

# Reti di Elaboratori

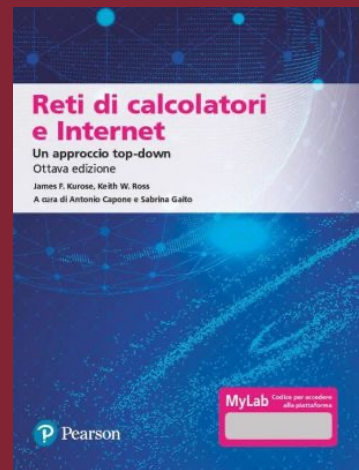
Livello di Trasporto: TCP



SAPIENZA  
UNIVERSITÀ DI ROMA

Alessandro Checco

[alessandro.checco@uniroma1.it](mailto:alessandro.checco@uniroma1.it)



Capitolo 3

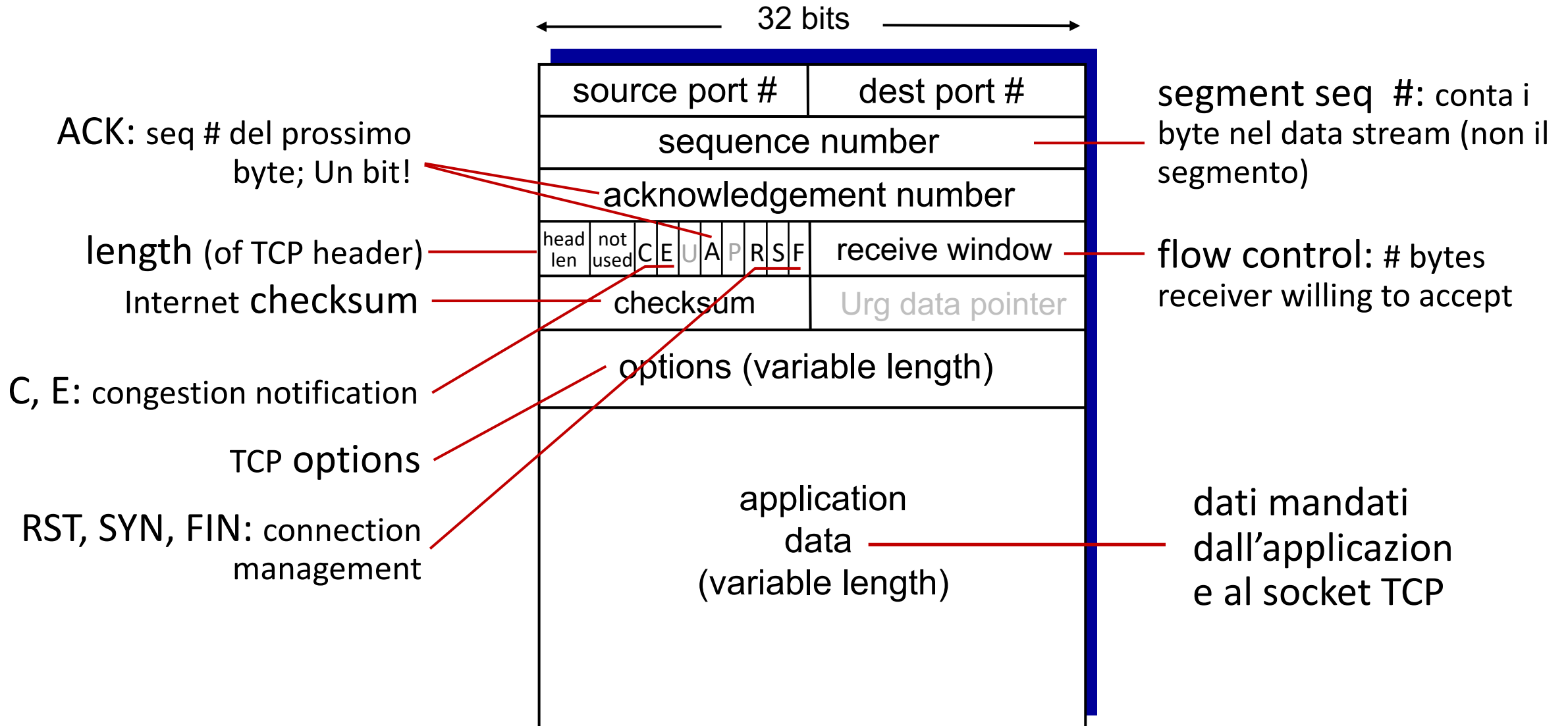
# Livello di trasporto: sommario

- Servizi a livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessione: UDP
- Principi di trasferimento affidabile dei dati
- **Trasporto orientato alla connessione: TCP**
  - struttura del segmento
  - trasferimento affidabile dei dati
  - controllo del flusso
  - gestione della connessione
- Principi di controllo della congestione
- Controllo della congestione TCP
- Evoluzione della funzionalità del livello di trasporto

# TCP: panoramica RFC: 793,1122, 2018, 5681, 7323

- **end-to-end:**
  - un mittente, un destinatario
- **byte stream affidabile e in ordine:**
  - messaggi del livello applicazione vengono concatenati in un unico stream
  - a differenza di UDP dove ogni messaggio era un diverso segmento
- **dati full duplex:**
  - flusso di dati bidirezionale nella stessa connessione
  - MSS: maximum segment size
- **Diverso da commutazione di circuito:**
  - la rete non è a conoscenza dello stabilimento della connessione
- **ACK cumulativi**
- **pipelining:**
  - Windows size dipende dal flow control e congestion control
- **orientato alla connessione:**
  - l'handshaking (scambio di messaggi di controllo) inizializza lo stato del mittente e del destinatario prima dello scambio di dati (3way: 3 segmenti)
- **flow control:**
  - il mittente non sovraccarica il destinatario
- **Proprietà:**
  - bidirezionale e full-duplex
  - Ibrido Go-Back-N e Selective-Repeat

# Struttura del segmento TCP



# Numeri di sequenza TCP, ACK

## *Numeri di sequenza:*

- numero seq del primo byte nel settore data del segmento

## *ACKs*

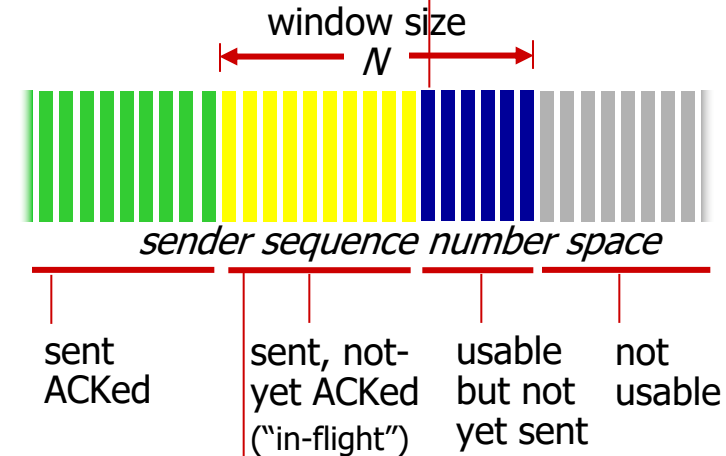
- seq # del byte successivo aspettato
- ACK cumulativo

*D:* il destinatario come gestisce i segmenti fuori ordine?

- *R:* Non indicato da specifiche TCP, lasciato all'implementazione

segmento dal mittente

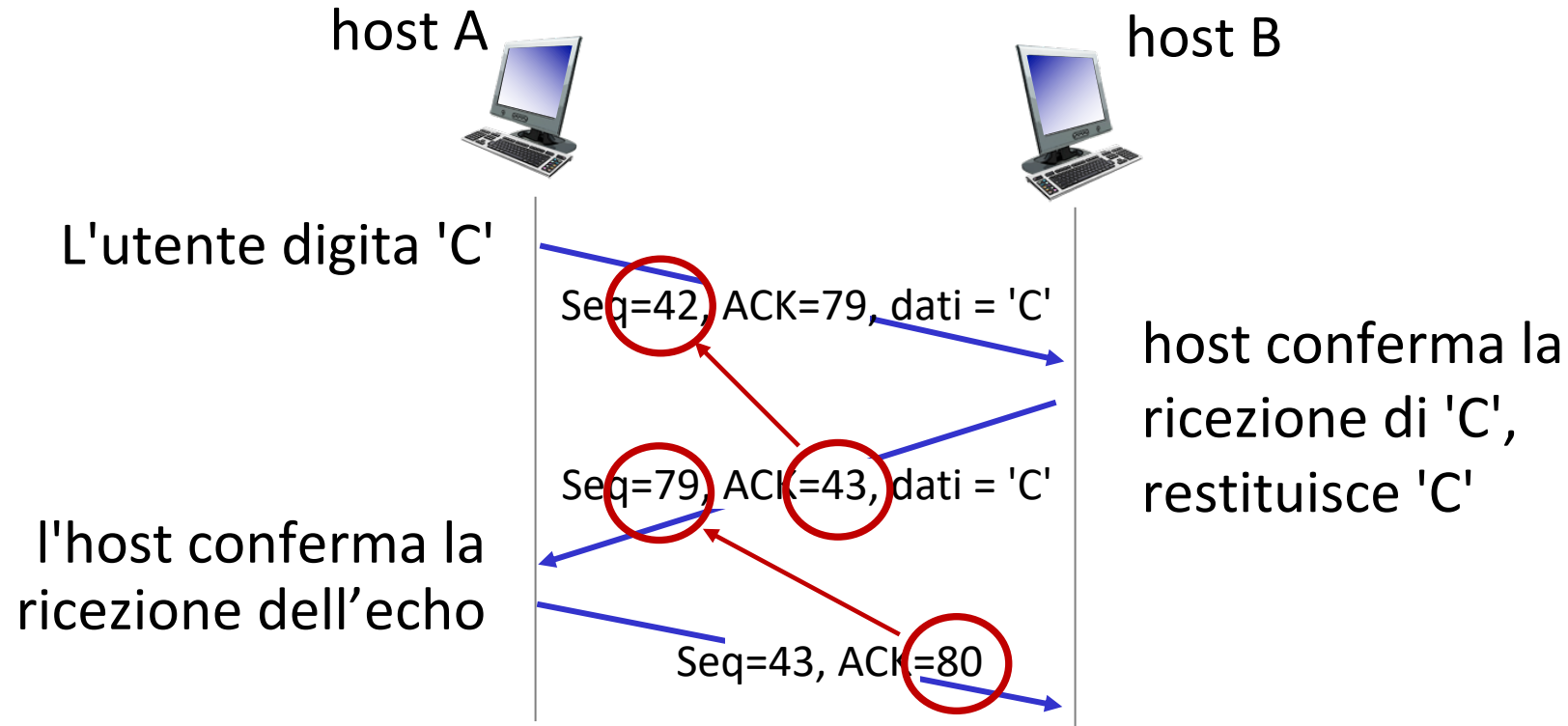
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



segmento dal destinatario

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

# Numeri di sequenza TCP, ACK



semplice scenario telnet con echo

# Tempo di andata e ritorno TCP, timeout

*D:* come impostare il valore di timeout TCP?

- più lungo di RTT, ma RTT varia!
- *troppo corto:* timeout prematuro, ritrasmissioni non necessarie
- *troppo lungo:* reazione lenta alla perdita del segmento

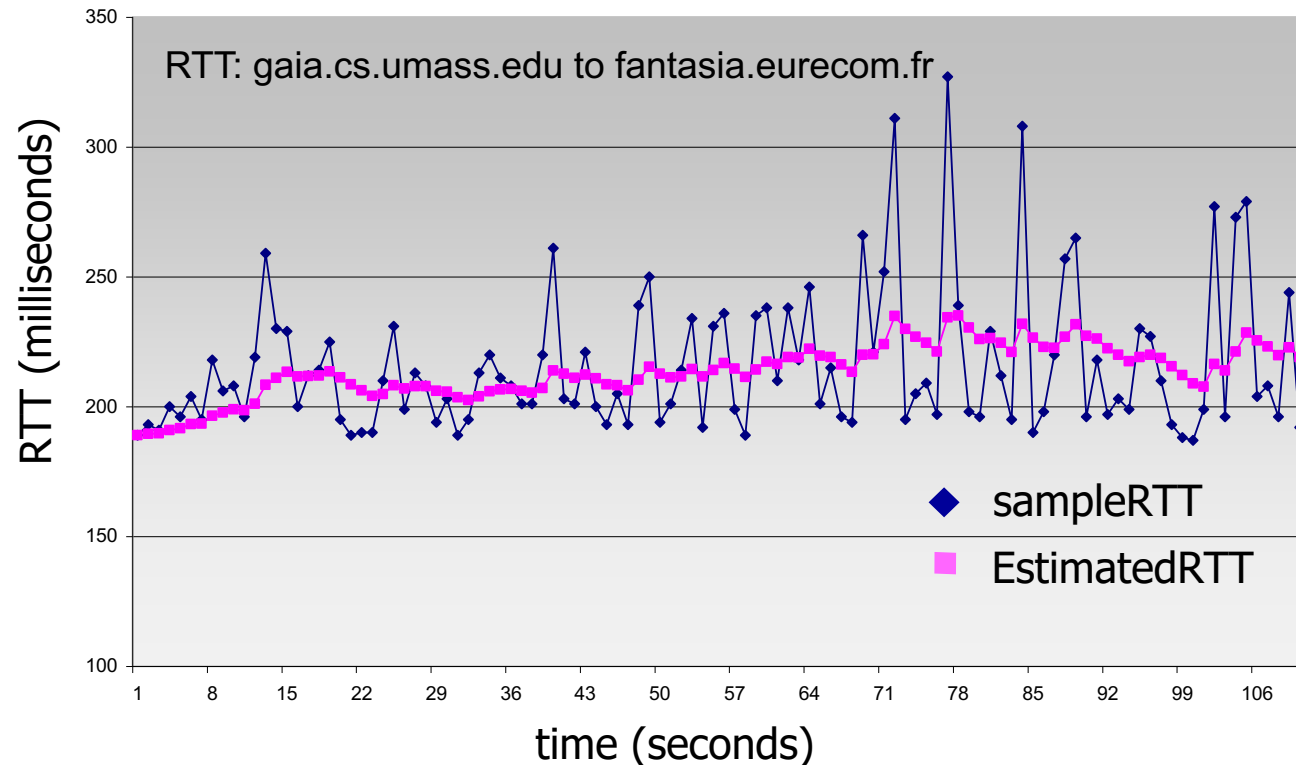
*D:* come stimare RTT?

- *SampleRTT*: tempo misurato dalla trasmissione del segmento fino alla ricezione dell'ACK
  - ignorare le ritrasmissioni (perché non sappiamo se a quale ritrasmissione l'ACK ricevuto si riferisca)
- *SampleRTT* varierà, si desidera un RTT più "regolare"
  - media delle misurazioni *recenti*, non solo l'ultimo *SampleRTT*

# TCP Round Trip Time e timeout

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

- exponential weighted moving average (EWMA)
- l'influenza del campione passato diminuisce in modo esponenziale
- valore tipico:  $\alpha = 0,125$





# Tempo di andata e ritorno TCP, timeout

- intervallo di timeout: **EstimatedRTT** più "margine di sicurezza"
  - grande variazione in **EstimatedRTT**: vogliamo margine di sicurezza più ampio

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



RTT stimato

"margine di sicurezza"

- **DevRTT**: EWMA della deviaz di **SampleRTT** da **EstimatedRTT**:

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

(tipicamente,  $\beta = 0,25$ )

# Mittente TCP (semplificato)

*evento: dati ricevuti  
dall'applicazione*

- crea segmento con seq #
- seq # è il numero nel byte-stream del primo byte di dati nel segmento
- avviare il timer se non è già in esecuzione
  - timer singolo collegato al segmento non confermato più vecchio
  - tempo di scadenza del timer: **TimeoutInterval**

*evento: timeout*

- ritrasmettere il segmento che ha causato il timeout
- riavvia il timer

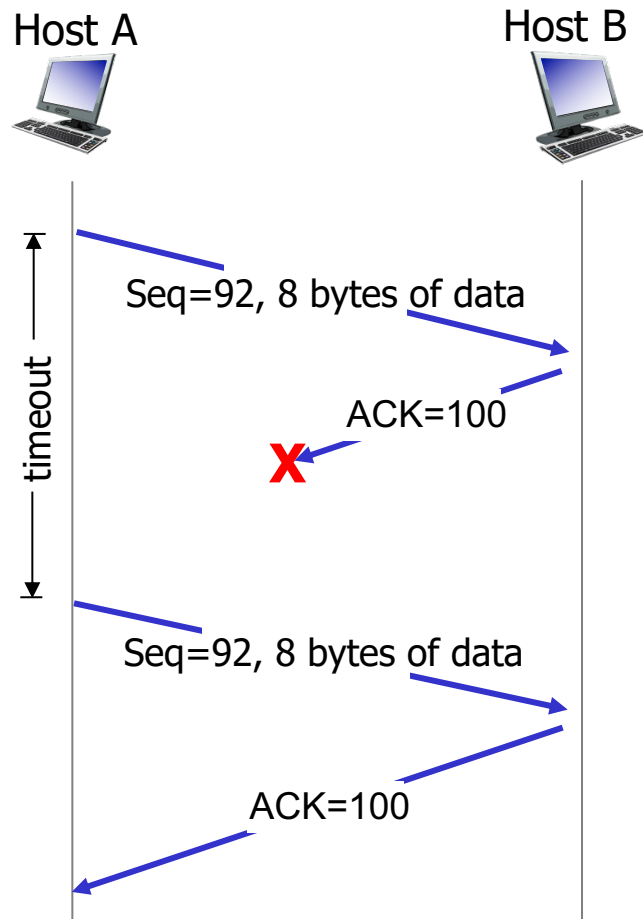
*evento: ACK ricevuto*

- se ACK copre segmenti precedentemente non confermati
  - aggiornare informazione su pacchetti confermati
  - avvia un timer se ci sono ancora segmenti non confermati (collegato al nuovo segmento più vecchio)

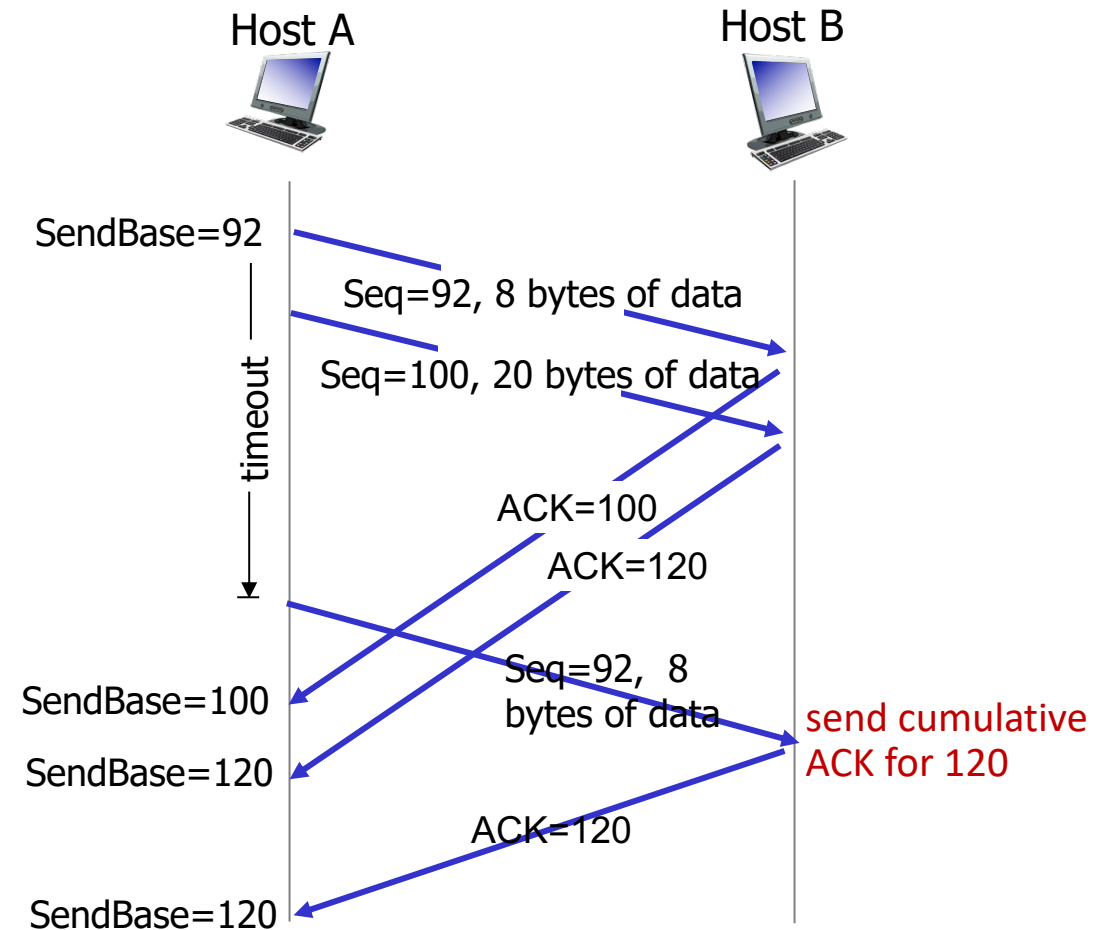
# Destinatario TCP: generazione di ACK [RFC 5681]

<i>Eventi a destinazione</i>	<i>Azione del destinatario TCP</i>
arrivo di un segmento in ordine con # seq atteso. Tutti i dati prec. già confermati (ACKed)	delayed ACK. Attendi fino a 500ms per il prossimo segmento. Se non arriva, invia ACK
arrivo di un segmento in ordine con # seq atteso. Un segmento precedente ancora non ACKed	invia immediatamente ACK cumulativo confermando entrambi i segmenti
arrivo segmento fuori ordine con seq # > atteso Gap detected	invio immediato di <i>ACK duplicato</i> , con #seq del prossimo byte atteso
arrivo di segmento che in parte o completamente riempie gap	Se il segmento riempie il gap in ordine invia ACK immediatamente

# TCP: scenari di ritrasmissione

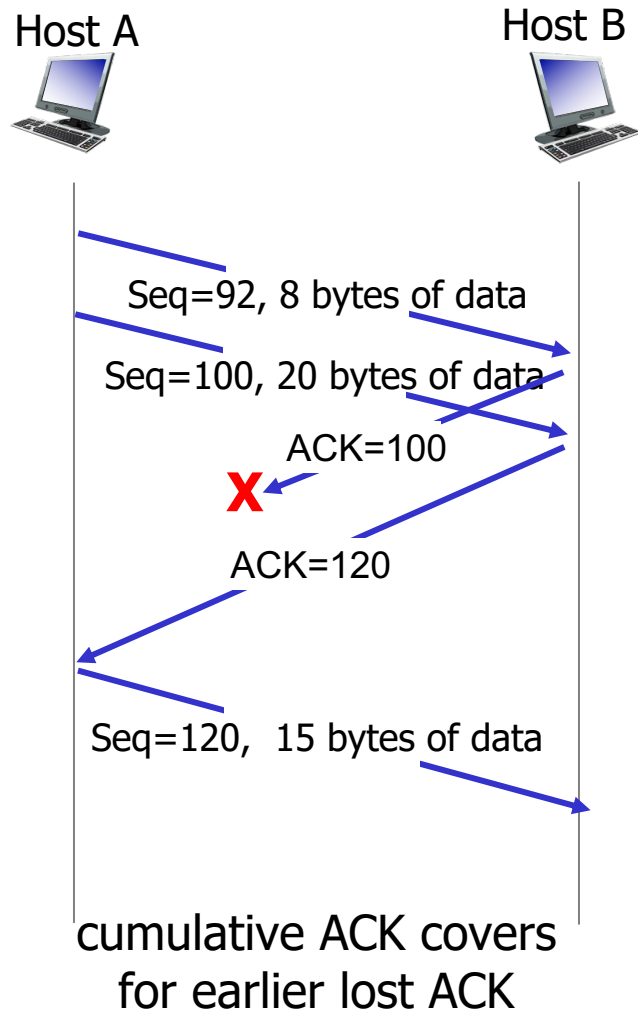


lost ACK scenario



premature timeout

# TCP: scenari di ritrasmissione



l'ACK delay non è mostrato

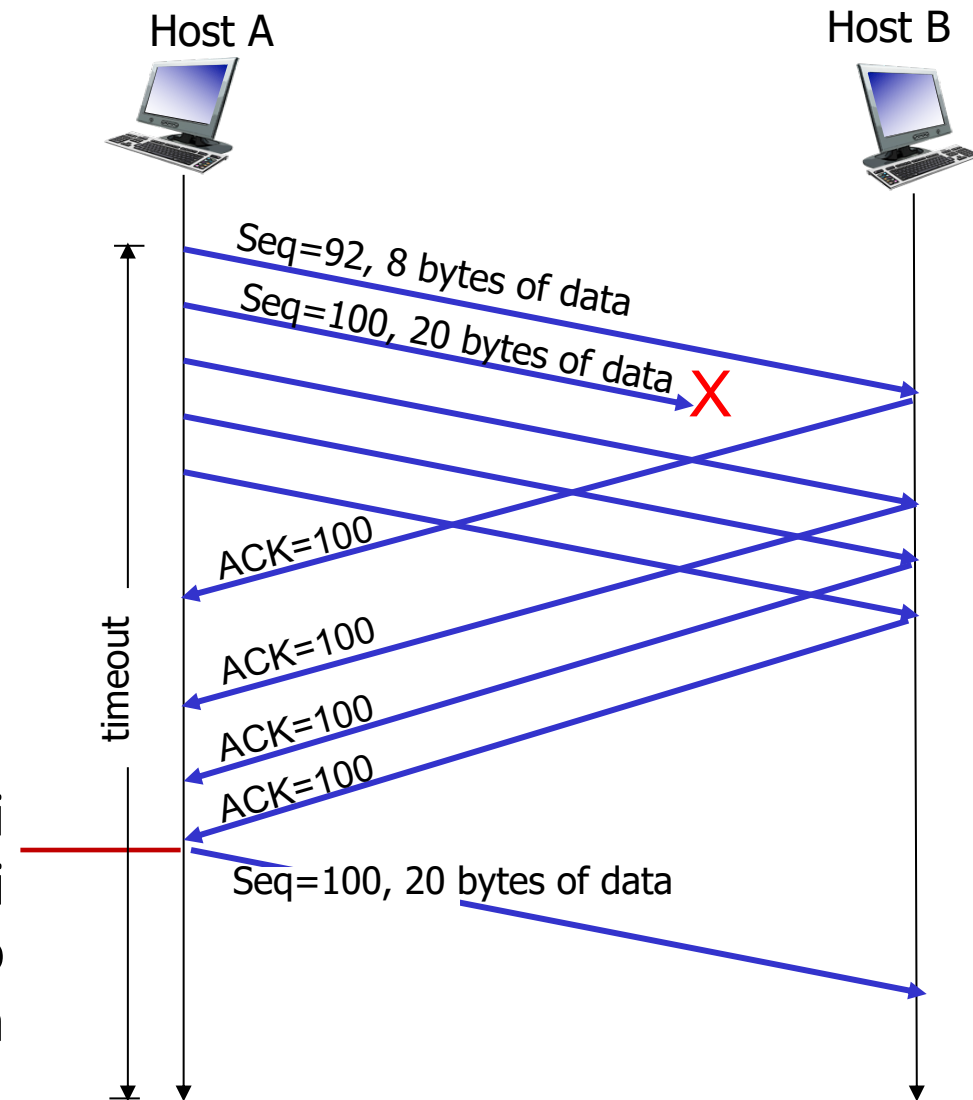
# TCP fast retransmit (ottimizzaz TCP originale)

## *Ritrasmissione rapida*

se il mittente riceve 3 ACK aggiuntivi per gli stessi dati ("triplo ACK duplicato"), invia nuovamente il segmento non confermato con numero di sequenza più piccolo

- probabilmente quel segmento non ACKed è andato perso, quindi non aspettare il timeout

💡 La ricezione di tre ACK duplicati indica che 3 segmenti sono stati ricevuti dopo un segmento mancante: è probabile che sia stato perso un segmento. Quindi ritrasmetti!

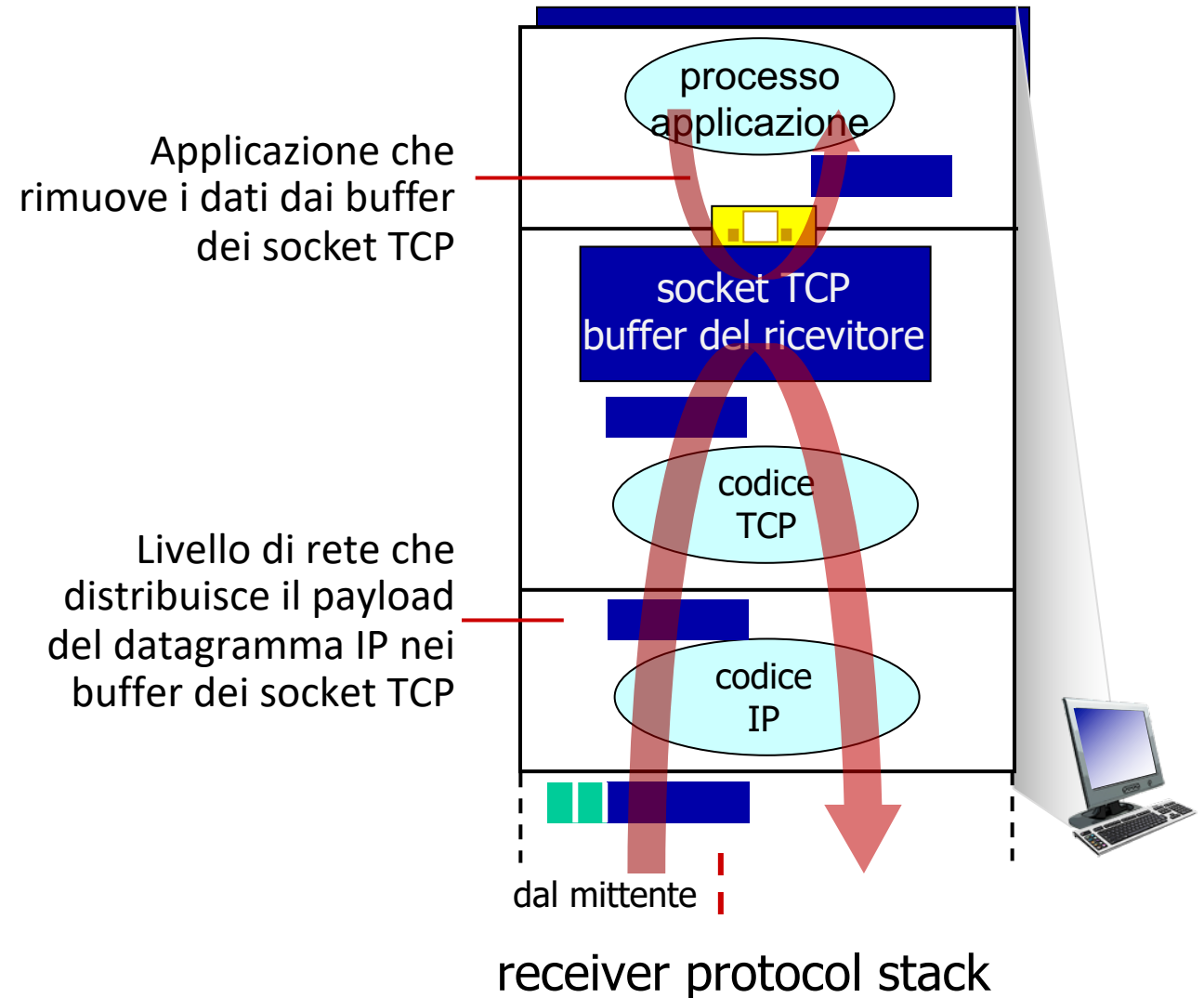


# Livello di trasporto: sommario

- Servizi a livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessione: UDP
- Principi di trasferimento affidabile dei dati
- **Trasporto orientato alla connessione: TCP**
  - struttura del segmento
  - trasferimento affidabile dei dati
  - **controllo del flusso**
  - **gestione della connessione**
- Principi di controllo della congestione
- Controllo della congestione TCP
- Evoluzione della funzionalità del livello di trasporto

# Controllo del flusso TCP

*D:* Cosa succede se il livello di rete fornisce i dati più velocemente rispetto al livello dell'applicazione che rimuove i dati dal buffer del socket?





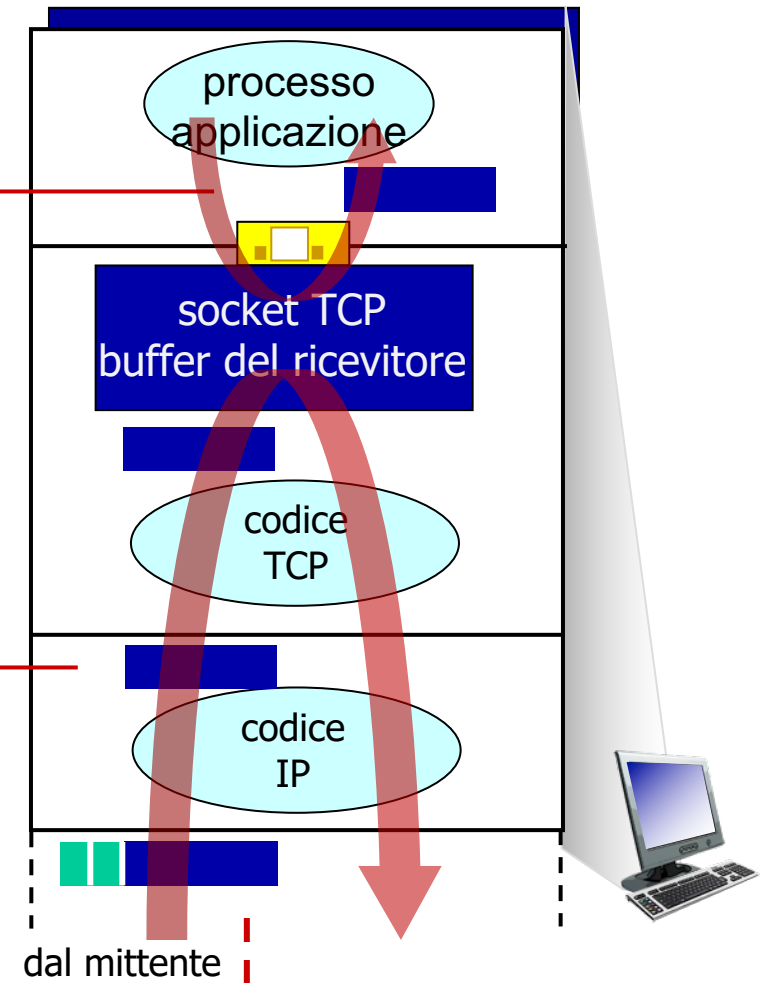
# Controllo del flusso TCP

**D:** Cosa succede se il livello di rete fornisce i dati più velocemente rispetto al livello dell'applicazione che rimuove i dati dal buffer del socket?



Applicazione che  
rimuove i dati dai buffer  
dei socket TCP

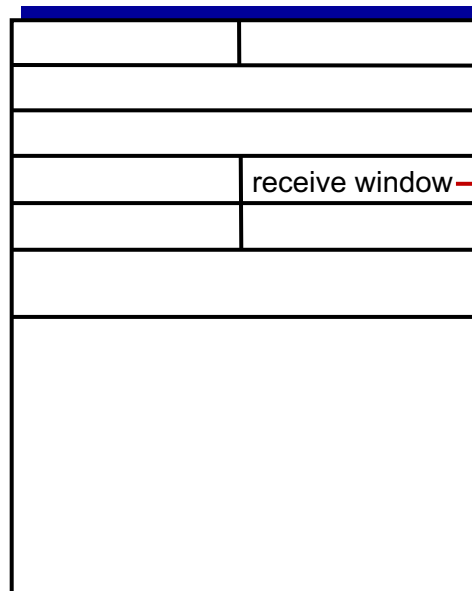
Livello di rete che distribuisce il payload del datagramma IP nei buffer dei socket TCP



pila di protocolli del ricevitore

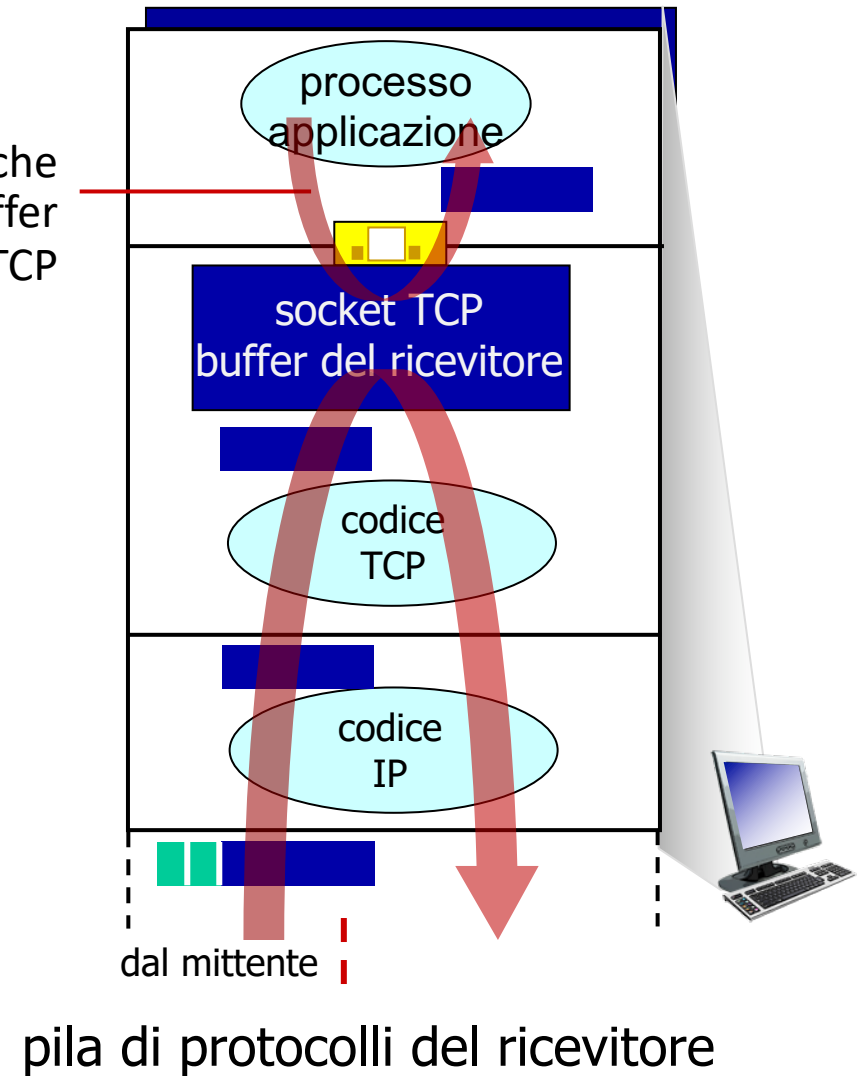
# Controllo del flusso TCP

**D:** Cosa succede se il livello di rete fornisce i dati più velocemente rispetto al livello dell'applicazione che rimuove i dati dal buffer del socket?



flow control: # bytes il destinatario è disposto ad accettare.

Applicazione che rimuove i dati dai buffer dei socket TCP



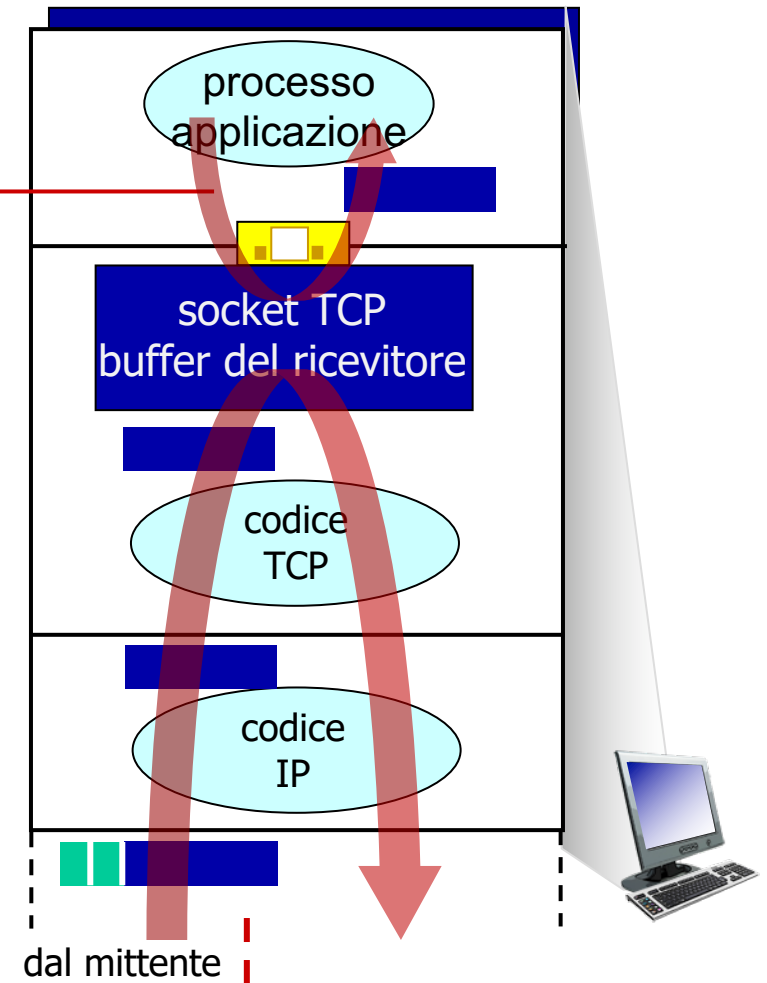
# Controllo del flusso TCP

*D:* Cosa succede se il livello di rete fornisce i dati più velocemente rispetto al livello dell'applicazione che rimuove i dati dal buffer del socket?

## controllo del flusso

ricevitore controlla il mittente, quindi il mittente non riempirà il buffer del destinatario trasmettendo troppi dati troppo velocemente

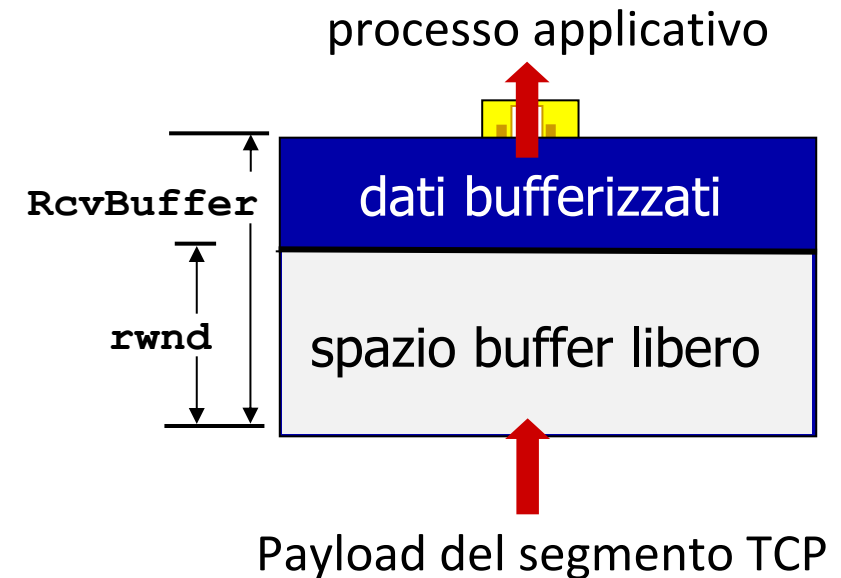
Applicazione che rimuove i dati dai buffer dei socket TCP



pila di protocolli del ricevitore

# Controllo del flusso TCP

- destinatario TCP "annuncia" lo spazio libero nel buffer nel campo **rwnd** nell'header TCP
  - dimensione du **RcvBuffer** impostata tramite le opzioni socket (l'impostazione predefinita tipica è 4096 byte)
  - molti sistemi operativi regolano automaticamente **RcvBuffer**
- mittente limita la quantità di dati non ACKed ("in volo") a **rwnd**
- garantisce che il buffer di ricezione non vada in overflow

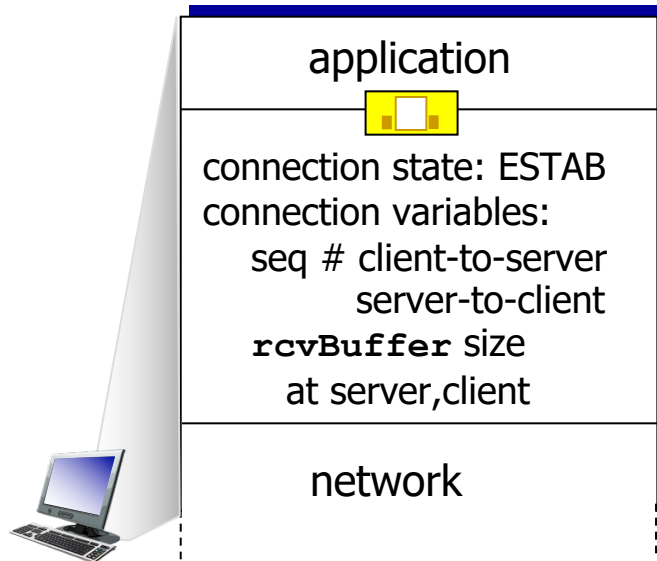


Buffering lato destinatario TCP

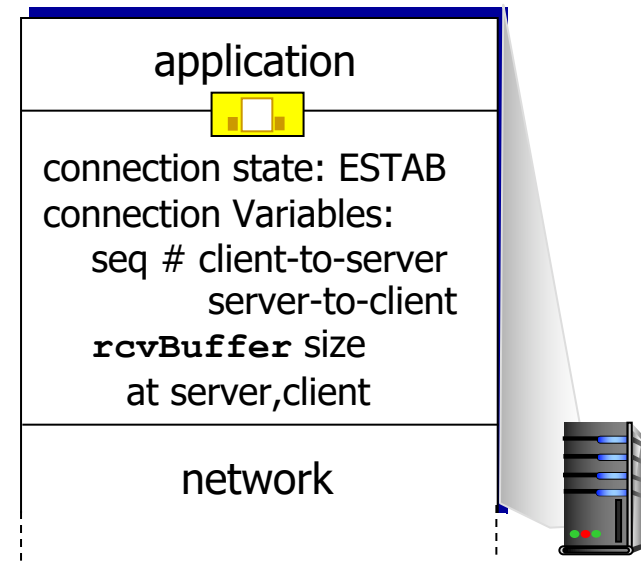
# Gestione della connessione TCP

prima dello scambio dati, mittente/destinatario fanno una “stretta di mano”:

- accettare di stabilire una connessione  
(ciascuno conosce la disponibilità dell'altro a stabilire una connessione)
- concordare i parametri di connessione (ad esempio, l'inizio di #seq)



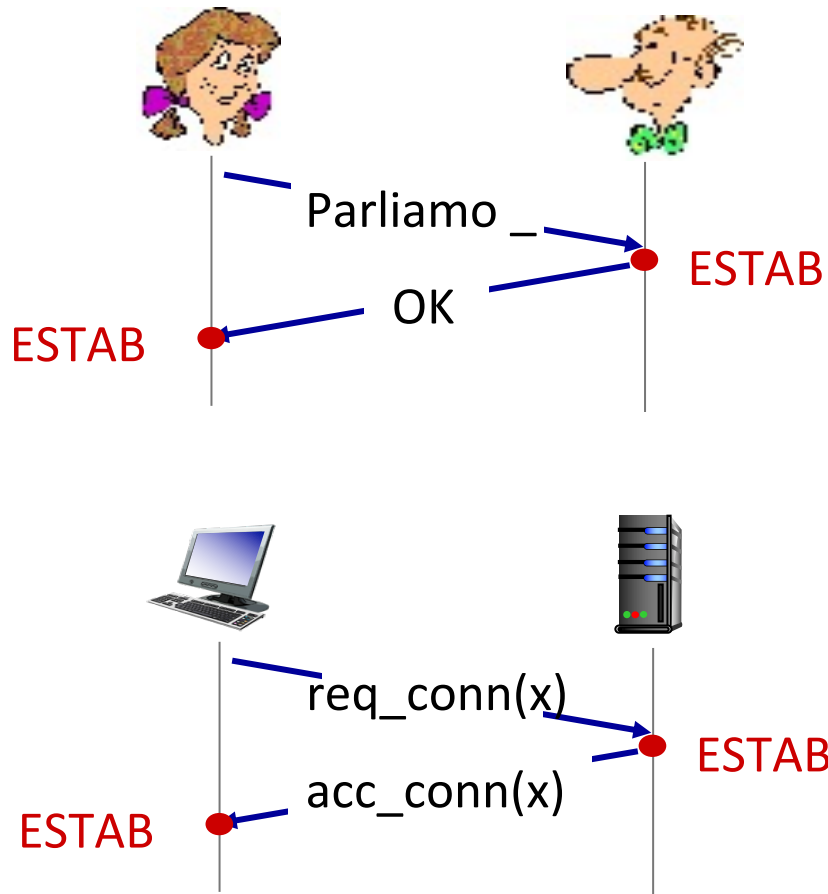
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



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

# Accettare di stabilire una connessione

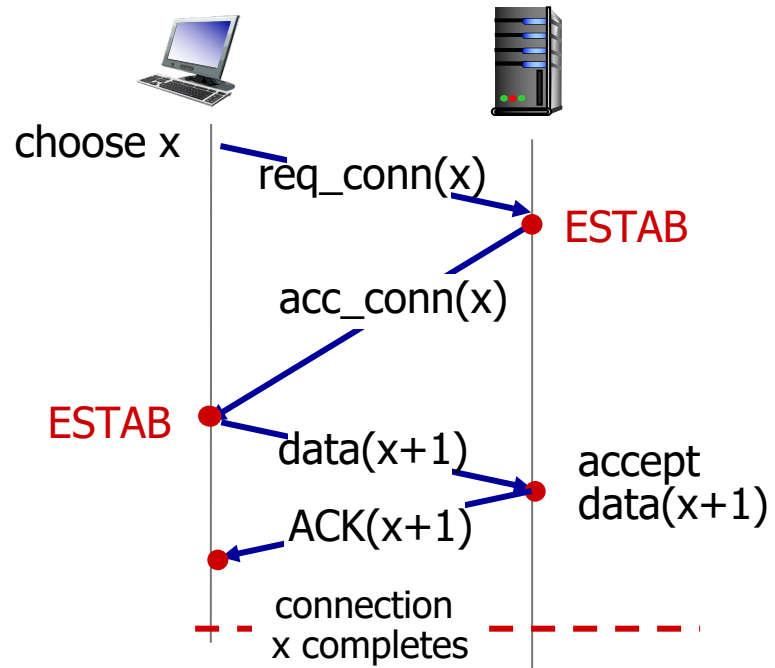
Stretta di mano a 2 vie:



*D:* l'handshake a 2 vie  
funzionerà sempre in rete?

- ritardi variabili
- messaggi ritrasmessi (ad esempio `req_conn(x)`) a causa della perdita del messaggio
- riordino dei messaggi
- non posso "vedere" l'altro lato

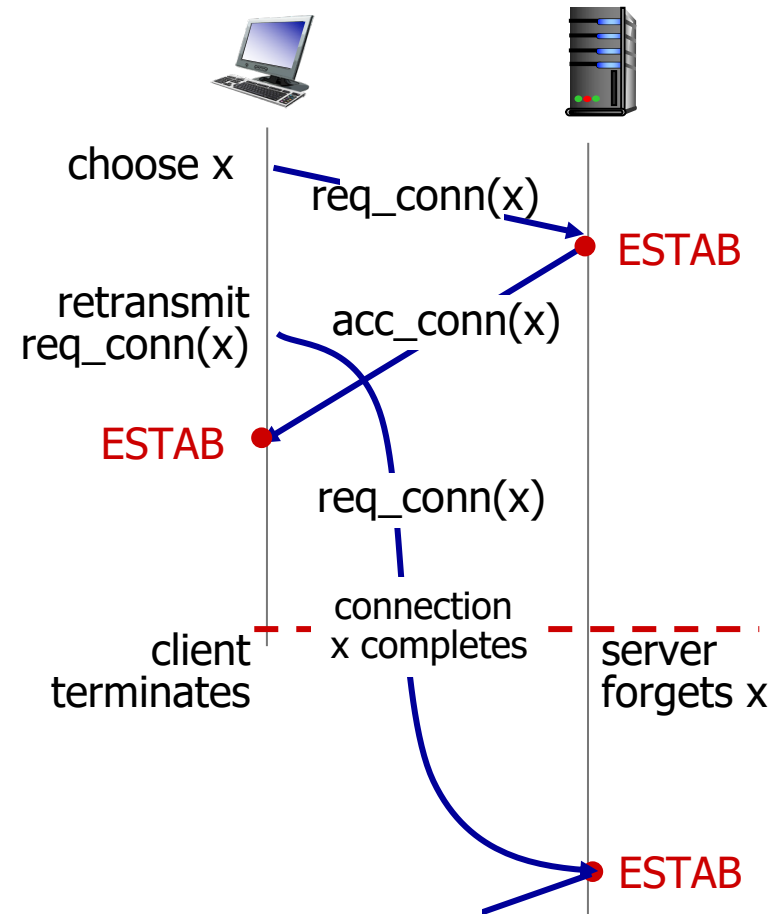
# Scenari di handshake a 2 vie




No problem!



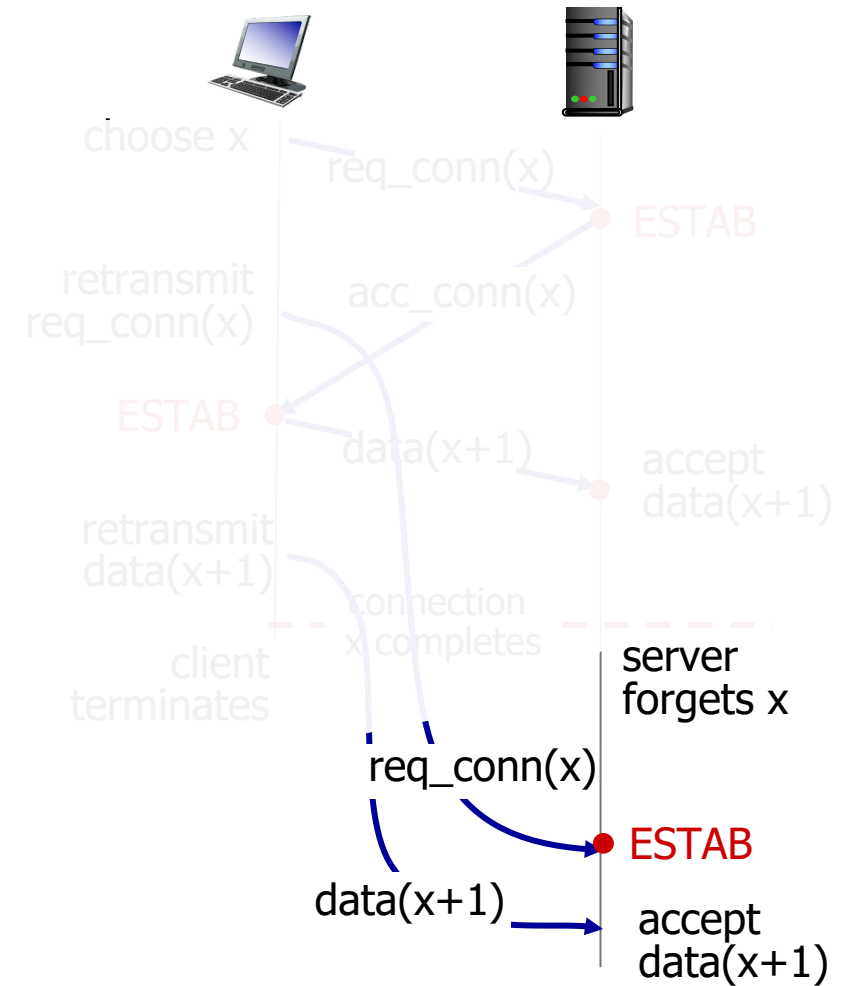
# Scenari di handshake a 2 vie



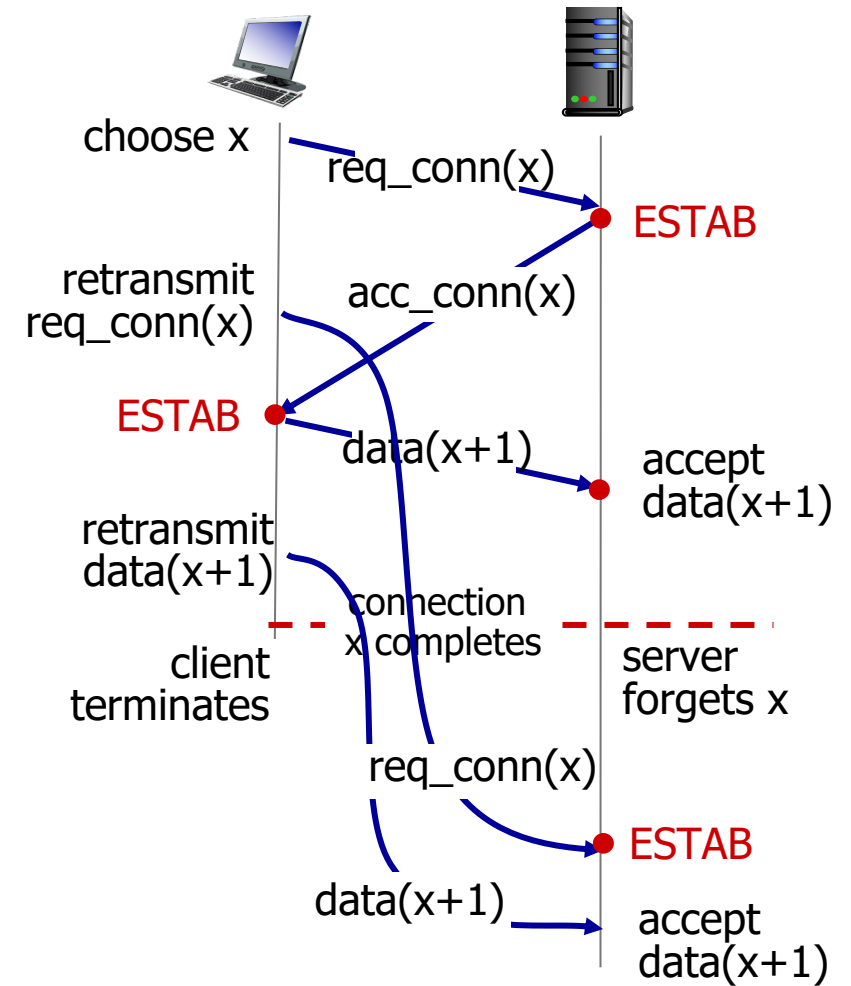
 Problema: connessione aperta a metà! (no client)



# Scenari di handshake a 2 vie



# Scenari di handshake a 2 vie



Problema: dati duplicati  
potenzialmente accettati

# Handshake TCP a 3 vie (3-way)

## Server state

```
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

## Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName, serverPort))
```

SYNSENT

ESTAB

scegli num seq iniz, x  
invia msg TCP SYN

SYNbit=1, Seq=x

SYNbit=1, Seq=y  
ACKbit=1; ACKnum=x+1

riceve SYNACK(x)  
indica che server è live;  
invia ACK for SYNACK;  
questo segmento può  
anche includere dati  
dal client al server

ACKbit=1, ACKnum=y+1

riceve ACK(y)  
indica client è live

LISTEN

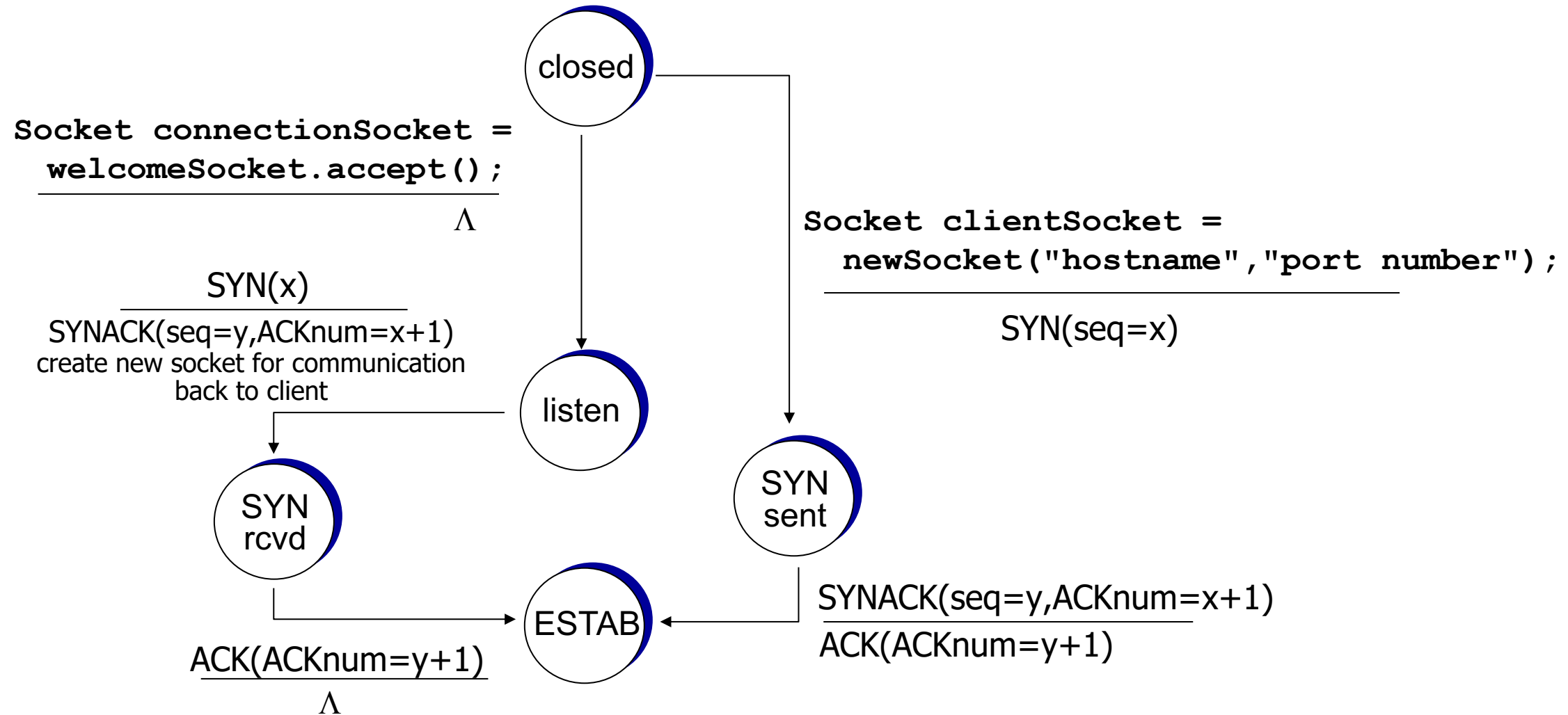
SYN RCVD

ESTAB

# Motivazione del 3-way handshake

- Il numero di sequenza x e y sono scelti in maniera casuale (random)
- Il 3-way handshake permette a entrambi gli host di conoscere il numero di sequenza dell'altro e quindi **stabilire univocamente una connessione bidirezionale** (garantire che entrambi si riferiscono alla stessa connessione)
- L'uso di numeri di sequenza univoci permette di individuare (nel SYNACK) richieste di connessione duplicate
- Il problema di connessione metà aperta si può verificare anche con il 3-way handshake
  - Può essere anche intenzionale (SYN flood = half open attack)

# TCP 3-way handshake FSM



# Un protocollo di stretta di mano umana a 3 vie



# Chiusura di una connessione TCP

- client, server chiudono ciascuno il proprio lato di connessione
  - invia il segmento TCP con il bit FIN = 1
- rispondere al FIN ricevuto con ACK
  - alla ricezione del FIN, l'ACK può essere combinato con il proprio FIN
- possono essere gestiti scambi FIN simultanei