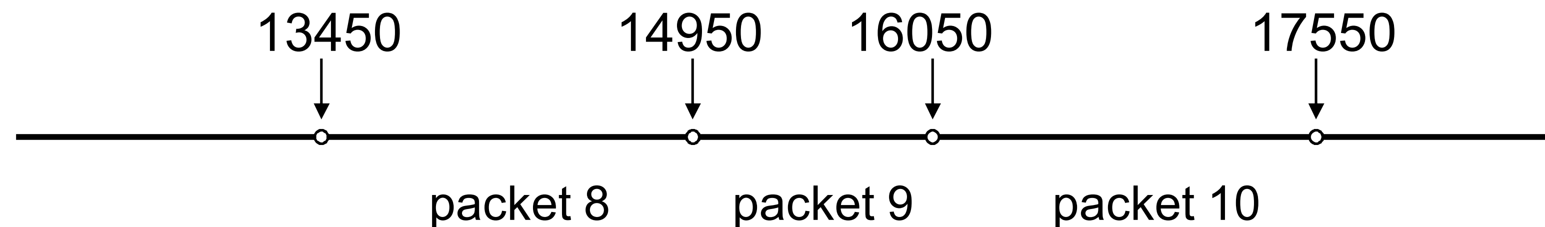
- 32 bit τιμή
- Κάνει wrap around
- Η αρχική τιμή επιλέγεται τη στιγμή εκκίνησης

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**



# Σημαντικές έννοιες του TCP

- Σημαντική παράμετρος: το **Παράθυρο συμφόρησης** (Congestion window)
  - ☞ **<u>Δυναμική</u>** ρύθμιση του μεγέθους του κατά τη διάρκεια της μετάδοσης των πακέτων της ροής (adaptation of its size)
- Υπάρχει κι άλλη μια παράμετρος που λέγεται **Threshold**

  Και δείχνει πότε σταματά η slow start φάση & ξεκινά η congestion avoidance φάση
- **Timeouts**
  - 💣 **<u>not</u>** receiving the ACK of a packet within a time interval

    Estimating RTT
- Συμπεραίνει τις απώλειες των πακέτων (Inferring packet loss)
- **Slow start**
- **Fast retransmission**

# TCP seq. #'s and ACKs (1/2)

Seq. #'s:

«Αριθμός» ροής byte του **πρώτου byte** στα δεδομένα του segment

ACKs:

- **seq # του *επόμενου byte που αναμένεται* από την άλλη πλευρά**
- **συσσωρευτικό (cumulative) ACK**

# TCP seq. #'s and ACKs (2/2)

Q: Πώς διαχειρίζεται ο παραλήπτης segments που φτάνουν με ***λάθος σειρά***

A: Δεν περιγράφεται στις «προδιαγραφές» TCP – επιλέγεται κατά βούληση σε κάθε υλοποίηση

Ο παραλήπτης έχει τις παρακάτω δύο γενικές επιλογές:

1. **αμέσως "*πετά*" τα segments που έφτασαν με *λάθος σειρά***, ή
2. "**κρατά**" τα segments που ήρθαν ***με λάθος σειρά*** και **περιμένει τα λάβει πακέτα με τα bytes που "χάθηκαν/δεν έφτασαν" ώστε να καλύψει τα κενά**

# TCP σύνδεση: χειραψία σε 3 βήματα

<u>Βήμα 1:</u> ο client host στέλνει το***TCP SYN segment*** στον server

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

<u>Βήμα 2:</u> ο server host λαμβάνει το SYN, απαντάει με ***SYNACK segment***

**O server δεσμεύει buffers !**

Προσδιορίζει τον αρχικό αριθμό σειράς

<u>Βήμα 3:</u> 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

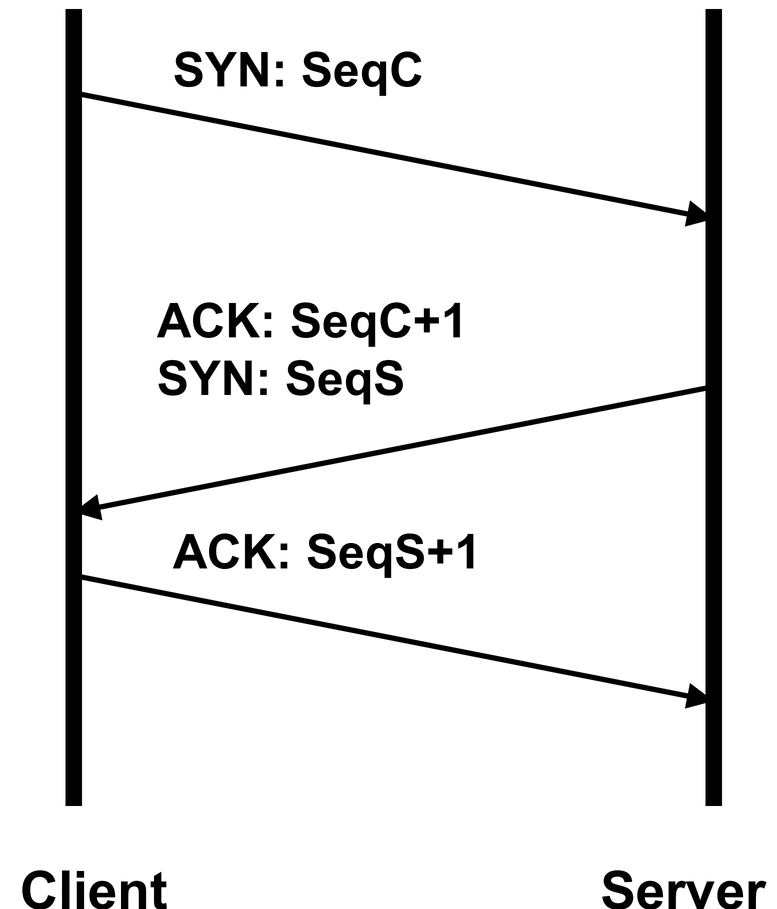
  Γιατί να μην επιλέξουμε απλά το 0;

  - 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 τουTCP :
  **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: o client host στέλνει τοTCP SYN segment στον server

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

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

- **O server δεσμέυει buffers**
- Προσδιορίζει τον αρχικό αριθμό σειράς

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

# Διαχείριση TCP σύνδεσης (συνέχεια)

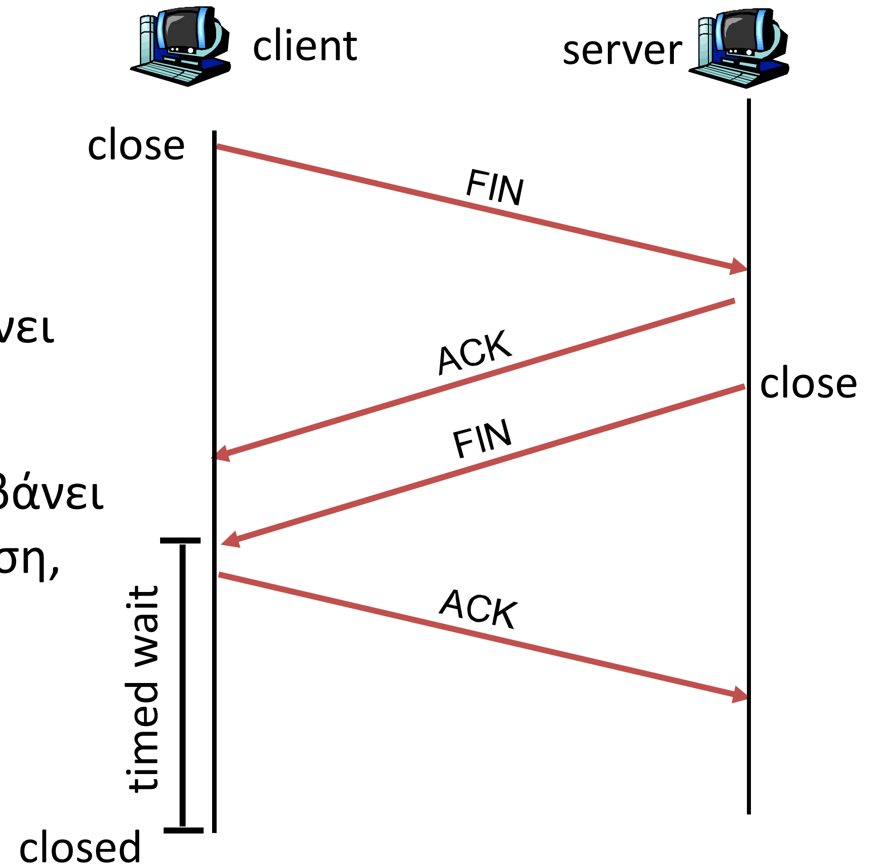
<u>Κλείσιμο μίας σύνδεσης:</u>

O client κλείνει το socket:

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

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

<u>Βήμα 2:</u> το τερματικό σύστημα του server λαμβάνει το ***FIN***, απαντάει με ***ACK***. Κλείνει την σύνδεση, **στέλνει** ***FIN***



# Διαχείριση TCP σύνδεσης (συνέχεια)

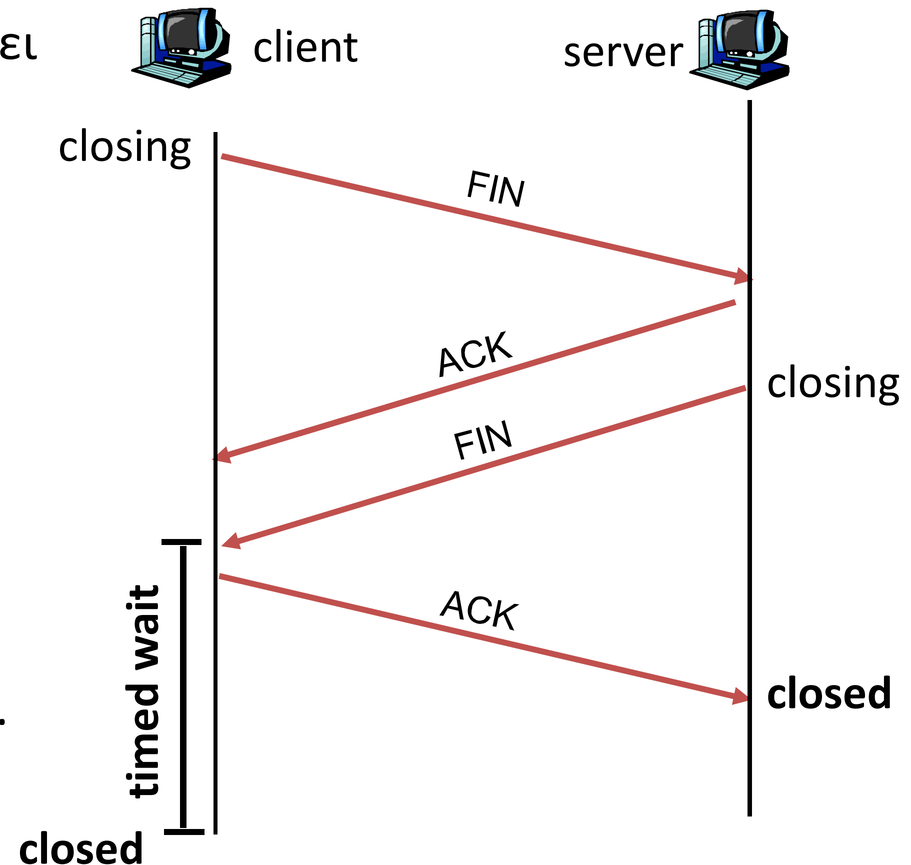
<u>Βήμα 3:</u> ο client λαμβάνει το ***FIN***, απαντάει με ***ACK***

Μπαίνει σε ***χρονισμένη αναμονή*** - θα απαντήσει με ***ACK*** στα ***FIN***s που λαμβάνει

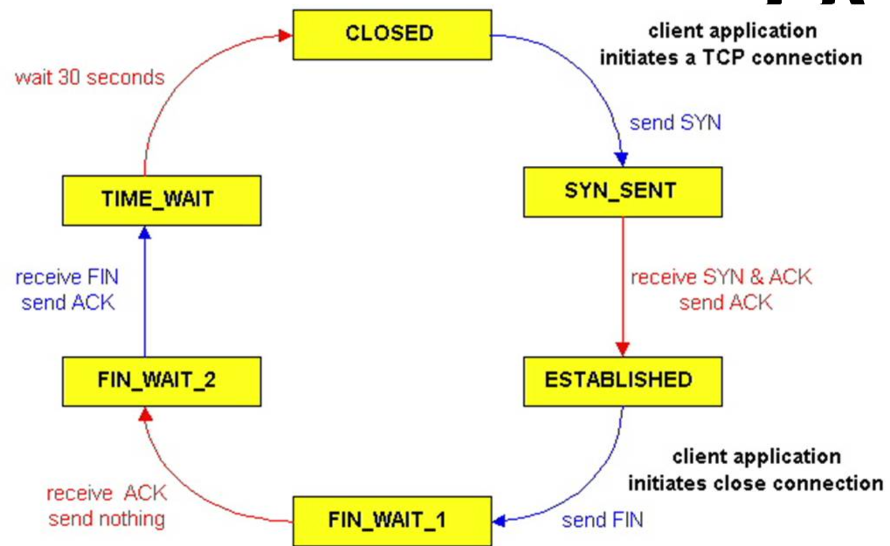
<u>Βήμα 4:</u> ο server, λαμβάνει το ACK.

Η σύνδεση έκλεισε.

<u>Σημείωση:</u> με μικρές μετατροπές, μπορεί να γίνει διαχείριση ταυτόχρονων FINs.

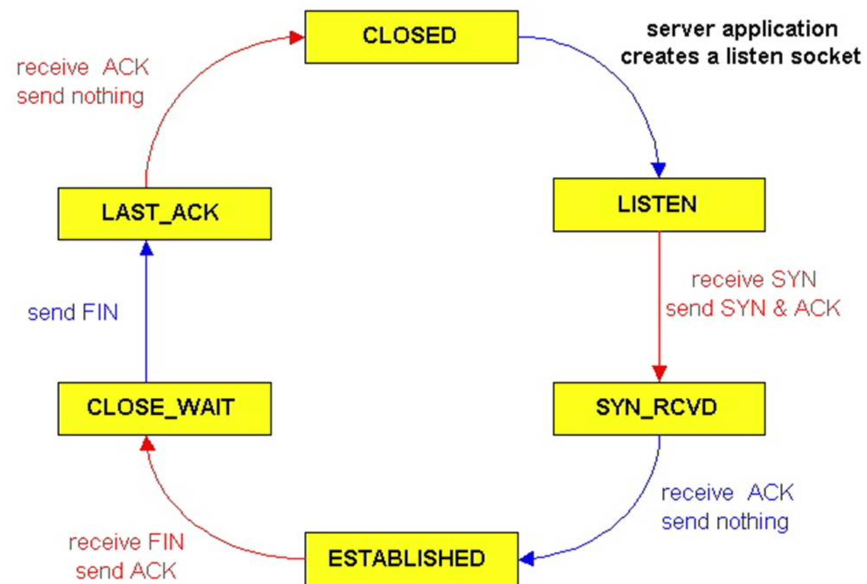


# Διαχείριση TCP σύνδεσης(συνέχεια)




Κύκλος ζωής TCP client

Κύκλος ζωής TCP server



# Παράδειγμα εγκαθίδρυσης TCP σύνδεσης 1

```
09:23:33.042318 IP 128.2.222.198.3123 > 192.216.219.96.80: S
  4019802004:4019802004(0) win 65535 <mss 1260,nop,nop,sackOK> (DF)

09:23:33.118329 IP 192.216.219.96.80 > 128.2.222.198.3123: S
  3428951569:3428951569(0) ack 4019802005 win 5840 <mss 1460,nop,nop,sackOK> (DF)
```

```
09:23:33.118405 IP 128.2.222.198.3123 > 192.216.219.96.80: . ack
  3428951570 win 65535 (DF)
```

# Παράδειγμα εγκαθίδρυσης TCP σύνδεσης 2

Client SYN

- SeqC: Seq. #4019802004, window 65535, max. seg. 1260

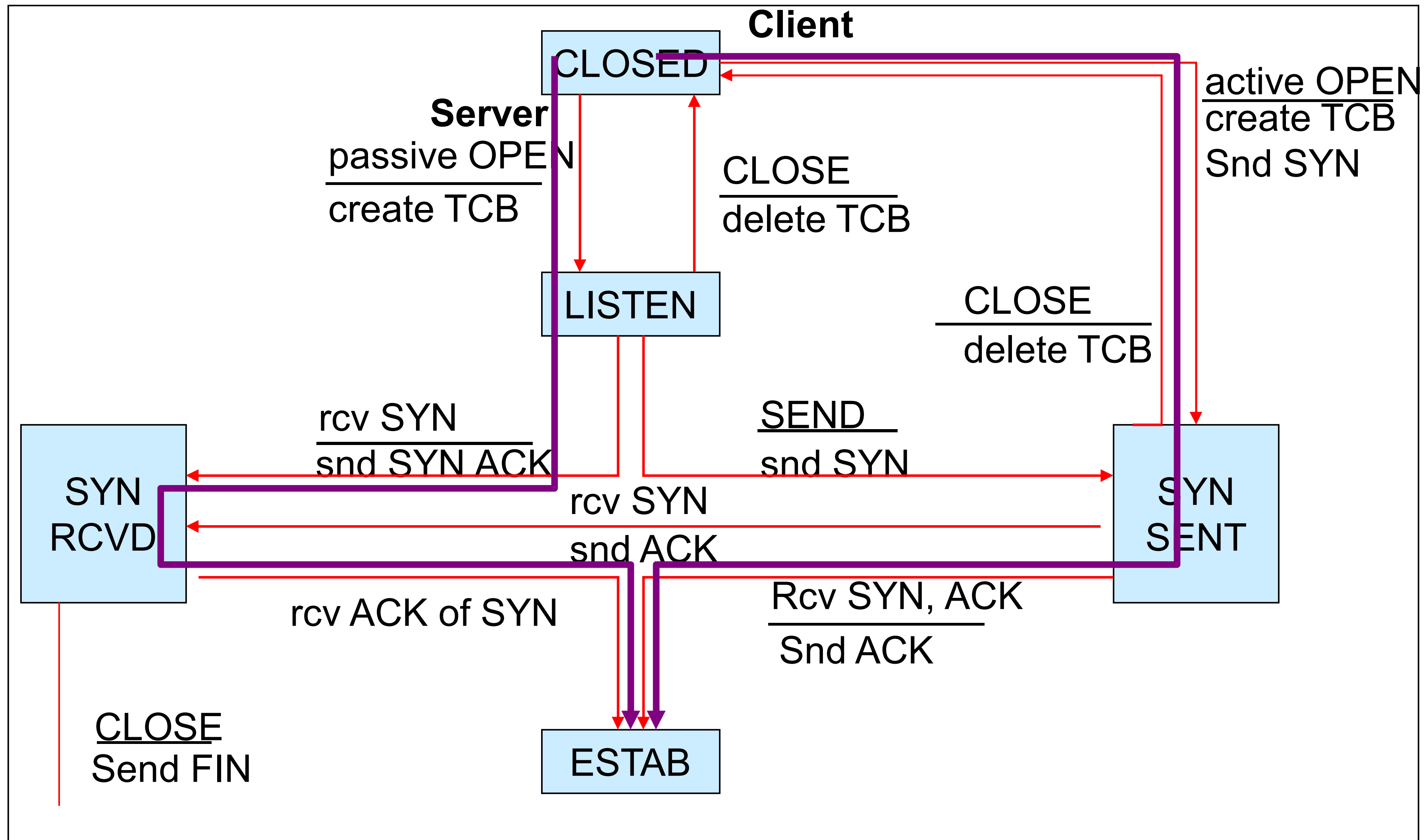
Server SYN-ACK+SYN

- λαμβάνει: #4019802005 (= SeqC+1)
- SeqS: Seq. #3428951569, window 5840, max. seg. 1460

Client SYN-ACK

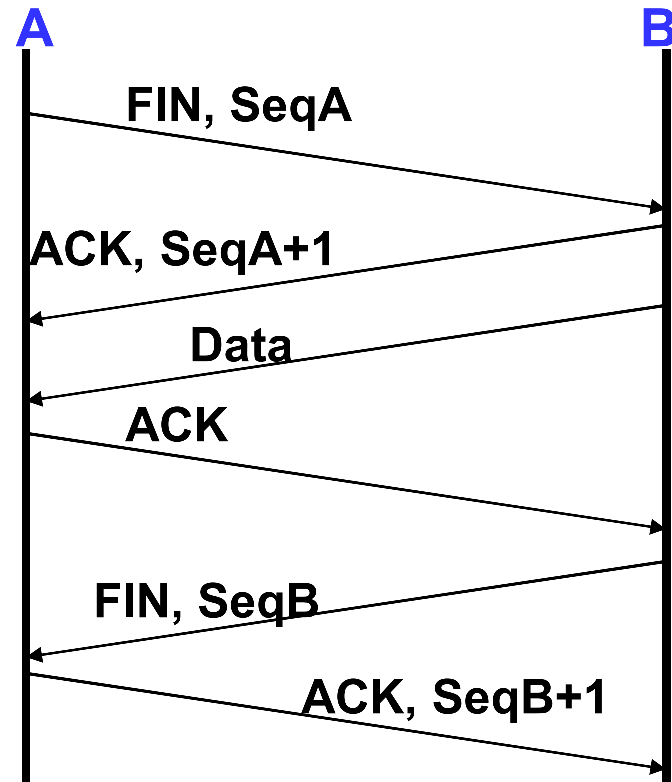
- λαμβάνει: #3428951570 (= SeqS+1)

# Διάγραμμα κατάστασηςTCP: εγκαθίδρυσης σύνδεσης



# Κλείσιμο σύνδεσης

- ***Οποιαδήποτε πλευρά*** μπορεί να ξεκινήσει το **κλείσιμο της σύνδεσης**
  - Στέλνει ***FIN***
  - "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



# Παράδειγμα κλεισίματος TCP σύνδεσης 1

```
09:54:17.585396 IP 128.2.222.198.4474 >
128.2.210.194.6616: F
  1489294581:1489294581(0) ack 1909787689 win 65434 (DF)

09:54:17.585732 IP 128.2.210.194.6616 >
128.2.222.198.4474: F
  1909787689:1909787689(0) ack 1489294582 win 5840 (DF)
```

```
09:54:17.585764 IP 128.2.222.198.4474 >
128.2.210.194.6616: . ack
  1909787690 win 65434 (DF)
```

# Παράδειγμα κλεισίματος TCP σύνδεσης 2

Session

Echo client on 128.2.222.198, server on 128.2.210.194

Client FIN

SeqC: 1489294581

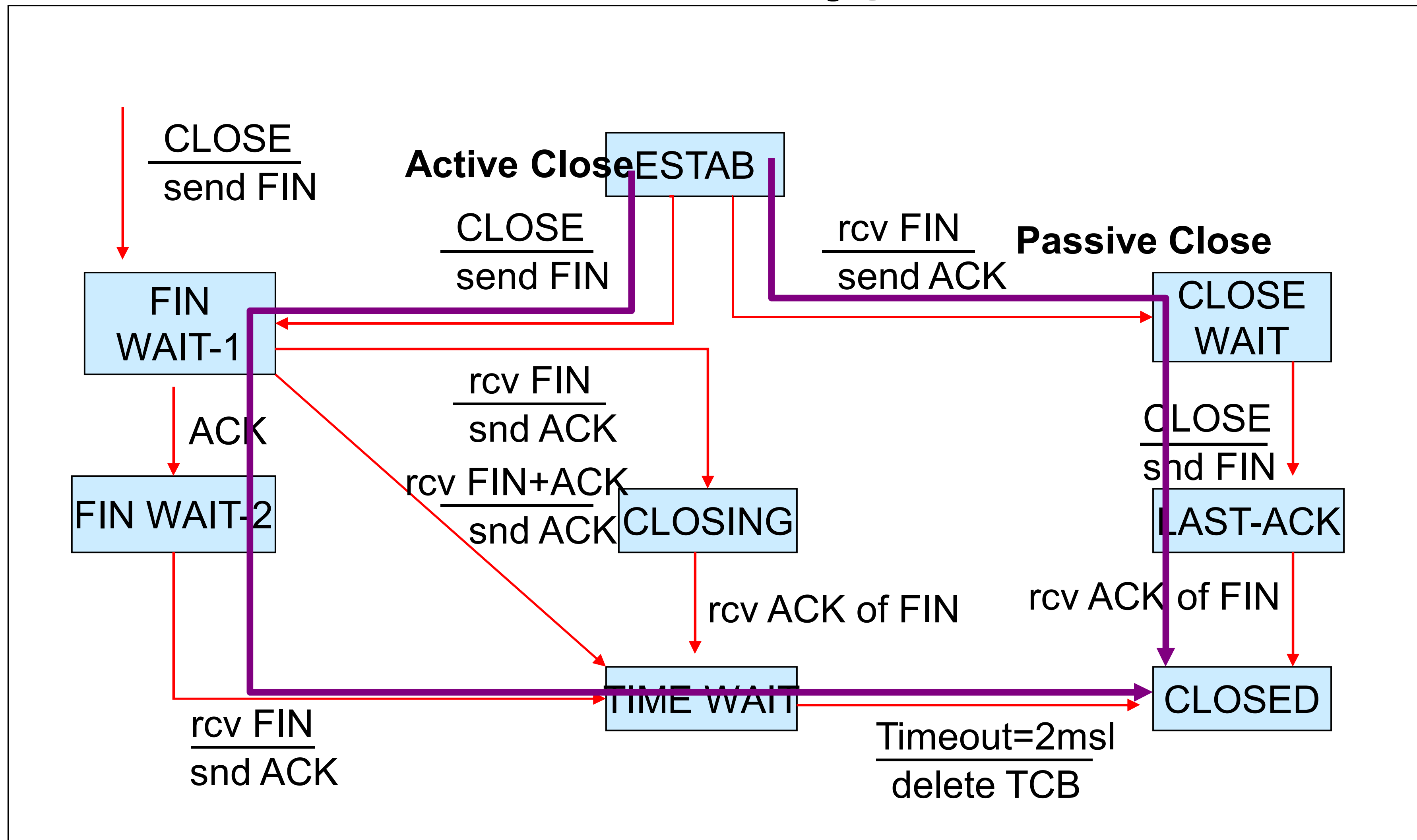
Server ACK + FIN

- Ack: 1489294582 (= SeqC+1)
- SeqS: 1909787689

Client ACK

Ack: 1909787690 (= SeqS+1)

# Διάγραμμα κατάστασης:Κλείσιμο σύνδεσης



# TCP Timeout

- Μηχανισμός που καθορίζει **πόσο να *περιμένει*** ***o αποστολέας*** μέχρι να ***ξαναστείλει το πακέτο***
- O 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***

# Sliding window of TCP

Το TCP είναι ένα πρωτόκολλο ***κυλιόμενου παραθύρου*** (***sliding window***

Αποστολέας:

Για ***μέγεθος παραθύρου n***, μπορεί να στείλει ***έως*** **και *n* bytes χωρίς να λάβει επιβεβαίωση**

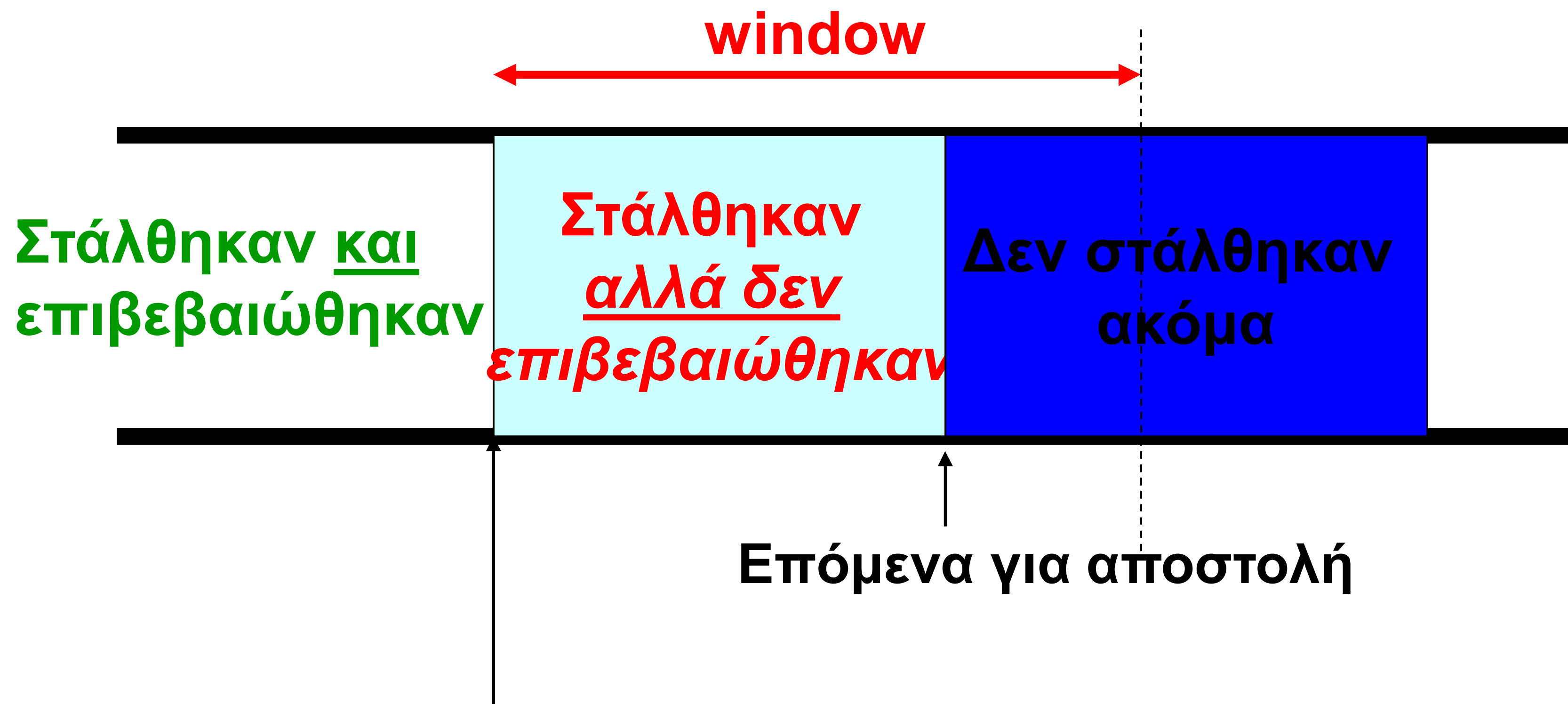
Όταν τα δεδομένα επιβεβαιωθούν τότε το παράθυρο μετακινείται προς τα μπρος

Παραλήπτης:

Στο κάθε πακέτο "σημειώνεται" το **μέγεθος παραθύρου, δηλαδή ο αριθμός των bytes για τα οποία έχει χώρο ο παραλήπτης**

# Έλεγχος ροής με παράθυρο

## Αποστέλλουσα πλευρά

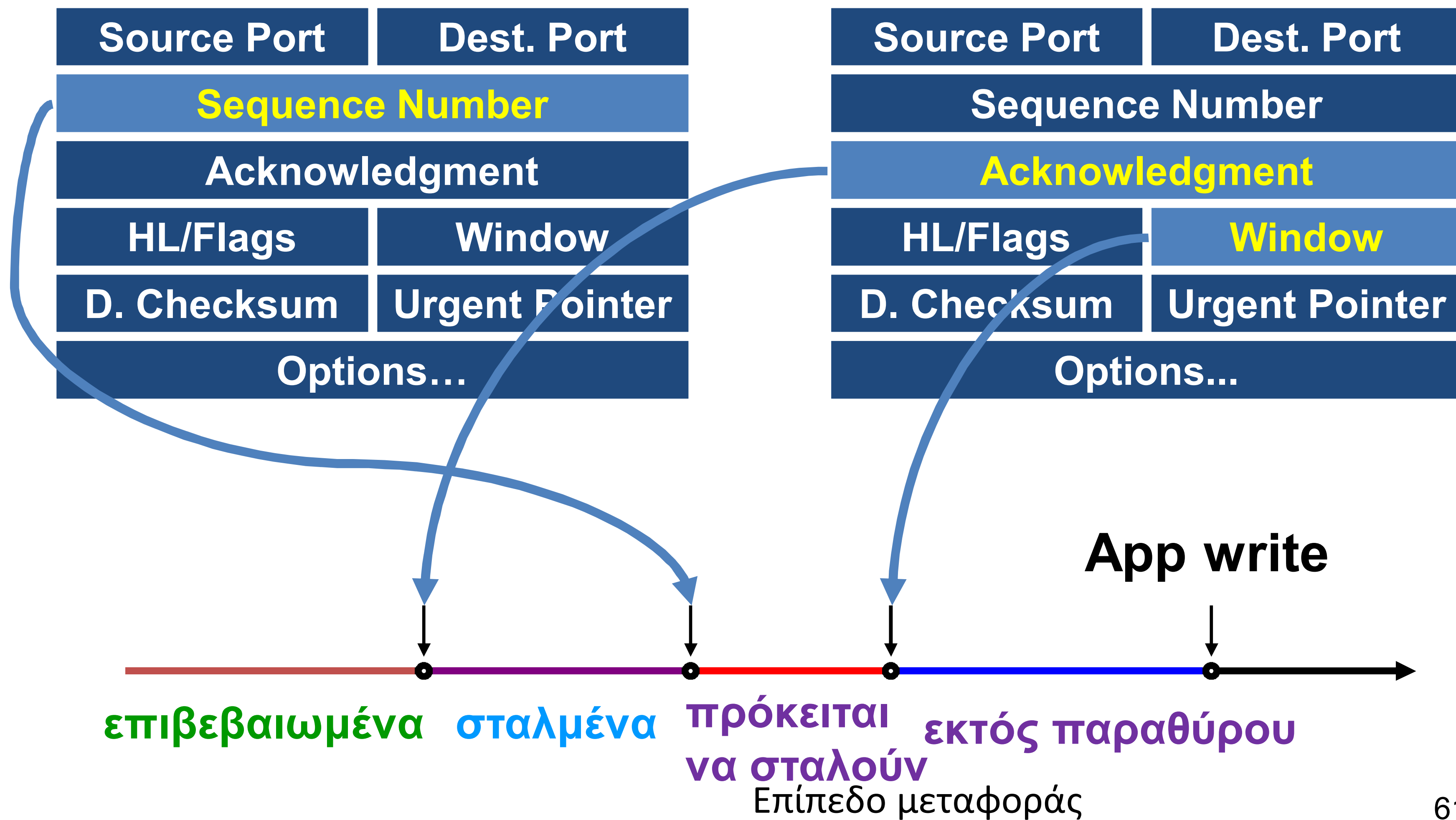


# Μηχανισμός γρήγορης επαναποστολής

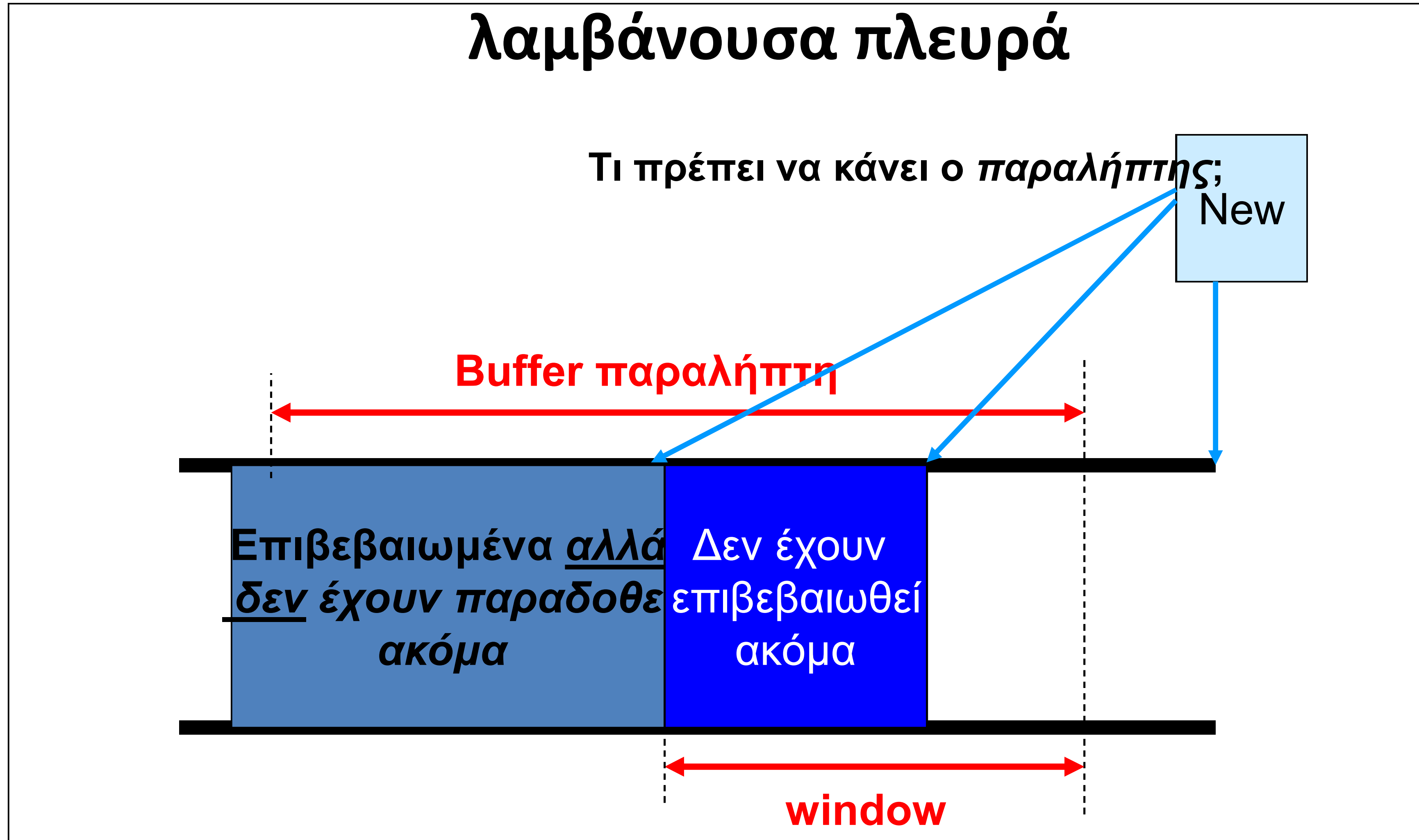
- Το TCP χρησιμοποιεί τα ***sequence numbers*** για να βρει ποια πακέτα έχουν χαθεί

Η παραλαβή **3 ομοίων ACKs** για ένα ***συγκεκριμένο segment*** παίζει το ρόλο ενός "***έμμεσου***" **NACK** (negative ACK – αρνητικής επιβεβαίωσης) για το segment που ακολουθεί, προκαλώντας την επαναποστολή του segment ***πριν γίνει timeout***

# Έλεγχος ροής με παράθυρο: αποστέλουσσα πλευρά



# Έλεγχος ροής με παράθυρο



# TCP Round Trip Time και Timeout

<u>Q:</u> πώς να θέσουμε την **τιμή του TCP timeout**?

Θα πρέπει να είναι μεγαλύτερο από το round-trip-time (**RTT)**

☞ ναι, αλλά το RTT ***ποικίλλει!***

Αν είναι πολύ μικρό … ⇨ **πρόωρο** timeout, που δημιουργεί

☹ επαναποστολές που ***<u>δεν</u>*** είναι απαραίτητες!

Πολύ μεγάλο? ➔ αργή και μικρή ***αντίδραση*** στην απώλεια segment !!!

<u>Q:</u> πώς να υπολογίσουμε το RTT?

**`SampleRTT`**: ο χρόνος που μετρήθηκε από την αποστολή του segment ως την παραλαβή του ACK

Αγνοεί segments που έχουν φτάσει με επαναποστολές

**`SampleRTT`** θα ποικίλλει, θέλουμε "ομαλότερο" το υπολογισμένο RTT

**Βρίσκουμε το μέσο όρο από *τις πρόσφατες* μετρήσεις**

***<u>όχι</u>*** μόνο **το τωρινό `SampleRTT`**

# TCP Round Trip Time και Timeout

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

- Μ.Ο. με εκθετικά βάρη (exponential weighted moving average)
- Η επίδραση των **παλαιότερων δειγμάτων φθίνει εκθετικά**
- Τυπική τιμή: $\alpha = 0.125$

# TCP Round Trip Time και Timeout

## Θέτοντας το timeout

- **EstimtedRTT** συν "περιθώριο ασφαλείας"
  - Μεγάλη μεταβλητότητα στο **EstimatedRTT ->** μεγαλύτερο περιθώριο ασφαλείας
- Πρώτα υπολογίζεται η τυπική απόκλιση του SampleRTT από το EstimatedRTT:

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

(τυπικά, β = 0.25)
```

Μετά η τιμή του χρονικού διαστήματος τίθεται σε:

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

# TCP reliable data transfer

- TCP creates  service on top of **IP's unreliable service**
- pipelined segments (e.g., sending several segments, back-to-back)
- **cumulative acks**
- TCP uses single retransmission timer
- retransmissions are triggered by:
  - **timeout events**
  - **duplicate acks**

# TCP sender events:

data received from application:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
- (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

# TCP sender events:

<u>timeout:</u>

- retransmit segment that caused timeout
- restart timer

<u>ACK packet received:</u>

If acknowledges previously unacked segments

  - update what is known to be acked
  - start timer if there are outstanding segments

```
NextSeqNum = InitialSeqNum
 SendBase = InitialSeqNum

  loop (forever) {
    switch(event)
    event: data received from application above
        create TCP segment with sequence number
lextSeqNum
        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 of y
        if (y > SendBase) {
            SendBase = y
           if (there are currently not-yet-acknowledged
egments)
                 start timer
           }
  }  /* end of loop forever */
```

# TCP sender («βασικός αλγόριθμος»)

Comment:

- SendBase-1:τελευταίο byte για το οποίο επιβεβαιώθηκε συσσωρευτικά η λήψη του

NextSeqNum-1:
το τελευταίο
Byte που μεταδόθηκε συσωρευτικά

# 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?

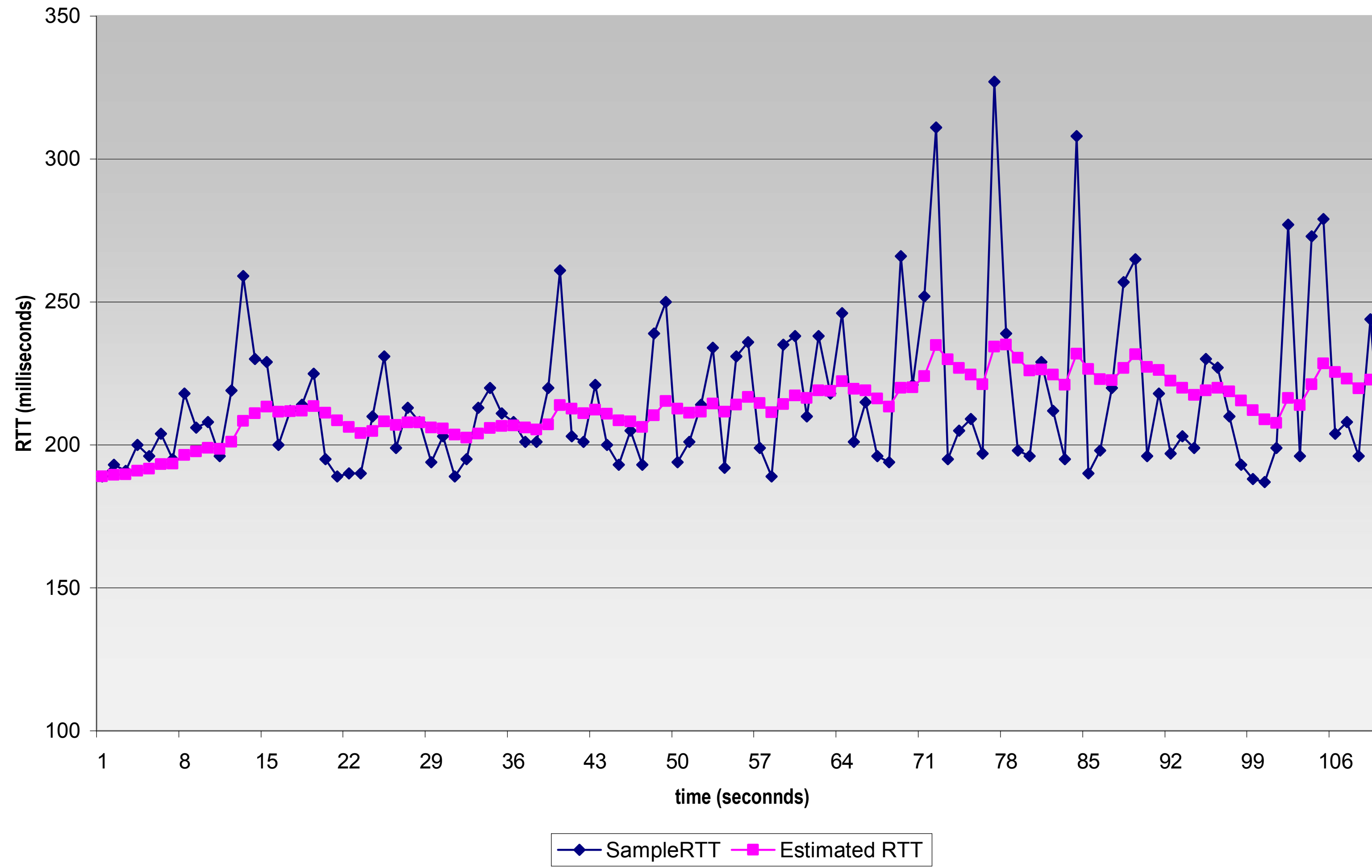
- **`SampleRTT`**: μέτρησε τον χρόνο από την μετάδοση του segment μέχρι τη λήψη του ACK
  - o Αγνόησε τις επαναμεταδόσεις
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - o average several recent measurements, not just current **`SampleRTT`**

# TCP Round Trip Time and Timeout

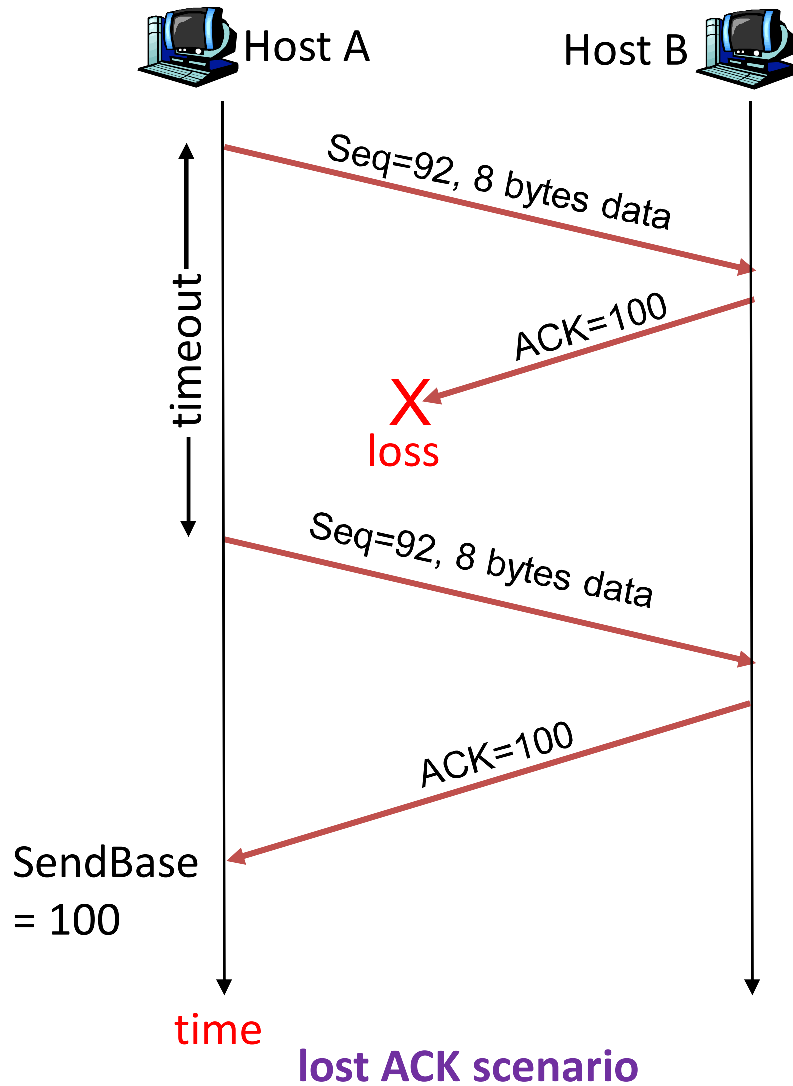
```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

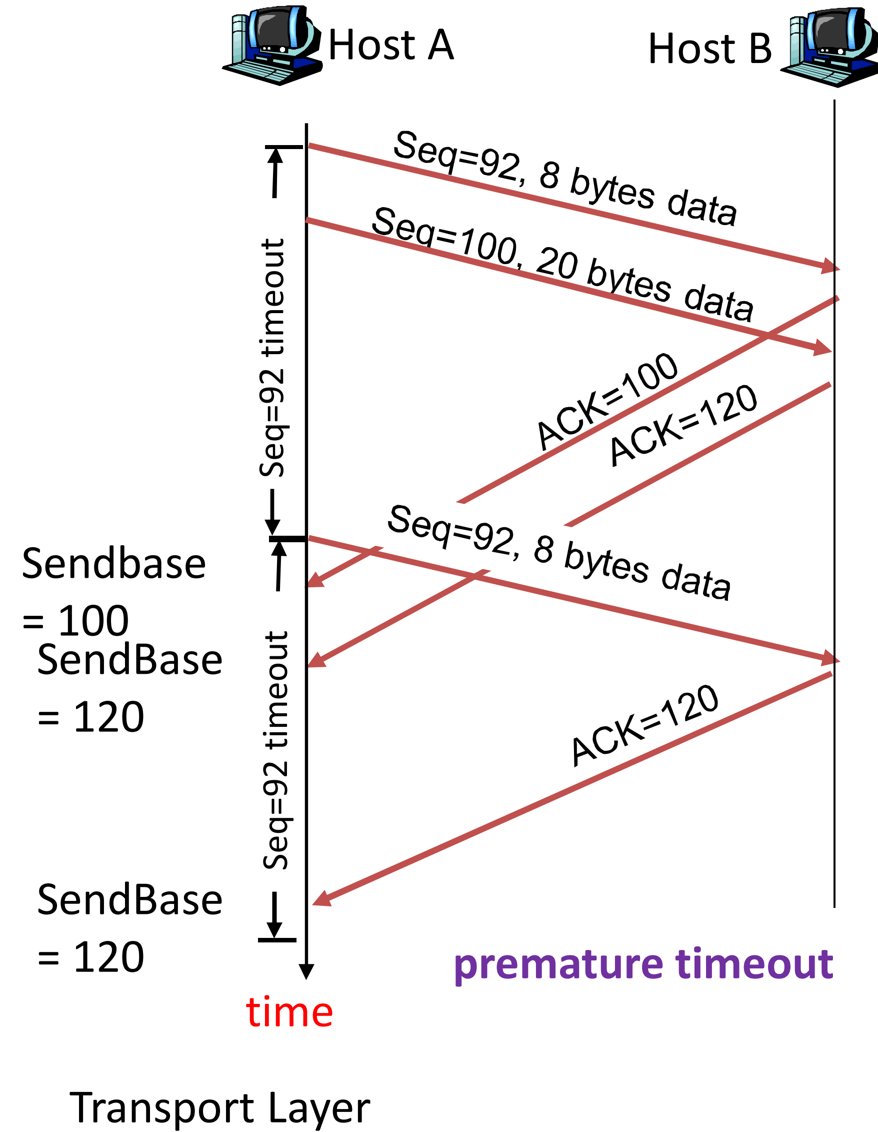
- **Exponential weighted moving average**
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

# Example RTT estimation:

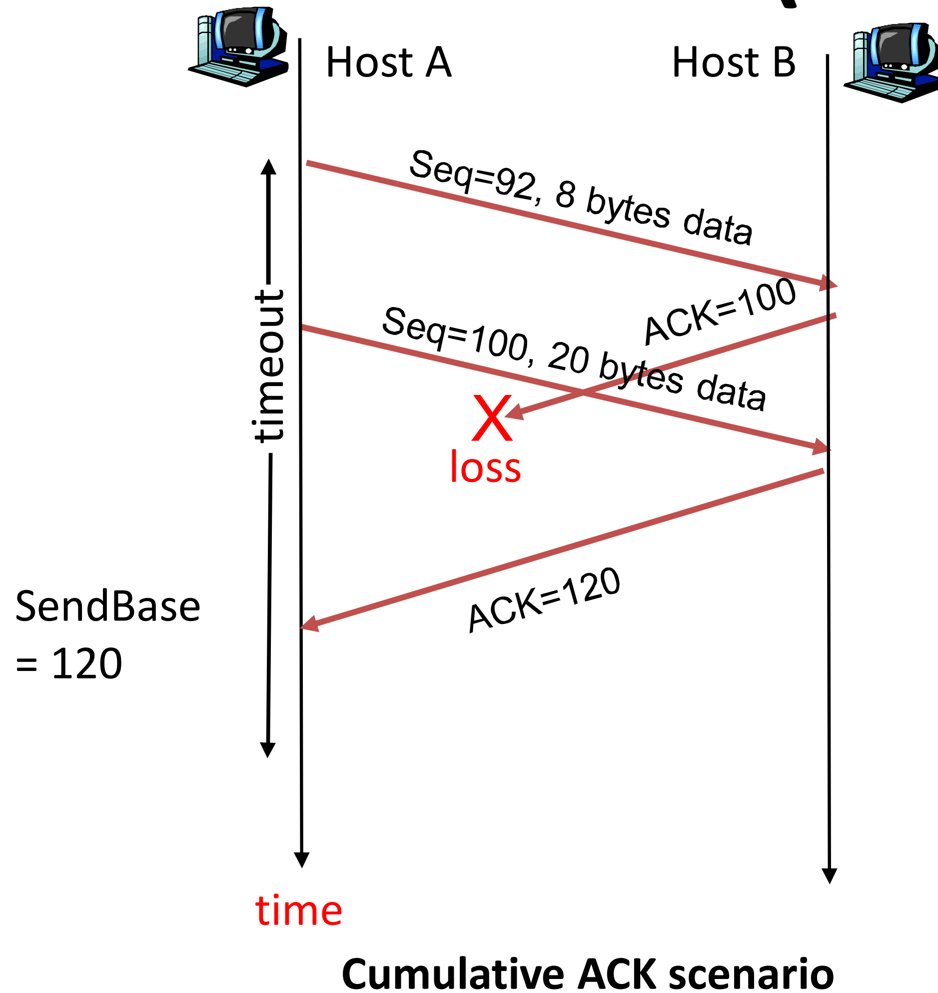


# TCP: retransmission scenarios



# TCP retransmission scenarios (more)




**Cumulative ACK scenario**

# TCP congestion control: control of sending rate

Sender:

- ***Reduces the sending rate*** via reducing the *congestion window*, when *a* ***loss event*** occurs
- ***Increase the sending rate***, when congestion is reduced

***timeout*** or ***3 DUP ACKs***

But how much should a sender reduce its congestion window ?

**.astByteSent- LastByteAcked ≤ min {*Congestion Window, Receive Window*}**

***congestion*** ***flow control***

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

# Κεντρικά χαρακτηριστικά του ελέγχου συμφόρησης του TCP (congestion control)

1. **Γραμμική αύξηση, πολλαπλασιαστική μείωση (additive increase, multiplicative decrease)**:

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

- ***additive increase:*** incr**ease 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)

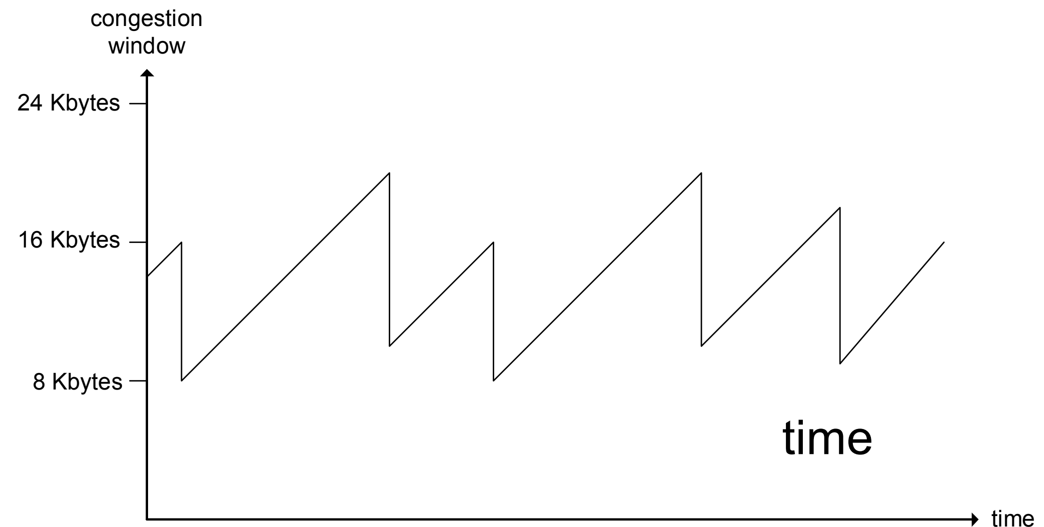
*Approach:* increase transmission rate (window size), probing for usable bandwidth until loss occurs

**congestion avoidance**

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

Using the additive increase
It probes the network to
Check if the congestion has
Been alleviated

congestion window size

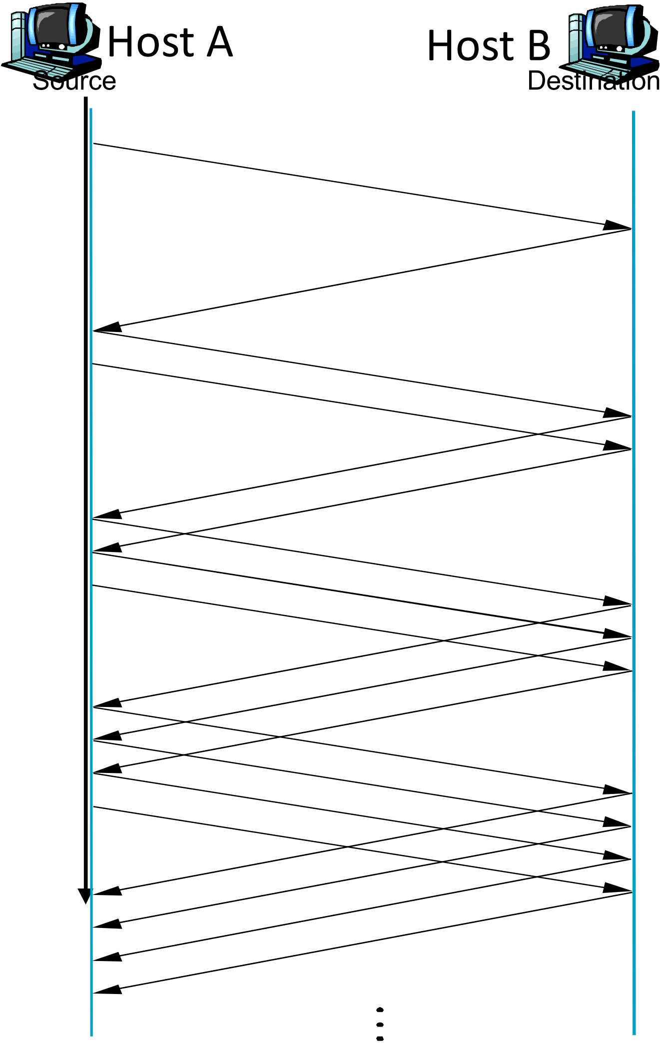



Saw tooth
behavior: probing for bandwidth

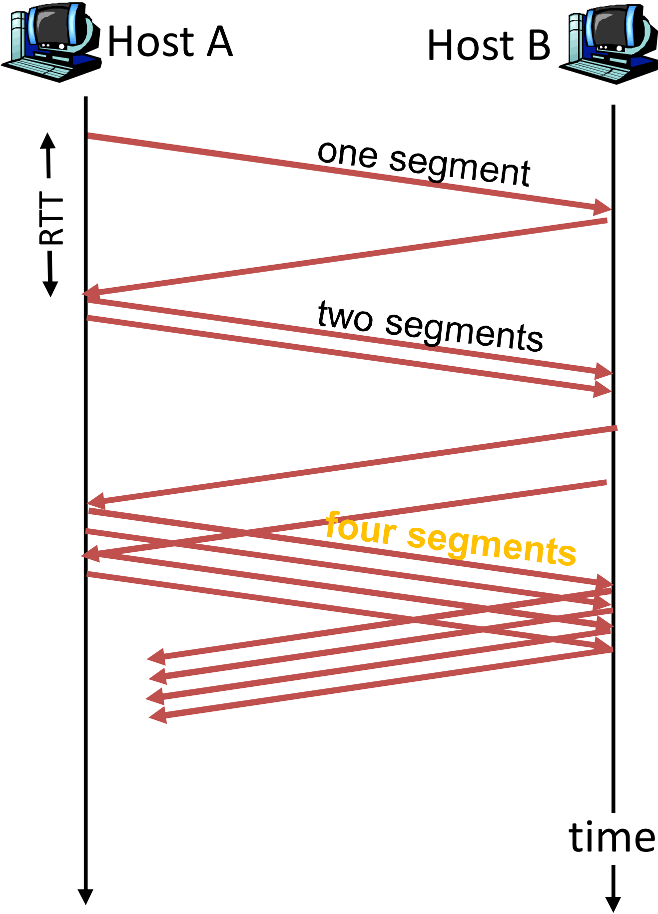
Transport Layer

# How fast the window size of the sender increases affects the TCP sending rate

**Additive Increase**

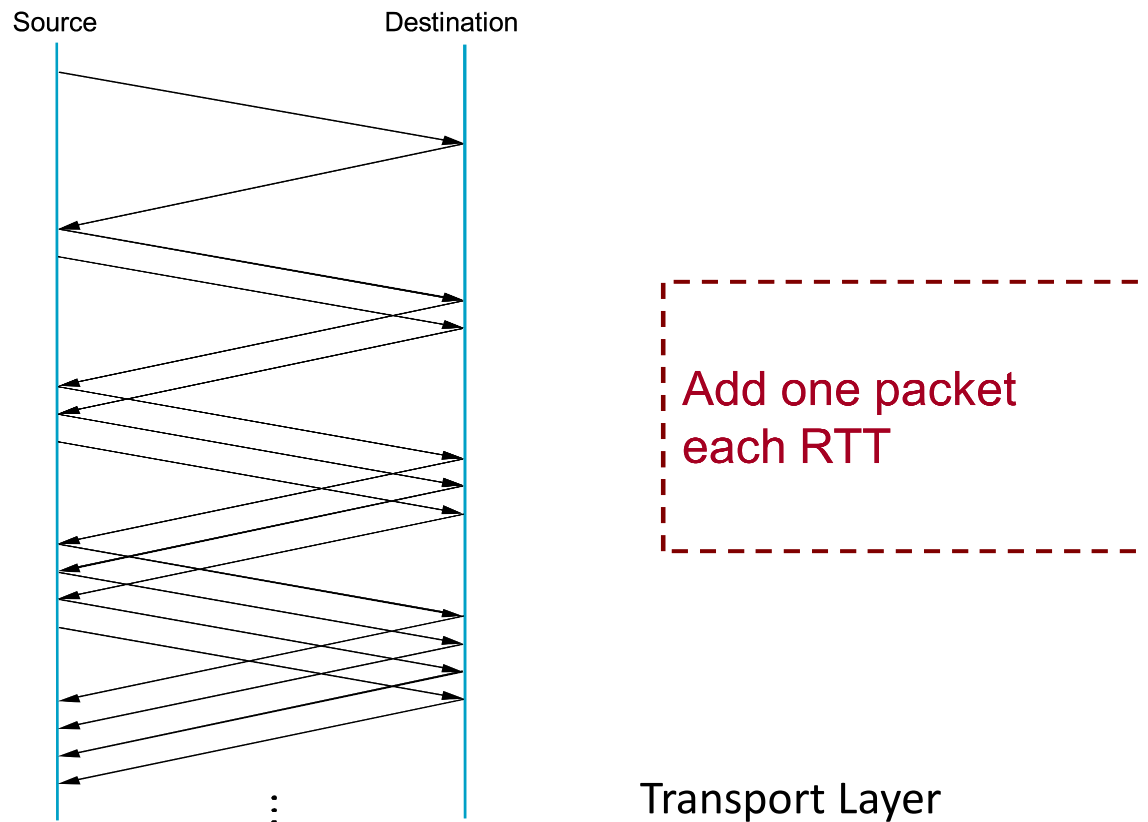



**Exponential Increase**



# Αποφυγή συμφόρησης (Congestion Avoidance)

Η **γραμμική αύξηση** του παραθύρου τουTCP congestion control λέγεται φάση αποφυγής συμφόρησης



# TCP Congestion Control – key ideas

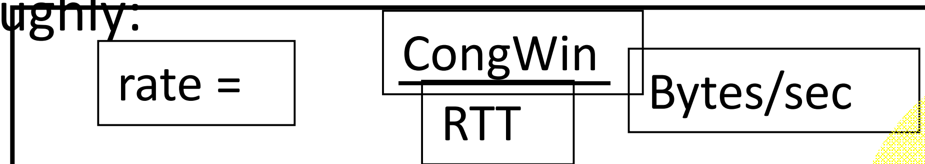
sender limits transmission:

**`LastByteSent-LastByteAcked`**

≤ **`CongWin`**

(για απλοτητα ας «αγνοησουμε» προς στιγμη τον ελεγχο ροης)

Roughly:




**`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 Windo`**) 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)

**Όταν ξεκινά η σύνδεση το** `CongestionWindow` είναι 1 MSS

επειδή το congestion window είναι πολύ μικρό λέγεται "slow start"

- Example: MSS = 500 bytes & RTT = 200 msec
- initial rate = 20 kbps

note: **Ωστόσο το διαθέσιμο bandwidth μπορεί να είναι >>MSS/RTT**

☺ Γι αυτό το λόγο αυξάνουμε γρήγορα το ρυθμό μετάδοσης

Δηλαδή τον αυξάνουμε εκθετικά

Μέχρι όμως την πρώτη απώλεια πακέτου

✓The increase (CongestionWindow++) takes place **at the reception of 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 *MSSS*), and
- sends out 4 maximum sized segments.

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

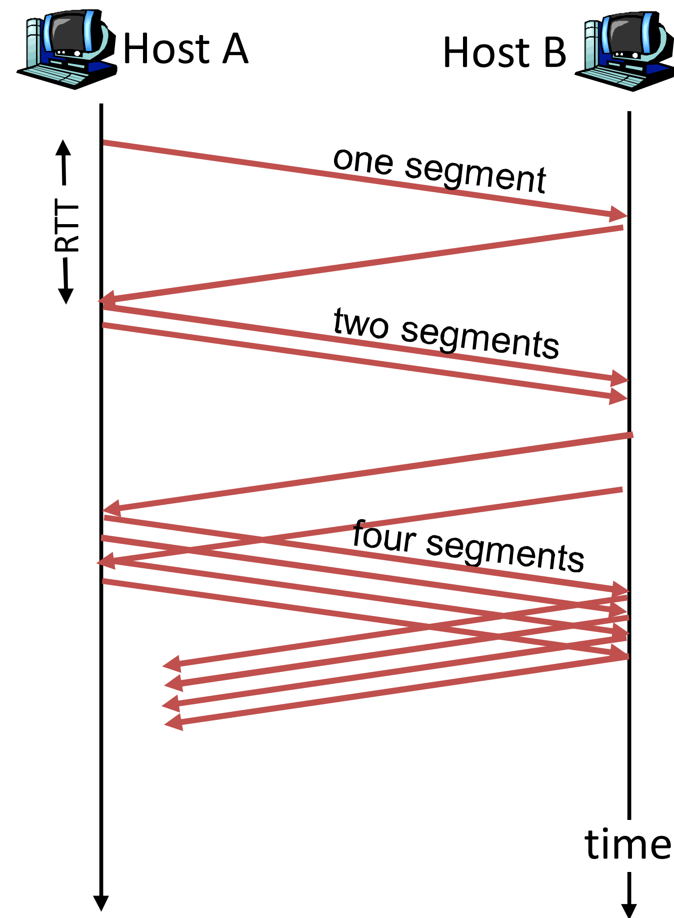
# TCP Slow Start (3/3)

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

**<u>double</u> `CongWin` <u>every</u> RTT**

☞ done by **incrementing `CongWin`** for **every ACK received**

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



# 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**

☞TCP Reno supports Fast Retransmit while TCP Tahoe does not.

# Προσοχή: διαφορετικές περιπτώσεις απώλειας πακέτου

☞ Διαφορετική αντίδραση αν έχουμε ***timeout*** και διαφορετική αν έχουμε ***3 DUPACKs !!!!***

Δικαιολόγηση:

3 dup ACKs (duplicate acknowledgements) δείχνουν μια πιο ήπια κατάσταση συμφόρησης από ότι η περίπτωση του timeout
Γιατί στην πρώτη περίπτωση καταφέρνει ο παραλήπτης και λαμβάνει ***κάποια*** segments

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

# Refinement: inferring loss

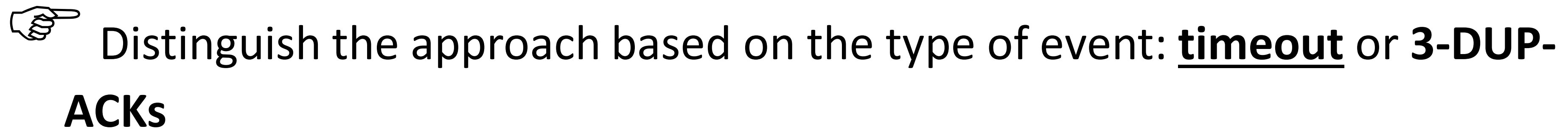
- **Μετά από τη λήψη 3 dup ACKs**:
  - `Congestion window (cwnd)` is cut in half

  Και μετά αυξάνεται γραμμικά
- **Αλλά μετά από timeout**:
  - `To congestion window (cwnd`) γίνεται 1 MSS
  - Αυξάνει ***εκθετικά*** μέχρι ένα ***threshold*** (που είναι ίσο με το μισό όσο ήταν πριν το timeout), και μετά αυξάνει ***γραμμικά***

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a “more alarming” congestion scenario

# Reaction to Congestion Events



Distinguish the approach based on the type of event: **timeout** or **3-DUP-ACKs**

**In the case of 3-DUP-ACKs**:

– **Congestion window /** ***2 and then increase linearly***

**In the case of timeout:**

The sender enters the ***slow-start phase!!!***

– **Congestion window =** ***1MSS***

– ***exponential*** **increase until the congestion window reaches half the value it had before the timeout**, and then

– continue with ***linear*** **increase** (as in the case of 3-DUP-ACK)

# TCP Congestion Control

Threshold: determines the **window size** at which the
***slow start will end*** and the ***congestion avoidance will begin***

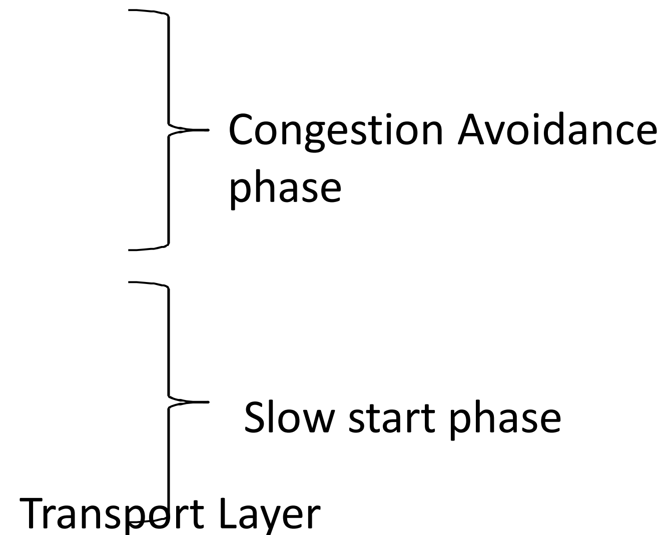
- When **`CongWin <= Threshold:`**

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

- When ***`CongWin > Threshold`*:**

  sender in ***congestion-avoidance*** **phase**, congest. window grows ***linearly***

- When **triple duplicate ACK** occurs:
  - **`Threshold=CongWin/2`**
  - **`CongWin=Threshold`**
- When **timeout** occurs:
  - Threshold=CongWin/2
  - CongWin= 1 MSS

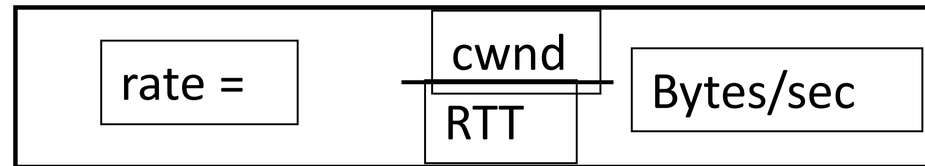


# TCP CongestionControl: details

sender limits transmission:

**`LastByteSent-LastByteAcked`** $\leq$ **`cnwd`**

roughly,



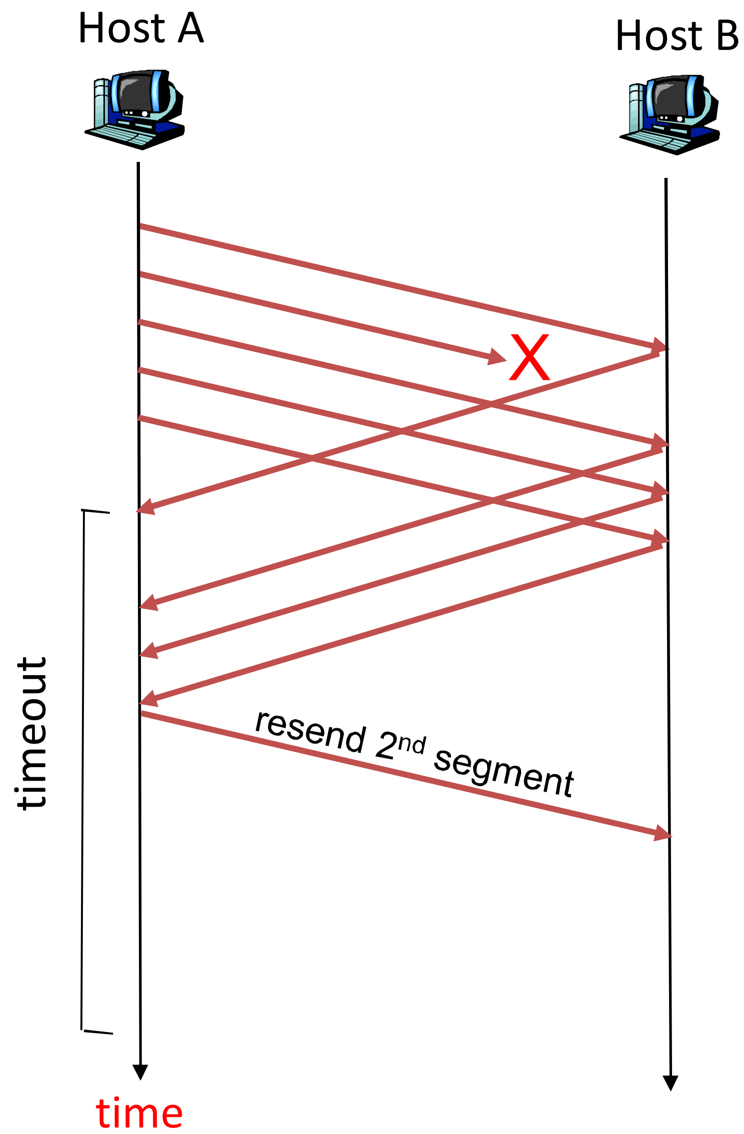

**`cwnd`** is dynamic, function of perceived network congestion

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (**`cwnd`**) after loss event

three mechanisms:

- AIMD
- slow start
- conservative after timeout events




Resending a segment after triple duplicate ACK

# 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

- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management

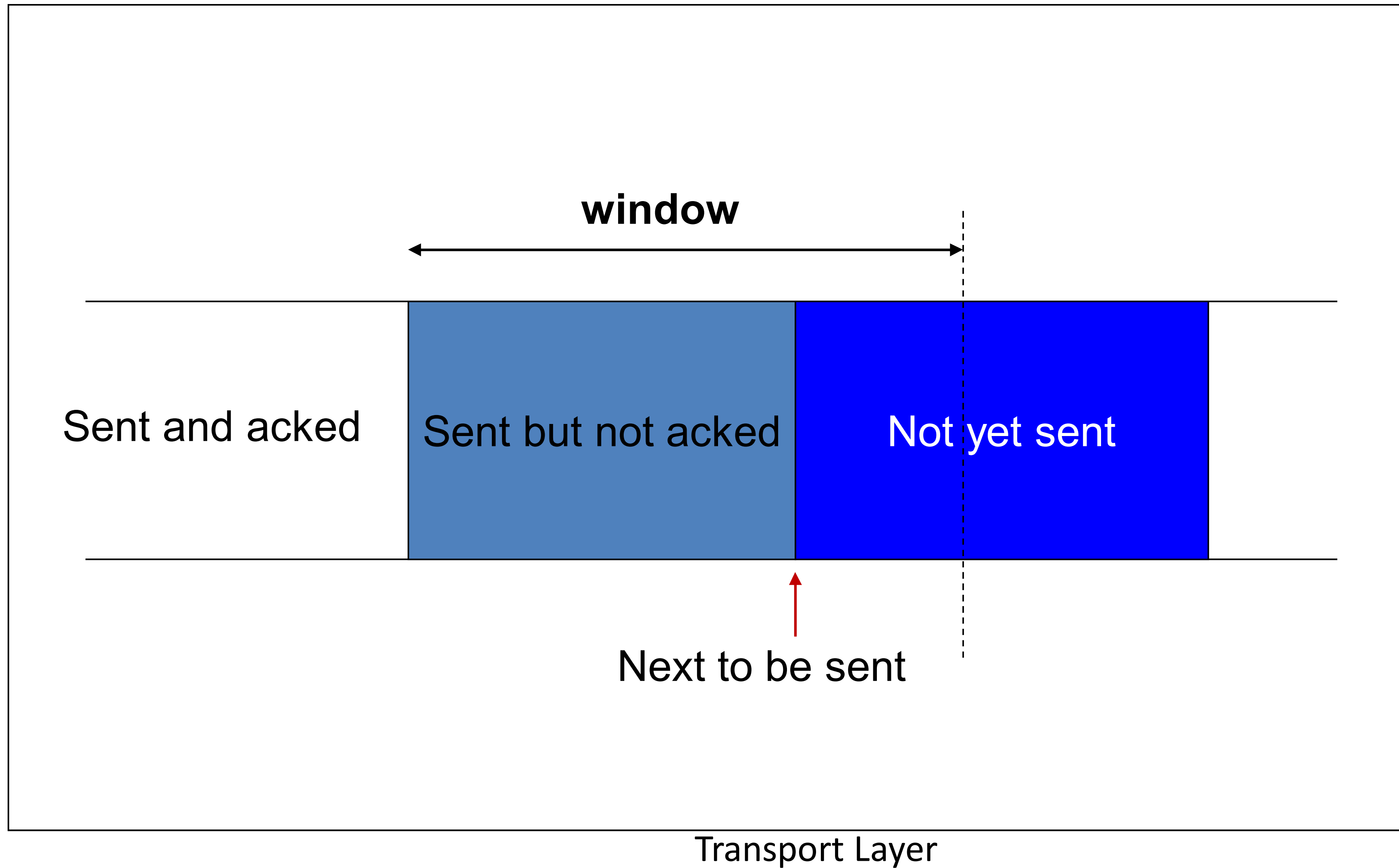
3.6 Principles of congestion control

3.7 TCP congestion control

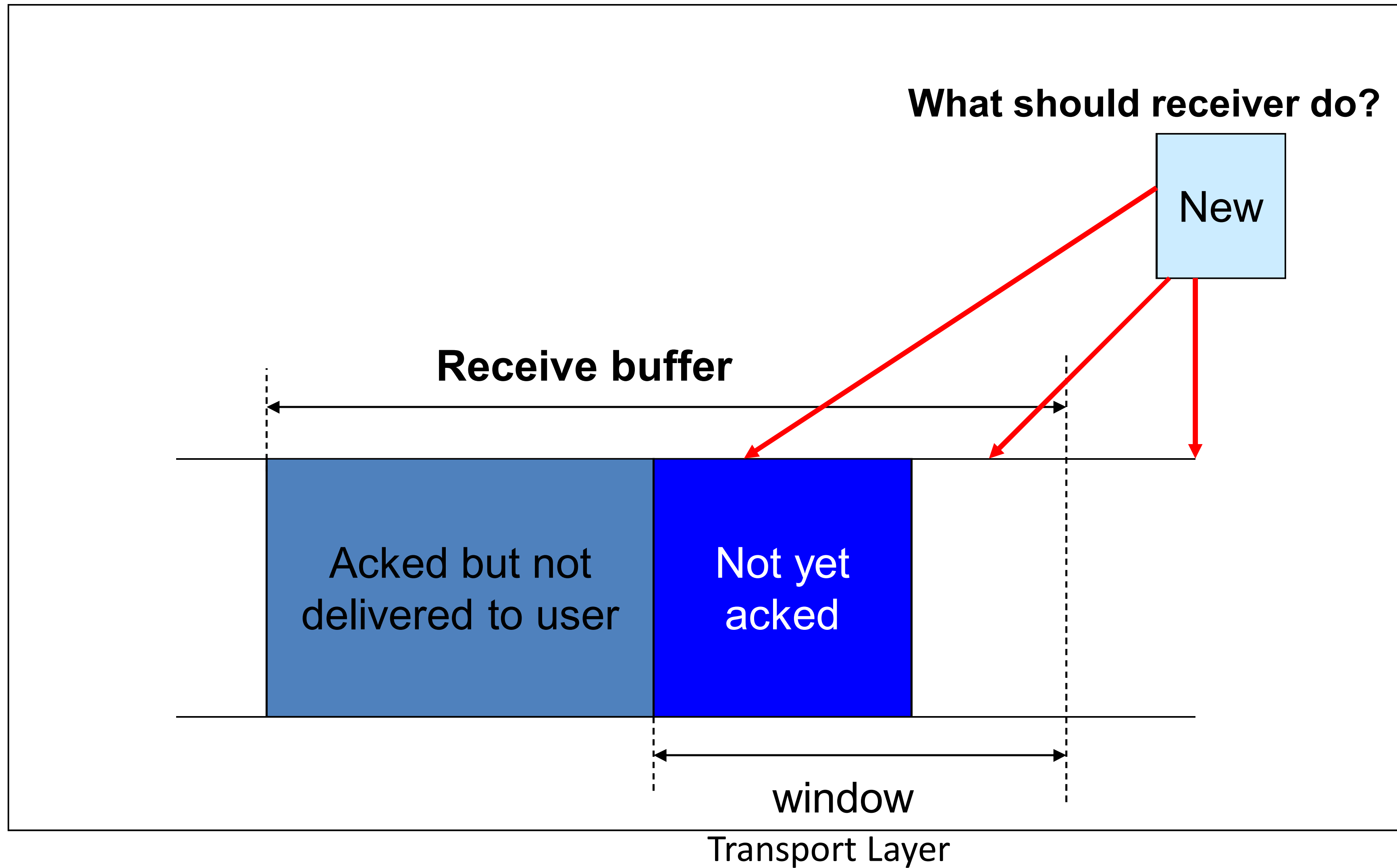
# TCP Flow Control

- TCP is ***a sliding window*** protocol
  - For window size *n*, can send up to *n* bytes ***without receiving an acknowledgement***
  - When the data is acknowledged then the window slides forward
- Each packet advertises a window size
  - Indicates number of bytes the receiver has space for
- Original TCP always sent entire window
  - **Congestion control now limits this**

# Window Flow Control: Send Side

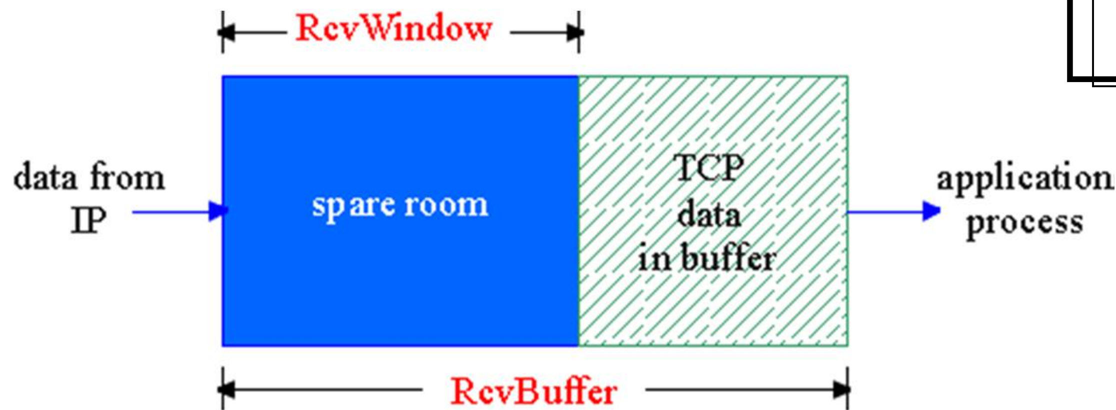


# Window Flow Control: Receive Side



# TCP Flow Control

Receive side of TCP connection has a **receive buffer**:



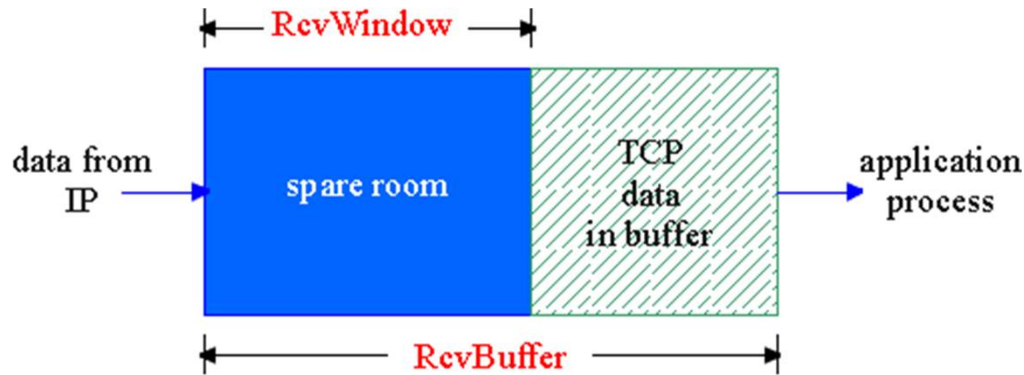

**flow control**

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

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

- app process may be slow at reading from buffer

# TCP Flow control: how it works




receiver ***advertises*** spare room by including value of **`RcvWindow`** in segments

sender limits unACKed data to **`RcvWindow`**

- guarantees receive buffer doesn't overflow

suppose TCP receiver discards out-of-order segments)

spare room in buffer

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

# TCP congestion control: summary

Sender:

- ***Reduces the sending rate*** via reducing the *congestion window*, when *a* ***loss event*** occurs
- ***Increase the sending rate***, when congestion is reduced

***timeout*** or ***3 DUP ACKs***

But how much should a sender reduce its congestion window ?

| **LastByteSent- LastByteAcked ≤ min {*Congestion Window, Receive Window*}** |
|---|

congestion

*flow control*

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

# Τέλος Ενότητας

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης