Università degli Studi di Roma "Tor Vergata"
Laurea in Informatica

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2024/2025

Livello di trasporto (parte3)

dr. Manuel Fiorelli

manuel.fiorelli@uniroma2.it

https://art.uniroma2.it/fiorelli

Basate sulle slide del libro di testo:
https://gaia.cs.umass.edu/kurose_ross/ppt.php

Capitolo 3: tabella di marcia

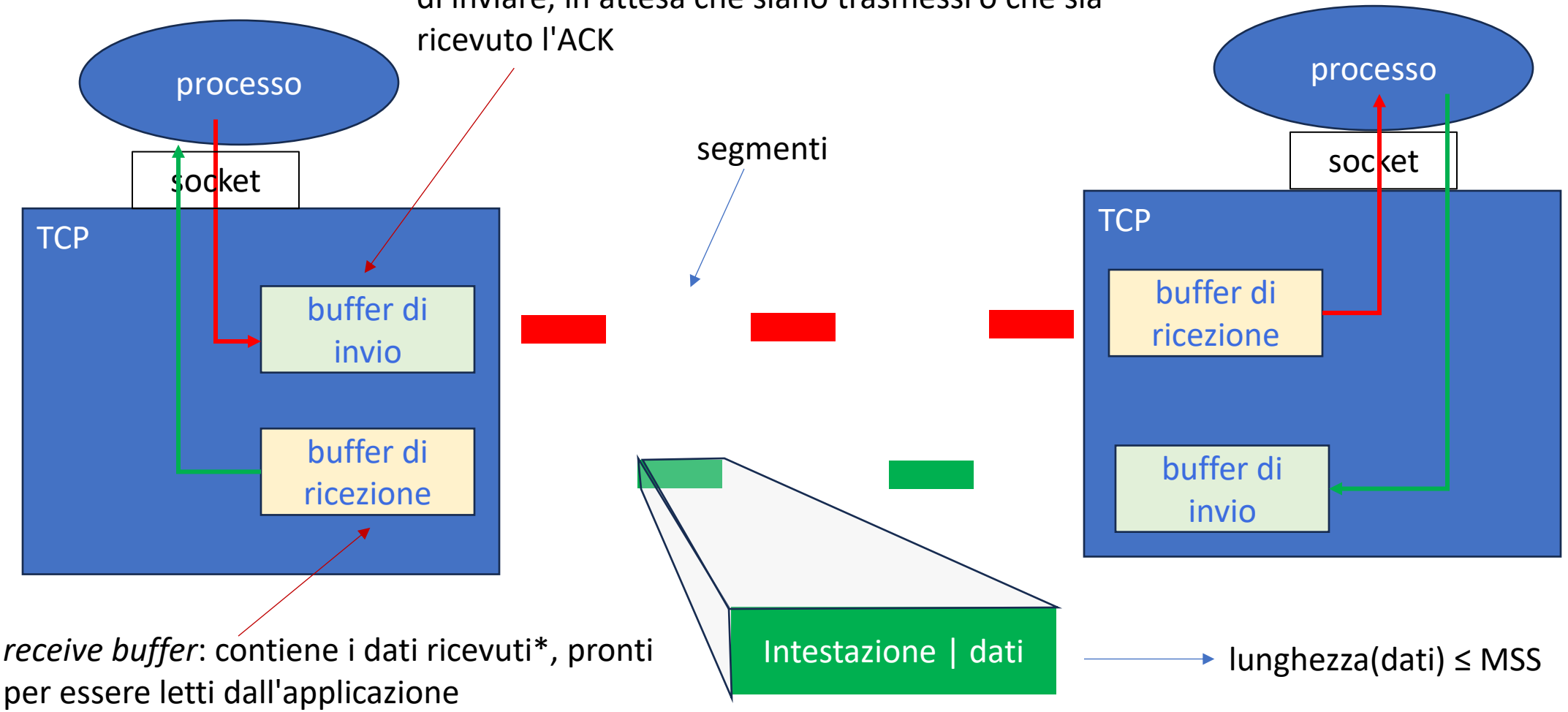
- Servizi a livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessione: UDP
- Principi del trasferimento dati affidabile
- Trasporto orientato alla connessione: TCP
  - struttura dei segmenti
  - trasferimento dati affidabile
  - controllo di flusso
  - controllo della congestione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

TCP: panoramica RFCs: 793,1122, 2018, 5681, 7323

- punto a punto (point-to-point):
  - singolo mittente, singolo destinatario
- flusso di byte affidabile (reliable), in sequenza (in order):
  - nessun “confine ai messaggi"
- full duplex data:
  - i dati possono fluire in direzioni opposte allo stesso istante nella stessa connessione
  - MSS: dimensione massima del segmento (maximum segment size); in realtà si riferisce ai dati applicativi nel segmento (intestazione esclusa)
- ACK cumulativi
- pipelining:
  - il controllo di flusso e della congestione definiscono la dimensione della finestra
- orientato alla connessione:
  - l'handshaking (scambio di messaggi di controllo) inizializza lo stato del mittente e del destinatario prima di scambiare i dati
  - una connessione TCP non definisce un circuito end-to-end del tipo che caratterizza le reti a commutazione di circuito. Infatti, le connessioni TCP sono interamente implementate nei sistemi periferici, mentre i commutatori di pacchetto vedono solo i pacchetti di rete in transito
- flusso controllato:
  - il mittente non sovraccarica il destinatario
- controllo della congestione:
  - il mittente riduce la velocità di invio in funzione della congestione della rete

TCP: Buffer di invio e ricezione TCP.

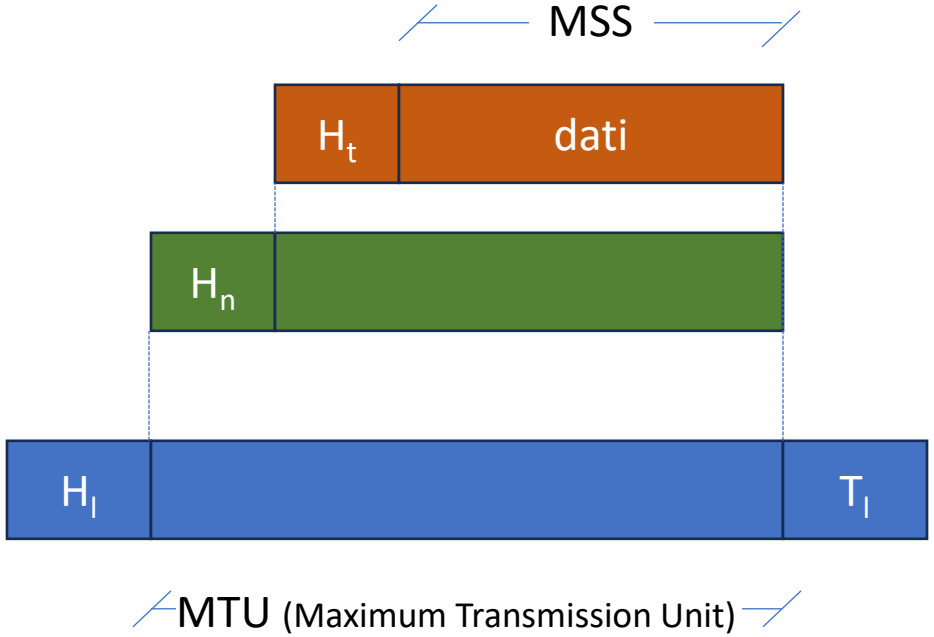
send buffer: contiene i dati che il processo ha chiesto di inviare, in attesa che siano trasmessi o che sia ricevuto l'ACK



receive buffer: contiene i dati ricevuti*, pronti per essere letti dall'applicazione

**la gestione dei segmenti fuori ordine richiede l'uso di ulteriori strutture dati e variabili (non mostrate); a prescindere da dove questi dati siano salvati dalla implementazione di TCP, il processo può accedere solo alla parte del buffer contenente dati in ordine*

TCP: MSS



La dimensione massima del segmento TCP predefinita per IPv4 è di 536. Per IPv6 è 1220. Se un host desidera impostare MSS su un valore diverso da quello predefinito, il suo valore viene specificata come opzione TCP, inizialmente nel pacchetto TCP SYN durante l'handshake TCP.

- MSS + lunghezza(H_t) + lunghezza(H_n) ≤ MTU

Valori tipici:

lunghezza(H_t) = 20 B

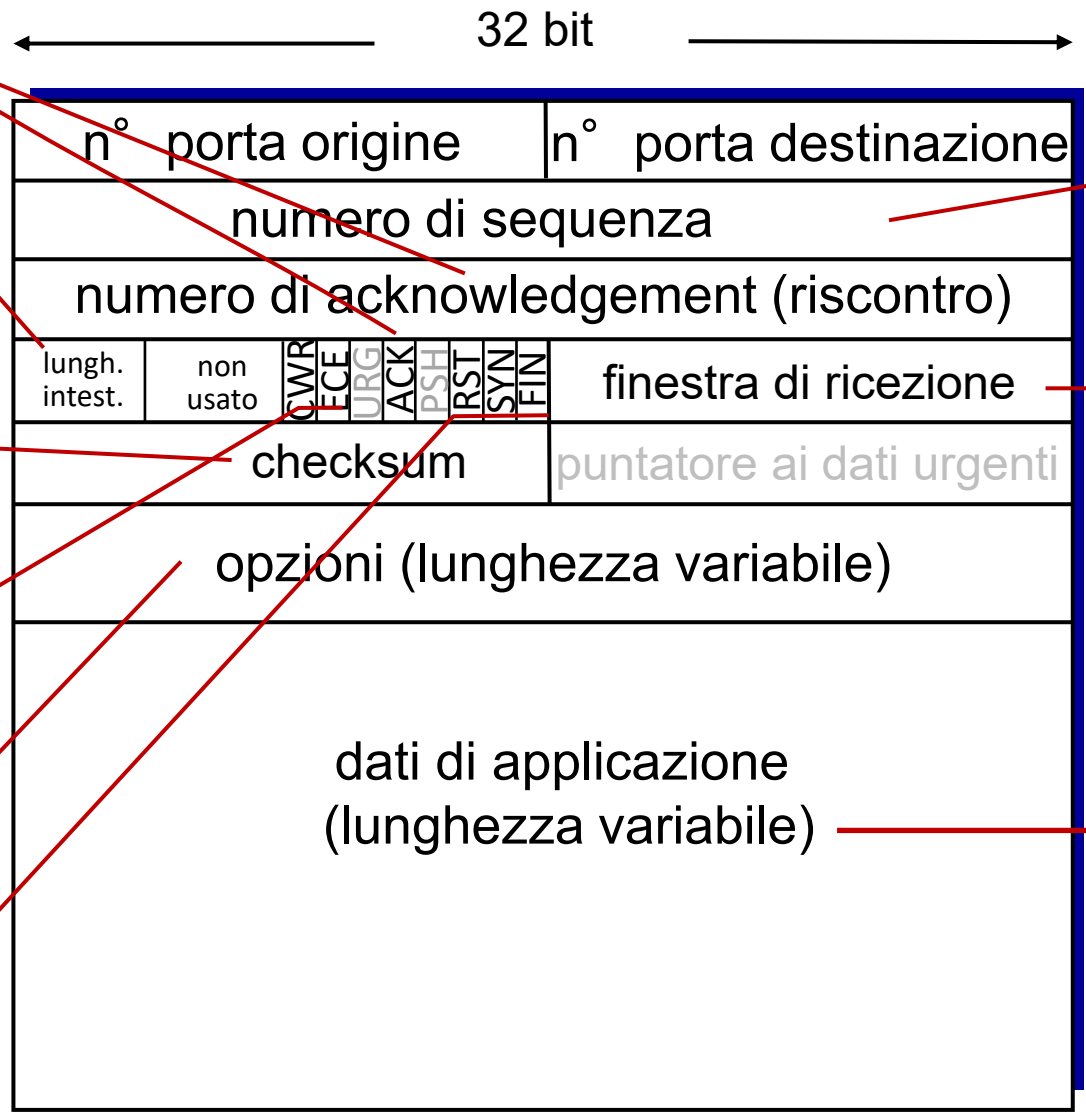
lunghezza(H_n) = 20 B

$MTU_{Ethernet}$= 1500 -> MSS = 1460

- Un host può determinare il MSS guardando la MTU del collegamento locale; ma ciò non offre garanzie circa altri collegamenti intermedi
- Può essere negoziato durante la connessione con l'opzione MSS
- Path MTU Discovery: permette di scoprire il valore più piccolo della MTU lungo il percorso da mittente a destinatario

Se un pacchetto IP eccede la MTU su un collegamento di uscita, il router dovrà frammentarlo (in IPv4) oppure scartalo (in IPv6).

Struttura dei segmenti TCP



ACK: numero di sequenza del prossimo byte atteso; flag A: numero di riscontro valido

Lunghezza dell'intestazione TCP (in multipli di 32 bit)

checksum Internet (come in UDP)

CRW, ECE: flag usati per il controllo di congestione esplicito

opzioni TCP (generalmente non usate)

RST, SYN, FIN: gestione della connessione

conteggio per byte di dati (non segmenti!)

controllo di flusso: numero di byte che il destinatario desidera accettare

dati inviati dall'applicazione nella socket TCP

(possibilmente suddivisi in più segmenti, a differenza di UDP in cui l'applicazione legge/scrivi messaggi che devono corrispondere a un segmento)

Lunghezza dell'intestazione variabile, ma valore tipico è 20 byte (senza alcuna opzione).

Numeri di sequenza e ACK di TCP

Numeri di sequenza:

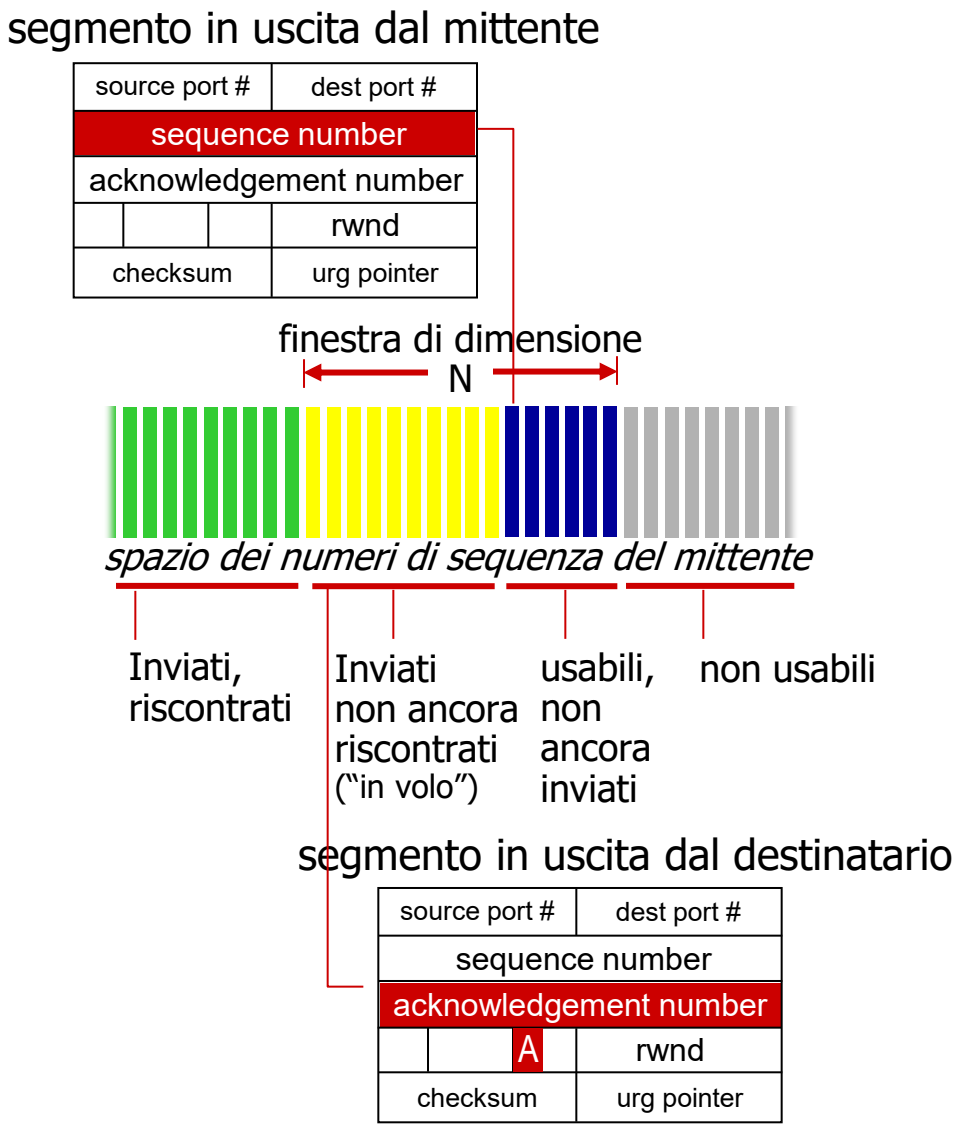
- "numero" del primo byte nel segmento nel flusso di byte

ACK:

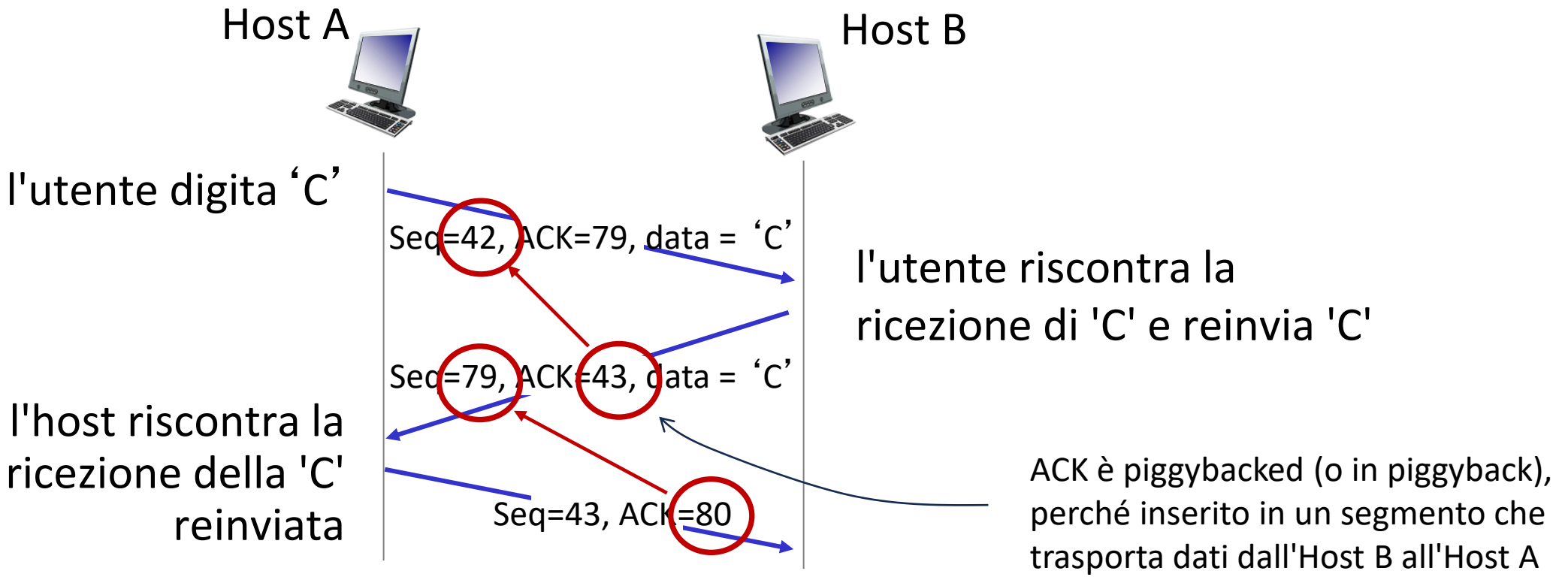
- numero di sequenza del prossimo byte atteso dall'altro lato
- ACK cumulativo
- RFC 2018: Acknowledgment Selettivo (che non studieremo)

D: come gestisce il destinatario i segmenti fuori sequenza?

- *R:* la specifica TCP non lo dice – dipende dall'implementatore



Numeri di sequenza e ACK di TCP



un semplice serve echo

TCP: tempo di andata e ritorno (round trip time) e timeout

D: come impostare il valore del timeout di TCP?

- più grande di RTT (tempo trascorso da quando si invia un segmento a quando se ne riceve l'acknowledgment), ma RTT non è noto varia!
- *troppo piccolo:* timeout prematuro, ritrasmissioni non necessarie
- *troppo grande:* reazione lenta alla perdita di segmenti

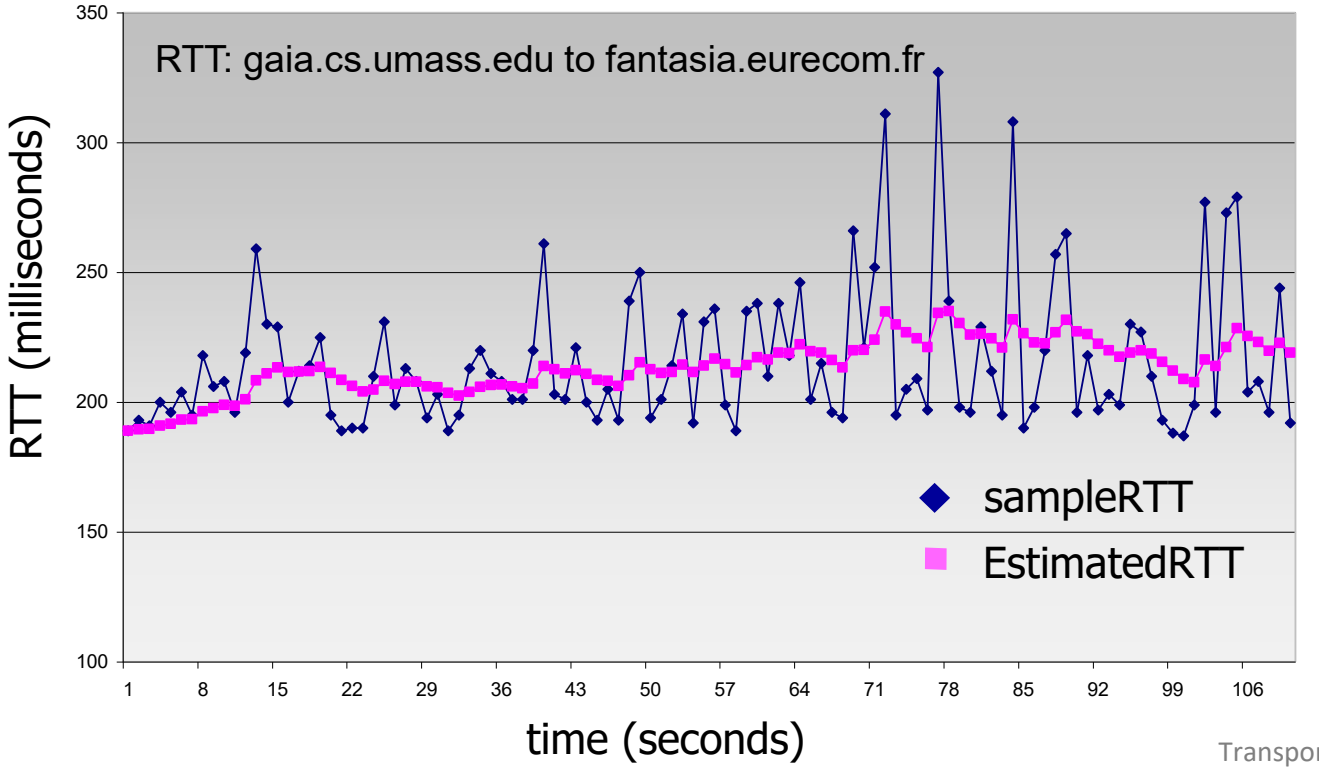
D: come stimare RTT?

- `SampleRTT:` tempo misurato dalla trasmissione del segmento fino alla ricezione di ACK
  - ignora le ritrasmissioni
- `SampleRTT` varia, quindi occorre una stima più "livellata" di RTT
  - *media di più misure recenti*, non semplicemente il calore corrente di `SampleRTT`

TCP: tempo di andata e ritorno (round trip time) e timeout

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

- exponential weighted moving average (EWMA): media mobile esponenziale pesata
- l'influenza dei vecchi campioni decresce esponenzialmente
  - i campioni più recenti riflettono meglio la congestione attuale della rete
- Valore raccomandato: $\alpha = 0.125$



TCP: tempo di andata e ritorno (round trip time) e timeout

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

$EstimatedRTT_n$ = (1- α)*$EstimatedRTT_{n-1}$ + α*$SampleRTT_n$

$$EstimatedRTT_n = (1-\alpha)\cdot EstimatedRTT_{n-1} + \alpha \cdot SampleRTT_n$$

$$= (1-\alpha)\cdot\left((1-\alpha)\cdot EstimatedRTT_{n-2} + \alpha \cdot SampleRTT_{n-1}\right) + \alpha \cdot SampleRTT_n$$

$$= (1-\alpha)^2 \cdot EstimatedRTT_{n-2} + \alpha(1-\alpha)\cdot SampleRTT_{n-1} + \alpha \cdot SampleRTT_n$$

$$= (1-\alpha)^2 \cdot\left((1-\alpha)\cdot EstimatedRTT_{n-3} + \alpha \cdot SampleRTT_{n-2}\right) + \alpha(1-\alpha)\cdot SampleRTT_{n-1} + \alpha \cdot SampleRTT_n$$

$$= (1-\alpha)^3 \cdot EstimatedRTT_{n-3} + \alpha(1-\alpha)^2 \cdot SampleRTT_{n-2} + \alpha(1-\alpha)\cdot SampleRTT_{n-1} + \alpha \cdot SampleRTT_n$$

$$= (1-\alpha)^n \cdot EstimatedRTT_0 + \alpha(1-\alpha)^{n-1} \cdot SampleRTT_1 + \cdots + \alpha(1-\alpha)\cdot SampleRTT_{n-1} + \alpha \cdot SampleRTT_n$$

$$= (1-\alpha)^n \cdot EstimatedRTT_0 + \sum_{i=0}^{n-1} \alpha(1-\alpha)^i \cdot SampleRTT_{n-i}$$

Essendo 0 < α < 1, il peso dei campioni vecchi decresce in modo esponenziale

TCP: tempo di andata e ritorno (round trip time) e timeout

- Intervallo di timeout: **`EstimatedRTT`** più un "margine di sicurezza"
  - grande variazione di **`EstimatedRTT`**: margine di sicurezza maggiore

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

RTT stimato

"margine di sicurezza"

- **`DevRTT`**: EWMA della deviazione di **`SampleRTT`** da **`EstimatedRTT`**:

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

(tipicamente, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Mittente TCP (semplificato)

evento: ricevuti dati dall'applicazione

- crea un segmento con il numero di sequenza
- il numero di sequenza è il numero del primo byte del segmento nel flusso di byte
- avvia il timer, se non già in funzione
  - pensate al timer come se fosse associato al più vecchio segmento non ancora riscontrato
  - intervallo di scadenza: **`TimeOutInterval`**

evento: timeout

- ritrasmette il segmento che ha causato il timeout (cioè il segmento in attesa di ACK con il più piccolo numero di sequenza)
- riavvia il timer

la durata del timeout viene raddoppiata, ma torna a usare la formula standard non appena viene ricevuto un ACK per un nuovo segmento

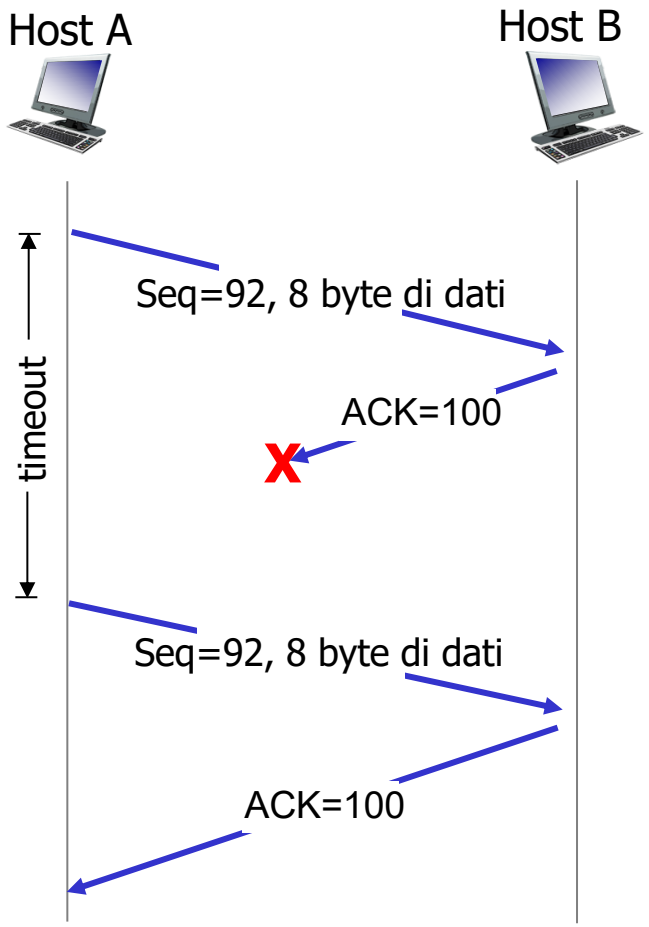
evento: ACK ricevuto

- se riscontra segmenti precedentemente non riscontrati (ACK y dove y > SendBase)
  - aggiorna ciò che si sa essere stato riscontrato (SendBase = y)
  - avvia il timer se ci sono altri segmenti ancora non riscontrati

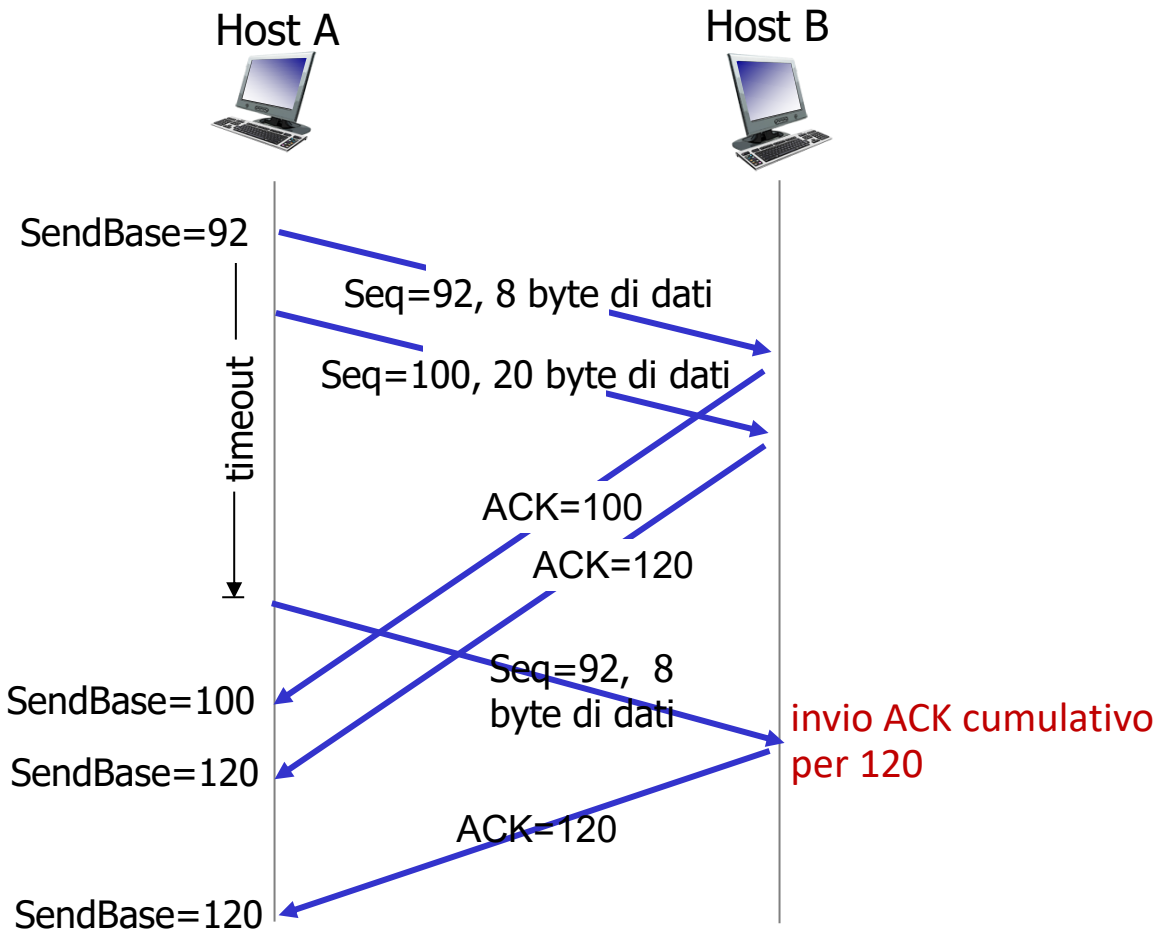
Ricevente TCP: Generazione degli ACK [RFC 5681]

Evento presso il destinatario	*Azione del ricevente TCP*
Arrivo ordinato di segmento con numero di sequenza atteso. Tutti i dati fino al numero di sequenza attivo sono già stati riscontrati.	ACK ritardato. Attende fino a 500 millisecondi per l'arrivo ordinato di un altro segmento. Se in questo intervallo non arriva il successivo segmento, invia un ACK.
Arrivo ordinato di segmento con numero di sequenza atteso. Un altro segmento ordinato è in attesa di trasmissione dell'ACK.	Invia immediatamente un singolo ACK cumulativo, riscontrando entrambi i segmenti ordinati.
Arrivo non ordinato di segmento con numero di sequenza superiore a quello atteso. Viene rilevato un buco.	Invia immediatamente un *ACK duplicato*, indicando il numero di sequenza del prossimo byte atteso (che è l'estremità inferiore del buco)
Arrivo di segmento che colma parzialmente o completamente il buco nei dati ricevuti.	Invia immediatamente un ACK, ammesso che il segmento cominci all'estremità inferiore del buco

TCP: scenari di ritrasmissione

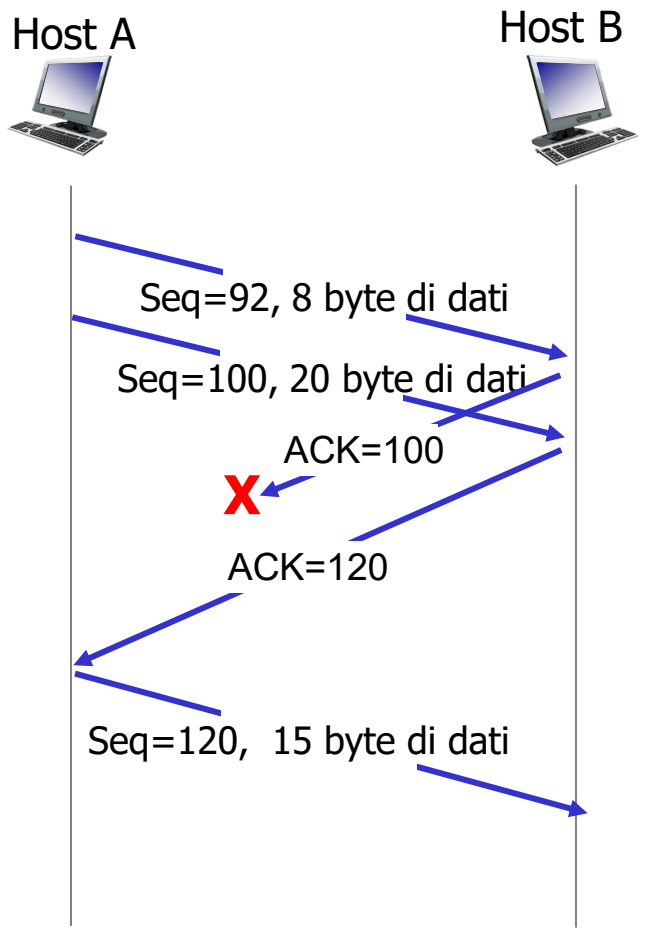


scenario con ACK perso



timeout prematuro

TCP: scenari di ritrasmissione



l'ACK cumulativo copre
l'ACK precedente perso

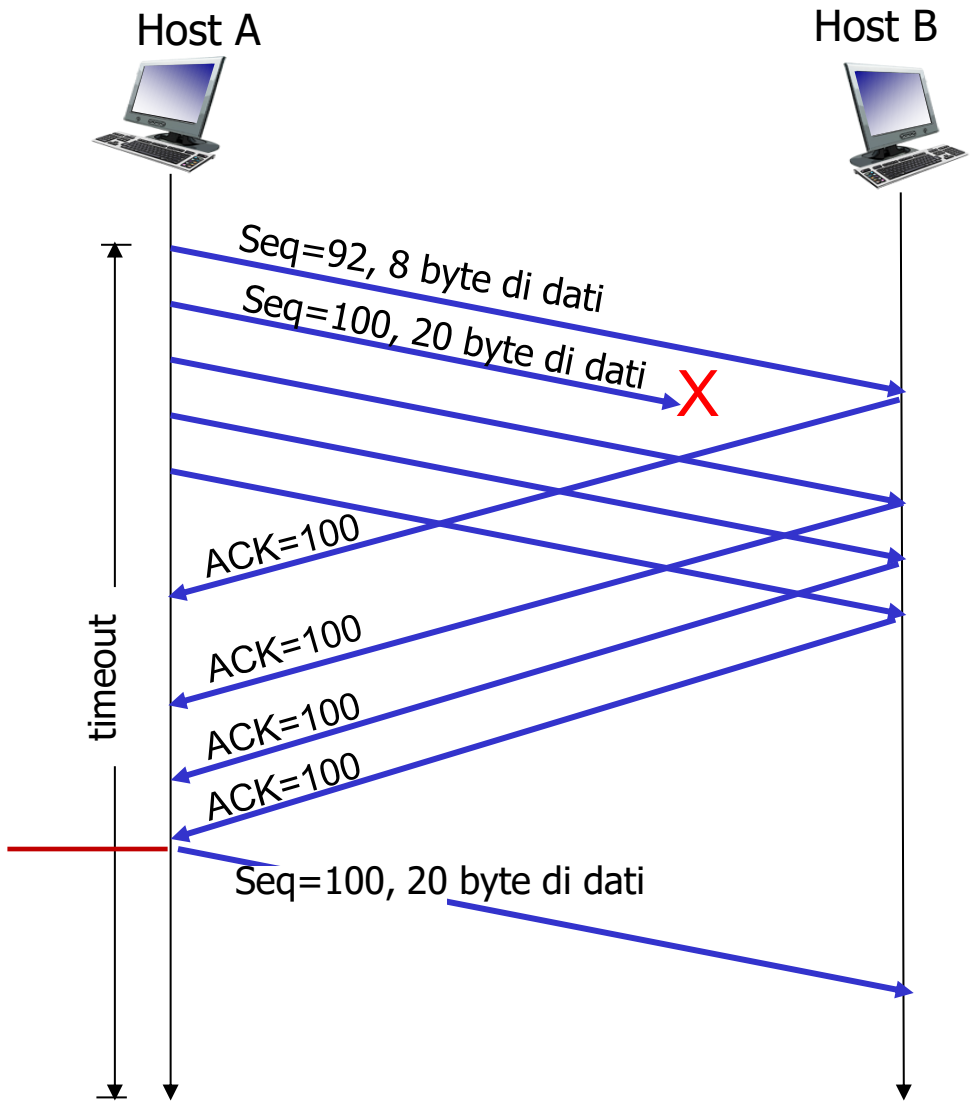
Ritrassmissione rapida

Ritrasmissione rapida

Se il mittente riceve 3 ACK addizionali per gli stessi dati ("3 ACK duplicati"), rispedisce il segmento non riscontrato con il più piccolo numero di sequenza

- è probabile che il segmento non riscontrato sia stato perso, quindi non aspettare il timeout.

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

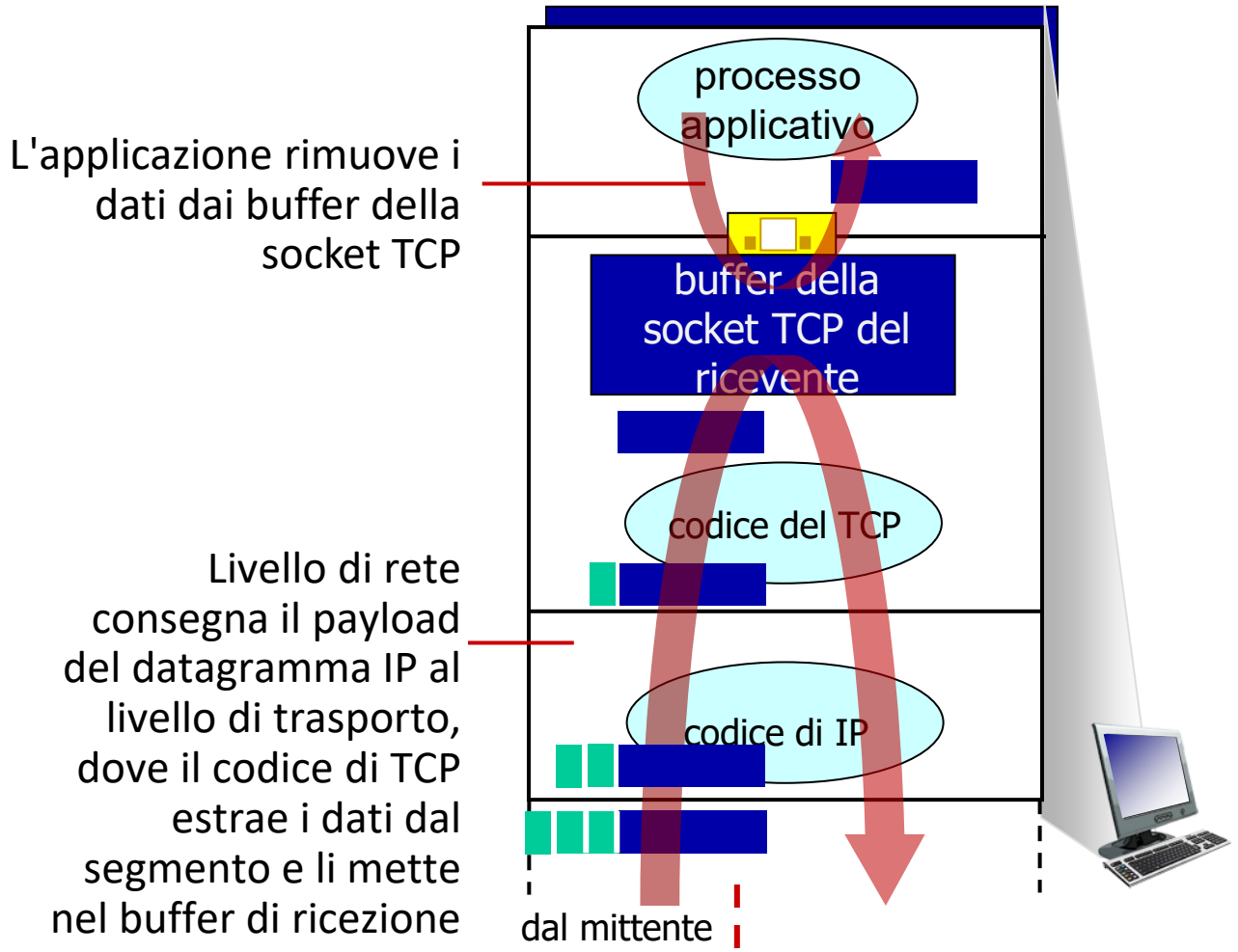


Chapter 3: roadmap

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

TCP: controllo di flusso

D: Cosa succede se il livello di rete fornisce i dati più velocemente di quanto il livello applicativo rimuova i dati dai buffer delle socket?



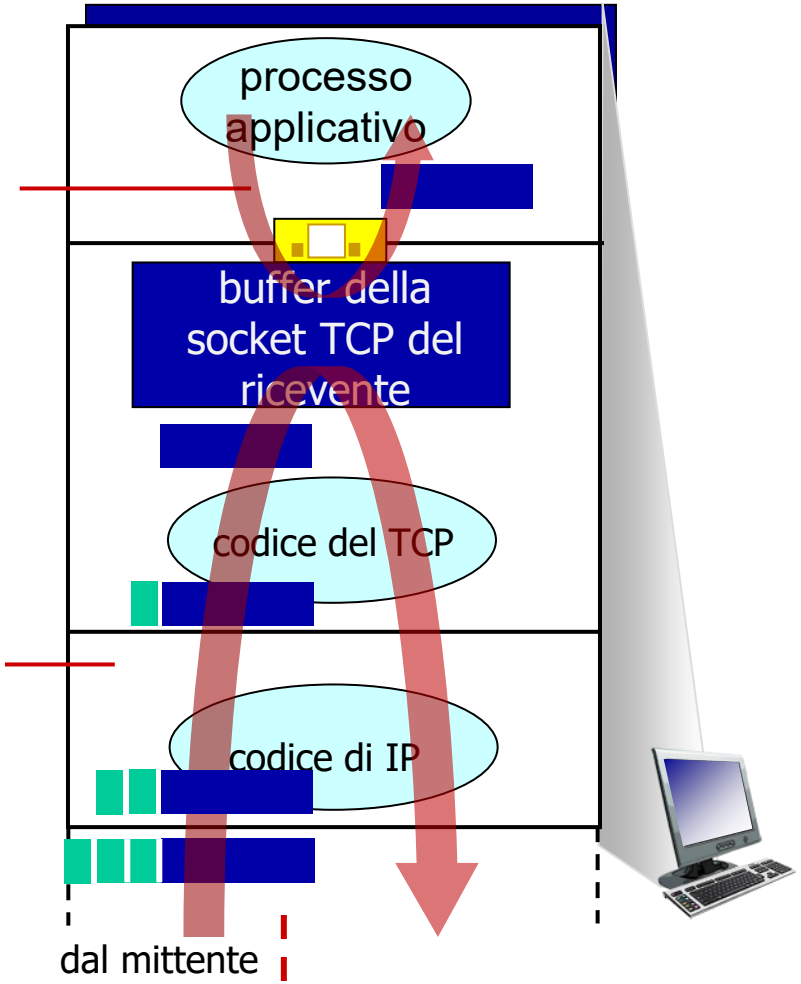
pila protocollare del ricevente

TCP: controllo di flusso

D: Cosa succede se il livello di rete fornisce i dati più velocemente di quanto il livello applicativo rimuova i dati dai buffer delle socket?

L'applicazione rimuove i dati dai buffer della socket TCP

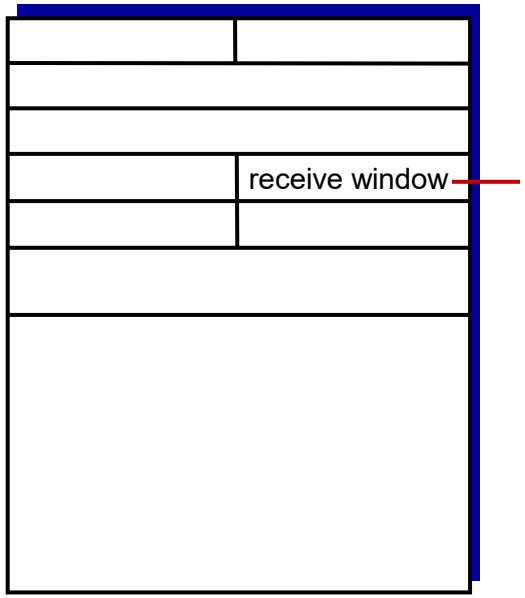
Livello di rete consegna il payload del datagramma IP al livello di trasporto, dove il codice di TCP estrae i dati dal segmento e li mette nel buffer di ricezione



pila protocollare del ricevente

TCP: controllo di flusso

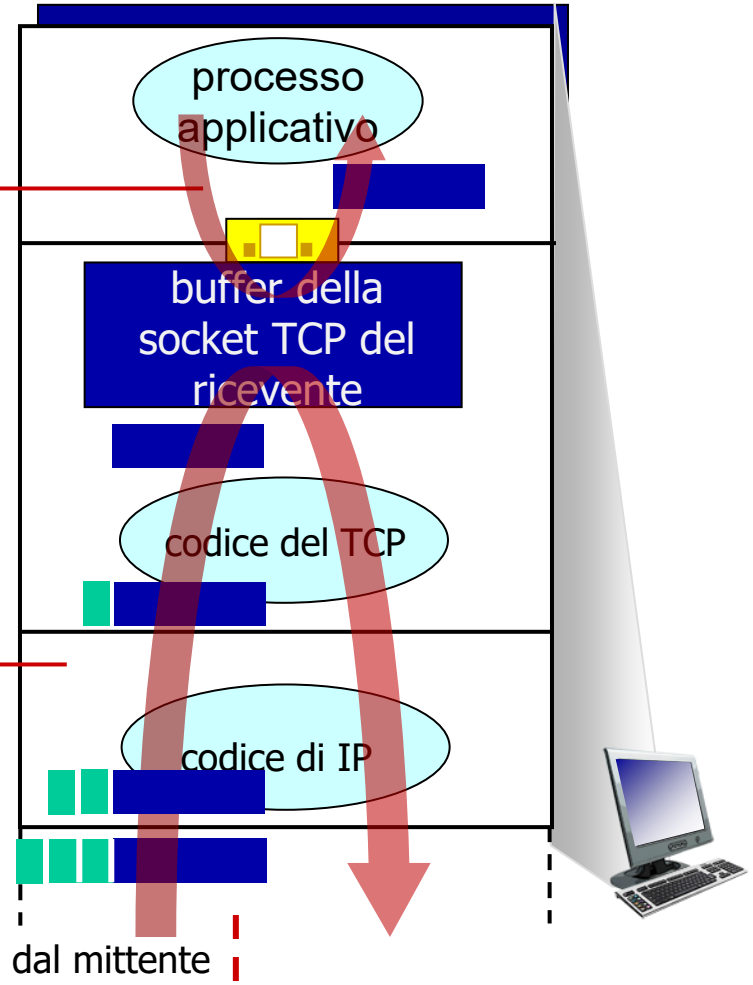
D: Cosa succede se il livello di rete fornisce i dati più velocemente di quanto il livello applicativo rimuova i dati dai buffer delle socket?



controllo di flusso: numero di byte che il ricevente è disposto ad accettare

L'applicazione rimuove i dati dai buffer della socket TCP

Livello di rete consegna il payload del datagramma IP al livello di trasporto, dove il codice di TCP estrae i dati dal segmento e li mette nel buffer di ricezione



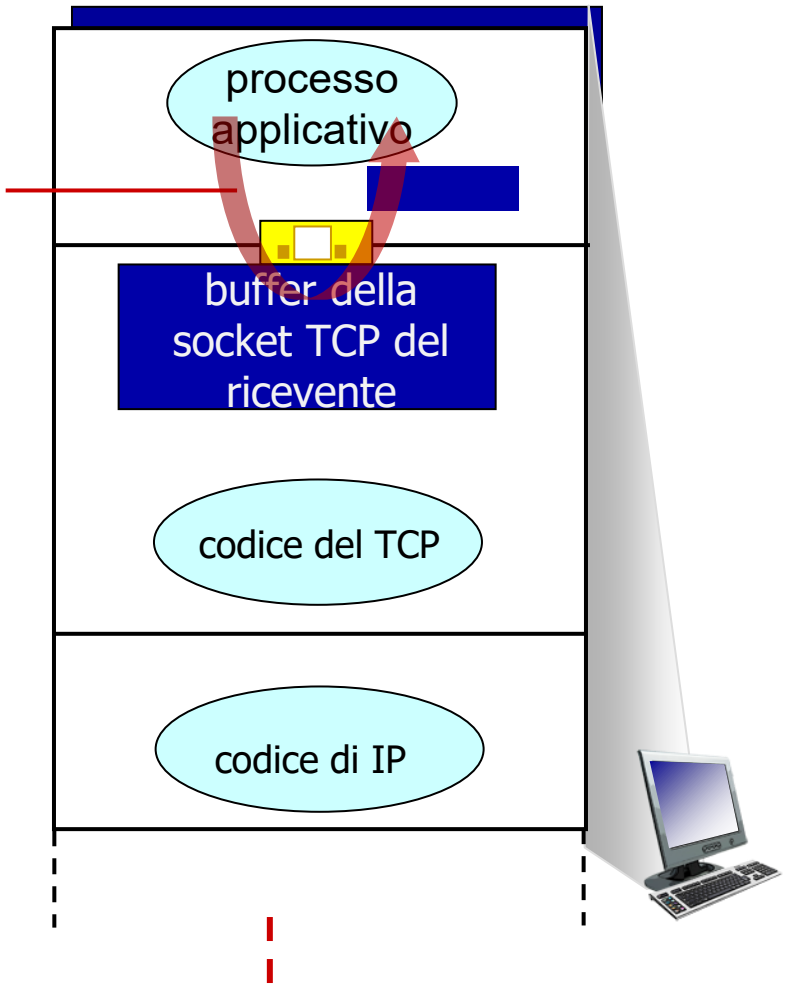
pila protocollare del ricevente

TCP: controllo di flusso

D: Cosa succede se il livello di rete fornisce i dati più velocemente di quanto il livello applicativo rimuova i dati dai buffer delle socket?

Controllo di flusso

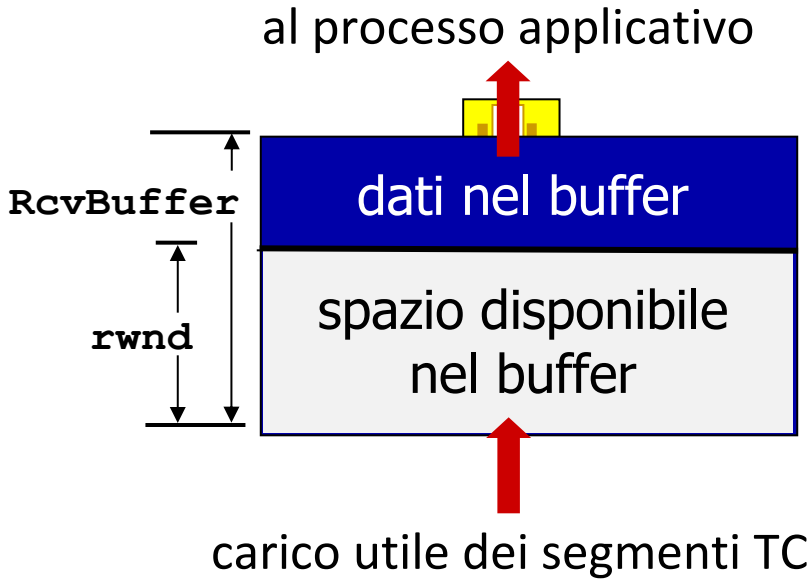
Il destinatario controlla il mittente, cosicché il mittente non ecceda il buffer del destinatario, inviando troppo e troppo velocemente



TCP: controllo di flusso

```
rwnd = RcvBuffer – (LastByteRcvd – LastByteRead)
```

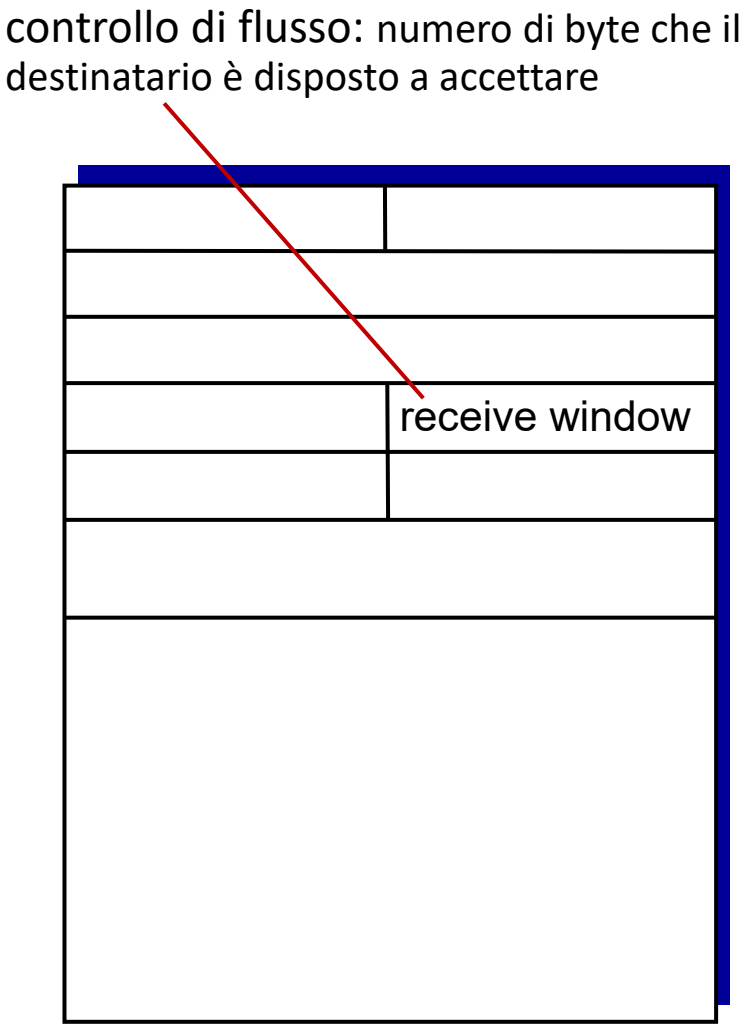
- Il mittente comunica lo spazio disponibile nel buffer nel campo **`rwnd`** (*receive window*, finestra di ricezione) nell'intestazione TCP
  - **`RcvBuffer`** dimensione impostata attraverso opzioni della socket (valore predefinito è tipicamente 4096 byte)
  - molti sistemi operativi regolano automaticamente **`RcvBuffer`**
- Il mittente limita i dati non riscontrati a **`rwnd`**
  - garantisce che il buffer di ricezione non vada in overflow



buffering dal lato del ricevente TCP

TCP: controllo di flusso

- Il mittente comunica lo spazio disponibile nel buffer nel campo **`rwnd`** (*receive window*, finestra di ricezione) nell'intestazione TCP
  - **`RcvBuffer`** dimensione impostata attraverso opzioni della socket (valore predefinito è tipicamente 4096 byte)
  - molti sistemi operativi regolano automaticamente **`RcvBuffer`**
- Il mittente limita i dati non riscontrati a **`rwnd`**
  - garantisce che il buffer di ricezione non vada in overflow

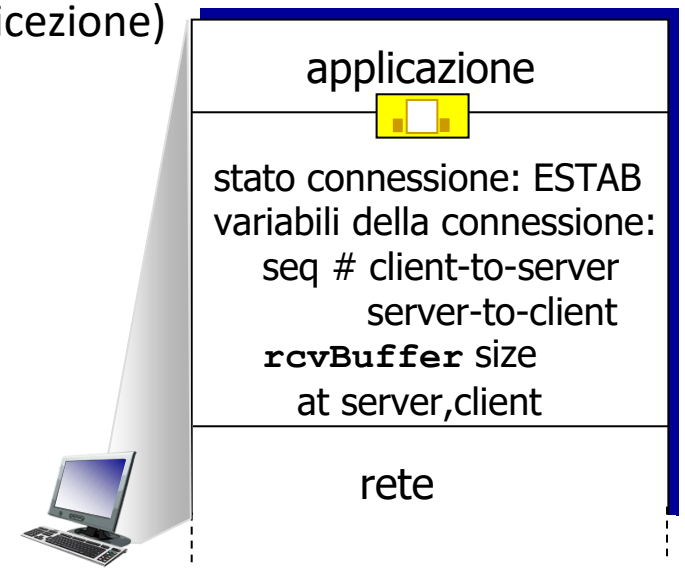


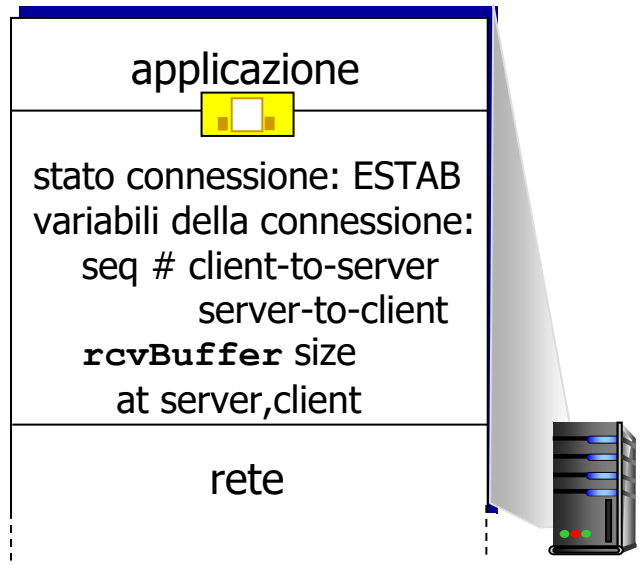
Formato del segmento TCP

Gestione della connessione TCP

Prima di scambiare i dati, il mittente e il destinatario si "stringono la mano" (“handshake”):

- accettano di stabilire una connessione (ognuno sa che l'altro è disposto a stabilire una connessione)
- concordare i parametri di connessione (ad esempio, i numeri di sequenza iniziali e la dimensione del buffer di ricezione)



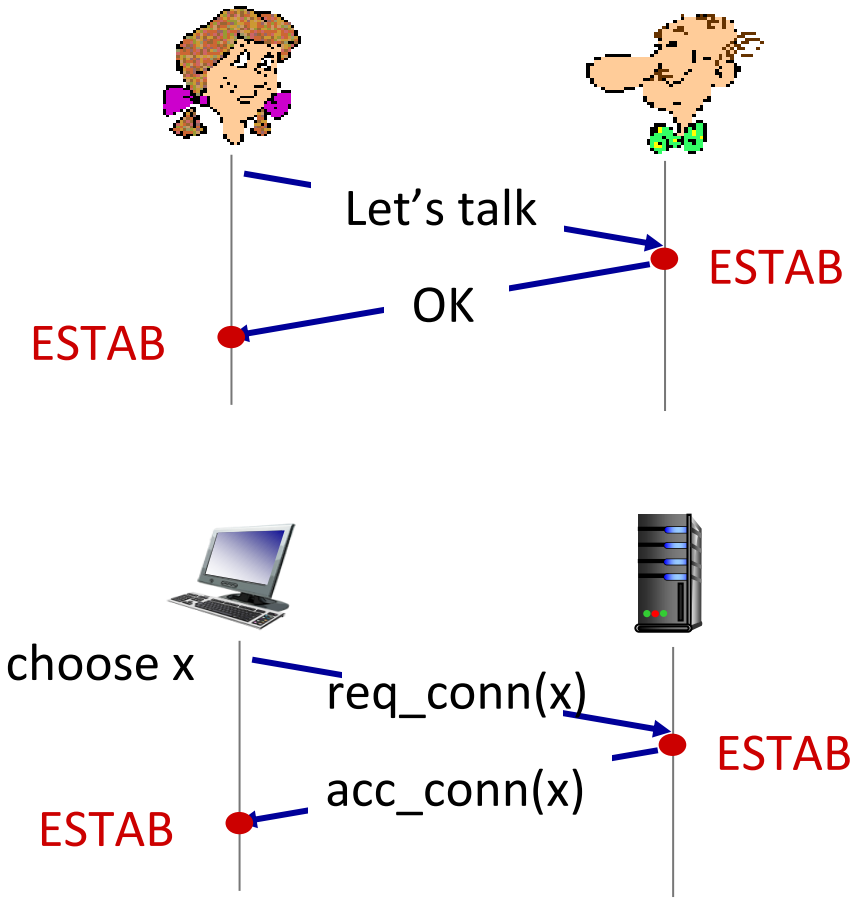



```
Socket clientSocket =
  newSocket("hostname","port number");
```

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

Accettare di stabilire una connessione

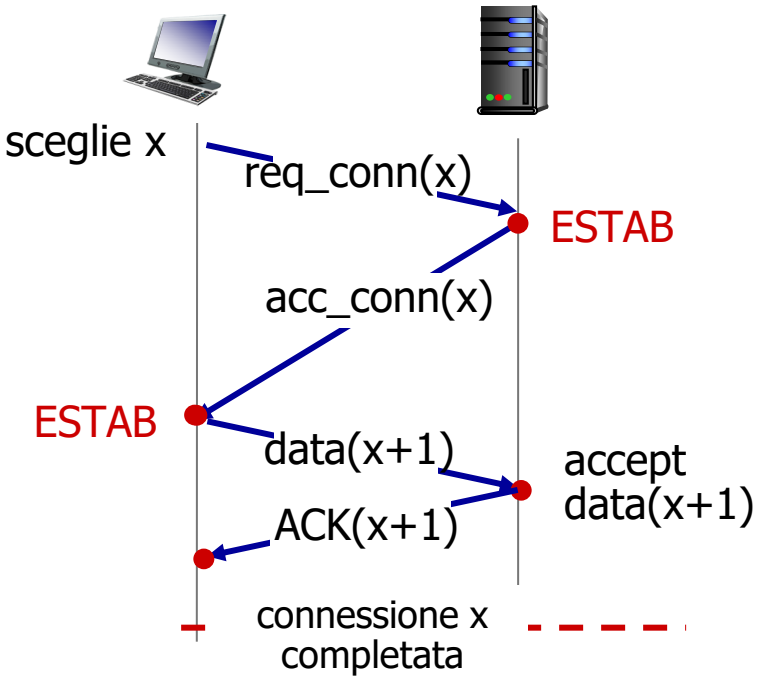
handshake a 2 vie:



D: l'handshake a 2 vie funzionerebbe sempre?

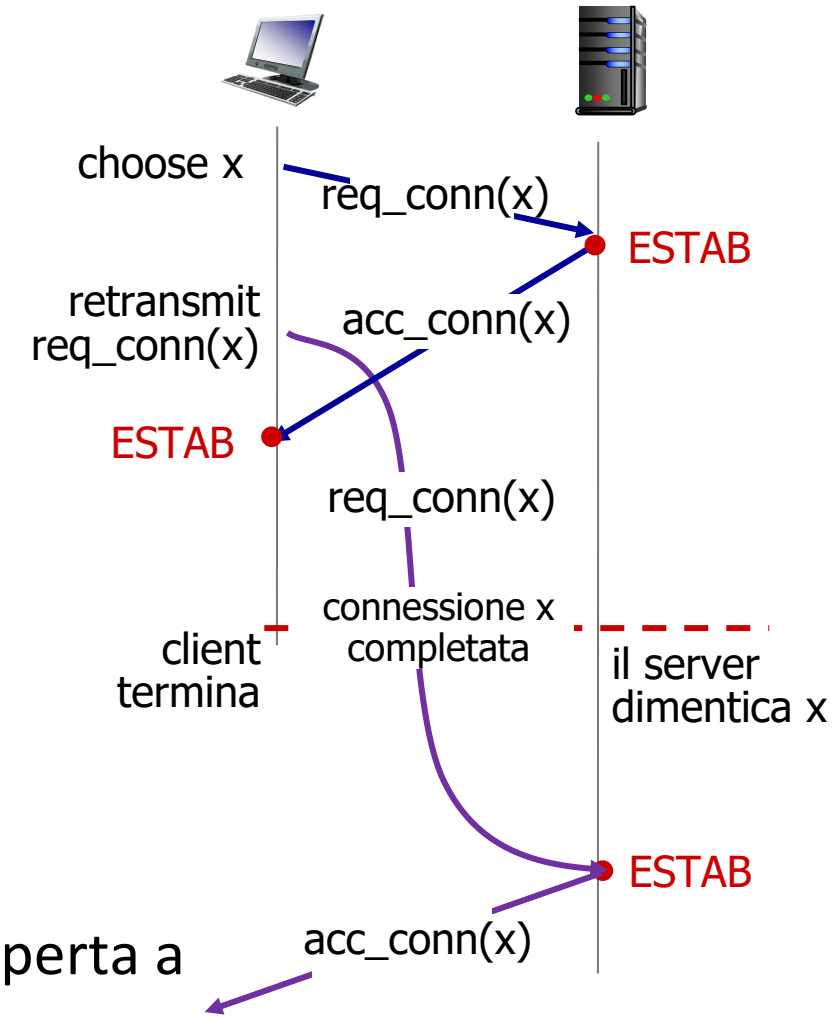
- ritardi variabili
- messaggi ritrasmessi (ad esempio, req_conn(x)) a causa della perdita di messaggi
- riordino dei messaggi
- impossibilità di "vedere" l'altro lato

Scenari per l'handshake a 2 vie



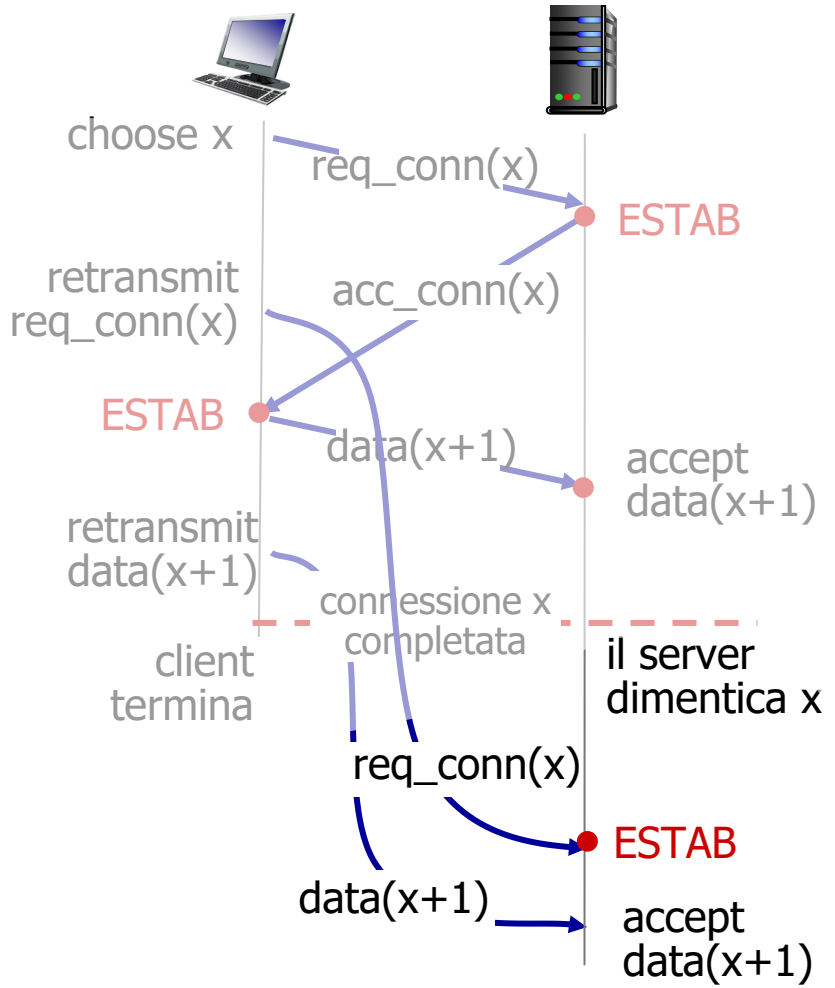
Nessun problema!

Scenari per l'handshake a 2 vie



Problema: connessione aperta a metà! (nessun client)

Scenari per l'handshake a 2 vie



Problema: dati duplicati accettati!

TCP 3-way handshake

Stato del Client

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

CLOSED

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

sceglie num. seq. iniziale x, invia segmento TCP SYN

SYNSENT

SYNbit=1, Seq=x

sceglie num. seq. iniziale y, invia segmento TCP SYN ACK, riscontrando SYN

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

ricevuto SYNACK(x) indica che il server è vivo; invia ACK per il SYNACK; questo segmento può contenere dati da client a serve

ESTAB

ACKbit=1, ACKnum=y+1

ricevuto ACK(y) indica che il client è vivo

Stato del Server

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

LISTEN

SYN RCVD

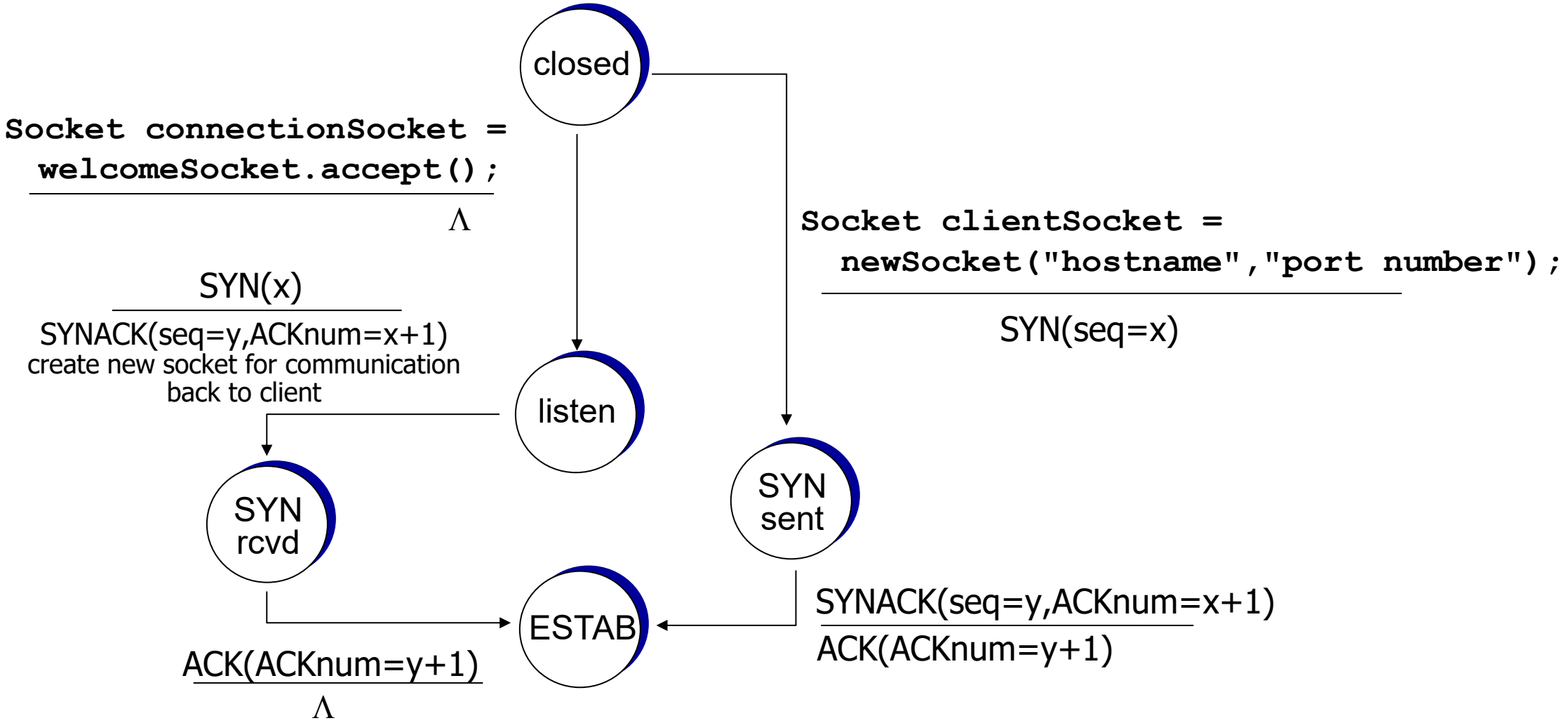
ESTAB

Entrambe le parti inizializzano il *sequence number* a caso, per evitare interferenze tra incarnazioni precedenti della stessa connessione

TCP 3-way handshake

- se un host riceve una richiesta di connessione per una porta su cui nessun socket è in ascolto, l'host invia al mittente un segmento con bit RST uguale a 1
- la ricezione di un segmento TCP RST informa un potenziale aggressore che la porta incriminata non è usata da alcun processo e che il segmento destinato a quella porta non è stato bloccato da un firewall lungo il percorso

TCP 3-way handshake: Macchina a Stati Finiti



Attacco SYN FLOOD

- L'aggressore invia un segmento TCP SYN ad un server con un indirizzo IP fasullo
- Il server alloca e inizializza le variabili e i buffer della connessione e risponde inviando un segmento TCP SYNACK alla porta e all'indirizzo IP (fasullo) di origine e transita nella stato SYN_RCVD
- La rete tenta di consegnare il segmento TCP SYNACK all'indirizzo IP fasullo, possibilmente raggiungendo un altro host che non c'entra nulla e che non risponderà
- Il server, nello strato SYN_RCVD, non ricevendo un ACK, eventualmente (dopo un minuto o più) rilascerà tutte le risorse associate a quella connessione mezza aperta
- Nel frattempo, però, l'aggressore è riuscito a consumare delle risorse del server. Inviando numerosi segmenti SYN ad un server obiettivo, un aggressore può montare un attacco DoS (spesso sottoforma di attacco DDoS)

Attacco SYN FLOOD: contromisure "SYN cookie" (semplificato)

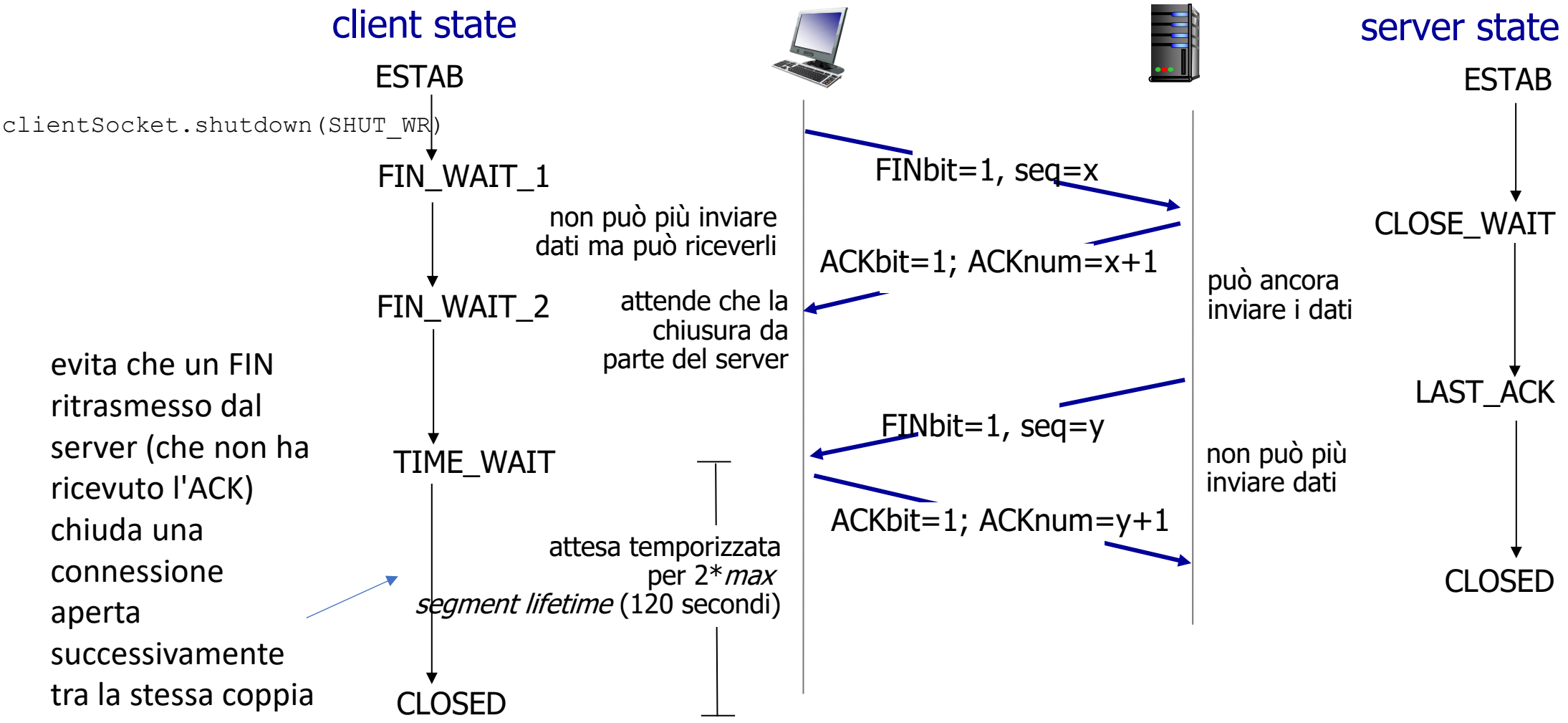
- Alla ricezione del segmento SYN iniziale, il server:
  - Calcola l'*hash* degli indirizzi IP e numeri di porta di origine e di destinazione e di una chiave segreta, producendo un *cookie*
    $cookie = hash(IP\ sorgente, IP\ destinazione, porta\ sorgente, porta\ destinazione, chiave\ segreta)$
  - usa il cookie con numero di sequenza iniziale da inserire dentro al segmento SYNACK
  - non alloca alcuna risorsa né memorizza il cookie
- Se il segmento SYN ricevuto in precedenza era legittimo, il client alla ricezione del segmento SYNACK risponde con un segmento ACK, che usa come numero di Acknowledgment *cookie + 1*
- Alla ricezione di un segmento ACK, il server può capire che è legato al SYN precedente perché gli basta calcolare Acknowledgment – 1 per ottenere il cookie. Quindi, riesegue il calcolo del cookie per vedere se ottiene lo stesso valore. Si noti che l'inclusione di una chiave segreta impedisce a un aggressore di creare un cookie valido.
- Per evitare attacchi replay, la funzione l'input hash dovrebbe includere una marca temporale, oppure la chiave segreta deve cambiare periodicamente.

Un protocollo umano di handshake a 3 vie

Chiusura di una connessione TCP

- client e server chiudono ciascuno il proprio lato della connessione
  - inviano il segmento TCP con il bit FIN = 1
- rispondere al FIN ricevuto con un ACK
  - alla ricezione del FIN, l'ACK può essere combinato con il proprio FIN
- è possibile gestire scambi FIN simultanei

Chiusura di una connessione TCP



evita che un FIN ritrasmesso dal server (che non ha ricevuto l'ACK) chiuda una connessione aperta successivamente tra la stessa coppia di porte