



*Corso di Laurea Triennale in Informatica  
Università degli Studi della Basilicata*

# Reti di Calcolatori

Docente: Ugo Erra

*ugo.erra+reti@unibas.it*

8° Lezione – Livello di trasporto – II° parte

# Sommario



- Trasporto orientato alla connessione: TCP
  - **Connessione TCP**
  - Struttura dei segmenti
  - Trasferimento dati affidabile
  - Controllo di flusso
  - Gestione della connessione
  - Principi del controllo di congestione

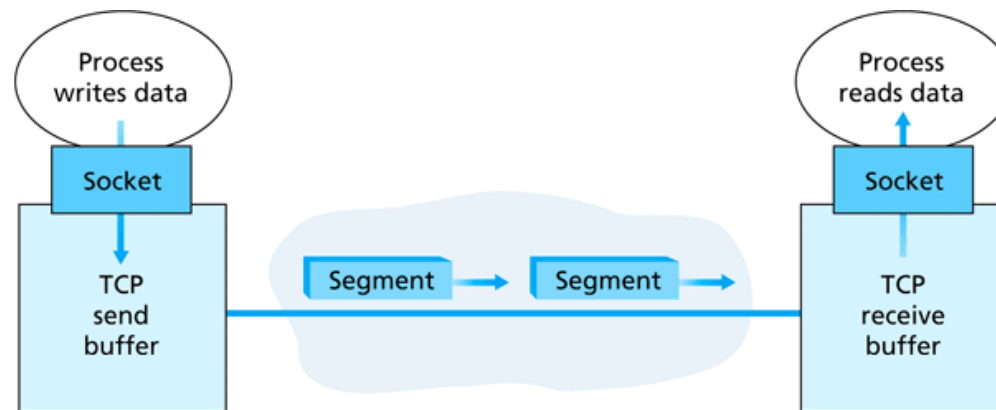
# Connessione TCP

- Il TCP è un protocollo di trasporto **orientato alla connessione**
  - ▣ La “connessione” non è un circuito end-to-end
  - ▣ Lo stato della connessione va in esecuzione solo sui sistemi terminali
  - ▣ I router intermedi non sanno nulla della connessione
- L'**handshaking a tre vie** (scambio di messaggi di controllo) inizializza lo stato del mittente e del destinatario prima di scambiare i dati
- Un processo client che voglia instaurare una connessione con un processo server invocherà in Java il comando

```
Socket clientSocket = new Socket("hostname", portNumber);
```

# Dimensione massima dei segmenti

- Il TCP offre un servizio **full-duplex**
  - ▣ Flusso di dati bidirezionale nella stessa connessione
- La **dimensione massima di segmento** (MSS, *maximum segment size*) rappresenta la quantità massima di dati che possono essere prelevati e spediti nel **buffer d'invio**
  - ▣ Un file di grosse dimensioni verrà frammentato in porzioni di dimensione MSS



# Come stabilire MSS

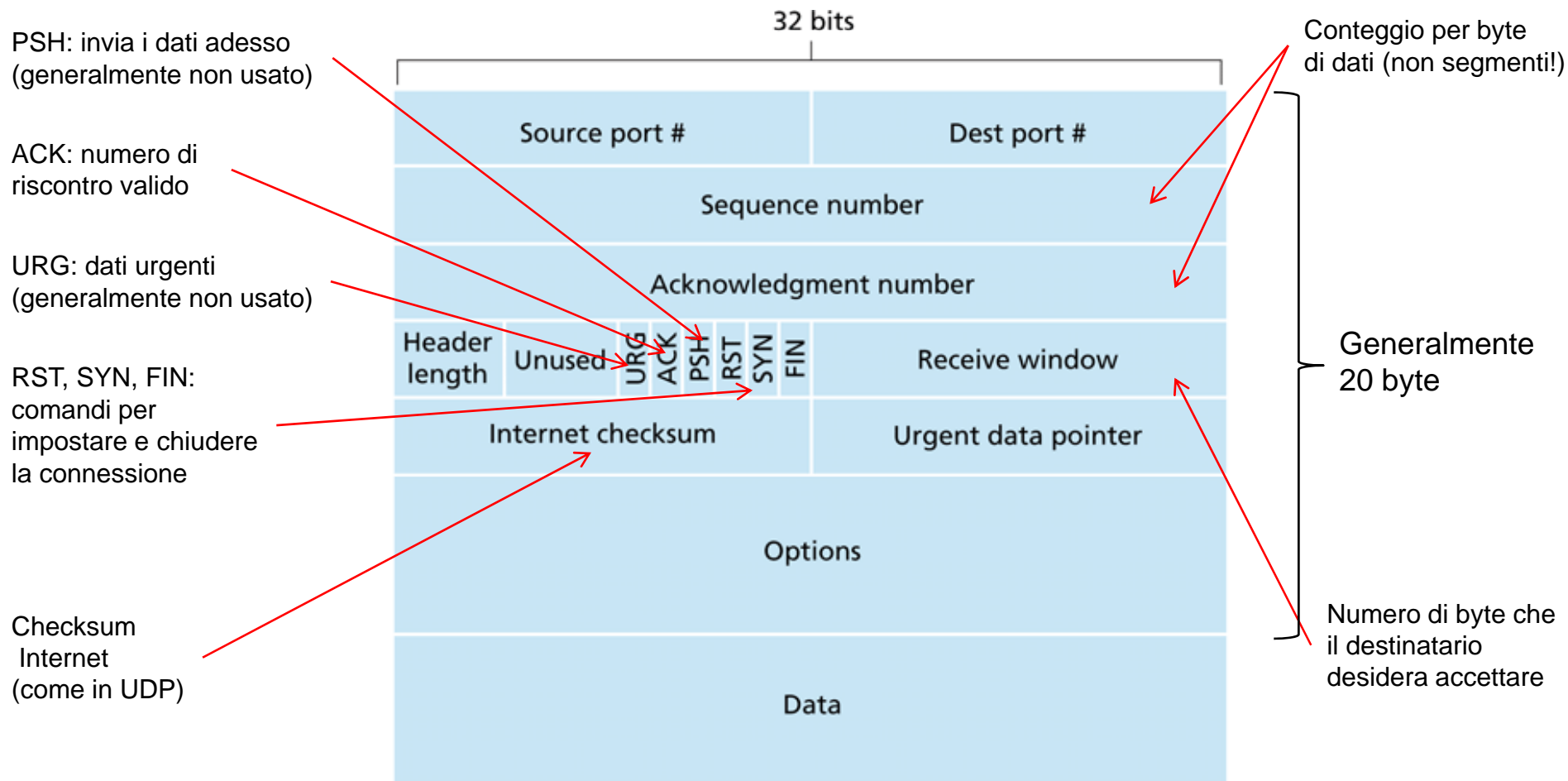
- Il valore MSS viene determinato l'**unità trasmissiva massima** (MTU, *maximum transmission unit*)
- La MTU è la dimensione massima del frame a livello di collegamento dall'host mittente locale
- MSS viene scelto in modo da non superare la MTU
  - ▣ Valori comuni per MTU sono 1460, 536, 512

# Sommario



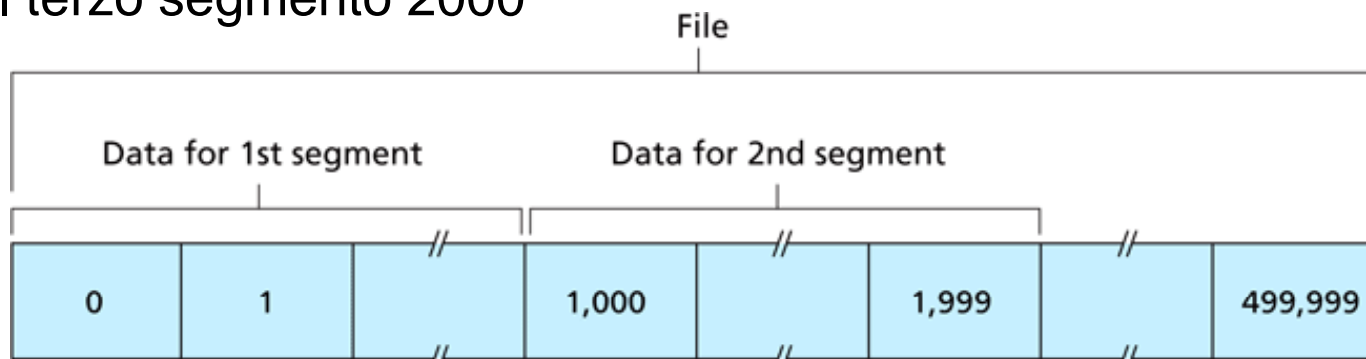
- Trasporto orientato alla connessione: TCP
  - Connessione TCP
  - **Struttura dei segmenti**
  - Trasferimento dati affidabile
  - Controllo di flusso
  - Gestione della connessione
  - Principi del controllo di congestione

# Struttura dei segmenti TCP



# Numero di sequenza in TCP

- TCP vede i dati come flusso di byte ordinati
- Il TCP utilizza il **numero di sequenza per un segmento** per indicare il numero di flusso di byte dal primo byte del segmento
- Supponiamo di trasmettere un file di 500.000 byte dall'host A all'host B con MSS uguale a 1000 byte
  - ▣ TCP invierà il file utilizzando 500 segmenti
  - ▣ Al primo segmento viene assegnata numero di sequenza 0
  - ▣ Al secondo segmento numero di sequenza 1000
  - ▣ Al terzo segmento 2000

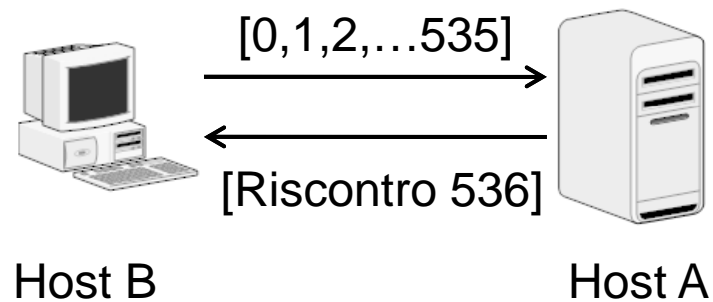


# Numero di riscontro (ACK) in TCP

- Il TCP essendo full-duplex l'host A inviare e ricevere dati verso l'host B
- *Il numero di riscontro che l'host A scrivi nei propri segmenti è il numero di sequenza del byte successivo che l'host A attende da B*
  - ▣ Numeri di sequenza:
    - “numero” del primo byte del segmento nel flusso di byte
    - Il primo byte non è zero ma è scelto a caso
  - ▣ ACK:
    - Numero di sequenza del prossimo byte atteso dall'altro lato
  - ▣ ACK cumulativo
    - D: come gestisce il destinatario i segmenti fuori sequenza?
    - R: la specifica TCP non lo dice – dipende dall'implementatore

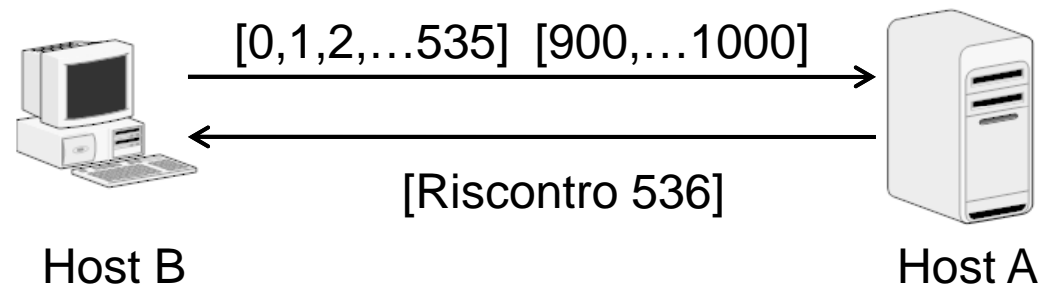
# Esempio 1

1. L'host A ha ricevuto da B un segmento con i byte da 0 a 535 ed
2. Nel momento in cui l'host A invia un segmento a B scrive nel campo riscontro il prossimo byte 536 che si aspetta da B



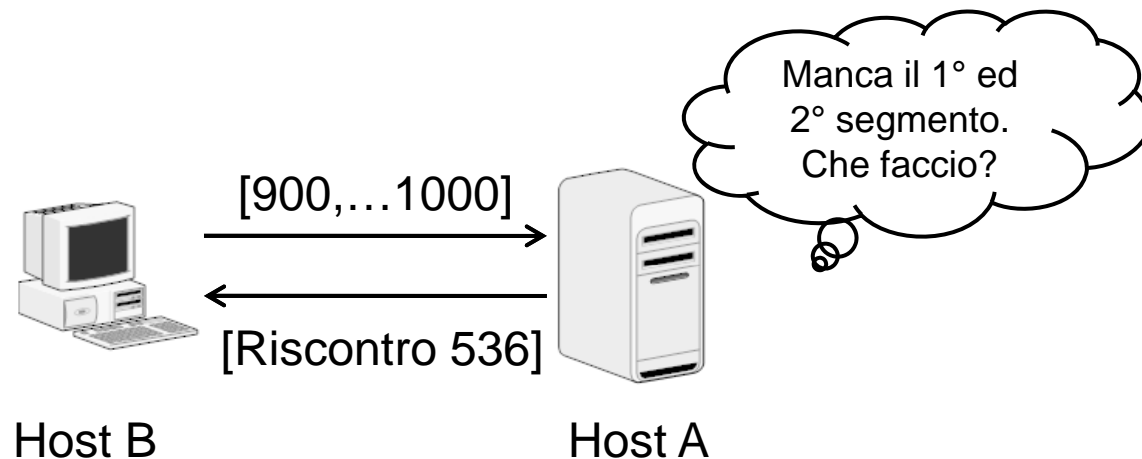
# Esempio 2

1. L'host A ha ricevuto da B un segmento con i byte da 0 a 535 ed un segmento con i byte da 900 a 1000
2. L'host A invia un segmento a B e scrive nel campo riscontro il prossimo byte 536 che si aspetta da B per ricreare il flusso da B

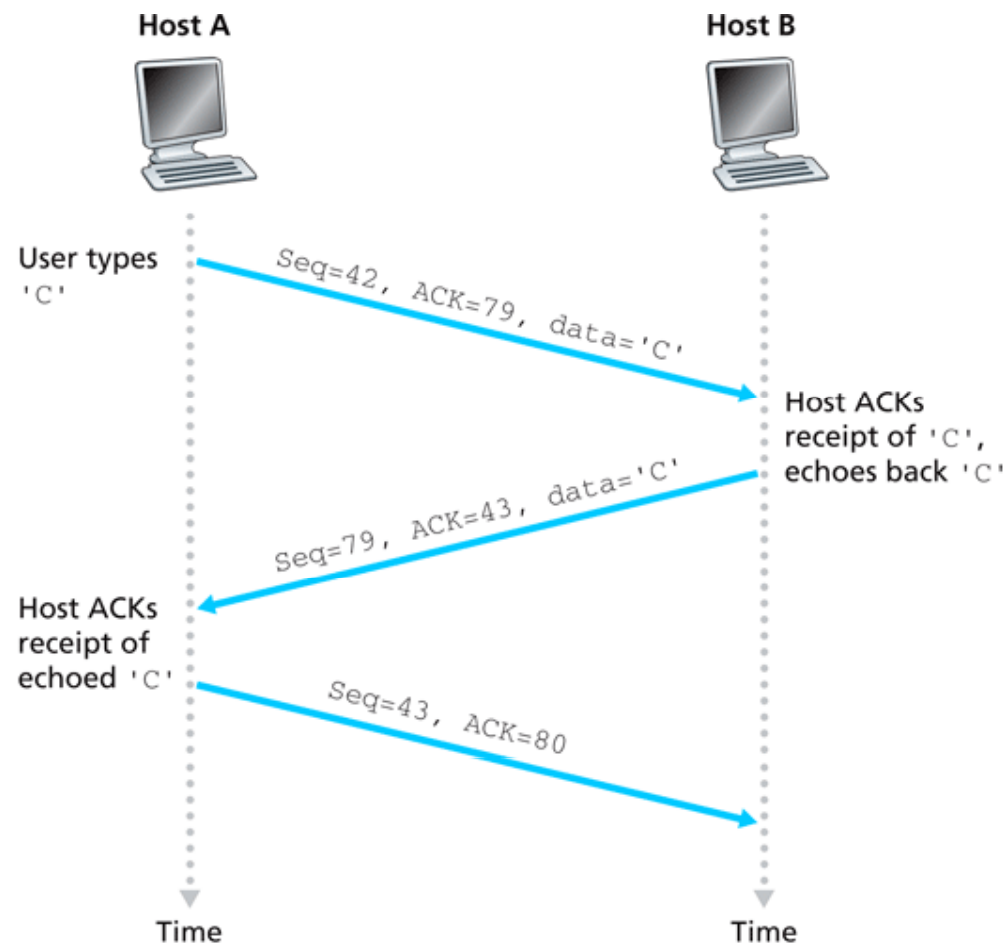


# Esempio 3

1. L'host A ha ricevuto da B il terzo segmento con i byte da 900 a 1000 prima del secondo segmento
2. Il terzo segmento non è arrivato in ordine e l'host A può scartarlo oppure "conservarlo"
  1. RFC non dice nulla e lascia all'implementazione la scelta



# Esempio: applicazione Telnet



# Stima di RTT



- Come impostare il valore del timeout di TCP?
  - Più grande di RTT
    - RTT è variabile nel tempo
  - Troppo piccolo
    - Il timeout è potrebbe essere prematuro e potremmo ritrasmettere segmenti non necessari
  - Troppo grande
    - Reazione lenta alla perdita dei segmenti

# Come stimare RTT?

- Poniamo `SampleRTT` come il tempo misurato dalla trasmissione del segmento fino alla ricezione di ACK
  - ▣ Viene stimato solo per le trasmissioni e non per le ritrasmissioni
- `SampleRTT` varia, quindi occorre una stima “più livellata” di RTT
  - ▣ Usiamo una media di più misure recenti, non semplicemente il valore corrente di `SampleRTT`

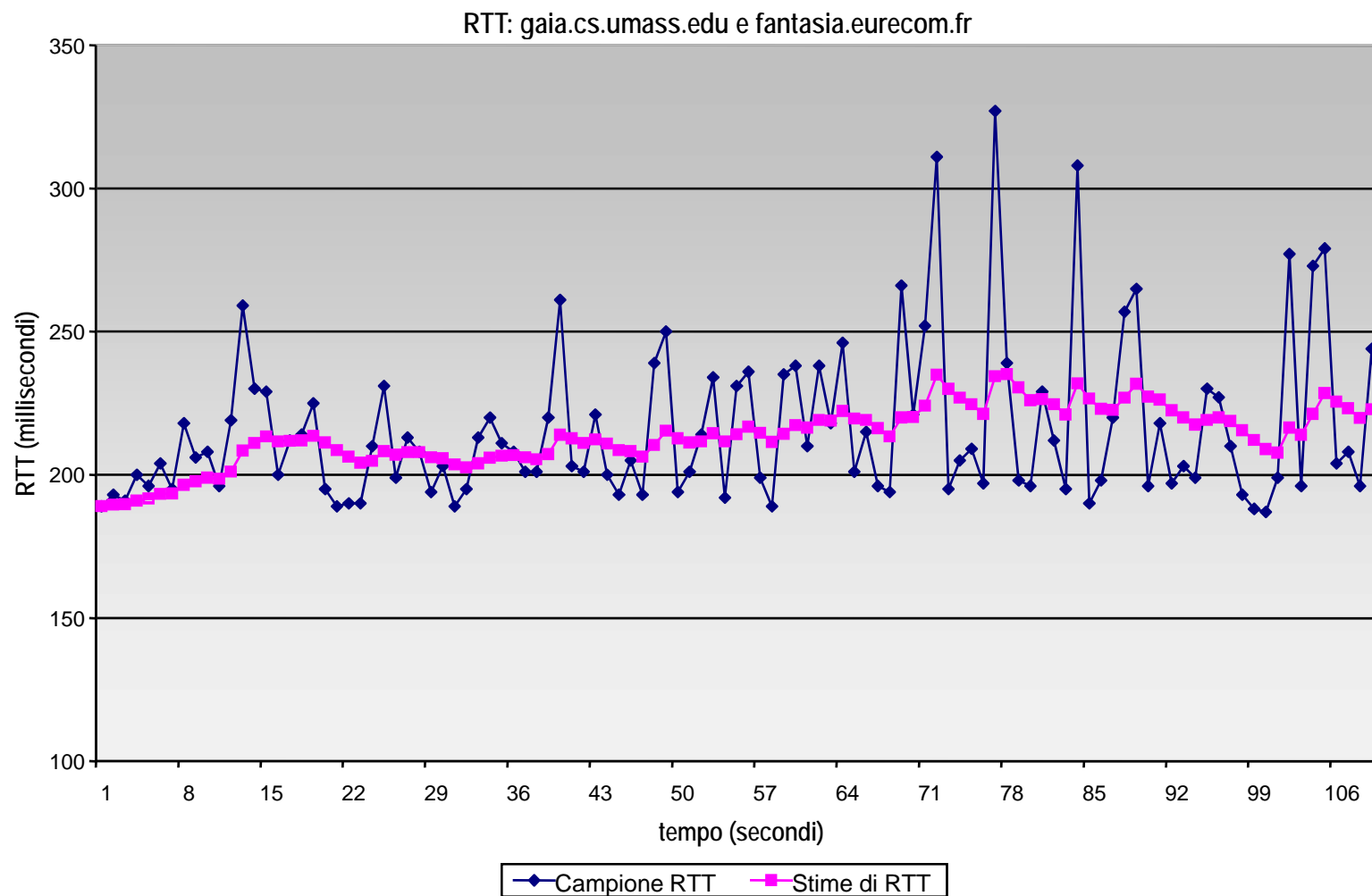
# EstimatedRTT

- In TCP stimiamo una media chiamata EstimatedRTT
- EstimatedRTT è misurato attraverso una **media mobile esponenziale ponderata**, ovvero

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

- L'influenza dei vecchi campioni decresce esponenzialmente
  - ▣ Valore tipico:  $\alpha = 0,125$

# Esempio di stima di RTT



# Impostazione del timeout

- Per impostare il timeout si utilizza `EstimatedRTT` più un “margine di sicurezza”
- Una grande variazione di `EstimatedRTT` comporta un margine di sicurezza maggiore
- Possiamo stimare quanto `EstimatedRTT` si discosta da `SampleRTT` attraverso

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

- L'intervallo di timeout è impostato come

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

# Sommario



- Trasporto orientato alla connessione: TCP
  - Connessione TCP
  - Struttura dei segmenti
  - **Trasferimento dati affidabile**
  - Controllo di flusso
  - Gestione della connessione
  - Principi del controllo di congestione

# Trasferimento dati affidabile

- Il protocollo TCP si basa su un servizio di livello rete (IP) generalmente non affidabile
  - ▣ Non garantisce la consegna
  - ▣ Non garantisce la corretta sequenza
  - ▣ Non garantisce l'integrità dei dati
- TCP offre un servizio di **trasporto dati affidabile** sopra il servizio inaffidabile e best-effort IP
  - ▣ Assicura che la sequenza dei byte non sia alterata
  - ▣ Assicura che la sequenza non abbia buchi
  - ▣ Assicura che non ci siano duplicazioni

# Eventi associati al TCP



- Poiché la gestione dei timer è costosa il TCP utilizza un solo timer di ritrasmissione
- Il TCP gestisce tre eventi relativi alla trasmissione ed alla ritrasmissione
  1. Dati provenienti dall'applicazione
  2. Timeout
  3. Ricezione di ACK

# Mittente TCP semplificato

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
loop (sempre) {
  switch(evento)

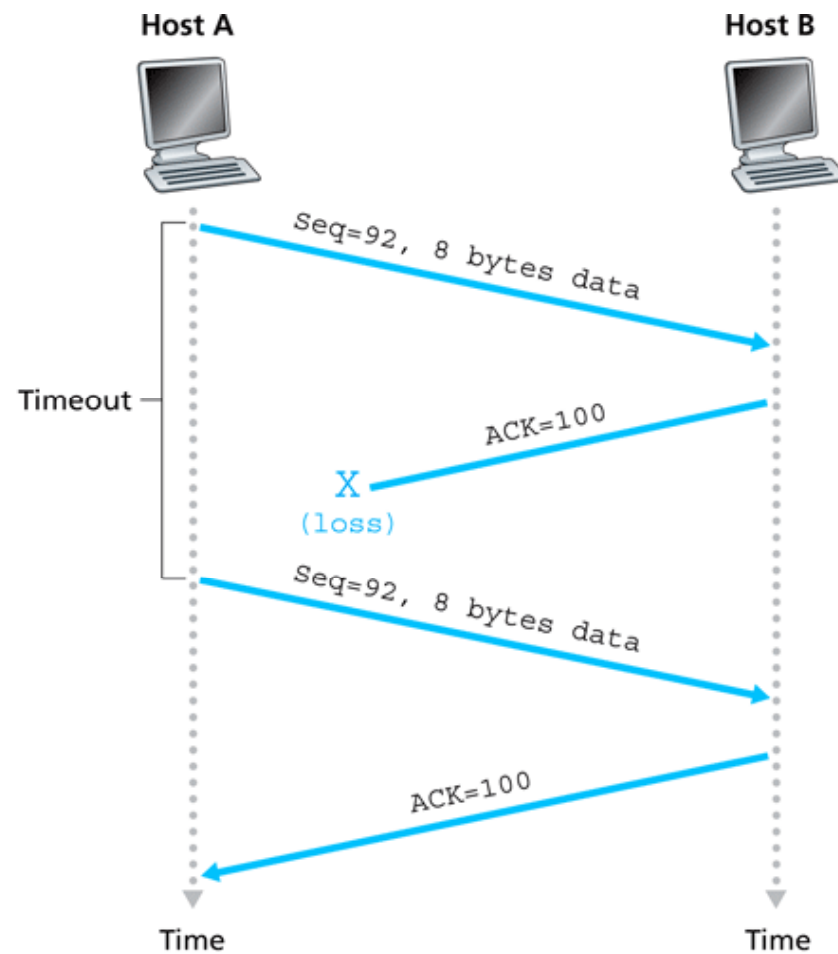
  evento: i dati ricevuti dall'applicazione superiore
          creano il segmento TCP con numero di sequenza NextSeqNum
          if (il timer attualmente non funziona)
              avvia il timer
          passa il segmento a IP
          NextSeqNum = NextSeqNum + lunghezza(dati)

  evento: timeout del timer
          ritrasmetti il segmento non ancora riscontrato con
          il più piccolo numero di sequenza
          avvia il timer

  evento: ACK ricevuto, con valore del campo ACK pari a y
          if (y > SendBase) {
              SendBase = y
              if (esistono attualmente segmenti non ancora riscontrati)
                  avvia il timer
          }
} /* fine del loop */
```

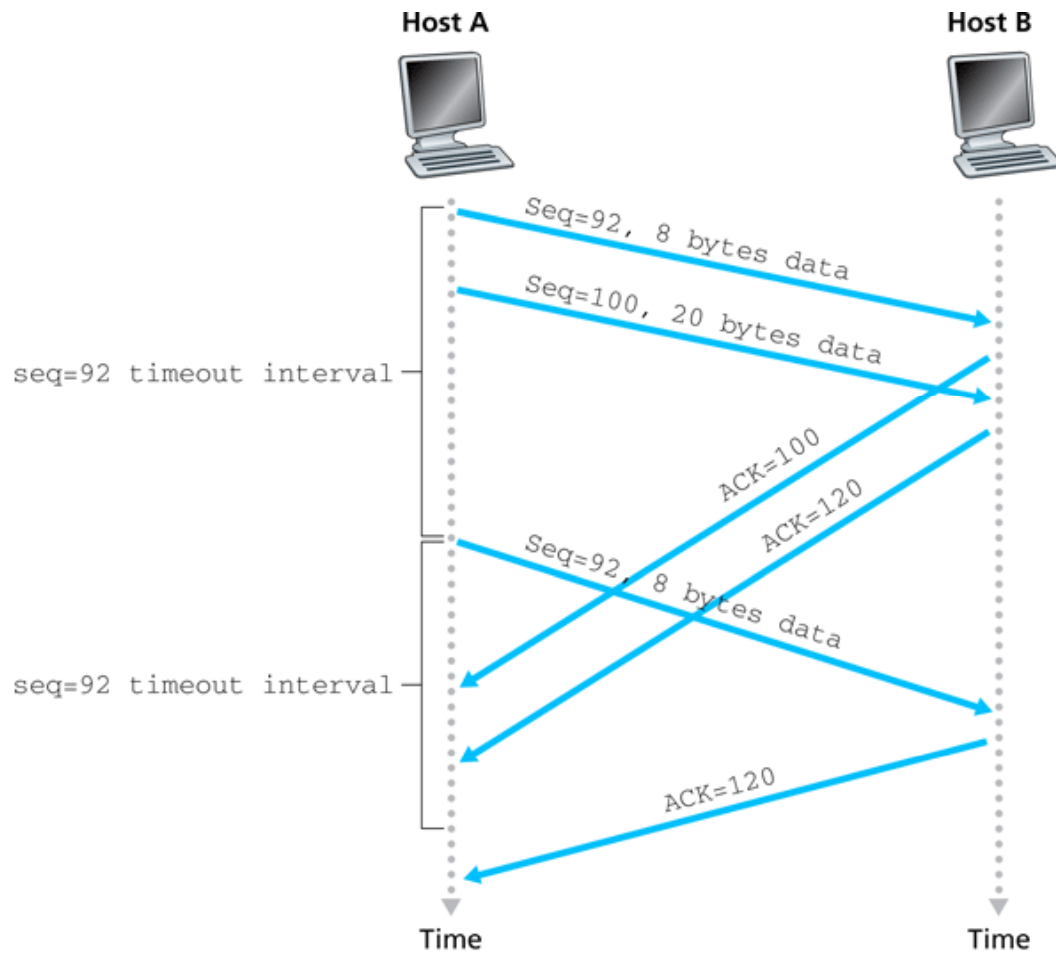
- SendBase è il numero di sequenza di più vecchio già riscontrato
- NextSeqNum è il prossimo numero di sequenza

# Scenario 1



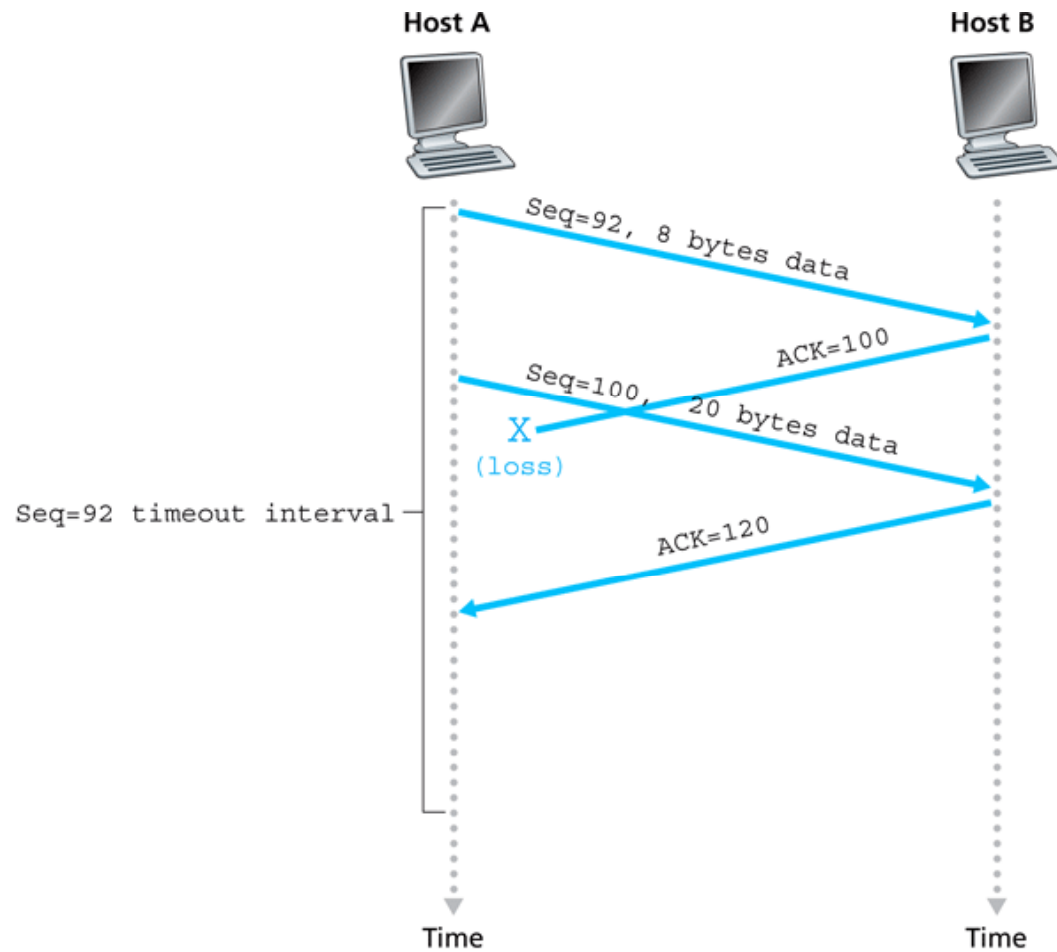
Ritrasmissione dovuta alla perdita di riscontro

# Scenario 2



Segmento 100 non ritrasmesso

# Scenario 3



Riscontro cumulativo che evita le ritrasmissioni del primo segmento

# Raddoppio dell'intervallo di timeout

- Una variante del TCP utilizza il raddoppio del timeout allo scadere del timer
  - Non utilizza `EstimatedRTT` e `DevRTT` per ricalcolarlo
  - Se `TimeoutInterval` era 0.75 il TCP ritrasmetterà il più vecchio segmento con una scadenza di 1.5sec
  - Se scade ancora allora il timeout sarà impostato a 3sec
- Assicura una gestione limitata della congestione

# Ritrasmissione rapida

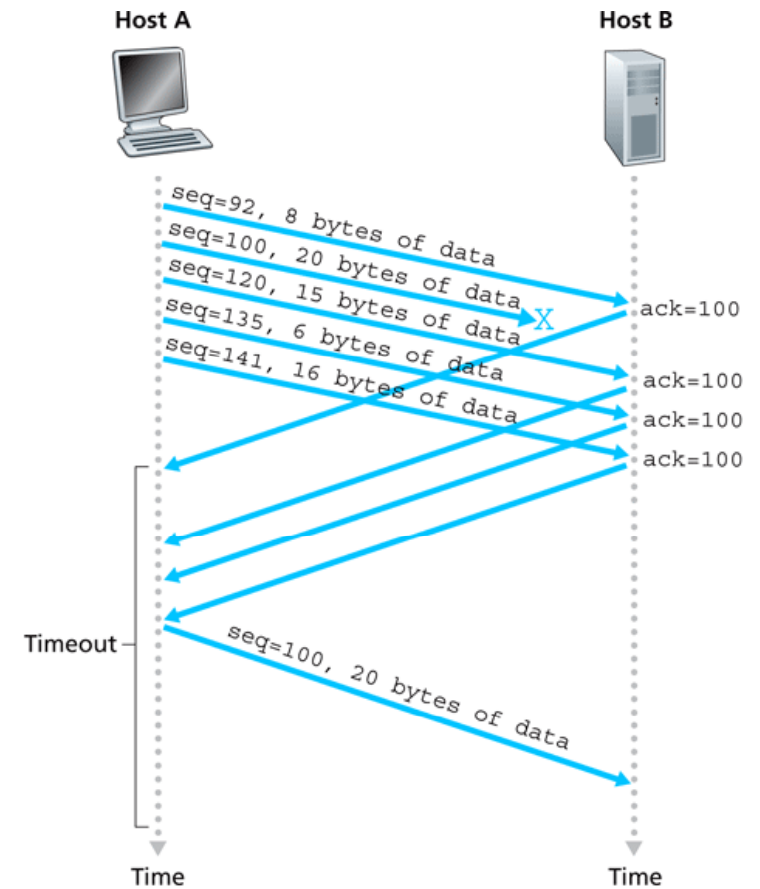


- In caso di ritrasmissione il lungo periodo di timeout permette la ritrasmissione ma a discapito del ritardo end-to-end
- Utilizzando gli **ACK duplicati** è possibile ritrasmettere un segmento prima dello scadere del timeout

# Ritrasmissione

**evento:** ACK ricevuto, con valore del campo ACK pari a  $y$

```
    if ( $y > \text{SendBase}$ ) {  
         $\text{SendBase} = y$   
        if (esistono attualmente segmenti non ancora  
riscontrati)  
            avvia il timer  
    }  
    else {  
        incrementa il numero di ACK duplicati ricevuti per  $y$   
        if (numero di ACK duplicati ricevuti per  $y = 3$ ) {  
            rispeditisci il segmento con numero di sequenza  $y$   
        }  
    }
```



# TCP: GBN o SR?



- TCP è un protocollo Go-Back-N oppure a Ripetizione selettiva?

# Sommario



- Trasporto orientato alla connessione: TCP
  - Connessione TCP
  - Struttura dei segmenti
  - Trasferimento dati affidabile
  - **Controllo di flusso**
  - Gestione della connessione
  - Principi del controllo di congestione

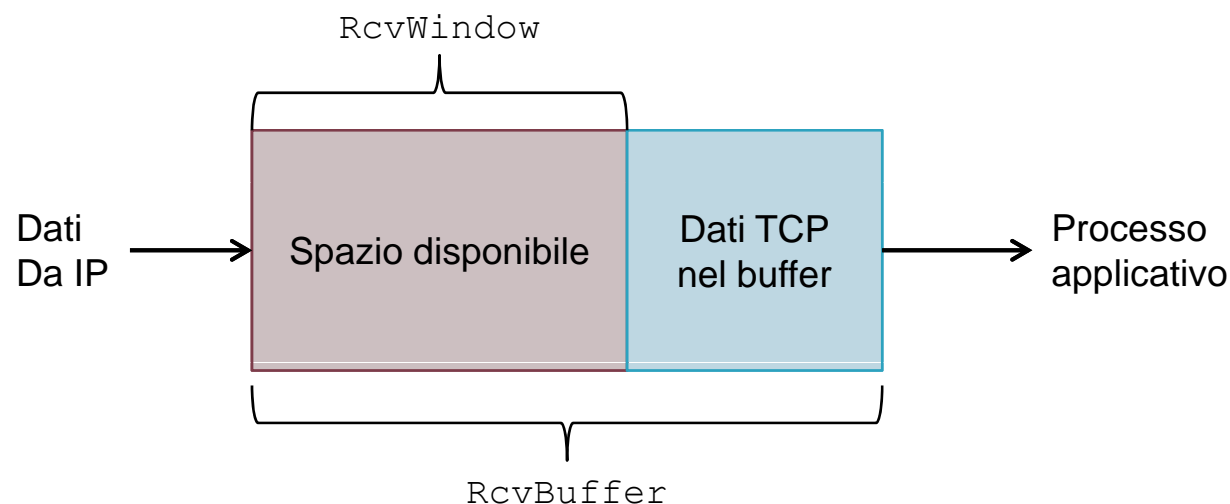
# Controllo di flusso



- Il mittente non vuole sovraccaricare il buffer del destinatario trasmettendo troppi dati e troppo velocemente
  - ▣ Ad esempio una applicazione potrebbe leggere i dati dal buffer di ricezione troppo lentamente
- TCP offre un **servizio di controllo di flusso** per non sommergere il mittente di pacchetti
- L'idea è basata su un servizio di corrispondenza delle velocità: la frequenza d'invio deve corrispondere alla frequenza di lettura dell'applicazione ricevente

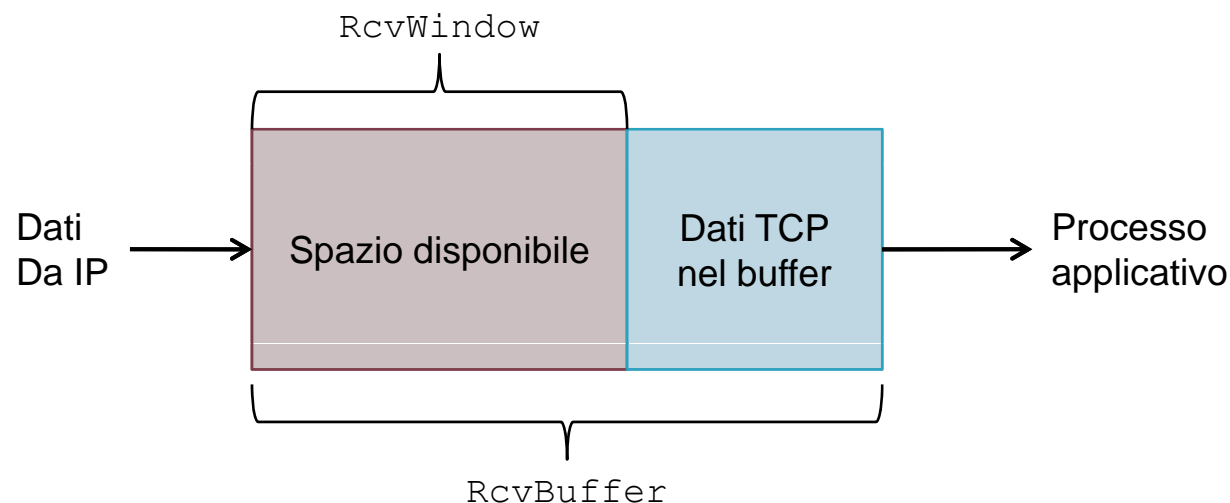
# Controllo di flusso

- La gestione del flusso è affidata ad un buffer di ricezione sul lato ricevente chiamata **finestra di ricezione**
  - ▣ Fornisce al mittente un'indicazione dello spazio libero disponibile nel buffer destinatario
  - ▣ Poiché il TCP è full-duplex sia mittente che destinatario mantengono una finestra di ricezione



# Funzionamento del controllo del flusso: destinatario

- Supponiamo che il destinatario TCP scarti i segmenti fuori sequenza
- Lo spazio disponibile nel buffer è pari a:  
$$\text{RcvWindow} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$
- `LastByteRcvd`: numero dell'ultimo byte letto dal buffer dall'applicazione
- `LastByteRead`: numero dell'ultimo byte ricevuto dalla rete
- Il mittente comunica lo spazio disponibile includendo il valore di `RcvWindow` nei segmenti
- Il mittente limita i dati non riscontrati a `RcvWindow`
  - ▣ Garantisce che il buffer di ricezione non vada in overflow



# Funzionamento del controllo del flusso: mittente

- Il mittente tiene traccia di due variabili
  - ▣ `LastByteSent`
  - ▣ `LastByteAked`
- Il mittente assicura che per tutta la durata della connessione

$$\text{LastByteSent} - \text{LastByteAked} \leq \text{RcvWindow}$$

in modo da non manderà in overflow il buffer di ricezione

# Sommario



- Trasporto orientato alla connessione: TCP
  - Connessione TCP
  - Struttura dei segmenti
  - Trasferimento dati affidabile
  - Controllo di flusso
  - **Gestione della connessione**
  - Principi del controllo di congestione

# Gestione della connessione

- Il mittente e ed il destinatario TCP stabiliscono una “connessione” prima di scambiare i segmenti di dati
- Inizializzano le variabili TCP:
  - ▣ Numeri di sequenza
  - ▣ Buffer, informazioni per il controllo di flusso (per esempio, RcvWindow)
- Il client avvia la connessione attraverso

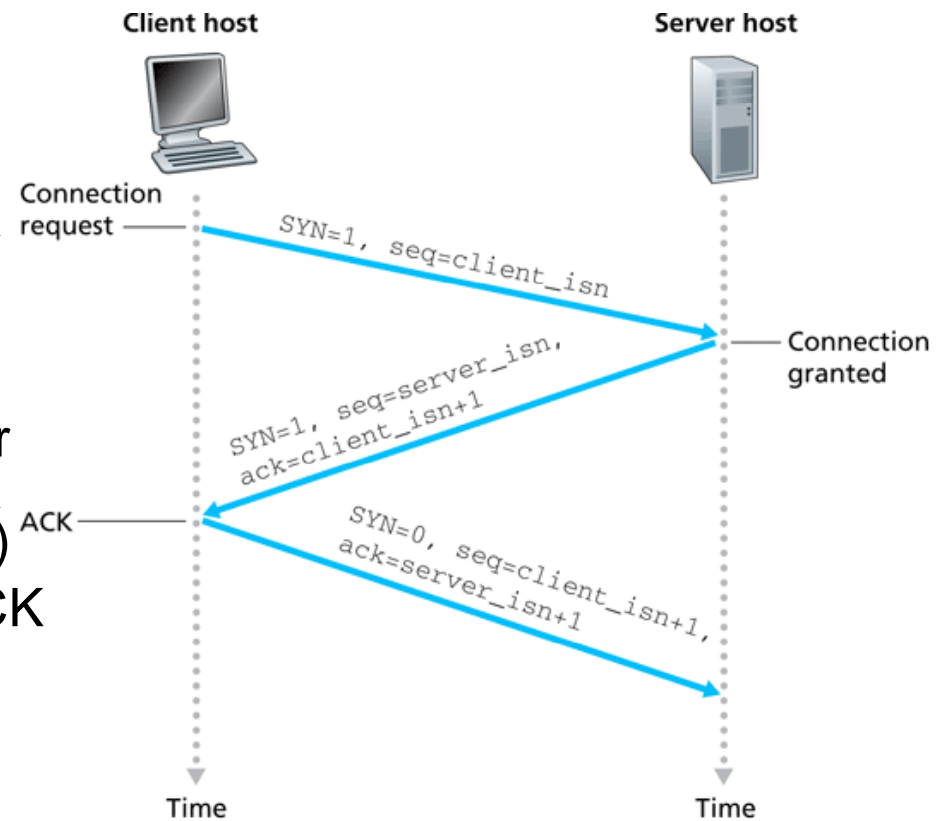
```
Socket clientSocket = new Socket("hostname", "portnumber");
```

- Il server accetta il contattato dal client attraverso

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

# Handshake a tre vie

- La connessione avviene attraverso un **handshake a tre vie**
  - **Passo 1:** il client invia un segmento SYN al server dove specifica il numero di sequenza iniziale (nessun dato)
  - **Passo 2:** il server riceve SYN e risponde con un segmento SYNACK il server alloca i buffer specifica il numero di sequenza iniziale del server (nessun dato)
  - **Passo 3:** il client riceve SYNACK e risponde con un segmento ACK, che può contenere dati



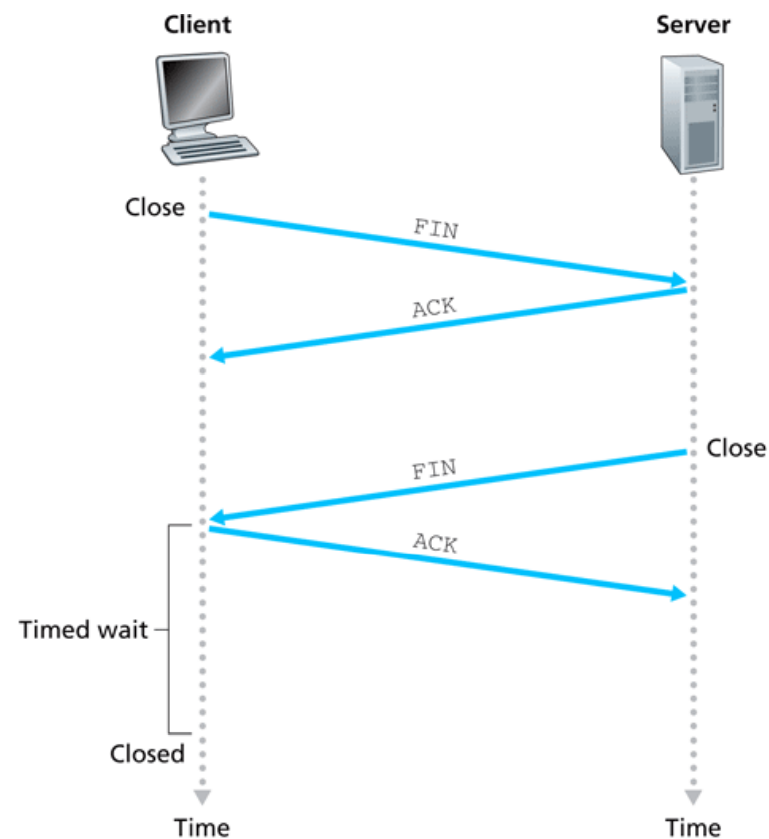
# Note sull'handshake a tre vie



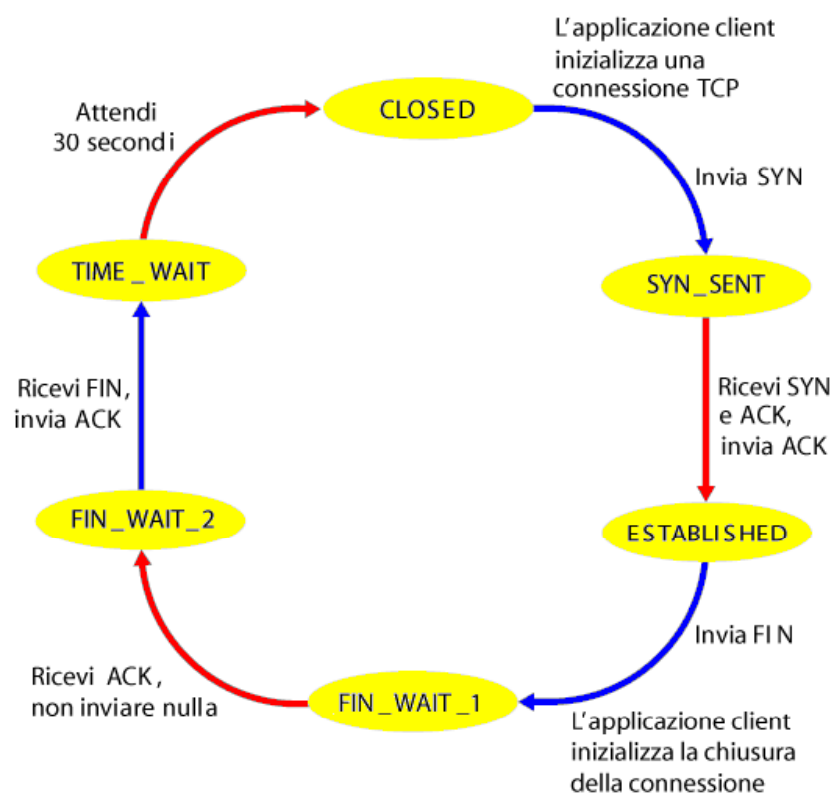
- Il terzo segmento risulta necessario al fine di permettere anche all'host server una stima del timeout iniziale
- Il flag SYN risulta utile nella analisi da parte dei firewall del traffico
  - ▣ I segmenti SYN stabiliscono *nuove* connessioni, mentre quelli con il flag non attivo appartengono a connessioni già instaurate

# Chiusura della connessione

- ❑ Per chiudere la connessione il client chiude la socket attraverso `clientSocket.close()`
- ❑ La chiusura della connessione avviene in due passi
  - ❑ **Passo 1:** il client invia un segmento di controllo FIN al server
  - ❑ **Passo 2:** il server riceve il segmento FIN e risponde con un ACK. Chiude la connessione e invia un FIN
  - ❑ **Passo 3:** il client riceve FIN e risponde con un ACK
    - Inizia l'attesa temporizzata - risponde con un ACK ai FIN che riceve
  - ❑ **Passo 4:** il server riceve un ACK. La connessione viene chiusa

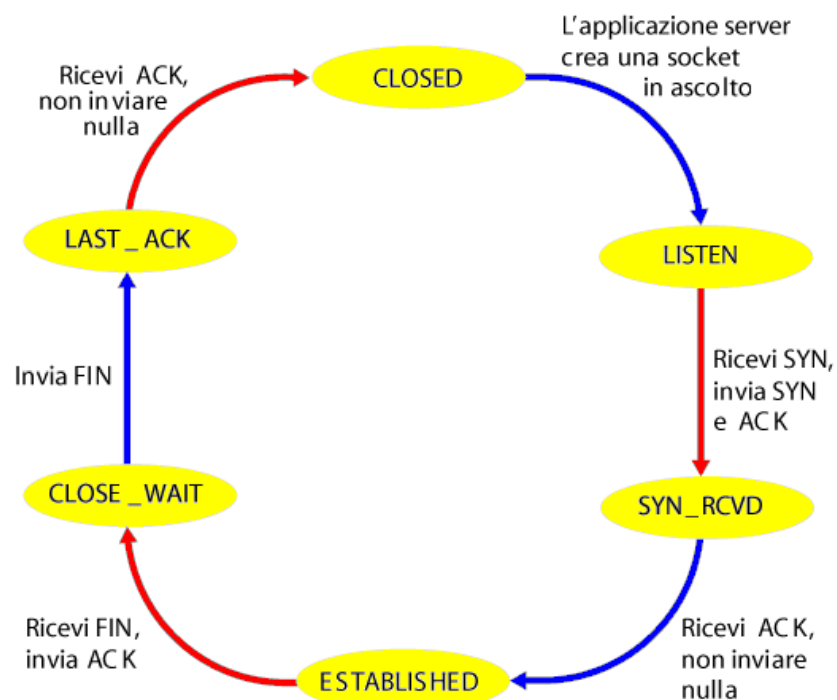


# Gestione della connessione



Sequenza di stati visitati da un client TCP

Sequenza di stati visitati dal TCP sul lato server



# Sommario



- Trasporto orientato alla connessione: TCP
  - Connessione TCP
  - Struttura dei segmenti
  - Trasferimento dati affidabile
  - Controllo di flusso
  - Gestione della connessione
  - **Principi del controllo di congestione**

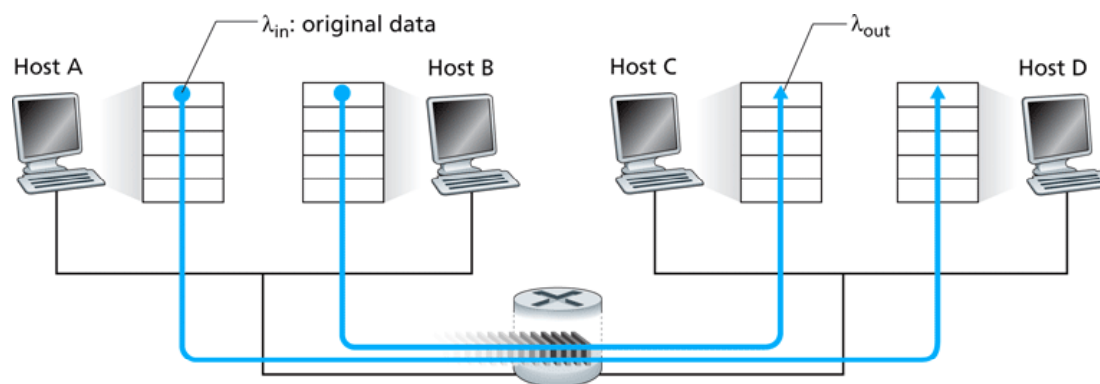
# Principi del controllo di congestione



- Nella congestione informalmente: “troppe sorgenti trasmettono troppi dati, a una velocità talmente elevata che la rete non è in grado di gestirli”
- Il problema differisce dal controllo di flusso!
- La congestione si manifesta come:
  - ▣ Pacchetti smarriti (overflow nei buffer dei router)
  - ▣ Lunghi ritardi (accodamento nei buffer dei router)
- Tra i dieci problemi più importanti del networking!

# Cause/costi: scenario 1

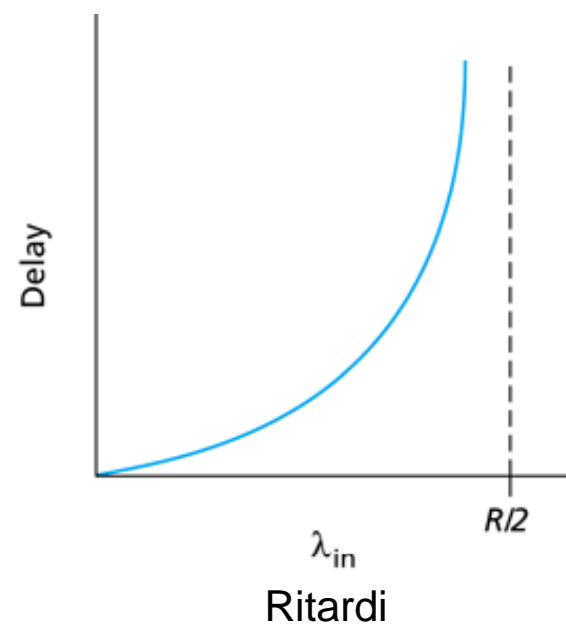
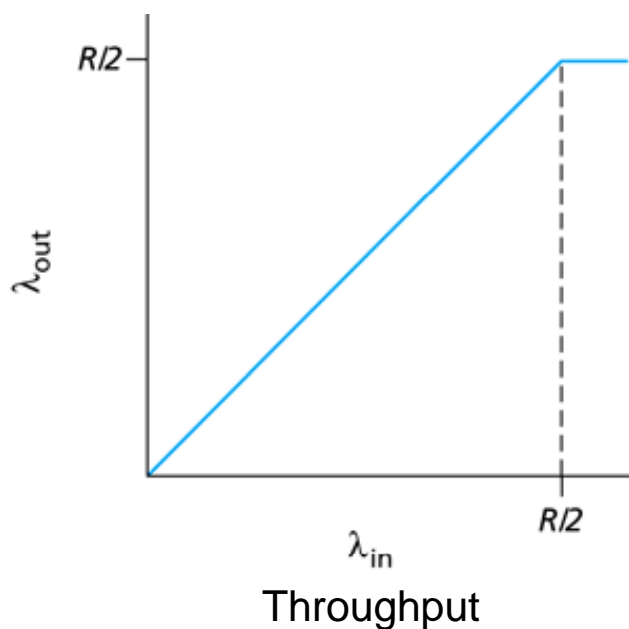
- Consideriamo uno scenario in cui:
  - ▣ Abbiamo due mittenti e due destinatari
  - ▣ Il router ha buffer illimitati
  - ▣ Nessuna ritrasmissione
- In questo scenario
  - ▣  $\lambda_{in}$ : frequenza di trasmissione in byte/sec
  - ▣  $R$ : capacità in uscita del router in byte/sec



Buffer illimitati e condivisi  
per i collegamenti in uscita

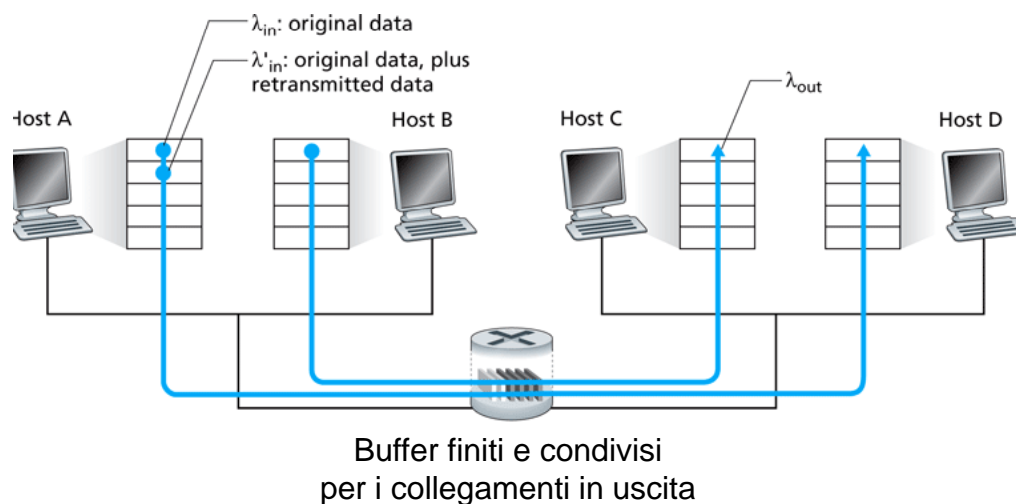
# Cause/costi: scenario 1

- Se la frequenza di invio si avvicina a  $R/2$ :
  - ▣ Throughput massimo
  - ▣ Il ritardo medio cresce sempre di più



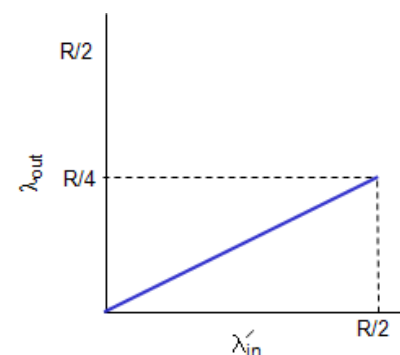
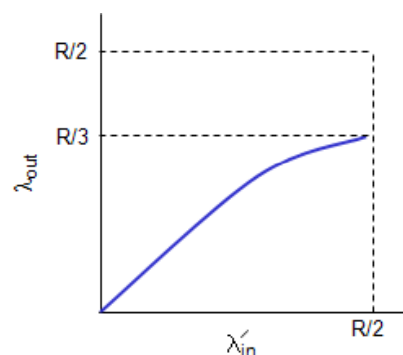
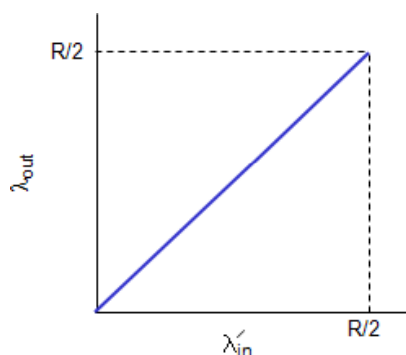
# Cause/costi: scenario 2

- Consideriamo uno scenario in cui:
  - ▣ Il router ha buffer finito
  - ▣ Il mittente ritrasmette il pacchetto perduto
- In questo scenario
  - ▣  $\lambda_{in}$ : è la frequenza di trasmissione in byte/sec
  - ▣  $\lambda'_{in}$ : è la frequenza di trasmissione compreso i dati ritrasmessi in byte/sec (**carico offerto**)
  - ▣  $R$ : capacità in uscita del router in byte/sec



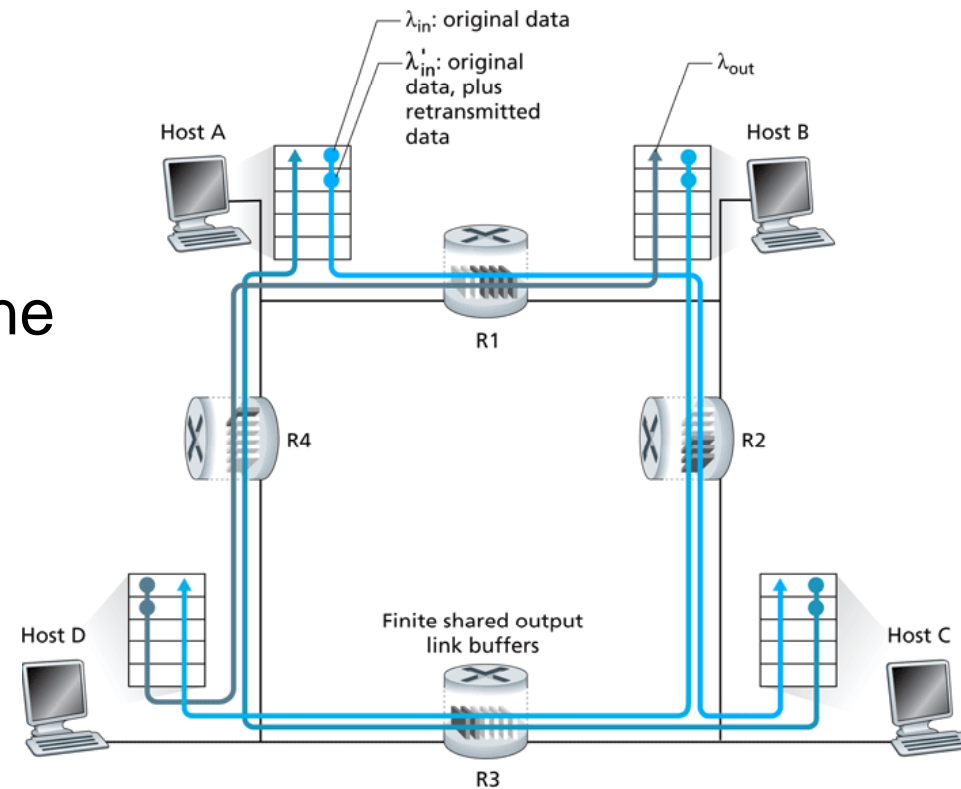
# Cause/costi della congestione: scenario 2

- Sono possibili tre situazioni:
  - ▣ Il mittente conosce l'istante in cui può inviare un pacchetto al router (non realistica)
  - ▣ Il mittente ritrasmette solo quando un pacchetto è andato perduto
    - Il mittente deve ritrasmettere per compensare i pacchetti scartati
  - ▣ Il mittente va in timeout prematuramente e ritrasmette il pacchetto anche se non è stato perduto
    - Il router invia due volte lo stesso pacchetto sprecando parte della sua capacità trasmissiva



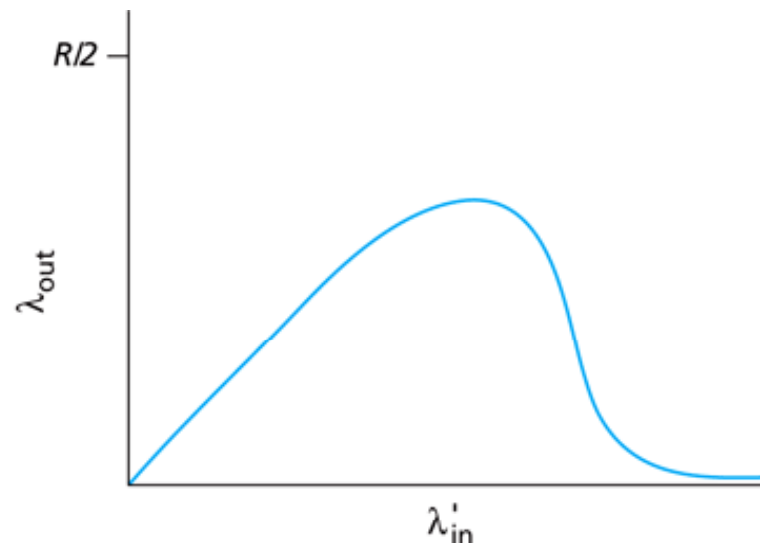
# Cause/costi: scenario 3

- Consideriamo uno scenario in cui:
  - ▣ Quattro mittenti
  - ▣ Percorsi multihop
  - ▣ timeout/ritrasmissione



## Cause/costi della congestione: scenario 3

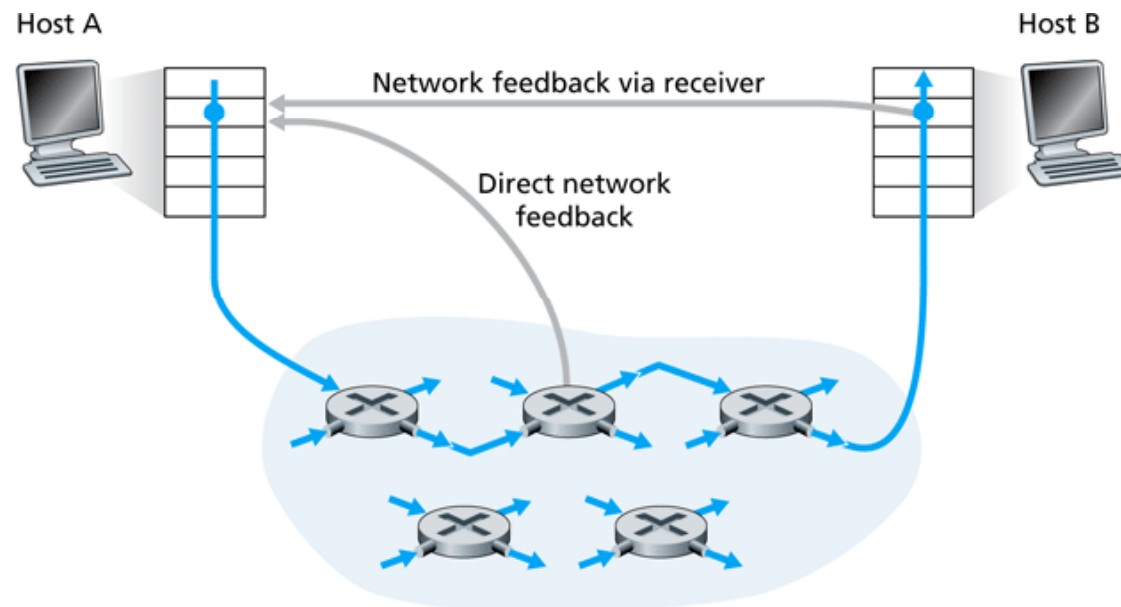
- Un altro “costo” della congestione:
  - ▣ Quando il pacchetto viene scartato, la capacità trasmissiva utilizzata sui collegamenti di upstream per instradare il pacchetto risulta sprecata!



# Approcci alla gestione della congestione

- Controllo di congestione punto-punto:
  - ▣ Nessun supporto esplicito dalla rete
  - ▣ La congestione è dedotta osservando le perdite e i ritardi nei sistemi terminali
  - ▣ Metodo adottato da TCP

- Controllo di congestione assistito dalla rete:
  - ▣ I router forniscono un feedback ai sistemi terminali
  - ▣ Un singolo bit per indicare la congestione (SNA, DECbit, TCP/IP ECN, ATM)
  - ▣ Comunicare in modo esplicito al mittente la frequenza trasmissiva



# Controllo della congestione nel TCP

- Il TCP limita al mittente la frequenza di invio sulla propria connessione in funzione della congestione percepita
- Tre domande:
  - ▣ Come può il mittente limitare la frequenza d'invio?
  - ▣ Come fa il mittente ad accorgersi che la rete è congestionata?
  - ▣ Quale algoritmo utilizzare per variare dinamicamente la frequenza sulla connessione punto-punto?

# Finestra di congestione

- Il mittente limita la trasmissione utilizzando una **finestra di congestione** chiamata `CongWin`
- La quantità di dati non riscontrati da un mittente non può eccedere il minimo tra di `CongWin` e `RcvWindow`, ovvero

$$\text{LastByteSent} - \text{LastByteAcked} \leq \min\{\text{CongWin}, \text{RcvWindow}\}$$

- Supponendo che il buffer di ricezione sia sufficientemente capiente la frequenza di invio del mittente è approssimativamente

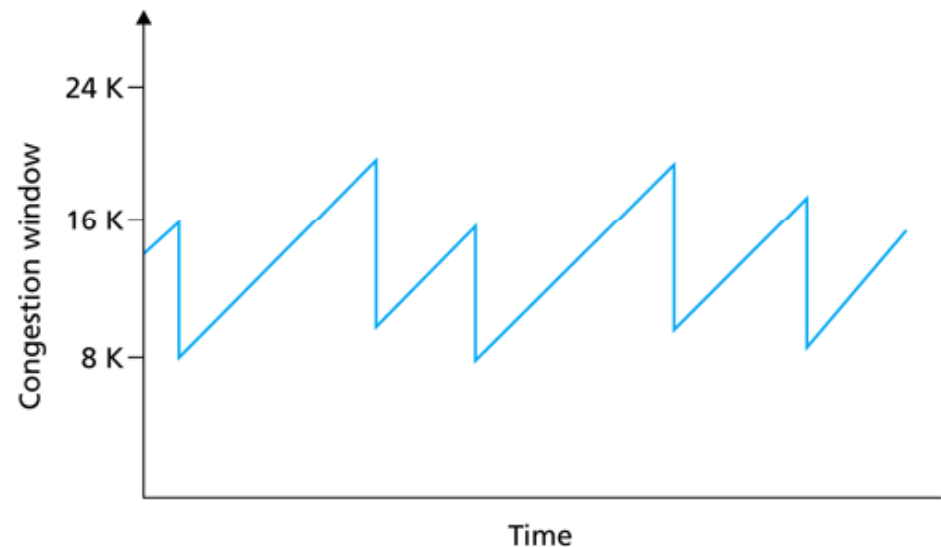
$$\text{CongWin} / \text{RTT byte/sec}$$

# In che modo il mittente percepisce la congestione?

- Definiamo evento di perdita come un timeout o la ricezione di 3 ACK duplicati
  - ▣ Un router ad esempio va in overflow lungo un percorso
  - ▣ Il mittente TCP riduce la frequenza d'invio (`CongWin`) dopo un evento di perdita
- Il TCP è **auto-temporizzante**
  - ▣ La frequenza con cui il mittente riceve i riscontri stabilisce la velocità di crescita della finestra di congestione
- L'**algoritmo di controllo della congestione TCP** regola la frequenza utilizzando tre componenti
  - ▣ Incremento additivo e decremento moltiplicativo
  - ▣ Partenza lenta
  - ▣ Reazione ai timeout

# Incremento additivo e decremento moltiplicativo (AIMD)

- Decremento moltiplicativo
  - ▣ Riduce a metà  $CongWin$  dopo un evento di perdita
  - ▣ Ad esempio se  $CongWin$  vale 20Kbyte lo porta a 10Kbyte poi a 5Kbyte etc...
- Incremento additivo
  - ▣ Aumenta  $CongWin$  di 1 MSS a ogni RTT in assenza di eventi di perdita
- La fase di incremento lineare viene detta **congestion avoidance**



# Riassumendo



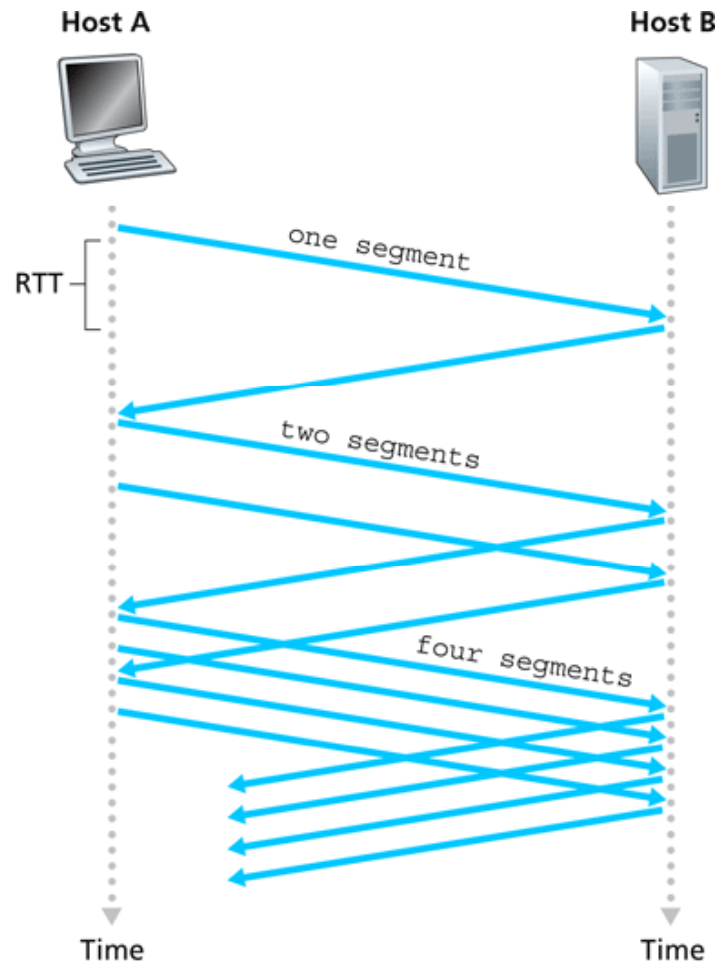
- Il protocollo TCP diminuisce o incrementa la finestra di congestione in base alla congestione del percorso end-to-end
  - ▣ Se è libero allora il mittente aumenta la propria frequenza di trasmissione in maniera additiva
  - ▣ Se è congestionato allora decrementa la frequenza di trasmissione in maniera moltiplicativa

MSS ( Maximum Segment Size)  
massima dimensione del corpo  
dati in un segmento TCP

# Partenza lenta

- Quando si stabilisce la connessione  $\text{CongWin} = 1$   
MSS
  - Esempio: MSS = 500 byte e RTT = 200 msec
  - Frequenza iniziale circa 20 kbps
- Inizialmente si ha una **partenza lenta** (*slow start*) poiché la banda disponibile potrebbe essere più grande di MSS/RTT
- La frequenza è incrementata in maniera esponenziale fino a quando non si verifica uno smarrimento
- Nel momento in cui si verifica uno smarrimento  $\text{CongWin}$  viene dimezzato e cresce in maniera lineare

# Partenza lenta



- Il mittente incrementa CongWin di 1 MSS ogni volta che un segmento trasmesso è riscontrato
- La procedura continua fino a quando non si verifica uno smarrimento

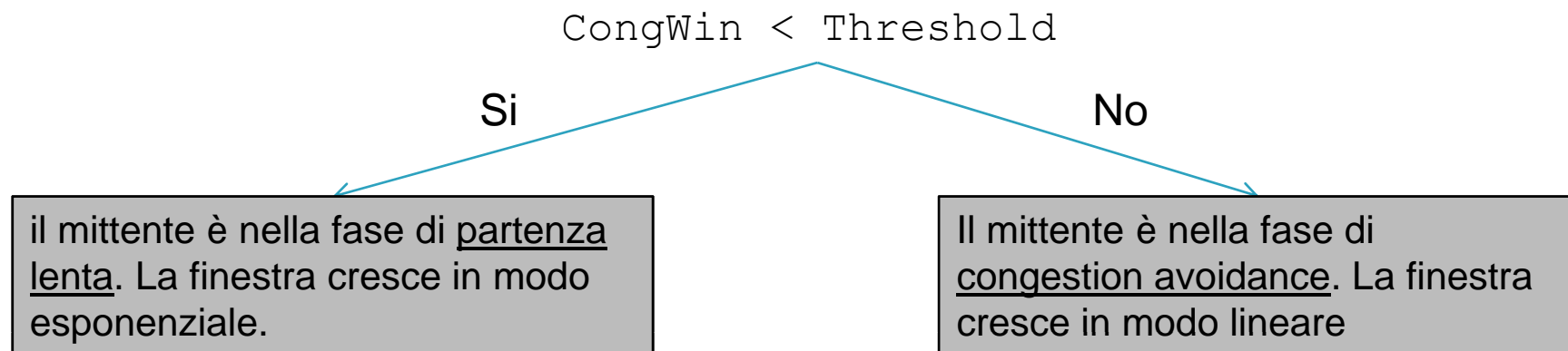
# Affinamento

- In effetti TCP gestisce diversamente il timeout rispetto alla ricezione di tre ACK
    - Dopo la ricezione di 3 ACK
      1. CongWin è ridotto a metà
      2. La finestra crescerà linearmente
    - Dopo un evento di timeout la situazione è “allarmante”
      1. CongWin a 1 MSS
      2. La finestra crescerà in modo esponenziale fino alla metà del valore precedente
      3. Successivamente cresce in maniera lineare
- Congestion avoidance*
- Partenza lenta*

# Gestione delle due fasi

- L'algoritmo di controllo di congestione presenta due fasi:
  - ▣ Partenza Lenta
  - ▣ Incremento additivo - decremento moltiplicativo
- Per distinguere tra le due fasi viene usata una variabile chiamata `Threshold`
- Quando il valore della `CongWin` è minore del valore di `Threshold` ci troviamo nella fase di partenza lenta, altrimenti siamo nella fase AIMD
- All'avvio della trasmissione la variabile viene settata ad un valore molto alto, mentre la dimensione della `CongWin` è pari alla dimensione di un segmento

# Controllo di congestione del mittente



- Si verificano tre ACK duplicati
  - ▣ Il valore di `Threshold` viene impostato a  $\text{CongWin}/2$
  - ▣ `CongWin` è impostato al valore di `Threshold`
- Si verifica un timeout
  - ▣ Il valore di `Threshold` viene impostato a  $\text{CongWin}/2$
  - ▣ `CongWin` è impostata a 1 MSS