Livello di Trasporto

parte 2

Sommario

:

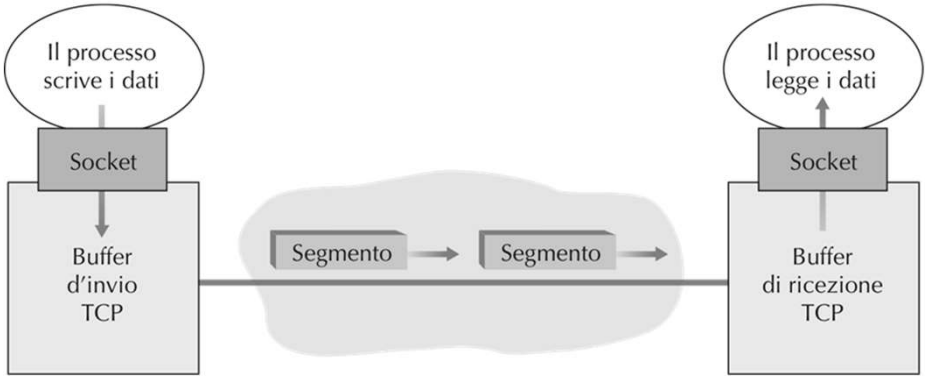
- I servizi del livello di transporto
- Multiplexing and demultiplexing
- Protocollo di trasporto connectionless: UDP
- Principi per rendere affidabile il trasferimento dati
- **Protocollo di trasporto connection-oriented: TCP**
  - Struttura dei segmenti
  - Trasferimento dati affidabile
  - Controllo di flusso
  - Gestione della connessione
- La congestione e la sua gestione
- Gestione della congestione in TCP

Protocollo TCP

- Comunicazione point-to-point:
  - Un mittente, un destinatario
- Trasferimento affidabile stream-oriented:
  - Non c'è un concetto di messaggio ma si parla stream di byte
- Modalità full duplex data:
  - Una connessione TCP offre un servizio full-duplex
- Connection-oriented:
  - prima di effettuare lo scambio dei dati, i processi devono effettuare l'handshake, ossia devono inviarsi reciprocamente alcuni segmenti preliminari per stabilire i parametri del successivo trasferimento dati handshaking iniziale (scambio di messaggi di controllo) inizia il mittente
- Controllo di flusso:
  - Il mittente non sovraccaricherà di pacchetti il destinatario

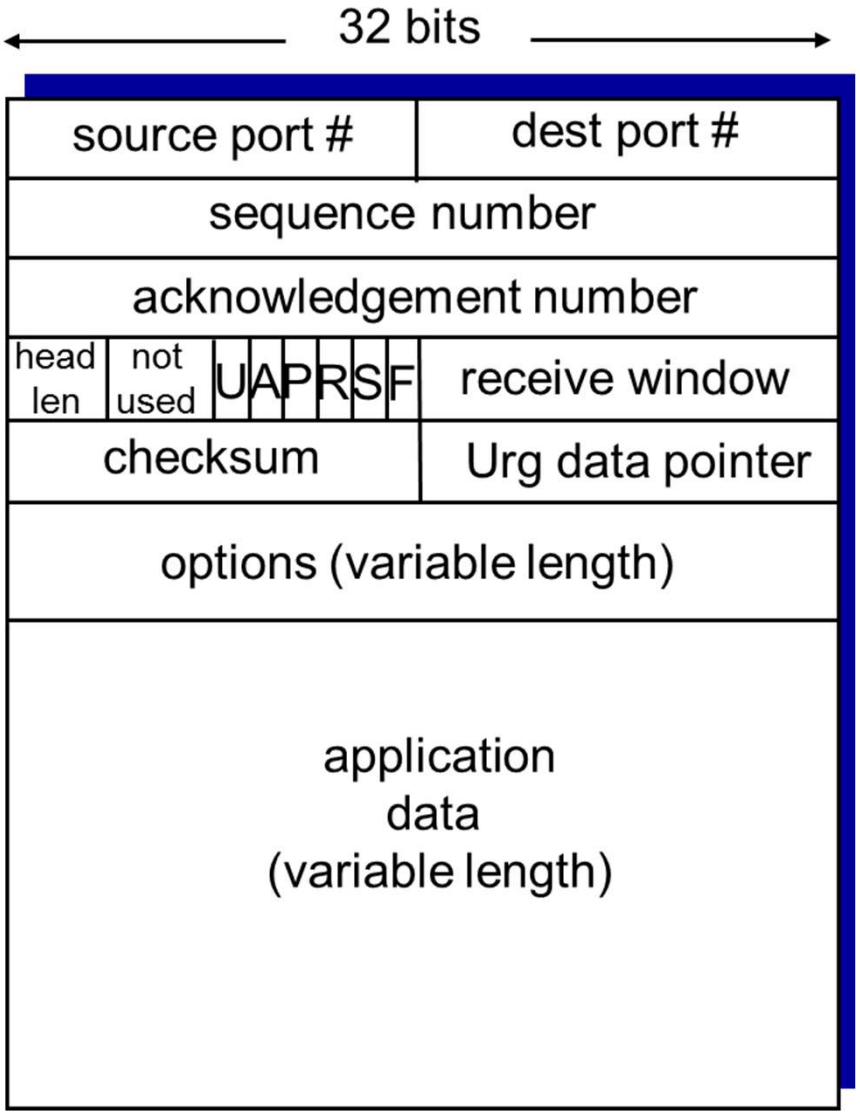
Protocollo TCP (II)

- Buffer di invio:
  - Un mittente, un destinatario
- Dimensione massima del segmento (MSS):
- Segmenti TCP



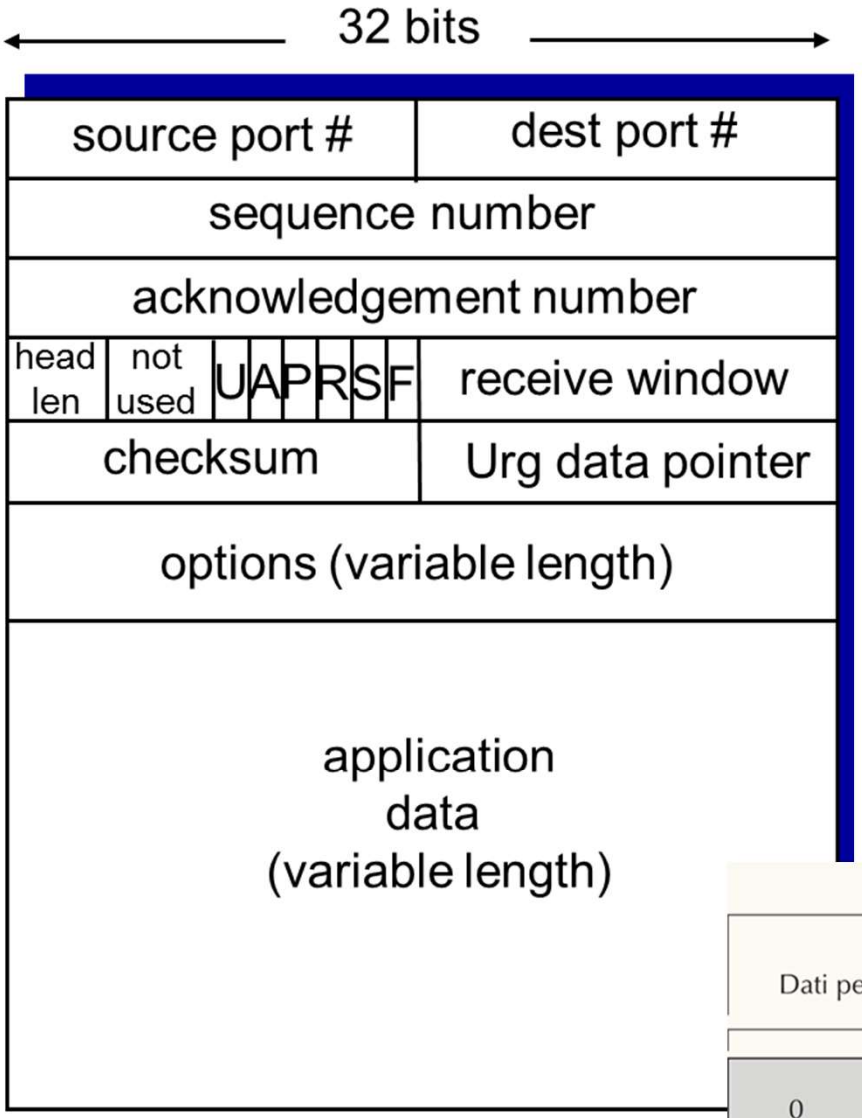
http://media.pearsoncmg.com/aw/aw_kurose_network_4/applets/flow/FlowControl.htm

TCP: struttura dei segmenti



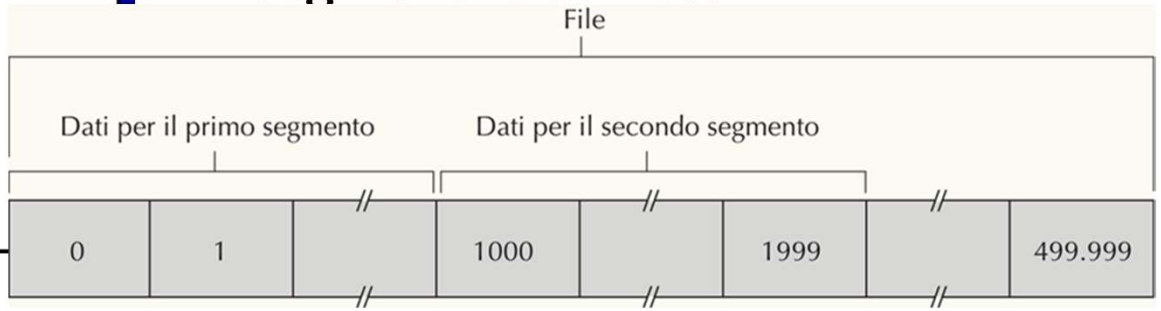
- Il segmento TCP consiste di campi intestazione e di un campo contenente un blocco di dati proveniente dall'applicazione
- Ogni segmento inizia con un' intestazione di 20 byte (5 parole di 32 bit) seguita da alcune opzioni
- Il formato è progettato per portare sia dati di utente che informazioni di protocollo (***piggybacking***), quali i riscontri e le *window advertisement*

TCP: struttura dei segmenti (II)

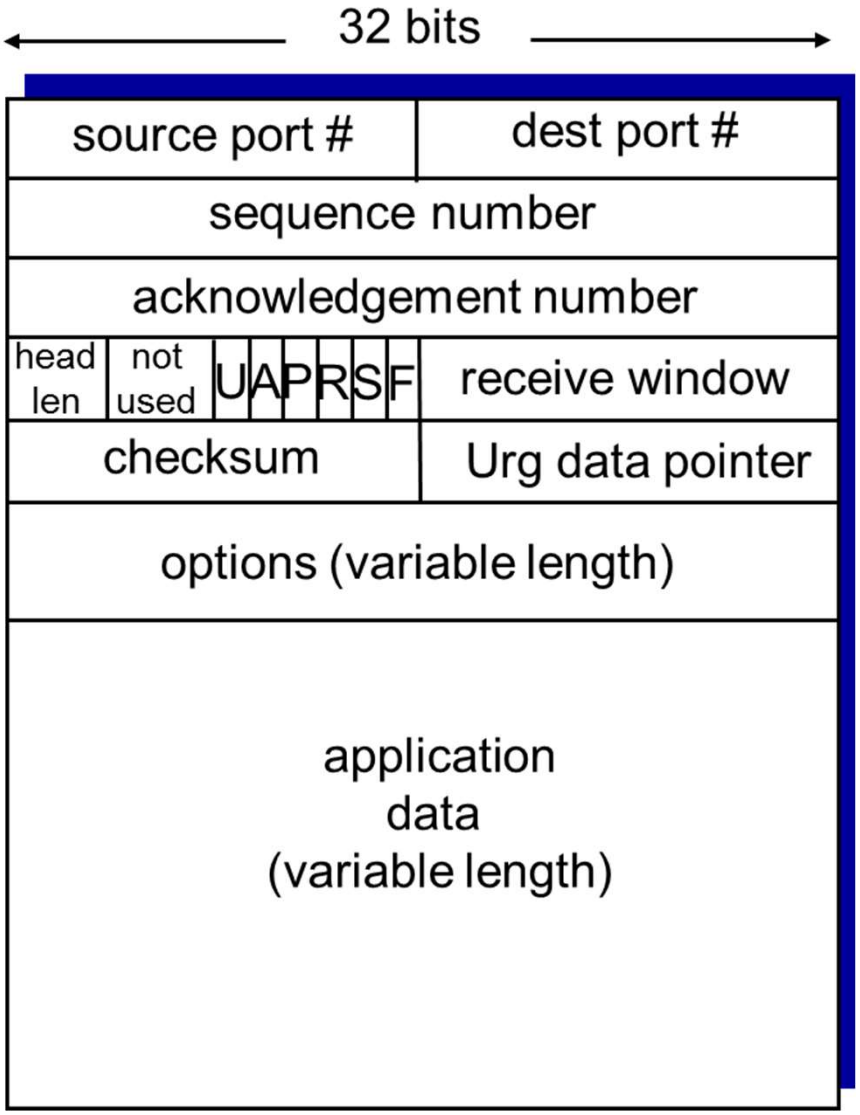


Numeri di sequenza e numeri di acknowledgement: rappresentano una parte critica del servizio di trasferimento dati affidabile proprio di TCP

- TCP vede i dati come un flusso di byte non strutturati, ma ordinati
- L'uso dei numeri di sequenza in TCP riflette questa visione, dato che i numeri di sequenza si riferiscono al flusso di byte trasmessi e non alla serie di segmenti trasmessi



TCP: struttura dei segmenti (III)



Numero di sequenza per un segmento = numero nel flusso di byte del primo byte del segmento

Numeri di acknowledgement = il numero di acknowledgement che l'Host A scrive nei propri segmenti è il numero di sequenza del byte successivo che l'Host A attende dall'Host B

Notare che notare che si riferiscono a due stream diversi!

Si tratta di un riscontro **cumulativo**

TCP: struttura dei segmenti (IV)
numeri di sequenza, ACK

Numeri di sequenza:

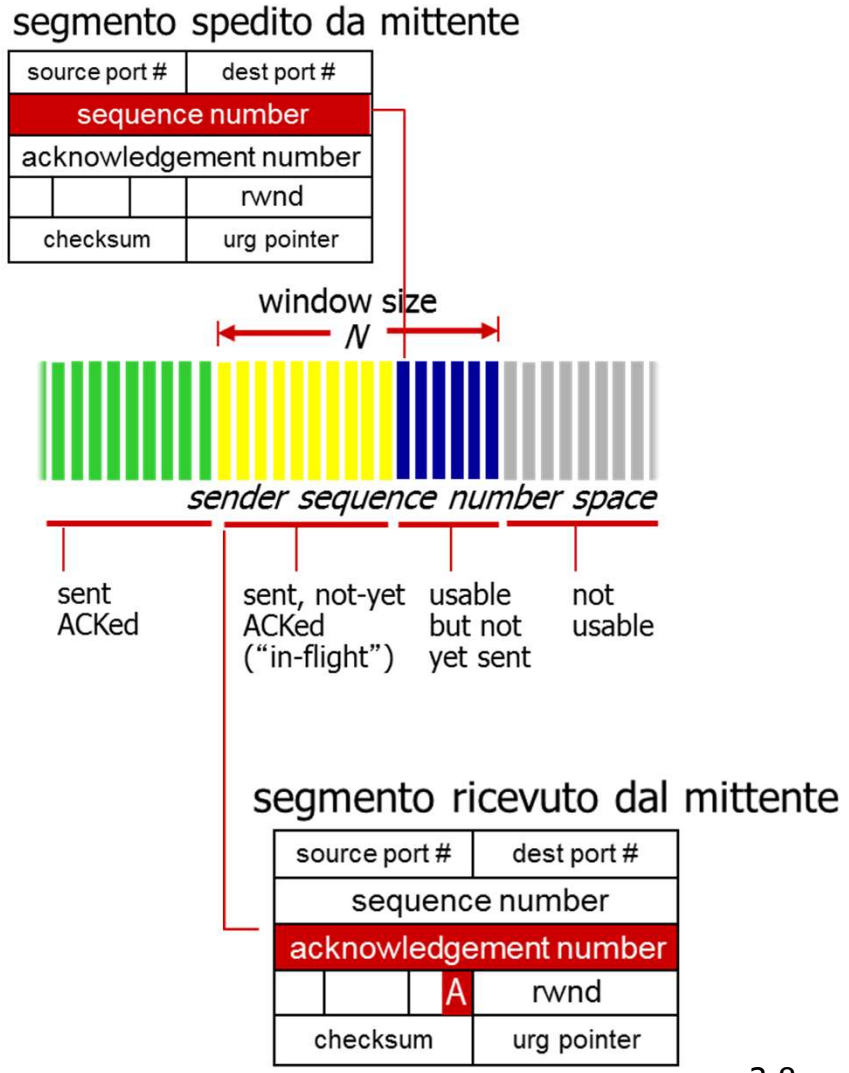
- Numero del primo byte nello stream di dati

Acknowledgements:

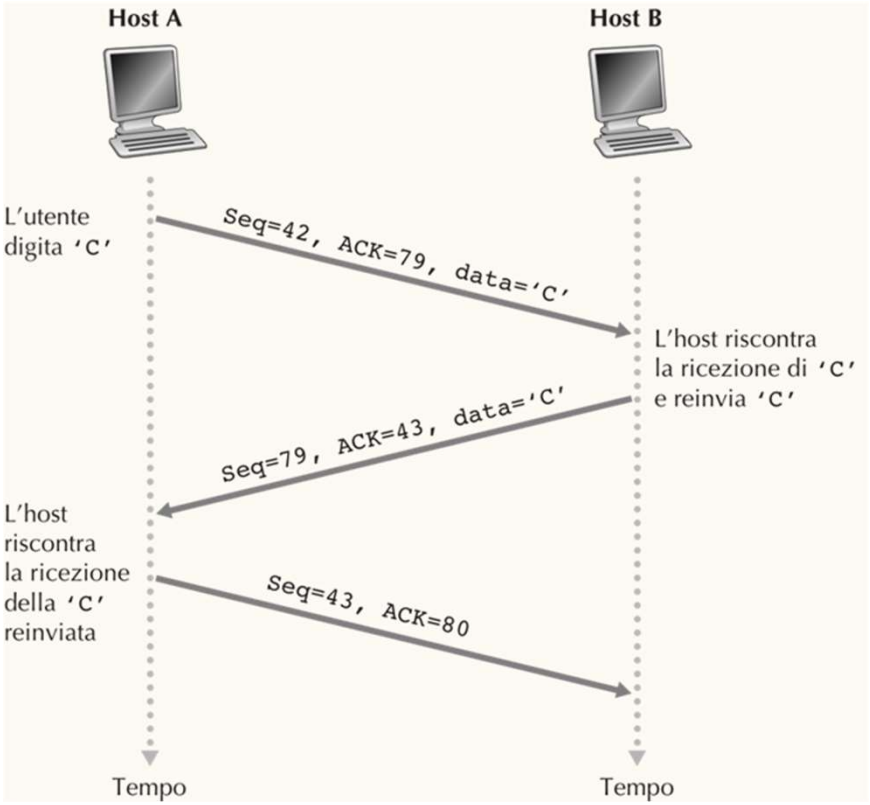
- Numero di sequenza del prossimo byte che il destinatario si aspetta di ricevere
- ACK cumulativi

D: come vengono gestiti I segmenti fuori ordine ?

R: nel protocollo questo aspetto non viene specificato ma la scelta viene demandata all'implementazione



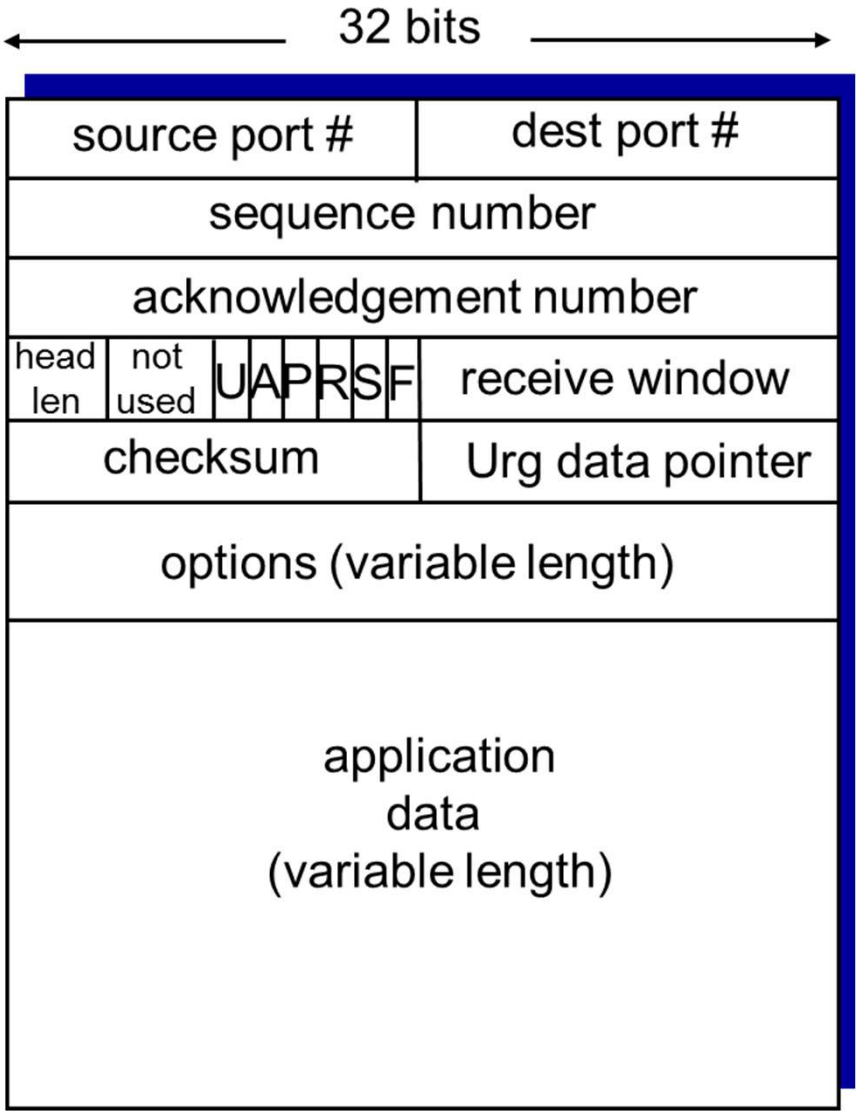
TCP: struttura dei segmenti (V)
numeri di sequenza, ACK



Telnet: protocollo a livello di applicazione impiegato per il login remoto che utilizza TCP.
Telnet: applicazione interattiva

- Vengono spediti tre segmenti: - Il primo è inviato dal client al server e ccontiene nel campo dati il byte ASCII per la lettera 'C'
- Il secondo segmento spedito dal server al client ricopre un duplice scopo (fa un acknowledgement dei dati ricevuti dal server; manda indietro un "eco" della lettera 'C' → acknowledgement piggybacked)
- Il terzo segmento viene inviato dal client al server e ha come unico scopo dare un acknowledgement ai dati inviati dal server

TCP: struttura dei segmenti (VI)



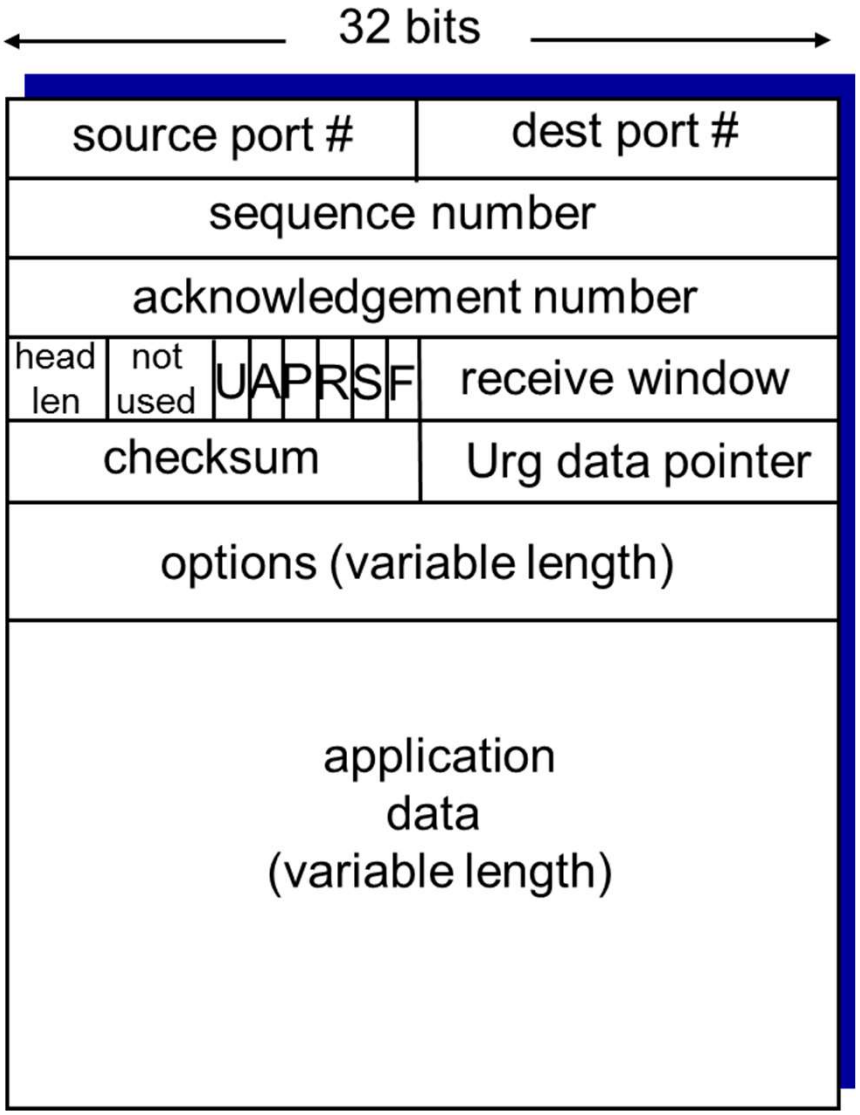
I campi *Source Port* e *Destination Port* identificano gli estremi locali della connessione

Una porta + indirizzo IP formano l'endpoint

Il campo *TCP header length* indica la lunghezza dell'header in parole di 32 bit. Informazione necessaria perché il campo *Option* ha lunghezza variabile

Segue un campo di 4 bit non utilizzati

TCP: struttura dei segmenti (VII)



Seguono 8 flag (nella figura vengono riportati solamente 6)

CWR e *ECE* (non riportati) sono usati per segnalare la congestione (ECN, *Explicit Congestion Notification*).

Il bit *PSH* segnala la presenza di dati *PUSH*

Il bit *URG* segnala la presenza di dati *URGENT*

TCP: struttura dei segmenti (VIII)

Il bit PSH

- Il bit *PSH* segnala la presenza di dati *PUSH*
- TCP è libero di ritardare (entro certi limiti) la trasmissione dei dati dell'utente mittente nei segmenti per **ottimizzare il trasferimento dei dati;** quindi tende a formare segmenti lunghi ("giusti") per diminuire l'overhead dell'header di TCP e IP. Anche il TCP ricevente può ritardare la consegna ddi dati al destinatario che li chiede, per altre ragioni di ottimizzazione (riduzione delle system call, confini di pagina, confini di buffer,...)
- Questa tecnica produce dei ritardi nella consegna dei byte che per certe applicazioni possono essere molto dannosi (ad es. per gli editor a comandi di carattere): l'utente umano si aspetta una risposta istantanea e TCP introduce un ritardo inaspettato e incomprensibile
- TCP offre una primitiva di servizio ***push*** che l'applicazione mittente può invocare per forzare l'invio dei buffer raccolti dal mittente.
- Il bit PSH dell'header TCP informa il TCP ricevente che il segmento è stato forzato, in modo che il TCP ricevente possa **fornirlo subito all'applicazione ricevente appena questa lo richieda**
- Il TCP ricevente è libero in assenza di push di trattenere dei dati ricevuti, e non consegnarli subito all'applicazione che glieli chiede
- Il bit PSH impedisce al TCP ricevente di trattenere dei dati ricevuti e non consegnarli all'applicazione che li chiede
- Il PUSH non tenta di "forzare" l'applicazione destinataria a consumare i dati

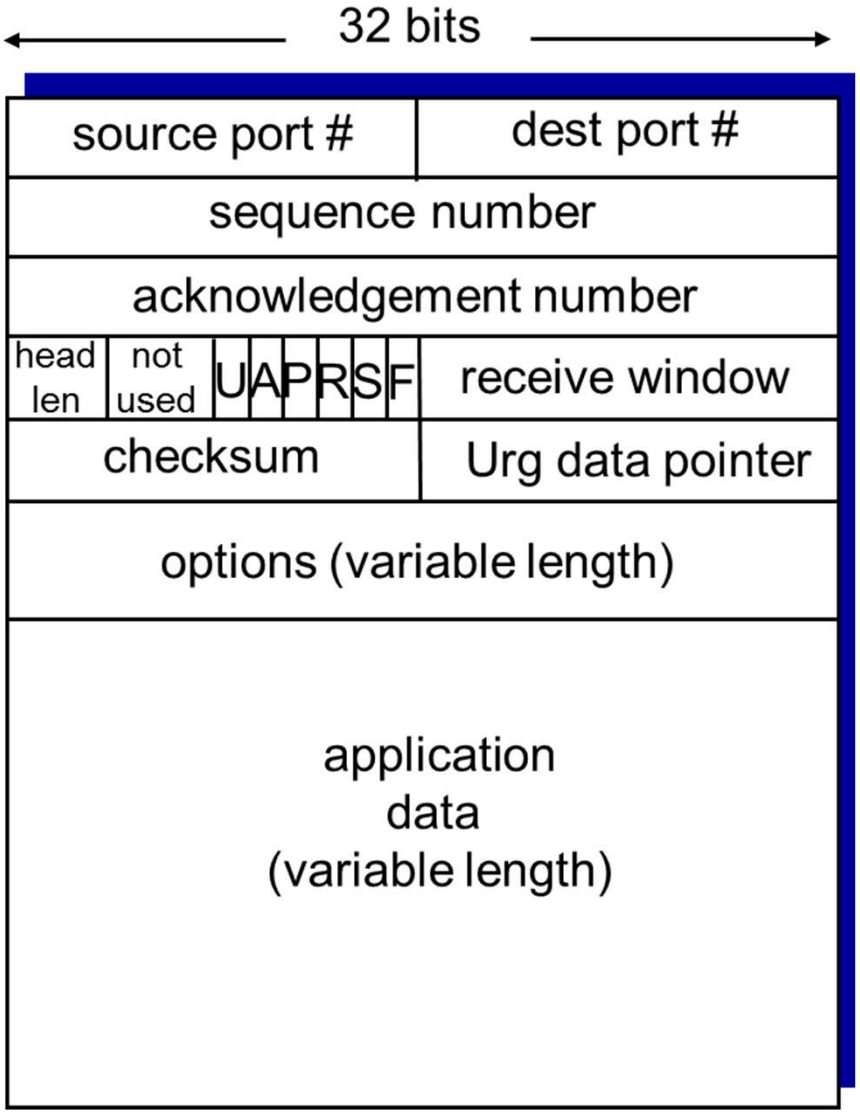
TCP: struttura dei segmenti (IX)

Il bit URG

- Nelle applicazioni distribuite è sempre necessario (prima o poi) poter inviare dei dati "urgenti" che almeno funzionalmente si comportino come se viaggiassero su un canale diverso da quello su cui scorrono i dati normali: viaggino ***out of band***.
- L'espressione è dovuta a quei protocolli in cui una diversa banda è davvero disponibile per i dati urgenti
- Quando non si ha una banda di segnalazione fisica (come in TCP/IP) è comunque necessario poter
  - avvertire il ricevente che ci sono dati urgenti da leggere, in modo che
  - smetta di fare quello che sta facendo, e
  - possa leggere da TCP (eventualmente senza elaborare) i dati normali, e
  - arrivare a quelli urgenti da elaborare subito

TCP: struttura dei segmenti (X)

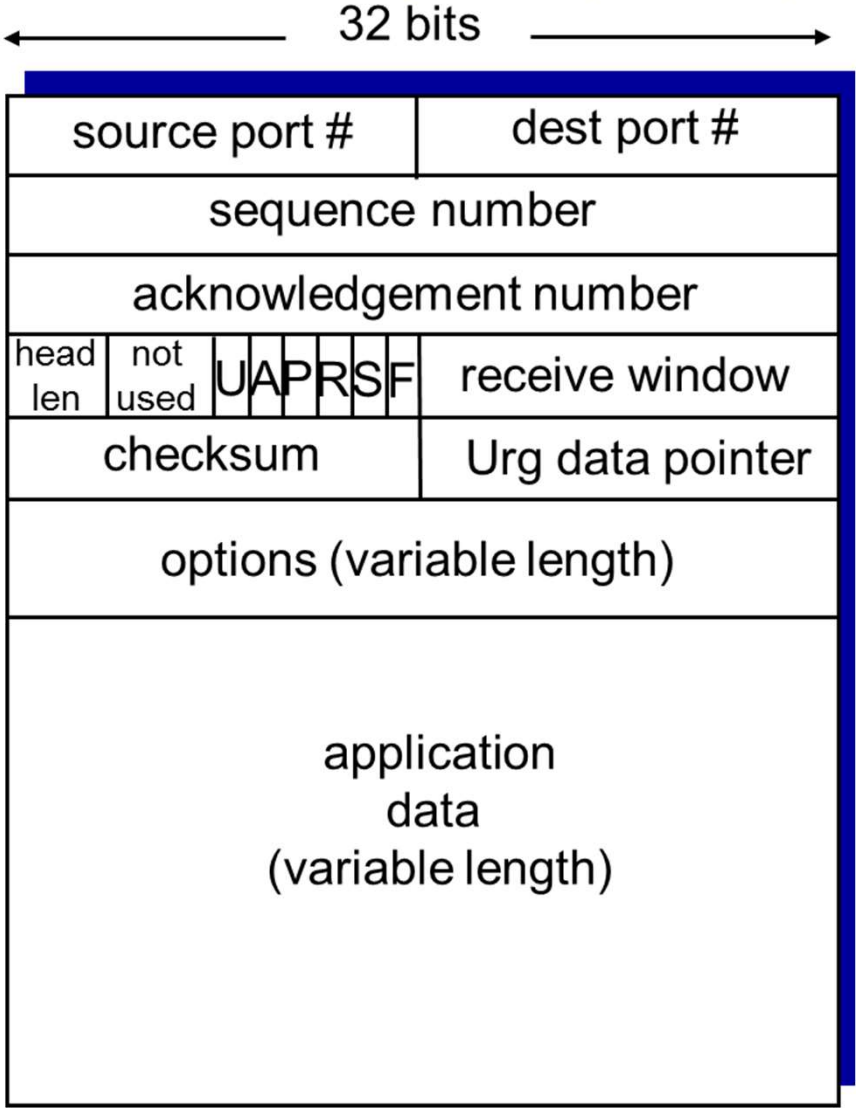
Il bit URG



- TCP usa questo secondo modello (perché IP non ha una funzionalità di dati out of band lui stesso)
- URGENT POINTER, se significativo, indica (come offset rispetto al SEQUENCE NUMBER corrente, infatti è solo 16bit) quale è l'ultimo byte degli urgenti: il ricevente deve leggere fino a quel byte per essere sicuro di avere consumato tutti i dati urgenti
- Notare che occorre aiuto dal sistema operativo per "avvertire" il processo ricevente di andare a leggere i dati. Ad es. in telnet voglio passare il ^C come urgente, in modo da interrompere un programma in ciclo (segnali di Unix!)
- **Molto diverso da PUSH!**

TCP: struttura dei segmenti (XI)

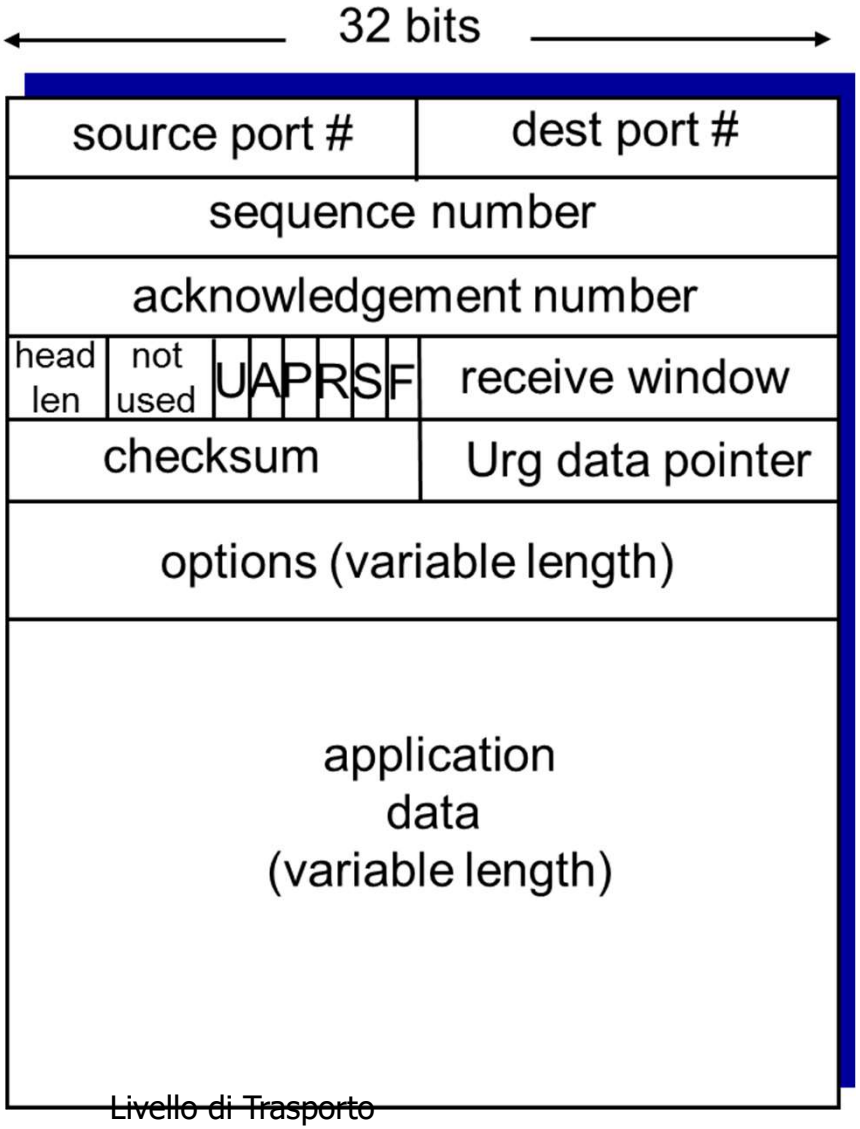
I bit ACK, RST, SYN, FIN



- Il bit *ACK* viene impostato a 1 per indicare che l'*Acknowledgement number* è valido (0 altrimenti)
- Il bit *RST* viene utilizzato per re-inizializzare una connessione (in caso di problemi)
- Il bit *SYN* viene usato per stabilire le connessione (SYN=1 e ACK=0 per una CONNECTION REQUEST, mentre SYN=1, ACK=1 per una CONNECTION ACCEPTED)
- Il bit *FIN* viene usato per rilasciare la connessione. Specifica che il mittente non ha altri dati da trasmettere

TCP: struttura dei segmenti (XII)

Finestra di ricezione



- I campo *Window size (finestra di ricezione)* indica quanti byte possono essere inviati a partire da quello che ha ricevuto in ackowledgement
- *Window size = 0* indica che i byte fino a ***Acknowledgement number – 1*** compreso sono stati ricevuti ma il ricevente non ha avuto modo di consumarli e non ha ulteriore spazio nel buffer (0 byte). Il ricevente può autorizzare l'invio di altro dati trasmettendo un segmento con lo stesso *Acknowledgement number* ed un campo *Window size > 0*

TCP: timeout e stima del RTT

Il **Round Trip Time** (RTT) sperimentato da un segmento l'intervallo di tempo tra l'istante t_0 in cui il segmento inizia ad essere trasmesso dal mittente e l'istante t_1 in cui lo stesso mittente riceve il corrispondente messaggio ACK inviatogli dal destinatario

$$RTT = t_1 - t_0$$

Ciascun segmento trasmesso sperimenterà un suo proprio valore di RTT (variabile da segmento a segmento)

RTTsample = valore del RTT del segmento corrente trasmesso (per la prima volta)

Ignora le ritrasmissioni (segmenti ritrasmessi)

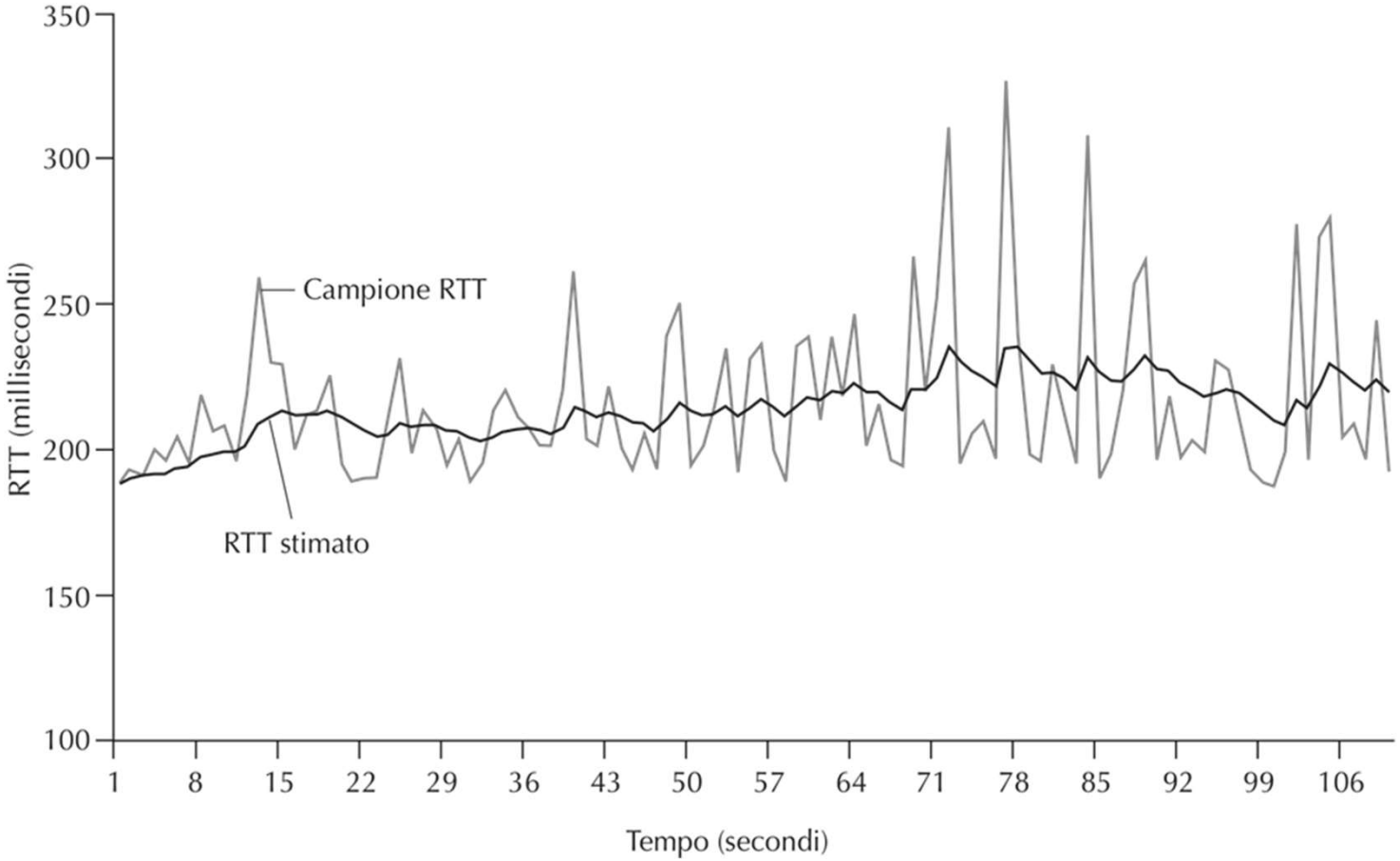
Indichiamo con **`EstimatedRTT`** (sec) il valore medio degli RTT dei segmenti trasmessi attraverso la connessione

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

- exponential weighted moving average (EWMA)
- influenza dei campioni passati decresce esponenzialmente
- valore tipico per α = 0.125

Ogni volta che un nuovo **`SampleRTT`** viene misurato, il TCP al lato sorgente aggiorna il valore **`EstimatedRTT`** del RTT medio in accordo alla seguente formula:

TCP: timeout e stima del RTT (II)



TCP: timeout e stima del RTT (III)

- Il Re-transmission Time Out (RTO) è l'intervallo di tempo massimo che può trascorrere tra l'istante di trasmissione di un segmento e l'istante di ricezione del corrispondete ACK prima che il TCP al lato sorgente ri-trasmetta di nuovo il segmento
  - valori troppo "piccoli" di RTO possono provocare ri-trasmissioni non necessarie di uno o più segmenti
  - valori troppo "grandi" di RTO possono introdurre ritardi troppo elevati nel trasferimento di segmenti che sono stati soggetti a perdita

TCP: timeout e stima del RTT (IV)

- È auspicabile impostare RTO a EstimatedRTT più un certo margine che dovrebbe essere grande quando c'è molta fluttuazione nei valori di SampleRTT e piccolo in caso contrario

RTO= EstimatedRTT + 4*DevRTT

RTT stimato "safety margin"

- **DevRTT** è una stima della varianza del RTT

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

(typically, β = 0.25)

Sommario

- I servizi del livello di transporto
- Multiplexing and demultiplexing
- Protocollo di trasporto connectionless: UDP
- Principi per rendere affidabile il trasferimento dati
- Protocollo di trasporto connection-oriented: TCP
  - Struttura dei segmenti
  - **Trasferimento dati affidabile**
  - Controllo di flusso
  - Gestione della connessione
- La congestione e la sua gestione
- Gestione della congestione in TCP

TCP Trasferimento dati affidabile

TCP crea un servizio di trasporto dati affidabile al di sopra del servizio inaffidabile e best-effort di IP, assicurando che:

- il flusso di byte che i processi leggono dal buffer di ricezione TCP non sia alterato,
- non abbia buchi,
- non presenti duplicazioni e rispetti la sequenza originaria

Il flusso di dati in arrivo è quindi esattamente quello spedito

Meccanismi utilizzati:

- Segmenti inviati in pipeline
- ACK cumulativi acks
- Timer per la ritrasmissione

Le ristrasmissioni vengono attivate da :

- timeout
- Ricezione di ACK duplicati

TCP eventi lato mittente

Mittente semplificato:
- ignora gli ACK duplicati
- ignora controllo di flusso e controllo della congestione

dati ricevuti dall'applicazione:

- crea un segmento con **`num-seq`**
- **`num-seq`** è il numero del primo byte nello stream di dati nel segmento
- fa partire il timer se non era già attivo
  - Il timer fa rifermiento al più vecchio dei segmenti spediti che non è stato ancora riscontrato
  - l'intervallo di timout: **`TimeOutInterval`**

timeout:

- Ritrasmette il segmento che ha causato lo scatto del timeout
- Riparte il timer

ricezione ACK:

- se l'ACK si riferisce ad un segmento non ancora riscontrato
  - aggiornare i segmenti riscontrati
  - far partire di nuovo il timer se ci sono segmenti ancora da riscontrare

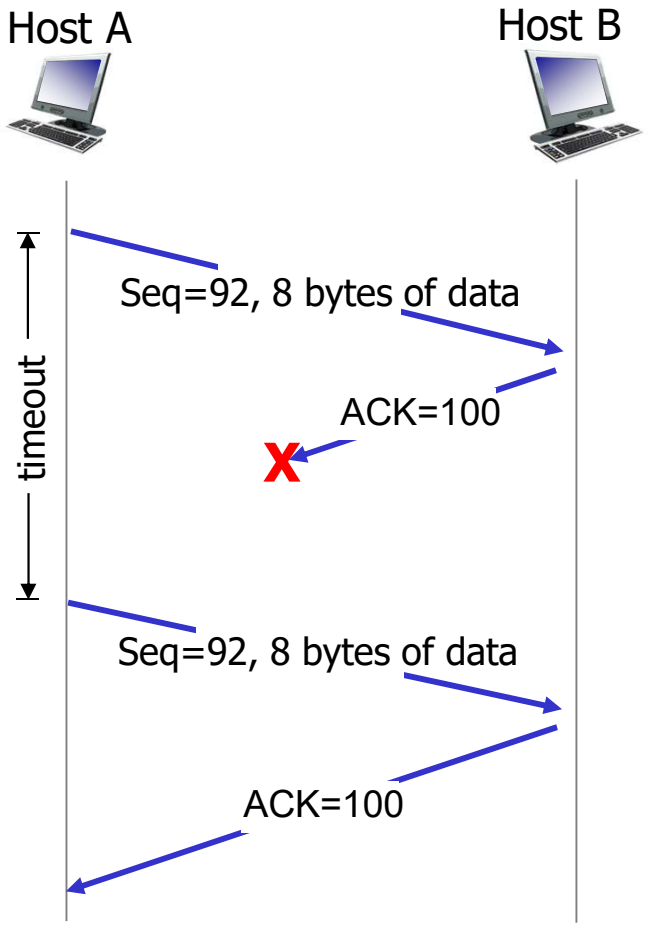
TCP eventi lato mittente

Mittente semplificato:

- ignora gli ACK duplicati
- ignora controllo di flusso e controllo della congestione

```
/* Ipotizziamo che il mittente non subisca imposizioni dal
controllo di flusso o di congestione TCP, che i dati
dall'alto abbiano dimensione inferiore a MSS e che il
trasferimento dati avvenga in un'unica direzione */

NextSeqNum = InitialSeqNumber
SendBase = InitialSeqNumber

loop (per sempre) {
  switch (evento)

    evento: dati ricevuti dall'applicazione a livello superiore
        crea il segmento TCP con numero di sequenza NextSeqNum
        if (il timer attualmente non è in funzione)
          avvia il timer
        passa il segmento a IP
        NextSeqNum = NextSeqNum + lunghezza(dati)
        break;

    evento: timeout del timer
        ritrasmetti il segmento che non ha ricevuto ACK con
           il più piccolo numero di sequenza
        avvia il timer
        break;

    evento: ACK ricevuto, con valore del campo ACK pari a y
        if (y > SendBase) {
          SendBase = y
          if (esistono attualmente segmenti senza ACK)
            avvia il timer
        }
        break;

} /* fine del loop */
```

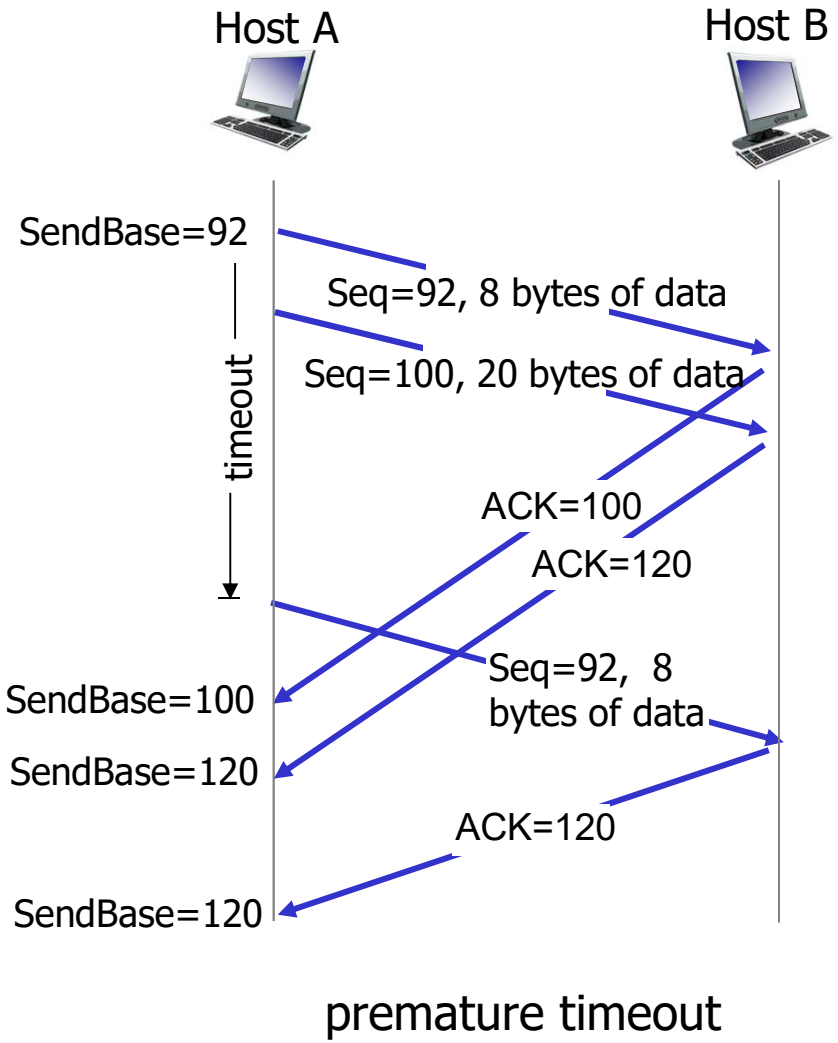
TCP e ritrasmissione: scenari



lost ACK scenario

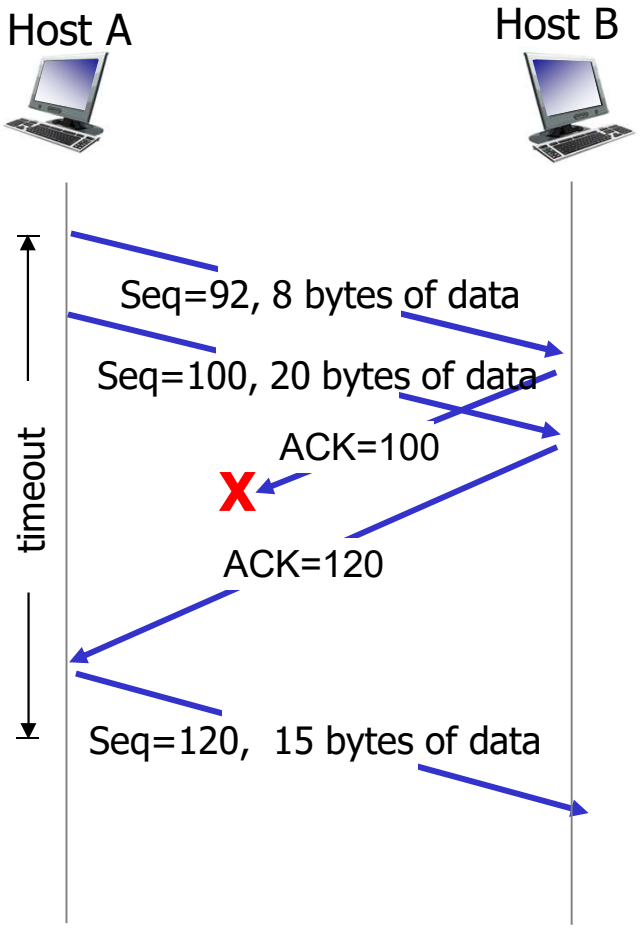
L'Host A spedisce un segmento all'Host B. L'acknowledgement sul percorso inverso viene smarrito

TCP e ritrasmissione: scenari (II)



premature timeout

TCP e ritrasmissione: scenari (III)



cumulative ACK

TCP raddoppio del RTO

Raddoppio dell'intervallo di Timeout

Ogni volta che si verifica un timeout TCP ritrasmette il segmento con numero di sequenza più piccolo non ancora riscontrato

Imposta il successivo timeout al doppio del valore precedente (anziché derivarlo dalle ultime stime)

- Esempio: se il valore del RTO è 0,75 sec se il timer scade prima della ricezione del ACK, RTO verrà impostato a 1,5 sec. Se scade ancora verrà impostato a 3 sec.

RTO cresce esponenzialmente

Tutte le volte che il timer viene avviato per un evento di

- ricezione di un ACK
- ricezione di dati dall'applicazione

il valore del RTO viene ricavato dalle stime (ritorna la gestione 'ordinaria' del RTO)

È una forma (limitata) di controllo della congestione

Meccanismo noto come exponential backoff

TCP: fast retransmit

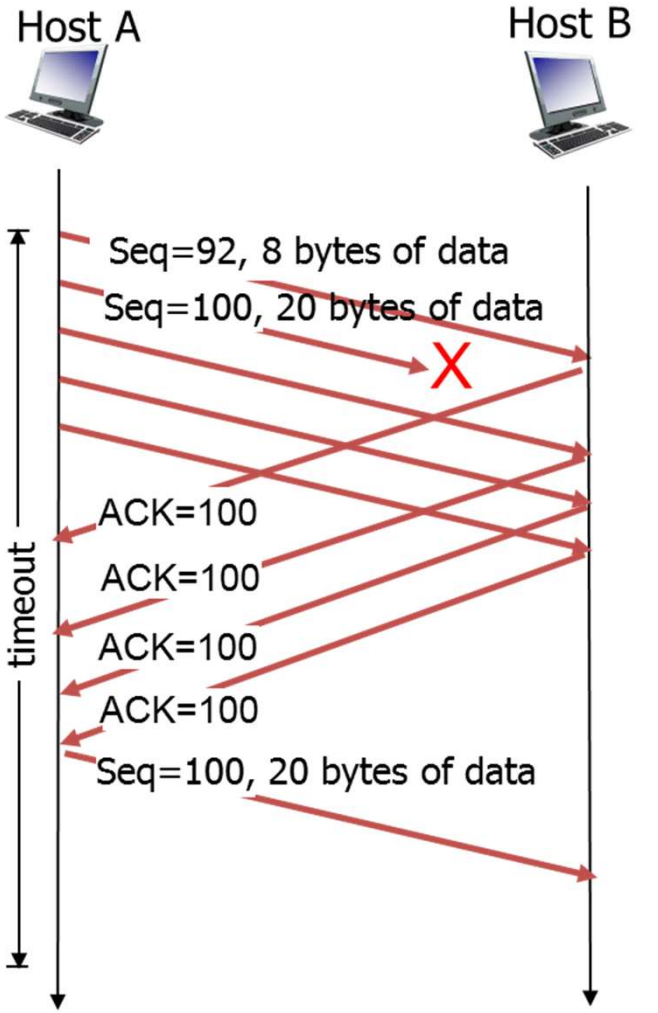
Il tempo di RTO può rivelarsi relativamente lungo

- Ritardo eccessivo prima di rispedire il pacchetto perso
- Grazie agli ACK duplicati il mittente può in molti casi rilevare la perdita dei pacchetti ben prima che si verifichi l'evento di timeout
  - il mittente spesso invia sequenze di segmenti
  - se un segmento viene perso, sarà molto probabile che ci saranno degli ACK duplicati

TCP fast retransmit

- Se il mittente riceve 3 ACK duplicati per lo stesso segmento ("triple duplicate ACKs"), rispedisce il segmento non riscontrato con il più piccolo numero di sequenza
- Probabilimente tale segmento è stato perso e si può risparmiare tempo evitando di attente lo scadere del timout

TCP: fast retransmit (II)



fast retransmit after sender receipt of triple duplicate ACK

Generazione degli ACK [RFC 1122, RFC 2581]

Evento	Azione del ricevente TCP
Arrivo ordinato di segmento con numero di sequenza atteso. Tutti i dati fino al numero di sequenza atteso 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.

Go-Back-N o Selective Repeat

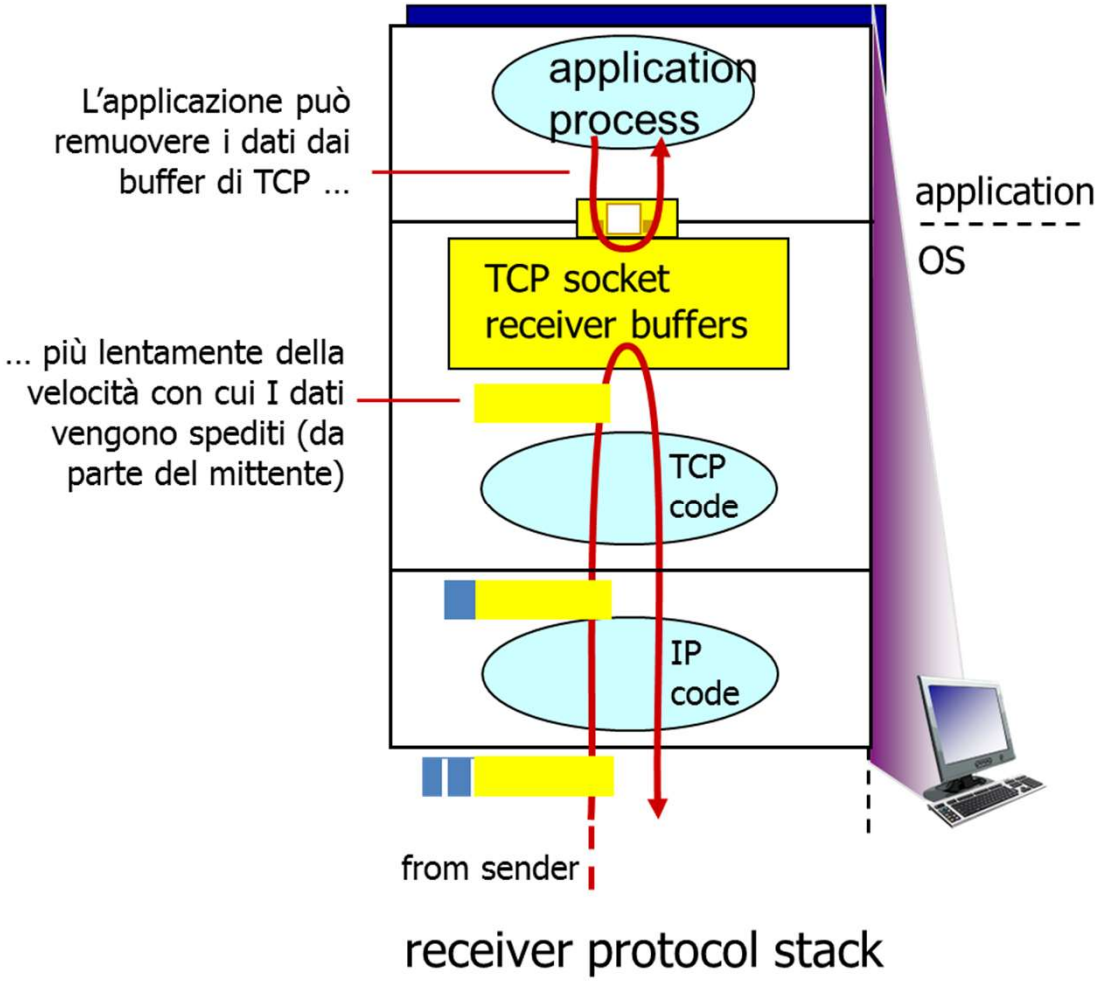
- TCP è un protocollo di tipo GBN oppure SR ?
- Poiché ogni messaggio ACK(y) riscontra positivamente tutti i byte ricevuti dalla destinazione con numero d'ordine ≤(y-1), il tipo di messaggio ACK impiegato dal TCP è cumulativo. L'impiego di ACK cumulativi è tipico della strategia Go-back-N
- D'altra parte, mediante l'impiego delle strategie di TimeOut e di Fast Retransmission, il TCP è in grado di ritrasmettere singoli segmenti. La capacità di ritrasmettere singoli segmenti è tipica della strategia Selective Repeat(SR)
- In conclusione, il meccanismo di ripristinod da errori usato dal TCP è un ibrido delle strategie Go-Back-N e Selective Repeat

Sommario:

- I servizi del livello di transporto
- Multiplexing and demultiplexing
- Protocollo di trasporto connectionless: UDP
- Principi per rendere affidabile il trasferimento dati
- Protocollo di trasporto connection-oriented: TCP
  - Struttura dei segmenti
  - Trasferimento dati affidabile
  - **Controllo di flusso**
  - Gestione della connessione
- La congestione e la sua gestione
- Gestione della congestione in TCP

TCP: controllo di flusso

- Quando viene instaurata una connessione tra mittente e destinatario, il TCP al lato destinazione alloca un buffer di ricezione per la memorizzazione temporanea dei segmenti ricevuti che non sono ancora stati passati all'applicazione destinataria
- Sia **`RcvBuffer`** la dimensione (in byte) di questo buffer
- Il Controllo di Flusso serve a regolare (aumentare o diminuire) la velocità (byte/sec) con cui il TCP mittente immette byte nella connessione in modo da garantire che il numero di byte da memorizzare nel buffer di ricezione non superi mai il valore: **`RcvBuffer`**
- Per implementare questa funzionalità, il lato sorgente e il lato destinazione della connessione TCP mantengono e aggiornano una finestra di trasmissione e una finestra di ricezione, le dimensioni delle quali possono variare nel tempo



Controllo di flusso: lato destinatario

- Il lato destinazione della connessione TCP mantiene e aggiorna le due variabili
  - **`Last_Byte_Rcvd`**: è il numero d'ordine dell'ultimo byte (ossia del byte più recente) che la destinazione ha ricevuto dalla connessione;
  - **`Last_Byte_Read`**: è il numero d'ordine del byte più recente che il processo destinazione ha prelevato (letto) dal buffer di ricezione;
- Affinchè il buffer di ricezione non vada in overflow, deve essere verificata la condizione

$$\texttt{Last\_Byte\_Rcvd} - \texttt{Last\_Byte\_Read} \leq \texttt{RcvBuffer}$$

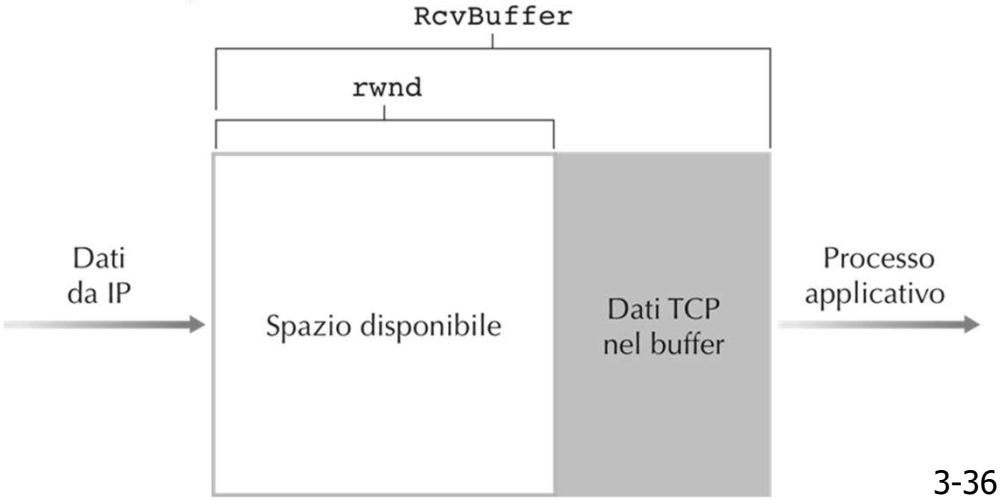
Controllo di flusso: lato destinatario (II)

Se **Last_Byte_Rcvd** e di **Last_Byte_Read** sono i valori correnti, il TCP lato destinatario effettua operazioni:

1. Aggiorna la variabile: **RcvWindow**

   rwnd = RcvBuffer–(Last_Byte_Rcvd –Last_Byte_Read)

   rwnd rappresenta il numero di byte che sono ancora liberi nel buffer di ricezione
2. Memorizza il valore della variabile: **rwnd** nel campo: **ReceiveWindow** di un nuovo segmento che il TCP destinatario invierà al mittente;
3. Invia il segmento al mittente (TCP lato mittente saprà quanti byte sono correntemente liberi nel buffer di ricezione)



Controllo di flusso: lato mittente

- Il TCP al lato mittente mantiene le due seguenti variabili:
  - **`Last_Byte_Sent`**: numero d'ordine dell'ultimo byte (ossia del byte più recente) che il mittente ha inviato nella connessione;
  - **`Last_Byte_Acked`**: numero d'ordine dell'ultimo byte (ossia del byte più recente) di cui il mittente ha ricevuto un ACK da parte del destinatario

    `Last_Byte_Sent – Last_Byte_Acked`

    rappresenta il numero di byte che sono stati trasmessi dal mittente ma non ancora riscontrati
- Quando il mittente riceve un nuovo segmento dal destinatario, il TCP effettua le seguenti operazioni:
  - Estrae dal segmento ricevuto il contenuto del campo: **`ReceiveWindow`** (ossia, legge il valore di **`rwnd`**);
  - Aggiorna la variabile: **`Last_Byte_Sent`** (ossia, eventualmente trasmette un certo numero di byte) in modo che risulti in ogni caso soddisfatta la seguente relazione:

    `Last_Byte_Sent` ≤ `Last_Byte_Acked + rwnd`
- Fino a quando il TCP a lato sorgente garantisce il soddisfacimento della relazione precedente, il buffer di ricezione non andrà mai in overflow

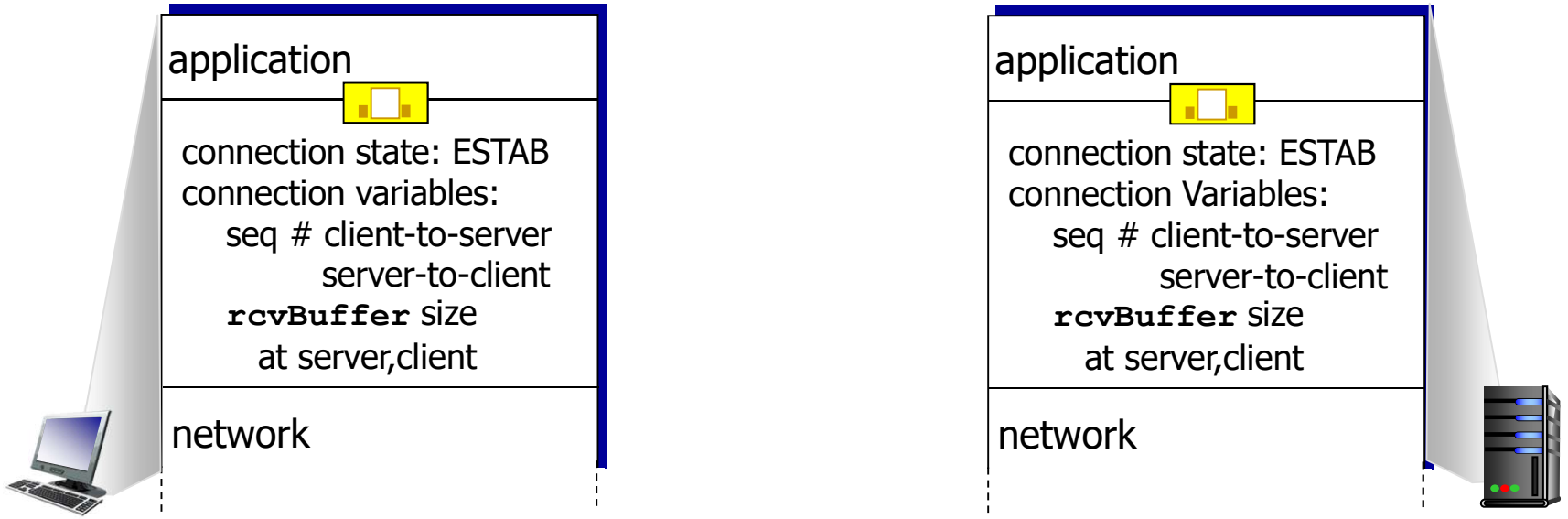
Sommario

- I servizi del livello di transporto
- Multiplexing and demultiplexing
- Protocollo di trasporto connectionless: UDP
- Principi per rendere affidabile il trasferimento dati
- Protocollo di trasporto connection-oriented: TCP
  - Struttura dei segmenti
  - Trasferimento dati affidabile
  - Controllo di flusso
  - **Gestione della connessione**
- La congestione e la sua gestione
- Gestione della congestione in TCP

Gestione della connessione

Prima di scambiarsi dei dati mittente/destinatario devono fare il setup della connessione (handshake)

- Devono essere daccordo a stabilire la connessione (ognuno deve sapere che la controparte vuole stabilire la connessione)
- Devono accordarsi su alcuni paramentri usati dalla connessione




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

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

Gestione della connessione (II)

- La vita di una connessione TCP è suddivisa nelle tre fasi:
  - Fase di setup della connessione;
  - Fase di connessione aperta;
  - Fase di chiusura della connessione

TCP: stabilire la connessione

La fase di setup della connessione è composta di tre passi.

- **Passo 1**: il processo client invia uno (speciale) segmento TCP al processo server. Il segmento non contiene dati nel suo campo payload. Il segmento ha:
  - I bit del campo SYN posto a 1;
  - Il campo sequence number che contiene il numero iniziale sequenza *x* che il cliente ha scelto per numerare i segmenti
- **Passo 2**: dopo che il segmento in oggetto ha raggiunto il processo server, il TCP al lato server alloca il buffer di ricezione, inizializza le variabili necessarie per gestire la connessione, e, poi, invia un segmento di riscontro(detto SYNACK segment) al processo cliente. Il segmento di riscontro che il processo server invia al processo cliente non contiene dati nel suo campo payload. Il segmento SYNACK ha:
  - Il bit del campo SYN posto a 1;
  - Il campo Acknowledgement number posto a: *(x+1)*;
  - Il campo sequence number posto a: *y*, dove *y* èi l numero iniziale di sequenza che il processo server usa per numerare i segmenti che tale processo invierà al client

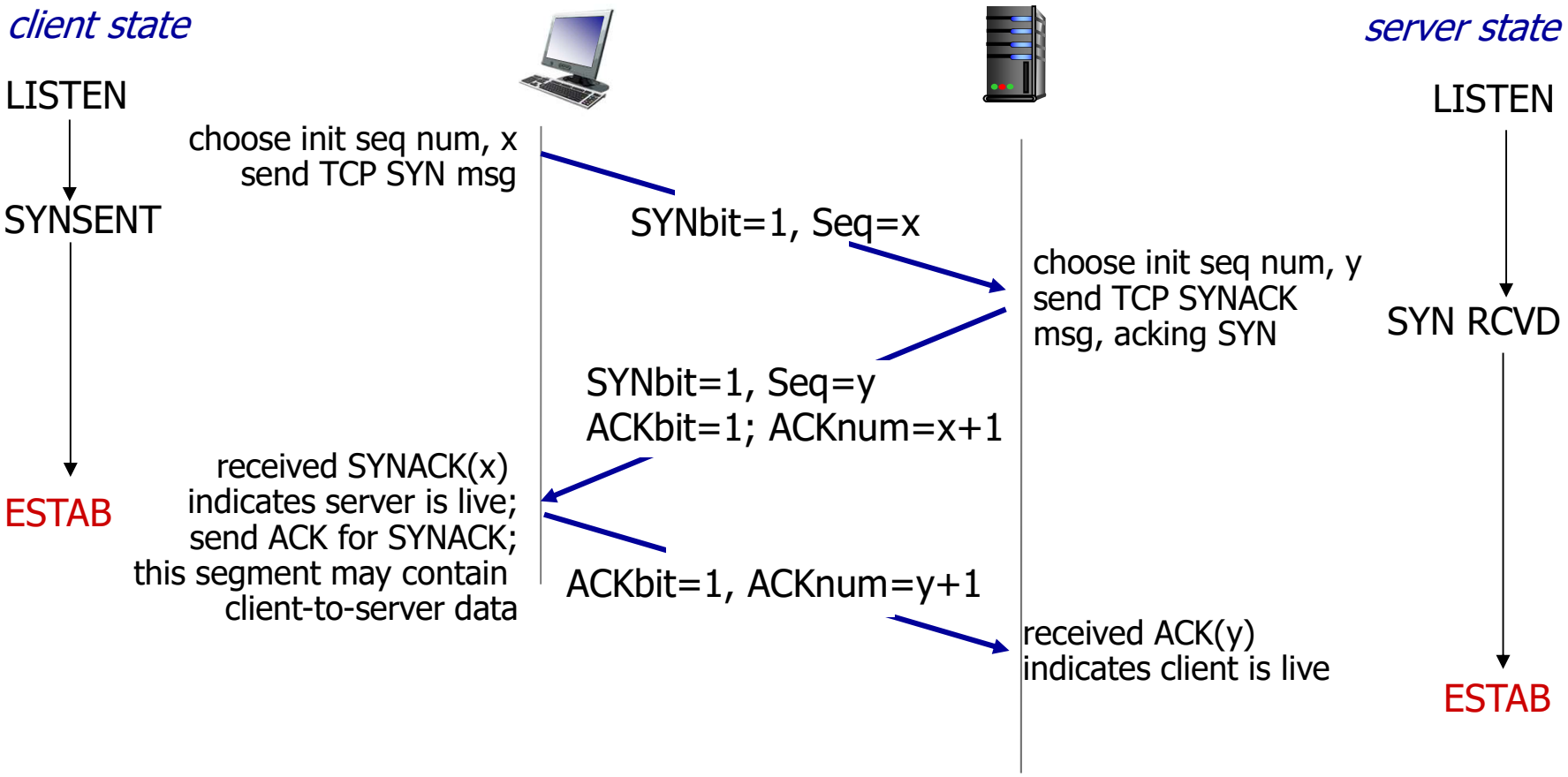
Nota: con l'invio del segmento SYNACK, il processo server sta dicendo al processo client: "ho ricevuto la tua richiesta di connessione con il tuo iniziale numero di sequenza, *x*. Concordo nello stabilire la connessione. Il mio iniziale numero di sequenza è: y"

TCP: stabilire la connessione (II)

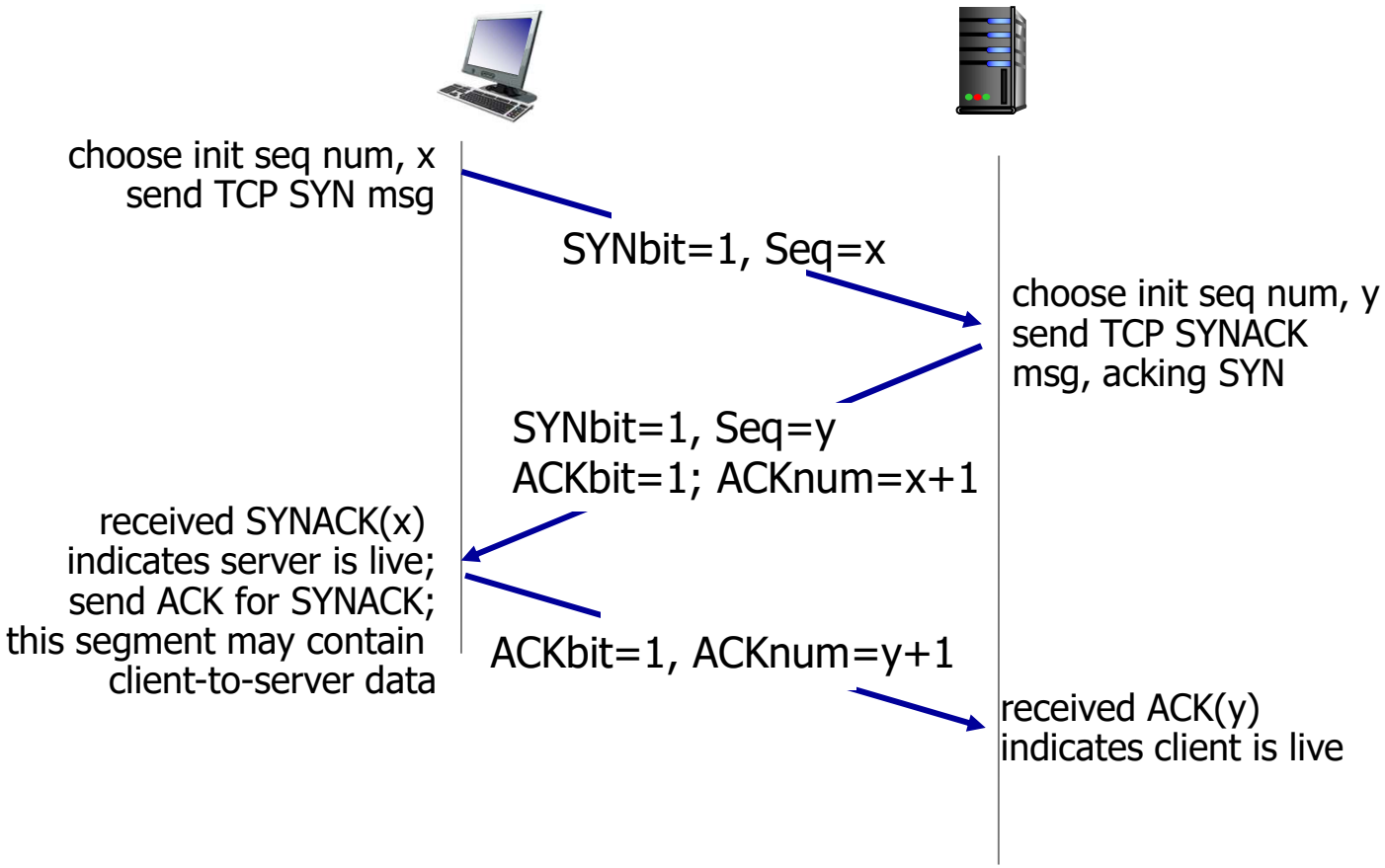
- **Passo 3**: dopo aver ricevuto il segmento SYNACK dal server, il TCP al lato client alloca il buffer di trasmissione e inizializza le variabili necessarie per gestire la connessione. Quindi, il cliente invia al server un (ultimo) segmento che notifica al servente che il cliente ha ricevuto il segmento SYNACK. Quest'ultimo segmento che il cliente invia al servente ha:
  - Il bit del campo SYN posto a 0
  - Il campo sequence number posto a: *(x+1)*
  - Il campo Acknowledgement number posto a: *(y+1)*

Dopo la ricezione di quest'ultimo segmento, il processo server può iniziare a inviare i suoi messaggi al processo client, attraverso la connessione instaurata

TCP: stabilire la connessione (III)

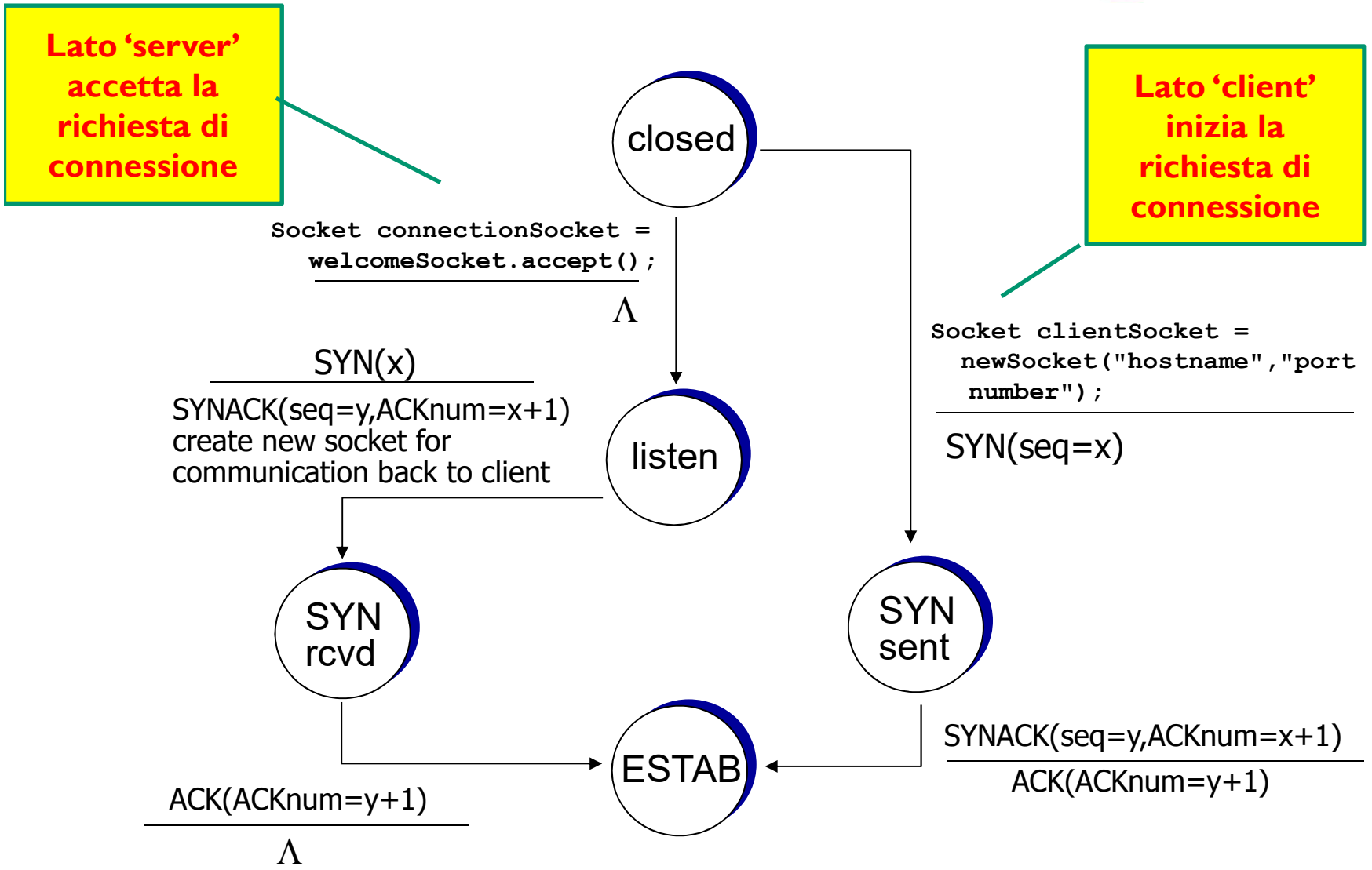


TCP: stabilire la connessione (IV)



TCP: stabilire la connessione (V)

FSM per handshake a 3-fasi



TCP: chiusura della connessione

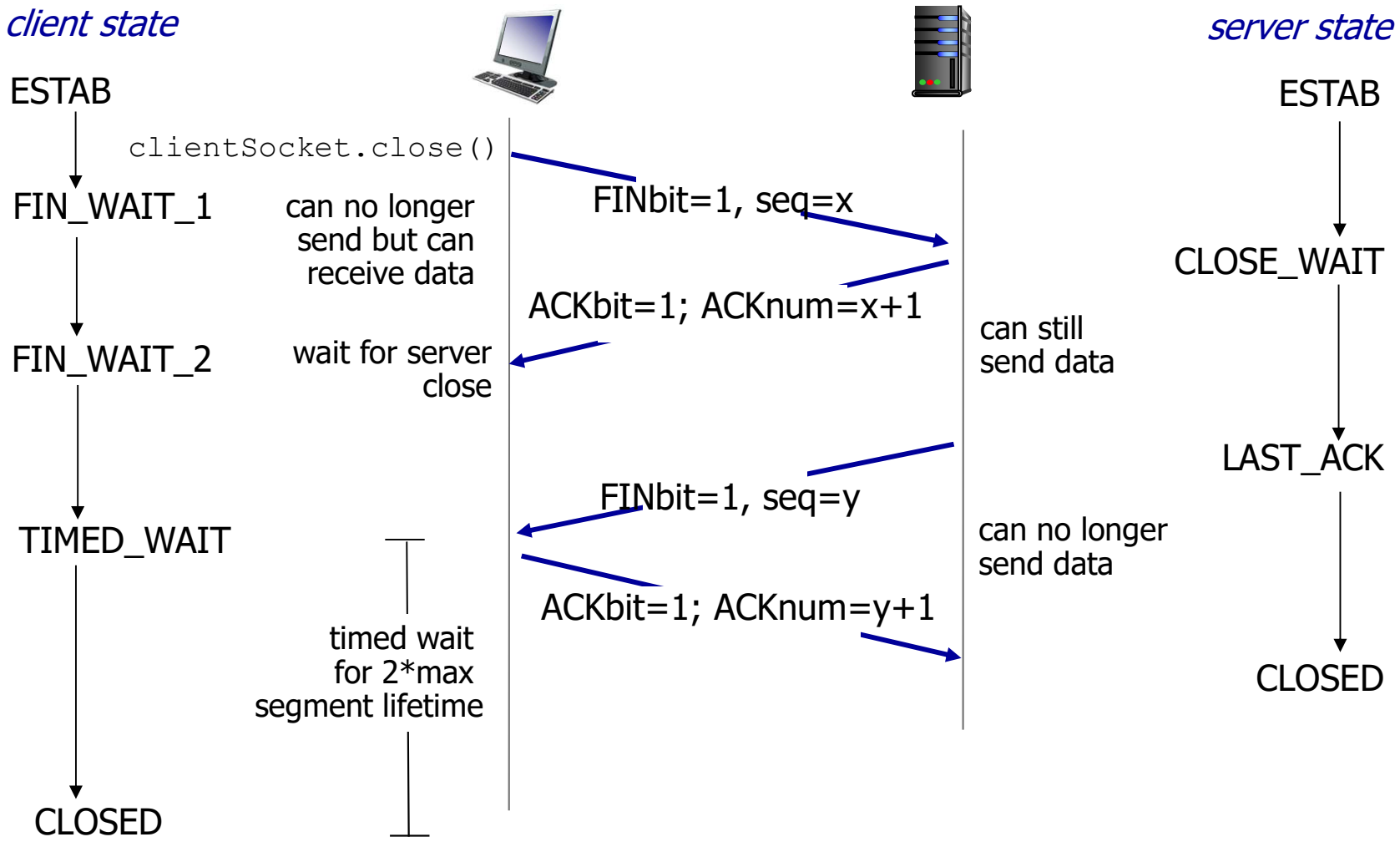
Ogni 'lato' di una connessione (una connessione TCP è full-duplex) viene rilasciata in modo indipendente

- la connessione invia un segmento con FIN = 1 per indicare che non ha più dati da trasmettere
- Quando questo segmento viene riscontrato quel lato della connessione viene chiuso

Chiusura senza perdita dei dati in transito

- Un flusso può essere chiuso prima dell'altro (full-duplex), e viene chiuso dall'applicazione mittente del flusso, quando decide che non ha più dati da trasmettere e il protocollo applicativo lo permette
- Il TCP mittente aspetta di avere il riscontro di tutti i dati mandati e poi invia il FIN con un numero di sequenza successivo
- I dati possono essere mandati solo nell'altra direzione, fino a che l'altro mittente non chiude la sua direzione di spedizione. Il suo TCP manda tutto e aspetta il riscontro, poi manda FIN con il suo sequence number successivo. Quando finalmente hanno il riscontro tutti e due, i TCP distruggono il descrittore della connessione

TCP: chiusura della connessione (II)



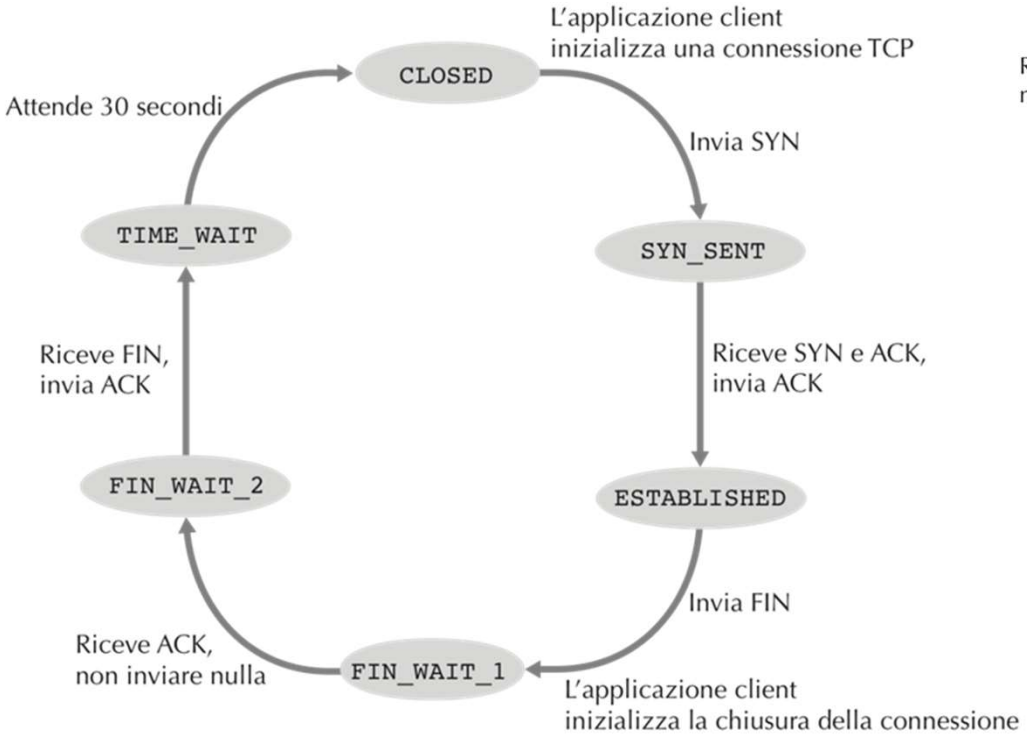
TCP: chiusura della connessione (III)

La fase di chisura della connessione di articola in 5 passi:

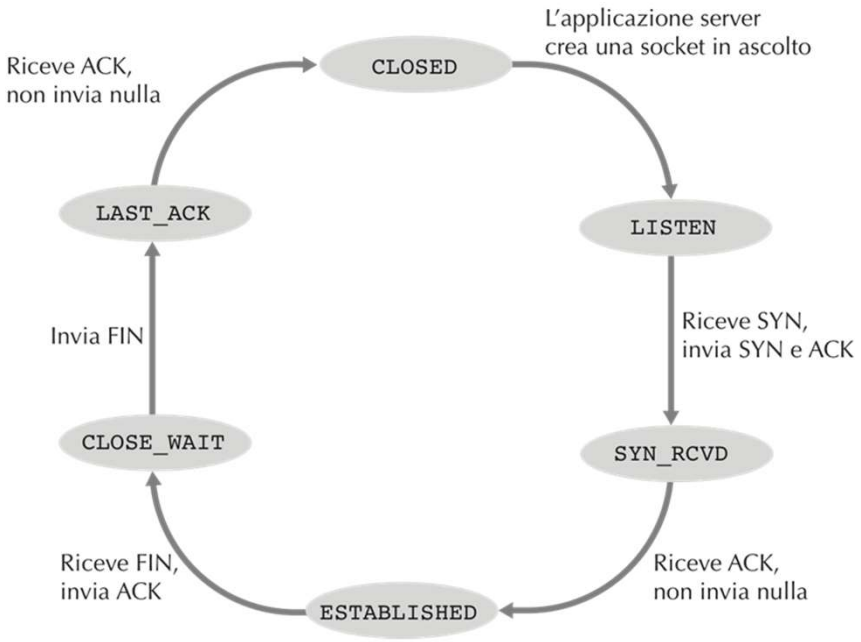
- **Passo 1**: il processo cliente invia al processo server un segmento con bit del campo FIN posto a 1;
- **Passo 2**: dopo aver ricevuto il segmento, il processo server invia al processo client un segmento con bit del campo ACK posto a 1. Il processo server dealloca tutte le risorse della connessione (buffer di ricezione e variabili)
- **Passo 3**: il processo server invia un altro segmento al processo client col bit del campo FIN posto a 1
- **Passo 4**: alla ricezione del suddetto segmento, il processo client invia un segmento al server con ACK=1. Dopo di che, il processo cliente entra in uno stato di attesa che, tipicamente, dura 30sec, e che ha lo scopo di permettere al segmento ACK di propagarsi fino al server
- **Passo 5**: alla ricezione dell'ultimo segmento con ACK=1, il processo server chiude definitivamente la connessione. Tutte le risorse (buffer e variabili) dedicate alla connessione sono definitivamente de-allocate

Gestione della connessione (cont.)

Nell'arco di una connessione TCP, i protocolli TCP in esecuzione negli host attraversano vari stati TCP



Tipica sequenza di stati visitati da TCP (lato client)



Tipica sequenza di stati visitati da TCP (lato server)

Connessione, reset, sicurezza

Cosa succede quando un host riceva un segmento TCP i cui numeri di porta non corrispondono ad alcun socket attivo su tale porta

- Es. un host riceva un pacchetto TCP SYN con porta destinazione 80 ma su quella porta non è in esecuzione alcun server

In questi casi l'host invierà un segmento con RST

Scansione delle porte

- Per analizzare una specifica porta su un host 'bersaglio' si spedisce un TCP SYN con quella porta destinazione
  - L'host sorgente riceve un TCP SYNACK: esiste sull'host bersaglio un'applicazione attiva su tale porta
  - L'host sorgente riceve un TCP RST: il segmento ha raggiunto l'host bersaglio ma su tale porta non è in esecuzione alcuna applicazione. L'attaccante sa però che la porta in questione non è bloccata da un firewall (si può raggiungere)
  - L'host sorgente non riceve nulla: il segmento è stato bloccato

Esistono strumenti per effettuare port-scanning (es. nmap)

Una "scansione di vulnerabilità" permette di realizzare un controllo della sicurezza di una rete effettuando una scansione delle porte

Connessione, reset, sicurezza (II)

Durante l'handshake a tre vie TCP destinatario (es. un server) alloca e inzializza le variabili ed i buffer della connessione in risposta ad un TCP SYN ricevuto (es. da un client)

Il server manda un TCP SYNACK in risposta ed attende un ACK dal client come terzo ed ultimo passo dell'handshake

Se il client non manda un ACK per completare il terzo passo, alla fine (spesso dopo un pinuto o più) il server termina la connessione e dealloca le risorse

Attacco SYN flood

L'aggressore manda un gran numero di segmenti TCP SYN senza completare il terzo passo dell'handshake

Attacco di tipo DoS (o Distributed DoS)

Esistono delle difese efficaci contro questo tipo di attacco incluse nella maggior parte dei systemi operativi (SYN coockie [RFC 4987])

Sommario

Sommario:

- I servizi del livello di transporto
- Multiplexing and demultiplexing
- Protocollo di trasporto connectionless: UDP
- Principi per rendere affidabile il trasferimento dati
- Protocollo di trasporto connection-oriented: TCP
  - Struttura dei segmenti
  - Trasferimento dati affidabile
  - Controllo di flusso
  - Gestione della connessione
- **La congestione e la sua gestione**
- Gestione della congestione in TCP

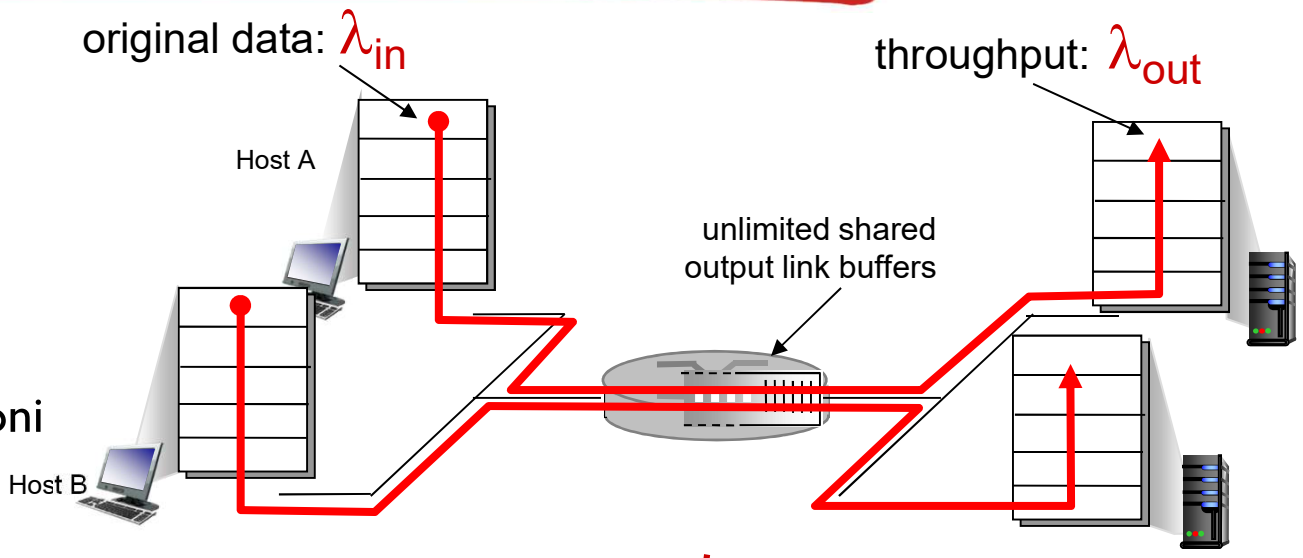
Principi del controllo della congestione

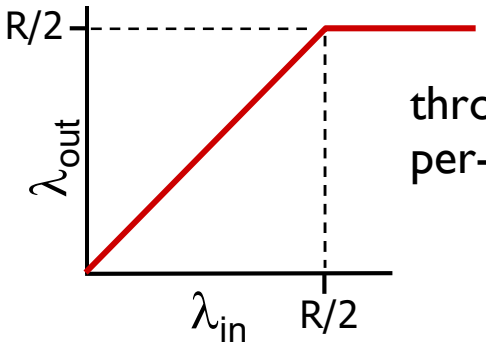
Congestione: condizione di fortissimo ritardo e perdita dei datagrammi, causata da un sovraccarico sulle interfacce, ad uno o più router. Come si crea?

- Più interfacce di un router "mandano" pacchetti sulla stessa interfaccia in uscita
- Quando il tasso di pacchetti in uscita supera la velocità di assorbimento della interfaccia, il router comincia ad accodare i pacchetti in uscita
- Se il sovraccarico continua, la coda cresce di lunghezza
- Quando il router ha finito i buffer di memoria disponibili per l'interfaccia sovraccarica, deve scartare i pacchetti in ingresso (solo dopo aver capito che sono diretti alla interfaccia congestionata!)
- Quando il router non ha velocità sufficiente a trattare i pacchetti ricevuti, le code di ingresso si allungano
- Quando il router ha finito i buffer di memoria disponibili per l'interfaccia di ingresso sovraccarica, rifiuta di leggere nuovi pacchetti (che possono essere persi nel mezzo fisico, ad es. in Ethernet)

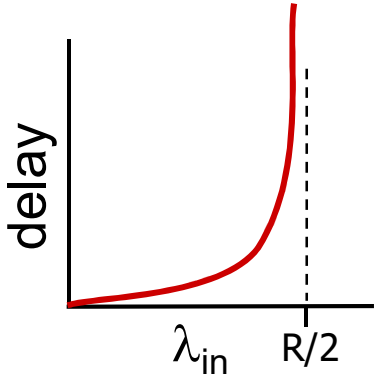
Cause/costi della congestion: scenario 1

- Due mittente, due ricevitori
- Un router, con buffer infinito
- Capacità link di output output: R
- Non ci sono ritrasmissioni





throughput massimo per-connection: R/2



Ritardo che cresce al mano mano che il tasso di arrivo λ_{in}, si avvicina alla capacità

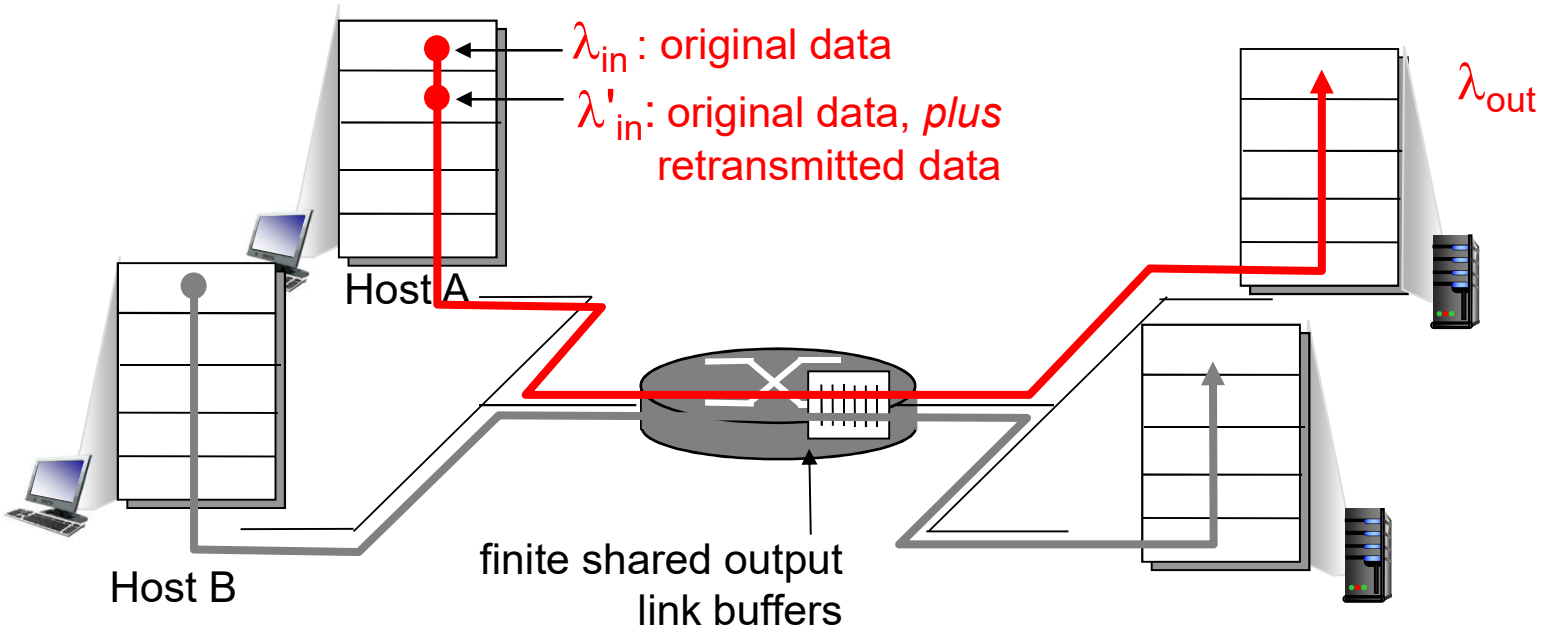
Quando il tasso di arrivo dei pacchetti si avvicina alla capacità del collegamento, i ritardi (dovuto all'accodamento dei pacchetti) crescono

Cause/costi della congestion: scenario 2

un router, buffer *di dimensione finita*

mittente ritrasmesste pacchetti per i quali ha ricevuto un timed-out

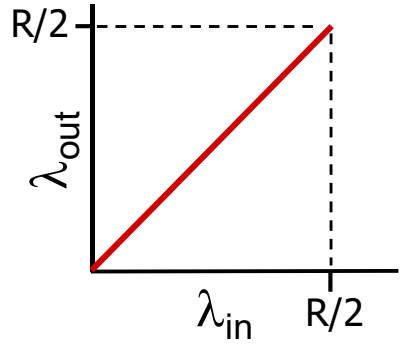
- Tasso di trasmissione (da parte dell'applicazione) = Tasso di ricezione: $\lambda_{in} = \lambda_{out}$
- L'input del livello trasporto include le ritrasmissioni: $\lambda'_{in} \geq \lambda_{in}$

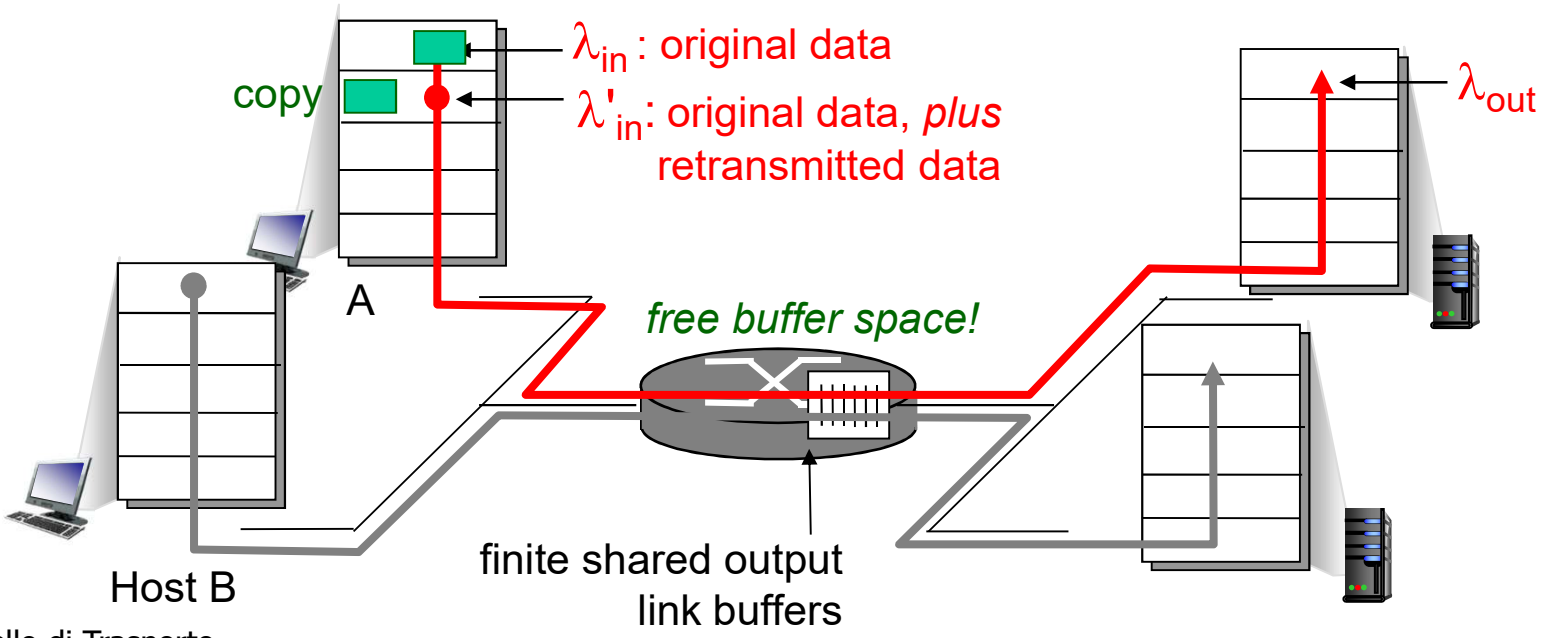


Cause/costi della congestion: scenario 2 (cont)

idealizzazione: conoscenza perfetta

Il mittente spedisce solo quando c'è spazio nel buffer del router

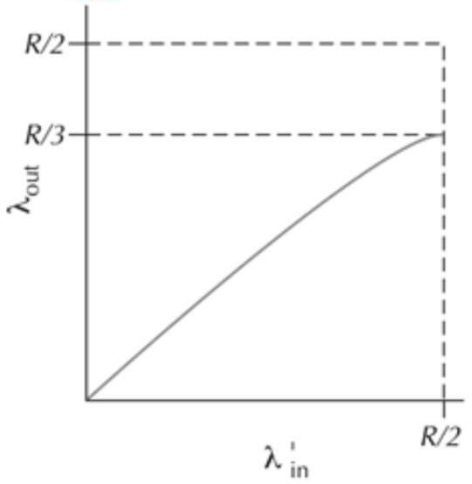


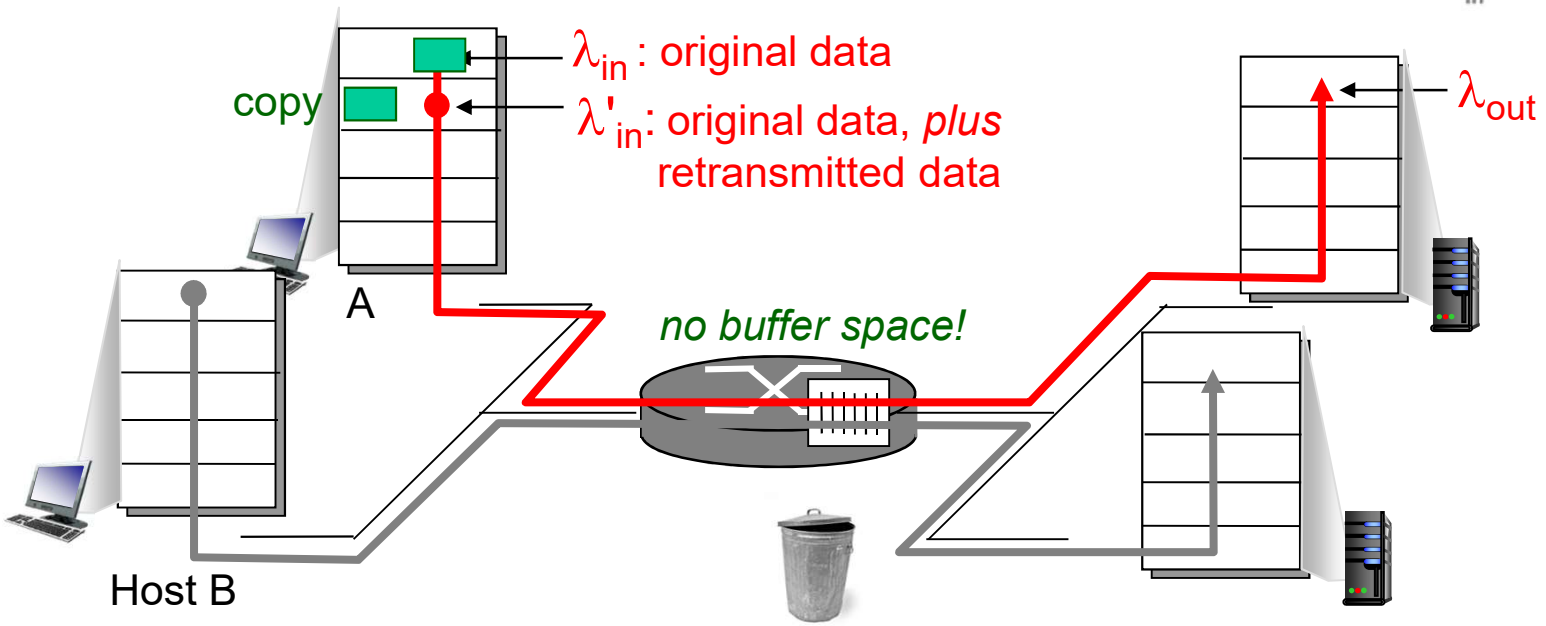


Cause/costi della congestion: scenario 2 (cont)

Idealizzazione: le perdite sono note

- I pacchetti possono essere persi solo al router, vengono scartati perché il buffer è pieno
- Il mittente ri-spedisce il pacchetto solamente se sa che è stato perso

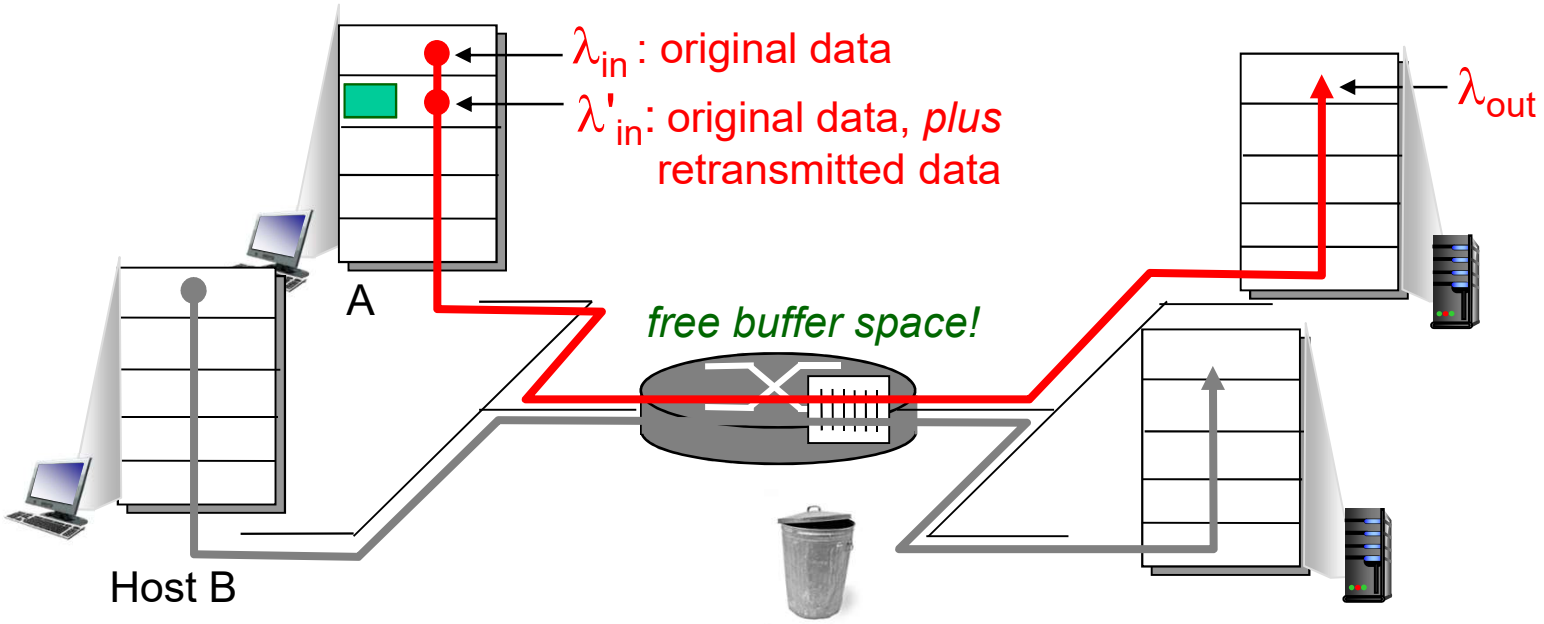




Cause/costi della congestion: scenario 2 (cont)

Idealizzazione: le perdite sono note

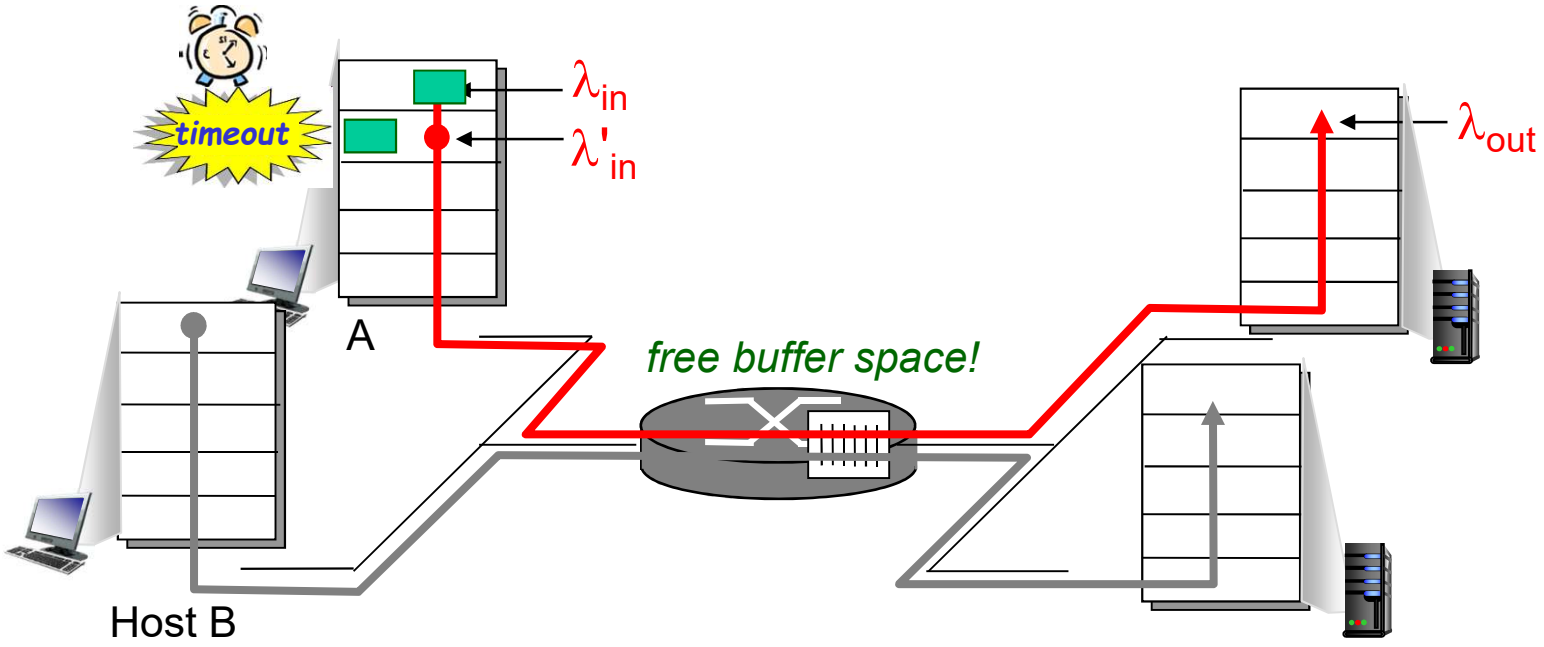
- I pacchetti possono essere persi solo al router, vengono scartati perché il buffer è pieno
- Il mittente ri-spedisce il pacchetto solamente se sa che è stato perso



Cause/costi della congestion: scenario 2 (cont)

Scenario realistico: duplicati

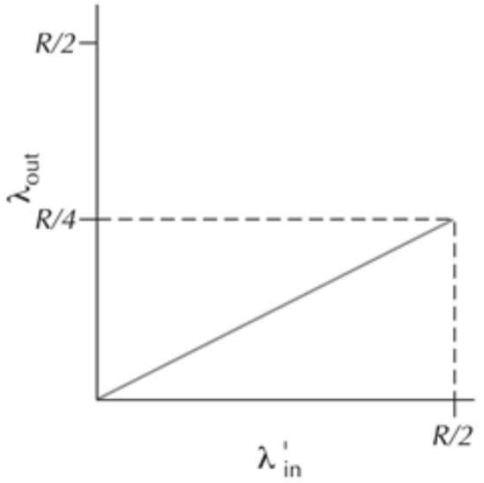
- I pacchetti possono essere persi, scartati al router per buffer pieno
- Al mittente scattano (prematuramente) dei times out e spedisce **due** copie che vengono entrambe spedite



Cause/costi della congestion: scenario 2 (cont)

Scenario realistico: duplicati

- I pacchetti possono essere persi, scartati al router per buffer pieno
- Al mittente scattano (prematuramente) dei times out e spedisce **due** copie che vengono entrambe spedite



"Costi" della congestione:

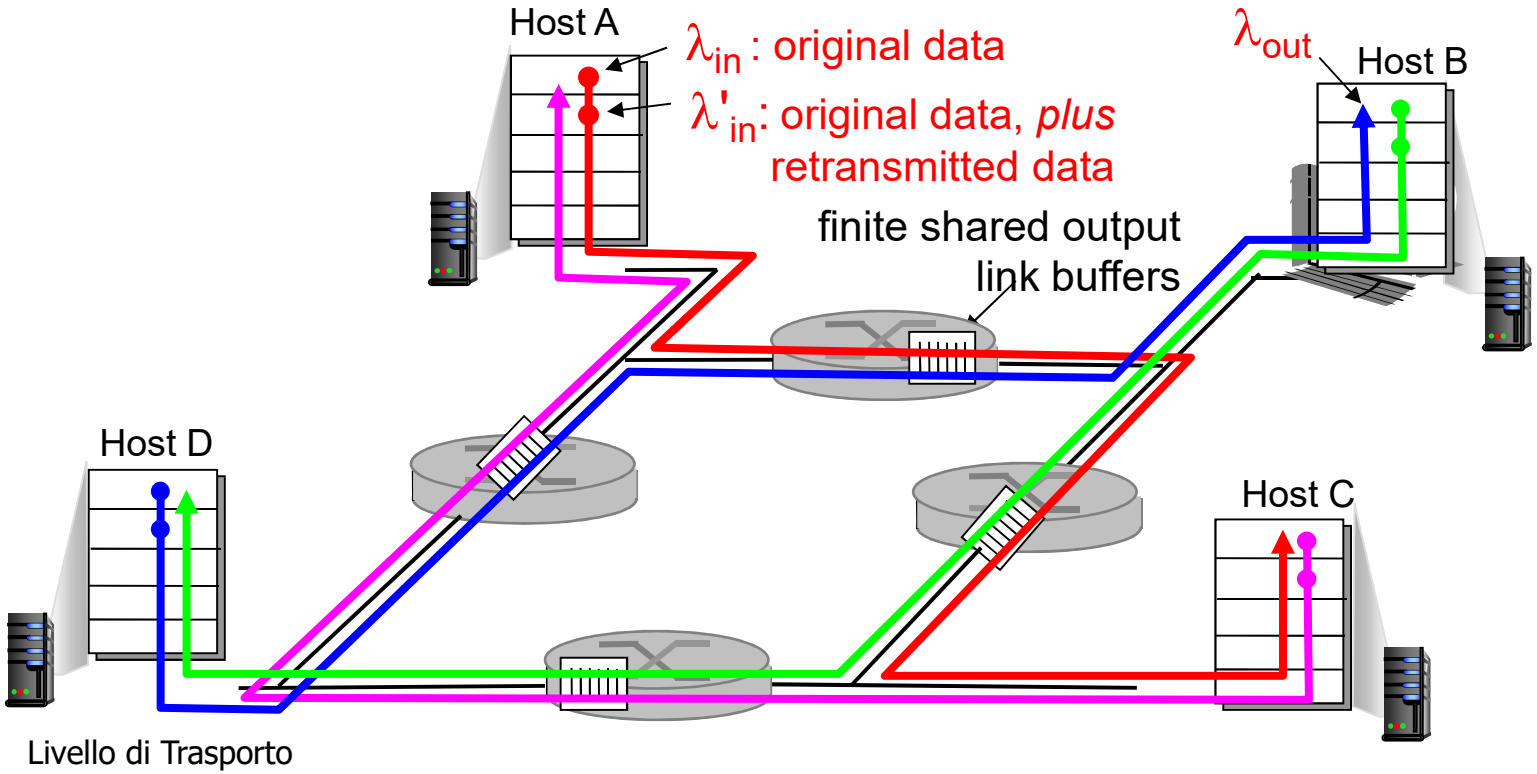
- Lavoro aggiuntivo (ritrasmissione di pacchetti) per ottenere un determinato *goodput*
- Trasmissione non necessarie: il link trasporta copie multiple dello stesso pacchetto
  - decremento del goodput

Cause/costi della congestion: scenario 3

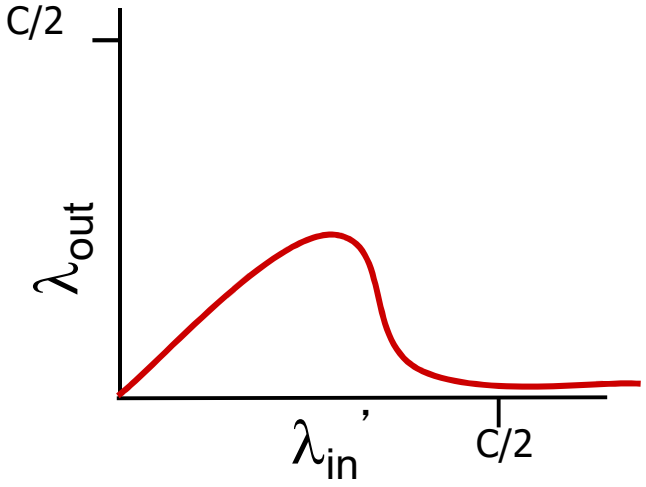
- Quattro mittenti
- Percorsi multihop
- Timeout/ritrasmissioni

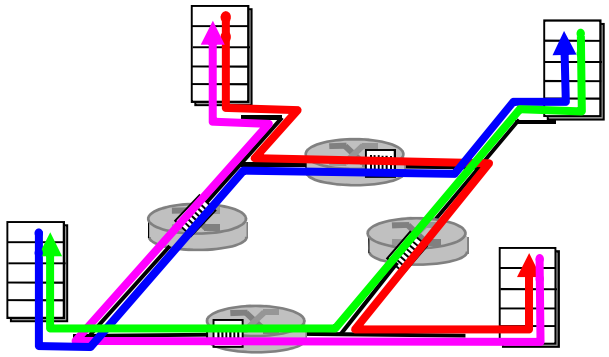
D: cosa succede all'aumentare di λ_{in} e λ'_{in} ?

R: all'aumentare di red λ'_{in} (rosso), tutti i pacchetti blu che arrivano alla coda vengono scartati, il throughput blu→ 0



Cause/costi della congestion: scenario 3 (cont)





Un altro costo della congestione:

- Quando i pacchetti vengono scartati, la capacità trasmissiva, utilizzata sui collegamenti per instradare il pacchetto fino al punto in cui è stato scartato, risulta sprecata

Approcci al controllo della congestione

Esistono due principali approcci al controllo della congestione:

Controllo di congestione end-to-end:

- Nessun feedback esplicito da parte della rete
- La presenza di congestione viene dedotta sulla base di osservazioni da parte degli end system (perdite di pacchetti, ACK duplicati, timeout, ritardi crescenti)
- È l'approccio usato da TCP

Controllo di congestione assistito dalla rete:

- I router forniscono feedback agli end-system
  - Un bit che indica l'insorgere di congestione (SNA, DECbit, TCP/IP ECN, RED, ATM)

Sommario

- I servizi del livello di transporto
- Multiplexing and demultiplexing
- Protocollo di trasporto connectionless: UDP
- Principi per rendere affidabile il trasferimento dati
- Protocollo di trasporto connection-oriented: TCP
  - Struttura dei segmenti
  - Trasferimento dati affidabile
  - Controllo di flusso
  - Gestione della connessione
- La congestione e la sua gestione
- **Gestione della congestione in TCP**

Controllo di congestione di TCP

- Consideriamo una connessione TCP da una processo sorgente (in esecuzione sull'end-system HS) ad un processo destinazione (in esecuzione sull'end-system HD)
- La connessione impiega router e canali trasmissivi che possono essere contemporaneamente utilizzati da altre connessioni attive nella rete e può accadere che il numero totale di segmenti che ciascun router e canale trasmissivo deve inoltrare nell'unità di tempo (traffico aggregato) sia più grande delle velocità di inoltro(switching speed) dei router e delle capacità dei canali trasmissivi
- Quando ciò accade, la rete viene detta essere congestionata
- In presenza di congestione, il TCP al lato sorgente deve ridurre opportunamente la velocità (byte/sec) con cui immette segmenti nella connessione (e, quindi, nella rete)
- Il livello network non fornisce al TCP sorgente alcuna informazione sulla congestione (e non fornisce strumenti adeguati per risolvere tale problema)
- TCP deve adottare un approccio del tipo end-to-end per controllare (ridurre) la congestione che si sta verificando in rete.

Controllo di congestione di TCP (II)

Il controllo di congestione "end-to-end" pone 3 problemi al TCP lato sorgente:

1. Come può il TCP regolare (aumentare o diminuire) la velocità (misurata in byte/sec) con la quale immette segmenti nella connessione?
2. Come può il TCP al lato sorgente accorgersi che si sta verificando nella rete una congestione e misurarne il livello?
3. Quale algoritmo adotta il TCP per controllare la velocità (byte/sec) di immissione di segmenti nella connessione in funzione del livello della congestione che si sta verificando nella rete?

Controllo di congestione di TCP (III)

- Per regolare la velocità di immissione di segmenti nella connessione TCP mantiene e aggiorna (lato sorgente) la variabile cwdn (Finestra di congestione)
- cwnd (in byte): massimo numero di byte che il TCP può immettere nella connessione senza bisogno di ricevere alcun ACK
- TCP lato sorgente garantisce che:

  $$\texttt{Last\_Byte\_Sent-Last\_Byte\_Acked} \leq \min\{\texttt{cwnd}, \texttt{rwnd}\}$$

- **rwnd**= $+\infty$, così che il TCP della connessione (lato mittente $\rightarrow$ destinatario) stia solo effettuando controllo di congestione
- TCP al lato sorgente immetta nella connessione un numero di byte pari a **cwnd**. Dopo un intervallo di tempo pari a RTT, il lato sorgente della connessione riceverà un ACK dal lato destinazione che riscontra tutti i byte trasmessi.
- Quindi, in assenza di perdite e errori, la velocità (in byte/sec) con cui mediamente il TCP immette segmenti nella connessione è:

  cwnd/RTT (byte/sec)

- **TCP può controllare (aumentare o diminuire) la velocità (in byte/sec) con cui immette segmenti nella connessione aumentando o diminuendo la dimensione della Finestra di Congestione**

Controllo di congestione di TCP:

perdita di un segmento

- TCP al lato sorgente si accorge (rileva) che si sta verificando una situazione di congestione nella rete quando, al lato sorgente, si verificano uno, o più, eventi di perdita di segmenti
- Per definizione, si verifica un Evento di Perdita quando, al lato sorgente della connessione, accade l'uno o l'altro dei due seguenti eventi:
- La sorgente riceve tre messaggi di riscontro con lo stesso numero di sequenza (three duplicate ACKs);
- Il timer (lato sorgente) scade, ovvero si verifica l'evento:

`Timer > RTO` (Time Out Event)

- Quando si verifica un Evento di Perdita, il TCP al lato sorgente della connessione ***riduce*** la dimensione **`cwnd`** della finestra di congestione
- L'evento: "ricezione di 3 ACKs con lo stesso numero di sequenza" è indice di un livello di congestione meno grave dell'evento di "Time Out".
- Quindi, la riduzione di **`cwnd`** susseguente all'evento di Time Out è maggiore della riduzione indotta dalla ricezione di 3 ACKs con lo stesso numero di sequenza

L'algoritmo di controllo di congestione

- Negli ultimi anni, sono state sviluppate varie versioni dell'Algoritmo di controllo di congestione di TCP: versione "TCP Reno "
- Algoritmo che si compone di 2 stati:
  - Congestion Avoidance (CA state);
  - Slow Start (SS state)
- L'Algoritmo di controllo di Congestione è implementato solo dal TCP del lato sorgente

Congestion Avoidance (CA)

- Nello stato di CA, il TCP al lato sorgente aggiorna la dimensione **cwnd** della finestra di congestione in corrispondenza al verificarsi di:
  - Ricezione di un ACK;
  - Ricezione di 3 ACKs con lo stesso numero di sequenza;

  cwnd è incrementata di:

  MSS (MSS/ CongWin) (bytes),

  ossia,

  cwnd := cwnd+MSS (MSS/cwnd)

  per ogni singolo segmento che è riscontrato positivamente per la prima volta (Fase di Incremento Additivo della finestra di congestione)
- Quando si verifica un evento di "ricezione di 3 ACKs duplicati", il valore di cwnd è dimezzato ma, in ogni caso, non è fatto scendere sotto il valore minimo di 1 MSS. Alla ricezione di 3 duplicati ACKs, il TCP aggiorna la cwnd:

  cwnd := max{cwnd/2, MSS}

  (Fase di Decremento Moltiplicativo della finestra di congestione)

- In assenza di perdite e errori, ogni ACK (in genere) riconosce cumulativamente tutti i **cwnd** byte presenti nella finestra di congestione e trasmessi consecutivamente. Poiché:
  - l'intervallo di tempo che passa tra la trasmissione di (tutti) i byte presenti nella finestra e la ricezione al lato sorgente del corrispondente ACK è di RTT;
  - il numero di segmenti di dimensione massima presenti in una finestra di dimensione **cwnd** è pari a :

    cwnd/MSS

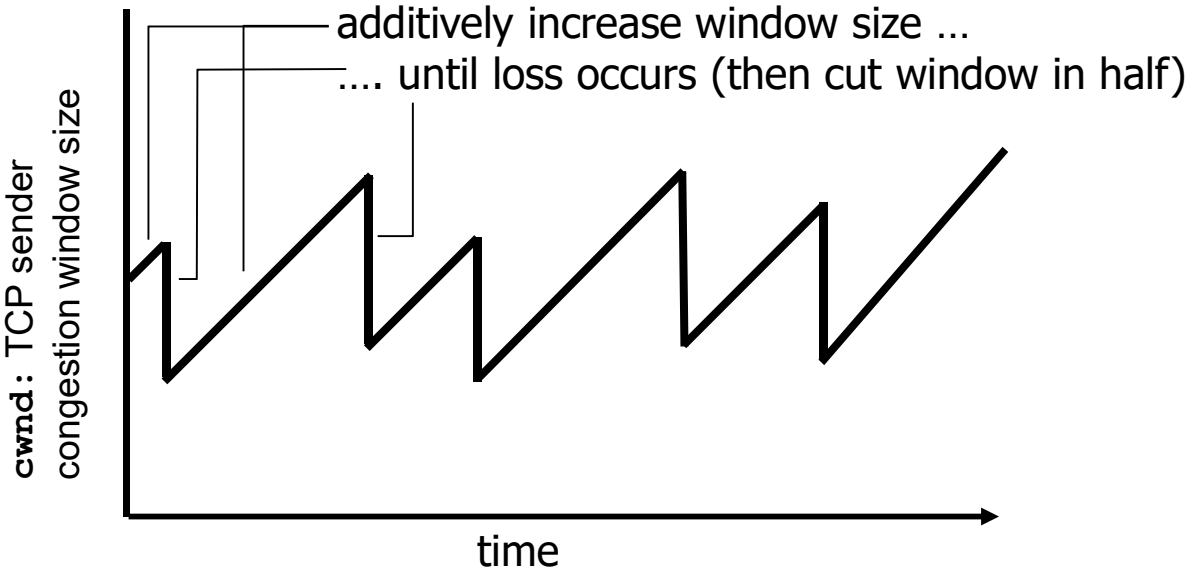
  ne segue che, nella fase di Incremento Additivo, la dimensione massima della finestra di congestione aumenta di 1 MSS (byte) ogni RTT (sec)

Congestion Avoidance (CA):
additive increase multiplicative decrease

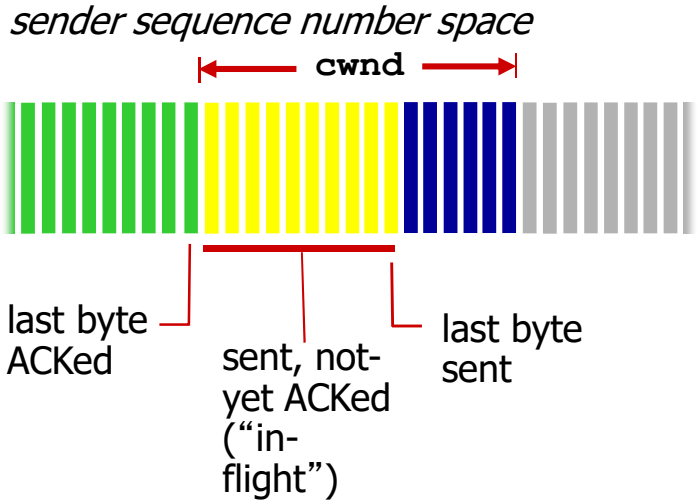
approccio: mittente incrementa il tasso di trasmissione (dimensione della finestra), "facendo" il probing della banda utilizzabile, fino a che non si verifica una perdita

- *additive increase:* incrementa **cwnd** du 1 MSS ogni RTT fino a che non si scopra una perdita (il TCP mittente)
- *multiplicative decrease:* divide **cwnd** in due dopo una perdita

AIMD tipico andamento a dente di sega (probing della banda)



Congestion Avoidance (CA): dettagli



Mittente limita il tasso di trasmissione:

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{cwnd}$$

~~**cwnd** è dinamica, ed è funzione della congestione~~ della rete (come percepita dal mittente)

Tasso di spedizione di TCP:

- *approssimativamente:* spedisce cwnd byte, attende RTT per ricevere gli ACK, e poi spedisce altri byte

$$\texttt{rate} \approx \frac{\texttt{cwnd}}{\texttt{RTT}} \text{ (bytes/sec)}$$

TCP: slow start

- L'algoritmo di controllo di congestione del TCP entra nello stato di Slow Start quando:
  - La connessione tra inizia;
  - Dopo che si è verificato un evento di Time Out
- Nello stato di SS, l'algoritmo di controllo di congestione:
  - **cwnd** è posto a 1 MSS;
  - **cwnd** è incrementato di 1MSS per ogni singolo segmento riscontrato positivamente per la prima volta, ossia:

cwnd := cwnd + MSS (byte)

Fase di Incremento Esponenziale della dimensione della finestra di congestione

- La Fase di Incremento Esponenziale continua sino a che si verifica uno dei tre seguenti eventi:
  - **cwnd** raggiunge un valore pari alla metà di quello che aveva prima di entrare dello stato di SS. In questo caso l'algoritmo abbandona lo stato di SS e entra nello stato di CA;
  - Si verifica un evento di Time Out. In questo caso l'algoritmo riinizia un nuovo stato di SS;
  - Si verifica l'evento Ricezione di 3 ACK duplicati. In questo caso l'algoritmo dimezza il valore corrente di **cwnd**, poi, entra nello stato CA

TCP: slow start (II)

- Per implementare l'algoritmo di aggiornamenti di **`cwnd`**, il TCP si usa di una variabile ausiliaria detta **`ssthresh`**
- Per definizione, **`ssthresh`** indica il valore di **`cwnd`** a cui lo stato di SS termina e inizialo stato di CA
- Ciò significa che **`ssthresh`** è aggiornato come segue
  - All'inizio della connessione, **`ssthresh`** è posto ad un valore infinito, così, da non avere inizialmente effetto sull'evoluzione dell'algoritmo di controllo di congestione
  - Ogni volta che si veridica un "evento di Time Out" oppure un "evento di ricezione di 3 ACK duplicati", **`ssthresh`** è posto alla metà del valore corrente di **`cwnd`**, ossia,

    `ssthresh` := **`cwnd`**/2

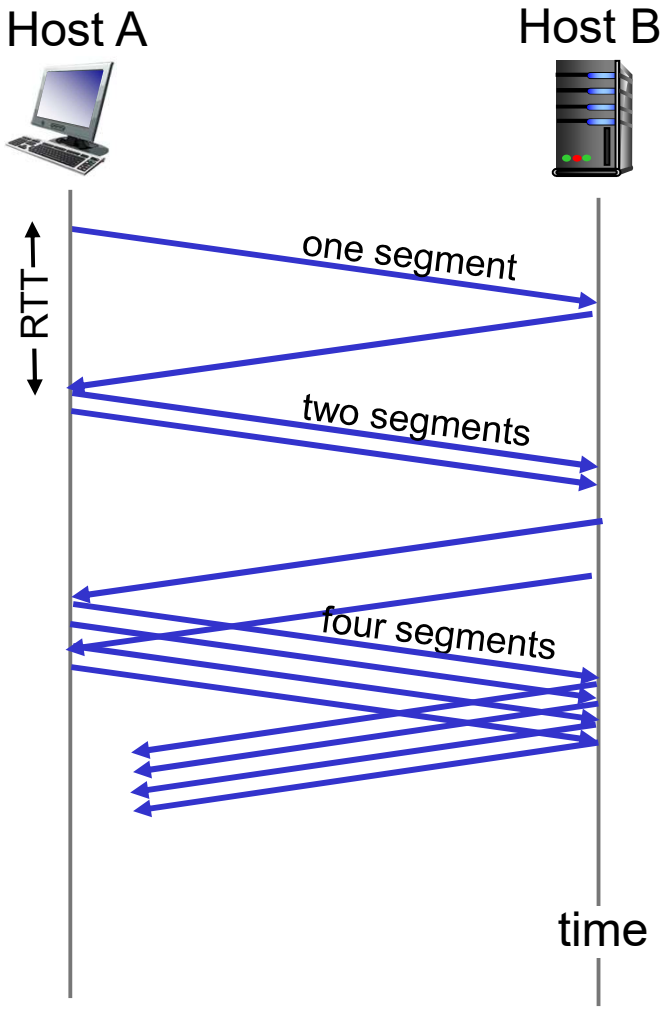
- Nella Fase di Incremento Esponenziale dello stato di SS, la dimensione **`cwnd`** della finestra di congestione aumenta di 1 MSS per ogni singolo segmento riscontrato positivamente per la prima volta:
  - In assenza di perdite e errori, il lato sorgente della connessione trasmette **`cwnd`** byte consecutivi
  - **`cwnd`** byte corrispondono ad un numero di segmenti di lunghezza massima pari a: **`cwnd/MSS`**
  - Dopo un intervallo di tempo pari a RTT, il lato sorgente riceve un ACK che riscontra tutti i **`cwnd`** byte precedentemente trasmessi

Nella fase di Incremento Esponenziale, la dimensione della finestra di congestione raddoppia di valore in ogni intervallo di durata RTT

TCP: slow start (III)

- Quando la connessione inizia, il tasso di spedizione viene incrementato in maniera esponenziale, fino all'occorrenza di una evento di perdita di un pacchetto:
  - inizialmente **`cwnd`** = 1 MSS
  - raddoppia **`cwnd`** every RTT
  - questo viene realizzato incrementando **`cwnd`** per ogni ACK ricevuto

Inizialmente il tasso di spedizione è basso ma cresce in modo esponenziale



L'algoritmo di controllo di congestione di TCP

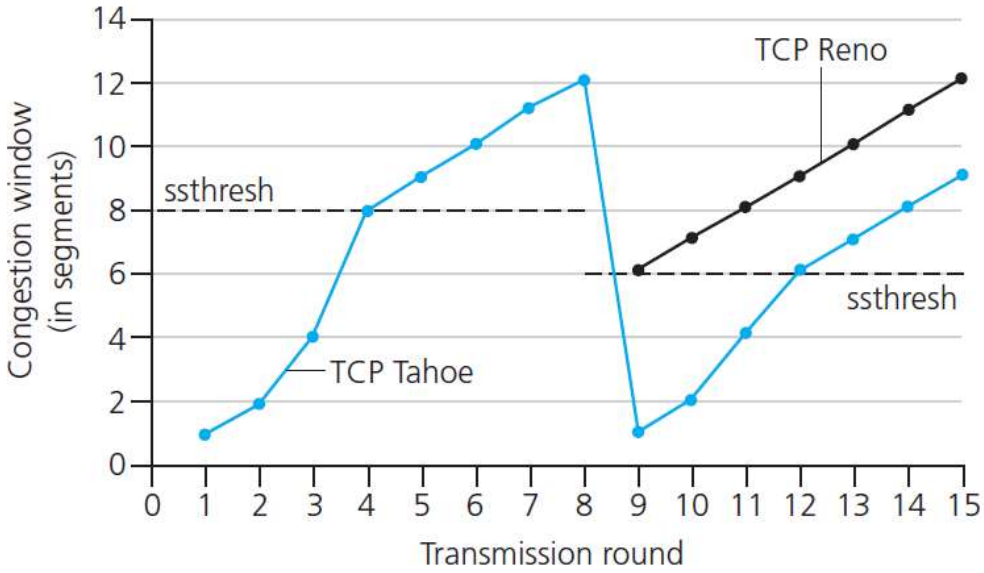
- L'algoritmo di controllo di congestione del TCP lavora al seguente modo:
- Fintantoché **`cwnd`** è minore o ugualea **`ssthresh`**, l'algoritmo è nello stato SS, in cui la dimensione della finestra raddoppia ogni RTT intervallo, finchè non si verificano eventi di perdita;
- Finchè **`cwnd`** è superiore a **`ssthresh`**, l'algoritmo è nello stato di CA, in cui la dimensione della finestra di congestione cresce di un MSS ogni RTT, fintantoché non si verificano eventi di perdita di segmenti
- Quando il lato sorgente della connessione riceve 3 ACK duplicati, allora:
  - Pone **ssthresh** a **cwnd**/2;
  - Pone **cwnd** = **ssthresh+3** (ossia dimezza **cwnd** e si aggiungono i 3 pacchetti che hanno originato i tre ACK duplicati);
- Quando si verifica un evento di Time Out, allora:
  - Pone **ssthresh** a **cwnd**/2;
  - Pone **cwnd** a 1 MSS e entra nello stato di SS

Questa è la versione di TCP Reno (1990). TCP Tahoe (1988) riporta sempre **cwnd** ad 1 (sia per timeout che per ACK duplicati)

TCP: dallo SS al CA

D: quando TCP dovrebbe passare dall'incremento esponenziale a quello lineare ?

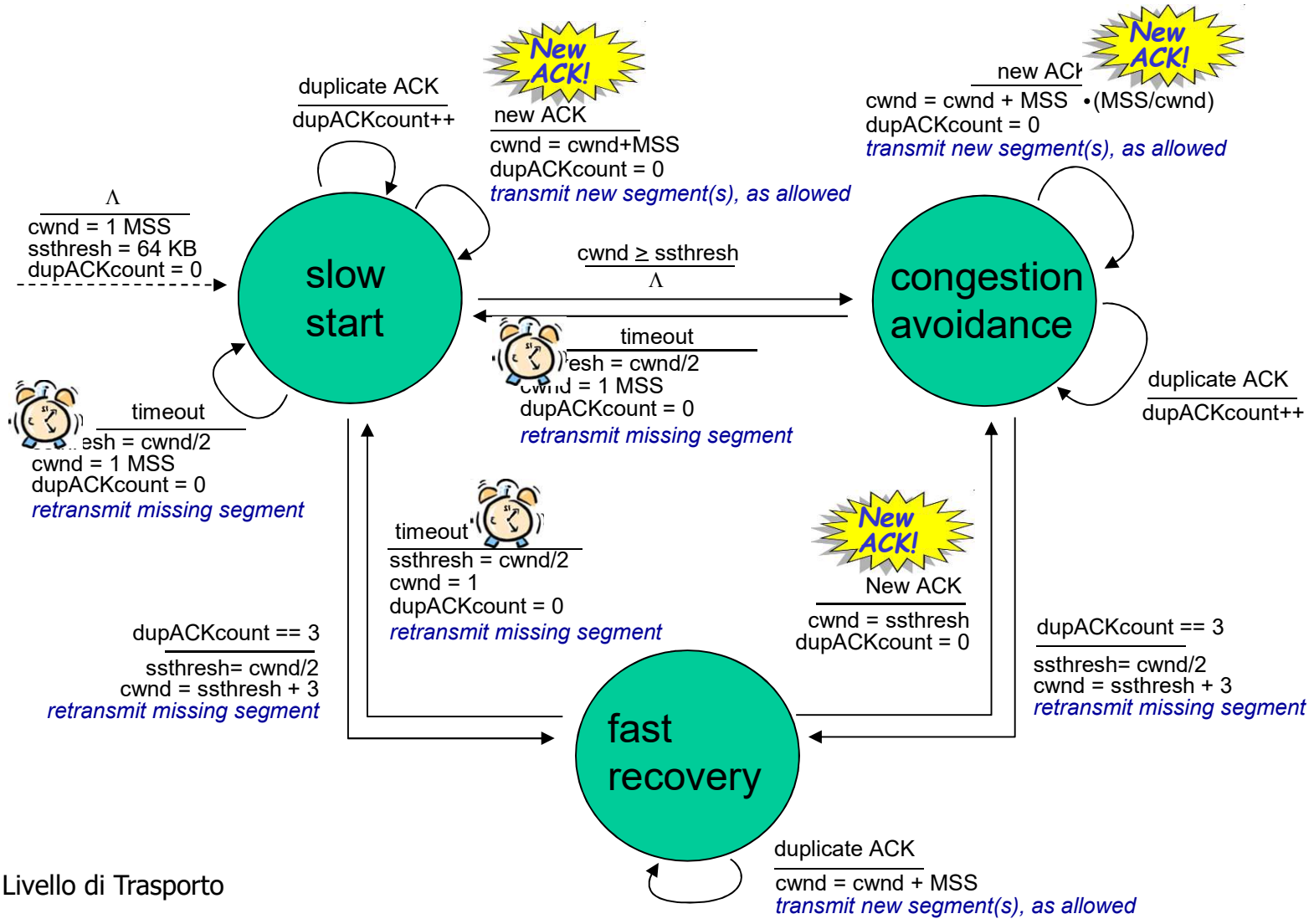
R: quando **`cwnd`** raggiunge la metà del valore che aveva prima del timeout.



Implementazione:

- variabile **`ssthresh`**
- In caso di perdita, a **`ssthresh`** viene assegnato la metà del valore che aveva **`cwnd`** prima dell'evento di perdita

Controllo di Congestione di TCP: FSM



Controllo di Congestione di TCP (II)

L'algoritmo di controllo di congestione di una connessione TCP-Reno

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment ACK cou duplicate nt for segment being acked	CongWin and Threshold not changed

Prestazioni di una connessione TCP

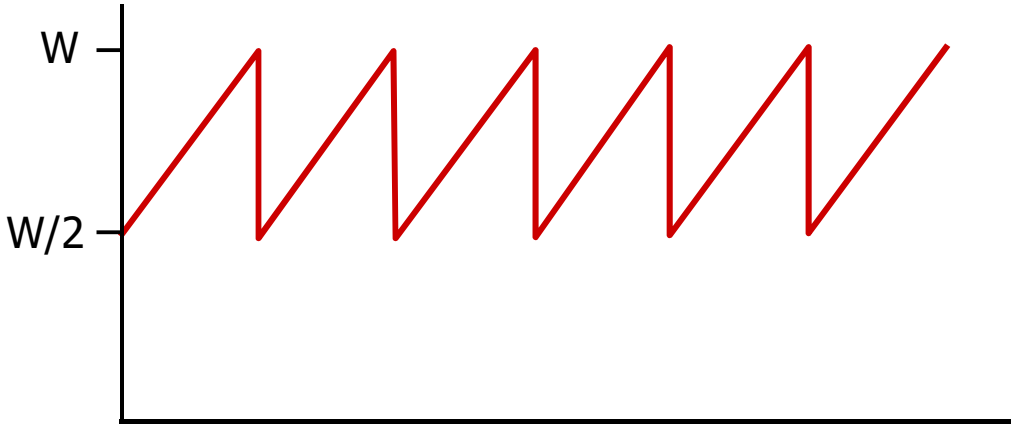
- Un mittente invia dei segmenti mediante una connessione TCP ad un destinatario
- I due parametri principali per misurare le prestazioni della connessione sono:
  - Il ***goodput medio*** (byte/sec)
  - Il ritardo di trasferimento o Latenza (sec)
- TCP ritrasmette i segmenti che sono andati persi o che sono stati ricevuti con errore, quindi in ogni istante, sulla connessione sono presenti o segmenti trasmessi per la prima volta oppure segmenti ritrasmessi

Prestazioni di una connessione TCP (II)

TCP goodput come functione della dimensione della finestra e del RTT ?

- ignoriamo lo slow start, ed assumiamo di avere sempre dei dati da trasmettere
- W: dimensione finestra (in bytes) quando si verifica una perdita

$$\overline{gd} \approx \frac{0.75 \cdot W}{RTT} \text{ (bytes/sec)}$$



Prestazioni di una connessione TCP (III)

- Si definisce ***goodput medio*** gd (byte/sec) della connessione il numero medio di byte che nell'unità di tempo, il processo destinazione riceve (ossia, non vi sono perdite) senza rivelarvi alcun errore
  - RTT (sec) il valore medio del RTT della connessione;
  - MSS (byte) la dimensione massima del campo payload di un segmento;
  - P_{LOSS} la probabilità di perdita (timeout o 3 duplicati ACK) di un segmento
- Allora, sotto ipotesi di larga generalità, il goodput medio della connessione è calcolabile mediante la seguente formula approssimata

$$\overline{gd} \approx \frac{1.22\ MSS}{RTT\ \sqrt{P_{LOSS}}} \text{ (bytes/sec)}$$

- La precedente formula vale
  - Per valori "piccoli" di P_{LOSS} (diciamo per $P_{LOSS} < 10^{-2}$);
  - Nei casi in cui gli Eventi di Time Out abbiano una probabilità trascurabile di verificarsi;
  - Quando l'algoritmo di controllo di congestione del TCP lavora essenzialmente nello stato di Congestion Avoidance (CA)

Prestazioni di una connessione TCP (IV)

Il futuro di TCP : TCP su "long, fat pipes"

- Esempio: vogliamo inviare dati a 10 Gbps attraverso una connessione TCP con segmenti da 1500 byte segments, RTT da 100 ms
- Usando la formula si ottiene W = 83,333 (in-flight segments)
- E cosa succedel per la probabilità di perdita ? [Mathis 1997]:
- Per raggiungere 10 Gbps la probabilità di perdita $P_{LOSS}=2 \cdot 10^{-10}$
  una probabilità di perdita molto molto piccola!

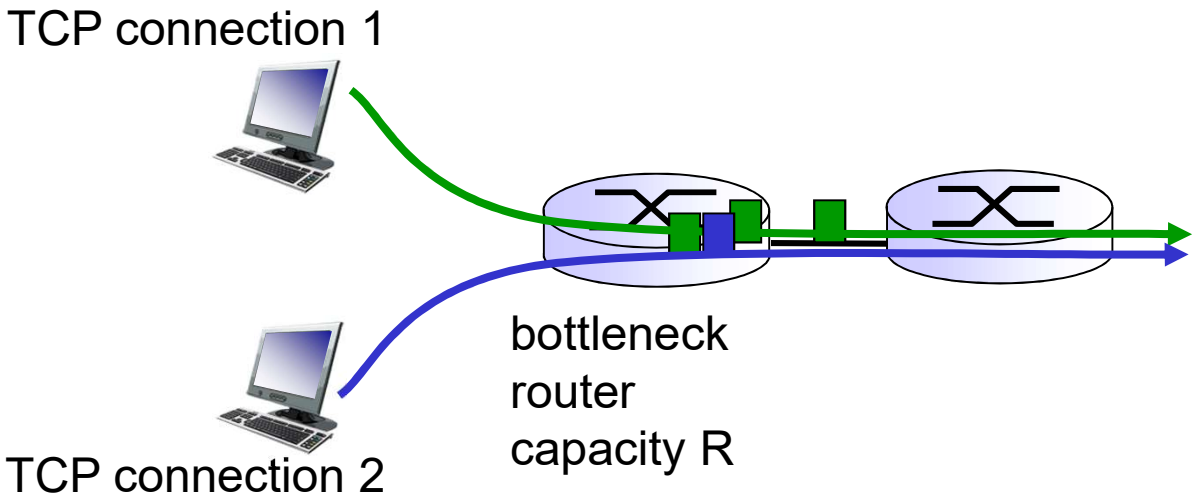
$$\overline{gd} \approx \frac{0.75 \cdot W}{RTT}$$

$$\overline{gd} \approx \frac{1.22\ MSS}{RTT\ \sqrt{P_{LOSS}}}$$

- Nuove versioni di TCP per gestire queste velocità/scenari

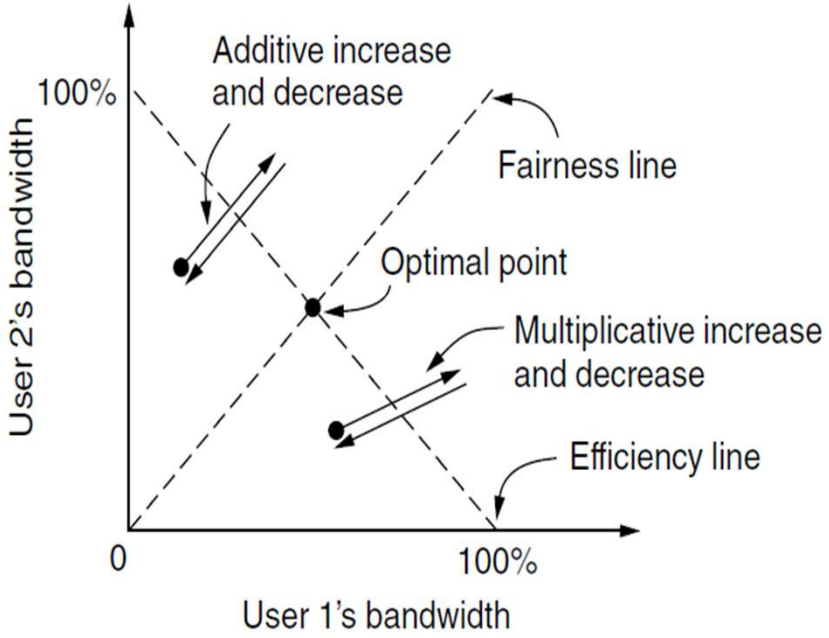
TCP e la fairness

Lo scopo della fairness: se abbiamo K sessioni TCP che condividonlo lo stesso link bottelneck, ognuna di queste connessessioni avrà un tasso medio di R/K



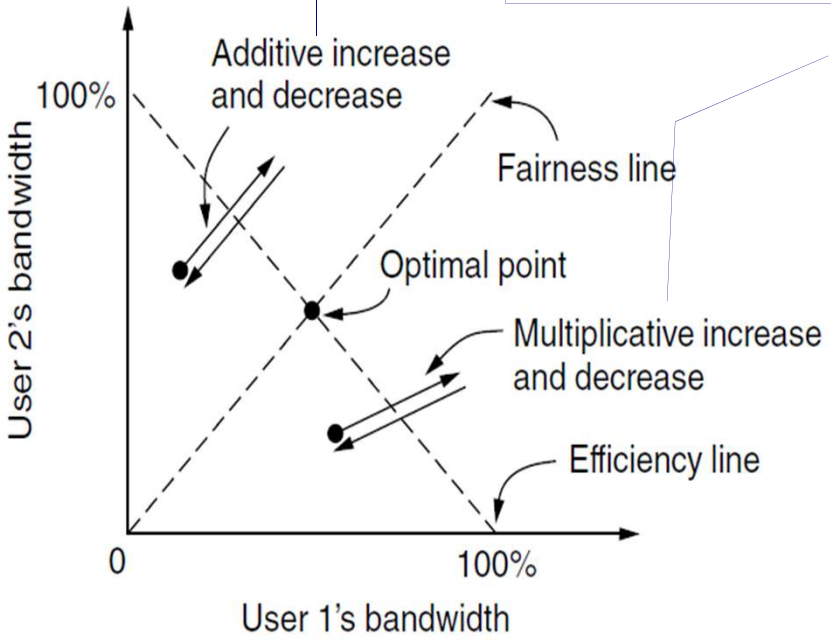
TCP e la fairness (II)

- Legge di controllo AIMD (additive increase multiplicative decrease) è una proposta di legge di controllo appropriata per arrivare ad un punto di funzionamento equo ed efficiente
- Grafico che mostra la banda allocata a due connessioni (assi *x* e *y*)
- Allocazione equa ½ e ½
- Quando la somma delle allocazioni arriva al 100% l'allocazione è efficiente. La rete invia un segnale di congestione alle due entità di trasporto se la somma delle loro allocazioni supera questa soglia (linea tratteggiata)
- L'intersezione delle due linee tratteggiate rappresenta il punto di funzionamento ottimale (fairness + efficienza)



TCP e la fairness (III)

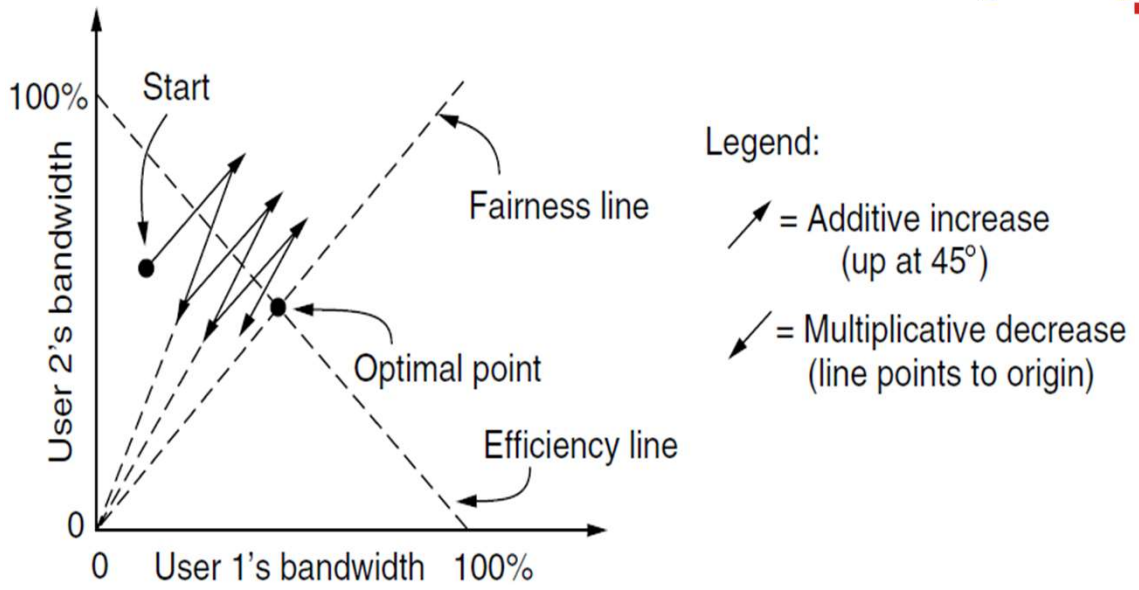
Incremento/decremento additivi:
i due utenti potrebbero additivamente aumentare nel tempo il loro uso delle risorse (es. aumentare il tasso di spedizione di 1 Mbps ogni secondo). Ad un certo punto il funzionamento oltrepassa la linea di efficienza e gli utenti riceveranno un segnale di congestione dalla rete. A questo punto dovranno ridurre le allocazioni. Una diminuzione additiva li porterà solo ad oscillare lungo una linea additiva



Incremento/decremento moltiplicativi:
i due utenti potrebbero aumentare moltiplicativamente nel tempo il loro uso delle risorse (es. aumentare il tasso di spedizione del 10% ogni secondo). Ad un certo punto il funzionamento oltrepassa la linea di efficienza e gli utenti riceveranno un segnale di congestione dalla rete. A questo punto dovranno ridurre le allocazioni. Una diminuzione additiva li porterà solo ad oscillare lungo una linea moltiplicativa

La linea moltiplicativa ha una pendenza diversa

TCP e la fairness (IV)



Nel caso in cui gli utenti aumentino in modo additivo e decrementano in modo moltiplicativo (quando superano il 100% della banda) si ottiene un andamento simile a quello del diagramma. Si osserva una convergenza verso il punto ottimale (e questo è indipendente dal punto di partenza)

AIMD è la legge di controllo usata da TCP oltre alle caratteristiche evidenziate dai diagrammi si basa su anche su altre considerazioni

- è facile portare la rete in una situazione di congestione ma è più difficile riportarla ad un funzionamento normale
- La politica di incremento deve essere *gentile* mentre la politica di decremento deve essere *aggressiva*

TCP e la fairness (V)

Fairness e UDP

- Le applicazioni multimediali spessno non utilizzando TCP
  - Non vogliono essere 'penalizzate' da meccanismi di strozzamento della banda duvuto al controllo di congestione
- Usano UDP:
  - Spediscono audio/video a tasso costante e possono tolletare delle perdite di pacchetti

Fairness e connessioni TCP parallele

- Le applicazioni posso aprire connessioni multiple tra due host
- I web browser lo fanno
  - Es. link con tasso R con 9 connessioni pre-esistenti:
  - Nuova applicazione richiede una connessione TCP, allora otterrà un tasso R/10,
  - ma se invece la nuova applicazione (sa le connessioni parallele) usa 11 connessini otterrà un'allocazione un-fair di R/2

Il livello trasporto: sommario

Illustrazione dei principi che sono alla base del livello trasporto :

- multiplexing, demultiplexing
- Trasferimento dati affidabile
- Controllo di flusso
- Controllo della congestione

Il livello trasporto di Internet: un caso di studio interessante

- UDP
- TCP