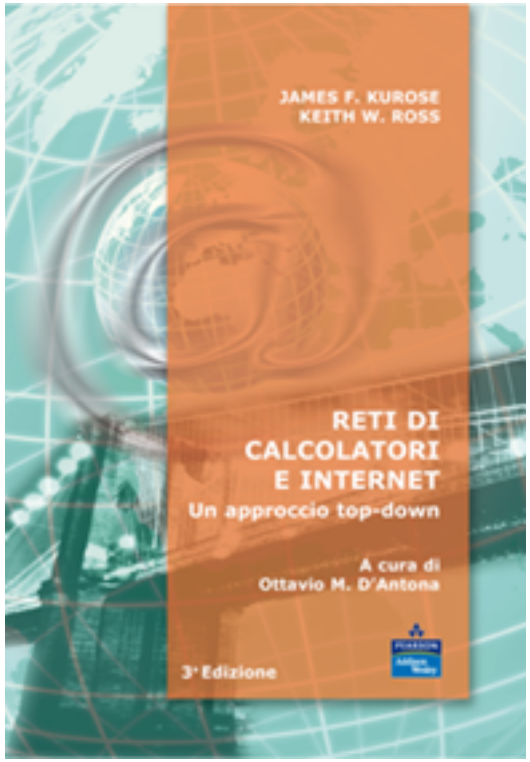
Reti di calcolatori: Livello Trasporto

(Capitolo 3 Kurose-Ross)

Marco Roccetti

13/20 Marzo 2024

(Capitolo 3 Kurose-Ross) seconda parte



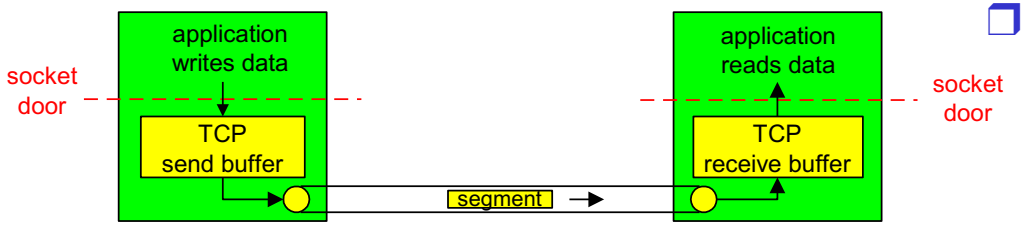
Reti di calcolatori e Internet: Un approccio top-down

3ª edizione
Jim Kurose, Keith Ross
Pearson Education Italia
©2005

TCP: rassegna

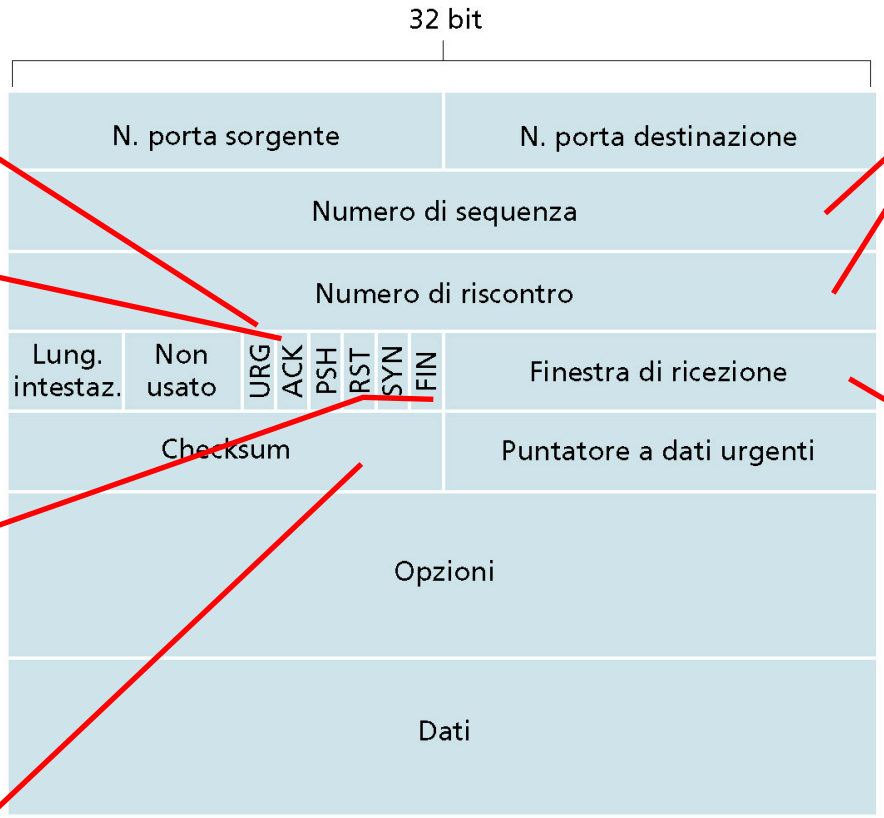
RFCs: 793, 1122, 1323, 2018, 2581

- Protocollo uno-a-uno:
  - Un sender, un receiver
- Flusso di Byte ordinato e affidabile
- Protocollo pipelined:
  - TCP ha controllo di flusso e di congestione basato su finestra scorrevole
  - Il protocollo è eseguito solo sui nodi terminali
- *Buffers su sender e receiver*



- Connessioni full-duplex:
  - Dati viaggiano nelle due direzioni
  - MSS: maximum segment size, 1460, 536, 512 bytes pari al MTU della rete sotto
- Orientato alla connessione:
  - Messaggi di controllo iniziali definiscono lo stato di sender e receiver prima di inviare i dati
- Controllo di flusso
  - Sender segue il ritmo del receiver
- Controllo di congestione
  - Sender segue il ritmo del router più lento

Struttura del segmento TCP



URG: dati urgenti (di solito non usato)

ACK: ACK # valid

RST, SYN, FIN: Comandi di setup della connessione

Internet checksum (come in UDP)

Contatore dei byte di dati del flusso (non segmenti)

Numero di byte che il ricevente può ricevere (buffer)

In tutto 20 bytes di solito

TCP: numeri di sequenza e ACKs

Il numero di sequenza per un segmento e' il numero nel flusso di byte del primo byte del segmento:

Es. Se ho un flusso da 500.000 byte e un MSS di 1000, ottengo 500 segmenti con il primo numerato 0, il secondo 1000, il terzo 2000 ….

Il numero di riscontro che l'host A scrive nei propri segmenti e' il numero di sequenza del byte successivo che A attende dall'host B:

Es. Se A ha ricevuto da B segmento di un flusso con dati da 0 a 535, A scrive 536 come numero di riscontro del segmento che manda a B

TCP: numeri di sequenza e ACKs

Num. Seq. (#seq):

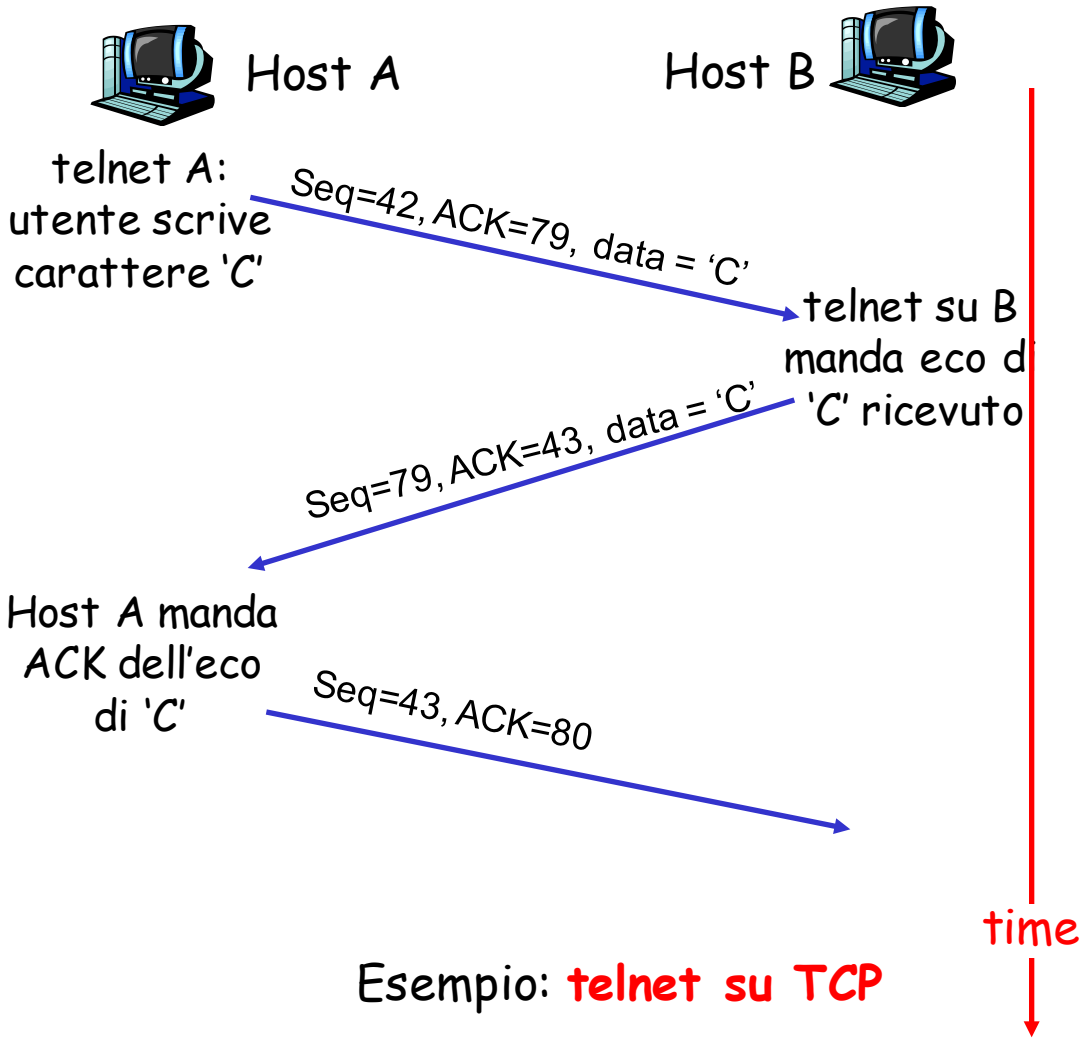
- Numero d'ordine del byte nella sequenza

ACKs:

- Numero d'ordine del prossimo byte atteso
- ACK cumulativo

??: come si gestiscono segmenti fuori ordine?

- TCP non lo specifica! Dipende...

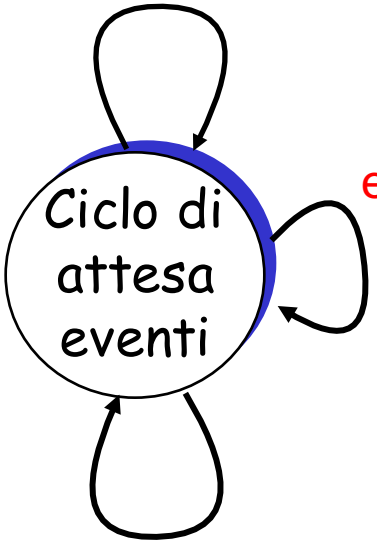


Esempio: **telnet su TCP**

TCP: reliable data transfer: 3 eventi principali

evento: dati ricevuti dall'applicazione sopra TCP

Crea e spedisci segmento



evento: scatta il timer per il segmento numero # y

ritrasmetti segmento

evento: ricevuto ACK per segmento # y

Elabora ACK

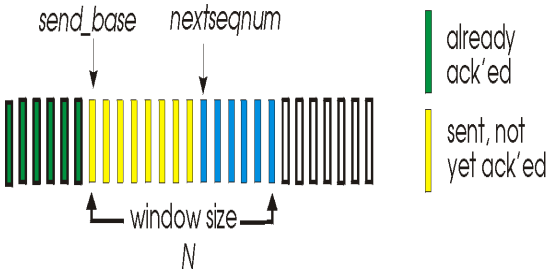
Assumiamo per semplicità che il Sender

- Spedisca dati solo in un senso
- Non abbia per ora controllo di flusso e congestione

TCP: trasferimento dati affidabile

Es. Sender TCP semplificato

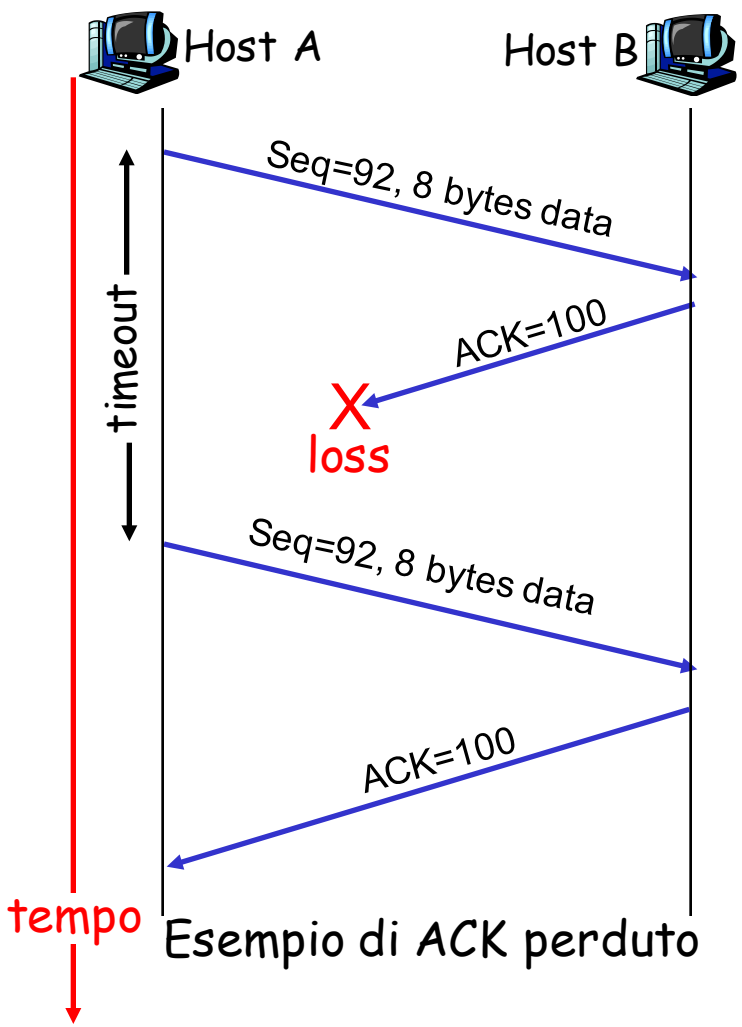
```
00  sendbase = initial_sequence number (inizio finestra scorrevole)
01   nextseqnum = initial_sequence number (puntatore alla parte libera)
02
03    loop (ciclo infinito) {
04      A seconda del tipo di evento esegui:
05      evento: ricevuti dati da spedire dall'applicazione
06            crea segmento TCP con numero nextseqnum
07            Avvia timer per il segmento nextseqnum
08            Spedisci segmento mediante servizio di livello rete (IP)
09            nextseqnum = nextseqnum + length(data)  (aggiorna finestra)
10       evento: scatta timer per segmento numero y
11            Ritrasmetti segmento numero y
12            Calcola nuovo timeout per il timer del segmento y
13            Riavvia il timer per il segmento y
14       evento: ricevuto ACK per il segmento numero y
15            se (y > sendbase) { /* ACK cumulativo per tutti i byte fino a y */
16               Cancella tutti i timer per i segmenti con numero < y
17                sendbase = y (aggiorna il lato di inizio della finestra)
18                }
19            altrimenti { /* ACK duplicato per segmento già ricevuto */
20                Incrementa contatore di ACK duplicati del segmento y
                  se (questo è il terzo ACK duplicato per segmento y) {
                     /* attiva fase "fast retransmit" di TCP*/
                    Rispedisci segmento con numero y
                    Riavvia timer per segmento y
                  }
          } /* fine ciclo infinito */
```



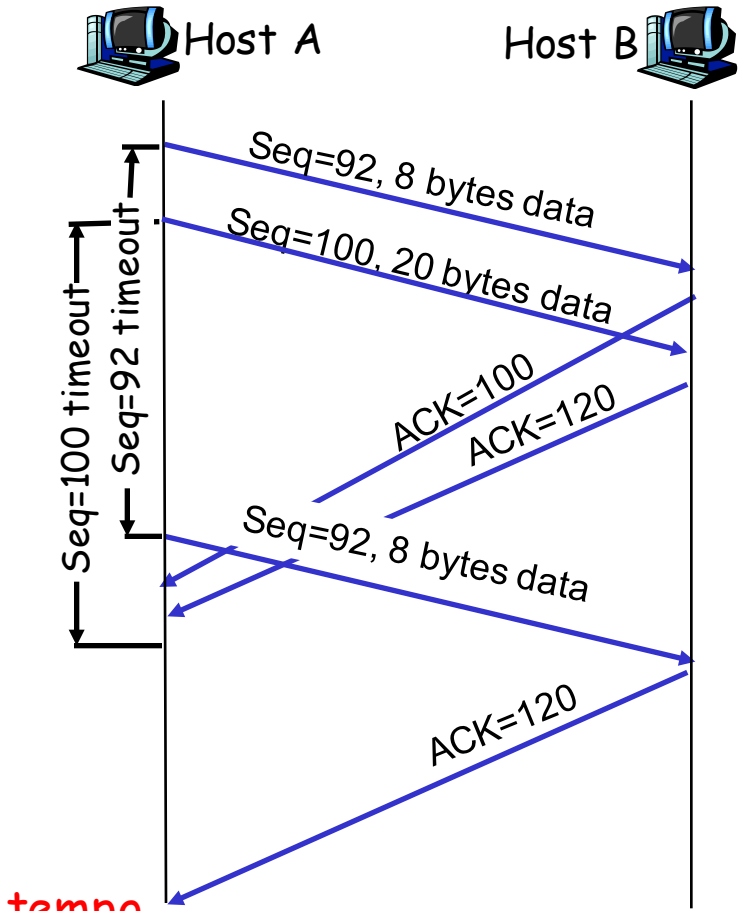
TCP: regole per ACK [RFC 1122, RFC 2581]

Evento	Azione del receiver TCP
Segmento arriva in ordine Non ci sono “buchi”	ACK ritardato. Aspetta per 500ms l’arrivo anche del prossimo segmento. Se non arriva entro 500ms spedisci ACK
Segmento arriva in ordine Non ci sono “buchi” Ho già un ACK ritardato...	Spedisci subito un ACK cumulativo per i due segmenti ricevuti
Segmento arriva fuori ordine con numero superiore. Si crea un “buco”.	Spedisci un ACK duplicato che indica di nuovo quale è il numero del prossimo byte atteso
Arriva segmento che copre del tutto “o parzialmente l’inizio di’’ Un “buco”	Spedisci ACK immediato se il segmento cade all’inizio del “buco”

TCP: esempi



Esempio di ACK perduto



Timeout prematuro, ACK cumulativo

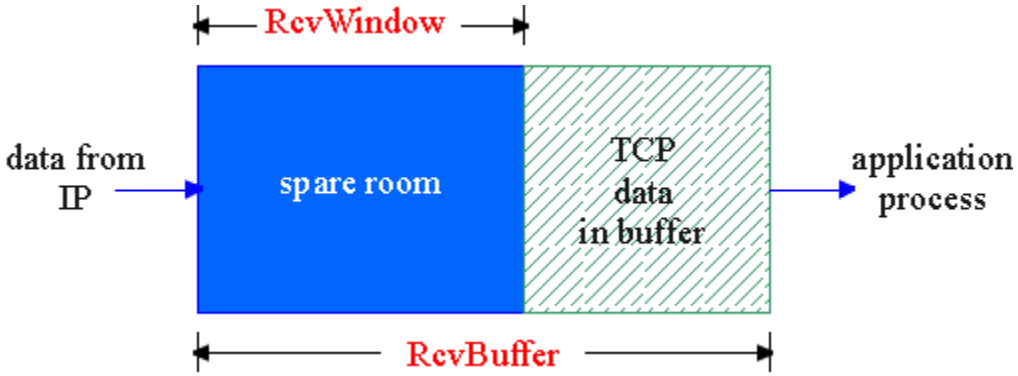
TCP controllo di flusso

controllo di flusso

Il sender evita di trasmettere dati troppo velocemente rispetto al buffer del receiver

`RcvBuffer` = dimensione del Buffer TCP del receiver

`RcvWindow` = dimensione del Buffer libero sul receiver



receiver buffer

receiver: informa esplicitamente il sender sulla quantità residua di buffer disponibile (parte azzurra) della figura: non me ne mandare piu' di questo!!!

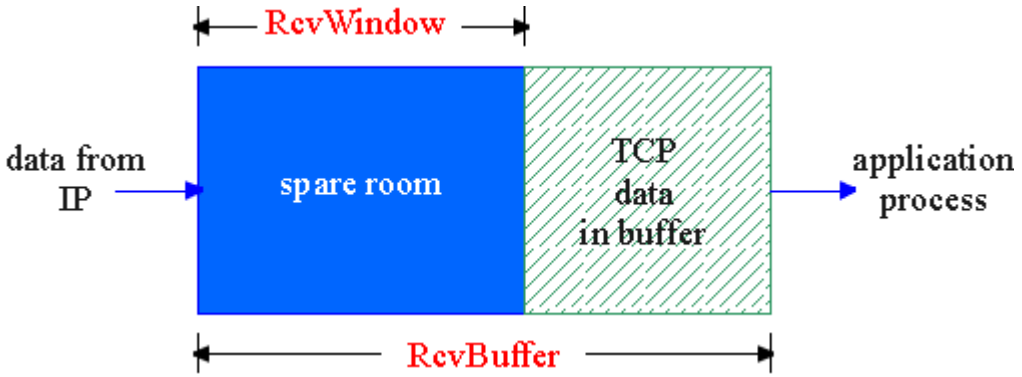
- **Questa informazione va scritta nel campo `RcvWindow`** del segmento TCP

TCP controllo di flusso: receiver

receiver: LastByteReceived – LastByteRead <= RcvBuffer (parte azzurra)

nel campo `RcvWindow` del segmento TCP devo dunque scrivere il valore:

RcvWindow = RcvBuffer – (LastByteReceived – LastByteRead)



receiver buffer

TCP controllo di flusso: sender

Anche il sender usa due variabili per gestire il proprio buffer:
LastByteSent e LastByteAcked

La differenza (LastByteSent – LastByteAcked) corrisponde a bytes che devono essere conservati nel buffer perche' potrebbero necessitare di ritrasmissione

Cioe' fa si' che: (LastByteSent – LastByteAcked) <= RcvWindow

Ovvero il sender: limita il numero di segmenti per i quali non ha ricevuto ACK inferiore all'ultimo **`RcvWindow` ricevuto**

TCP controllo di flusso: problema

Supponiamo receiver abbia comunicato a sender RCvWindow = 0

Il sender si ferma e non manda piu' nulla

Come fa il receiver a comunicare che RcvWindow non e' piu' a 0 se non puo' rispondere a segmenti del sender che s'e' fatto muto per rispettare le regole?

Per ovviare al problema, il sender manda comunque segmenti di dati nulli (anche se RcvWindow = 0) per dare modo al receiver di rispondere ai segmenti con gli eventuali nuovi valori (diversi da 0) della RcvWindow

TCP Round Trip Time e Timeout

D: come si definisce il valore del timeout?

- Maggiore di RTT
  - ma: RTT può variare
- Troppo corto: timeout prematuro
  - Implica ritrasmissione inutile
- Troppo lungo: reazione lenta in caso di perdita di segmento

D: come stimare RTT?

- **SampleRTT**: tempo misurato dalla trasmissione del segmento alla ricezione dell'ACK
  - Si ignorano le ritrasmissioni
  - Valgono gli ACK cumulativi (uno solo alla volta in un dato RTT)
- **SampleRTT** può variare rapidamente: occorre un modo per rendere "morbida" la variazione
  - Si usa una media pesata delle ultime stime di **SampleRTT** sperimentate

TCP Round Trip Time and Timeout

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

- Medie mobili pesate (esponenziali)
- L'effetto di una singola stima decade nel tempo esponenzialmente
- Valore tipico di x: 0.125 (1/8)

Come si definisce quindi il timeout?

- **EstimtedRTT** al quale si aggiunge un margine di sicurezza
- se **EstimatedRTT varia molto** -> occorre un margine di sicurezza superiore

```
Timeout = EstimatedRTT + 4*Deviation

Deviation = (1-y)*Deviation +
            y*|SampleRTT-EstimatedRTT|
con y=0.25 di solito
```

TCP: instaurazione di una connessione

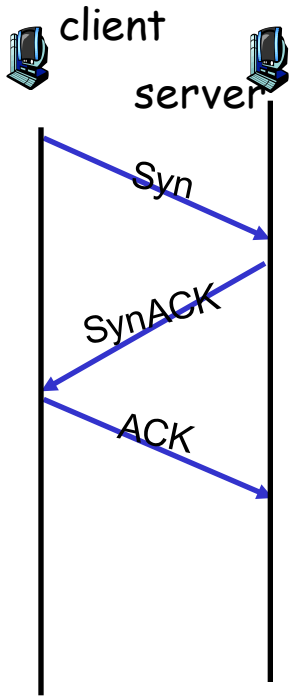
N.B.: TCP sender e receiver devono stabilire una connessione prima di scambiare segmenti

- Necessario per inizializzare le variabili di TCP:
  - Numeri di sequenza #s
  - Buffer e controllo di flusso (es. **RcvWindow**)
- *client:* chi inizia la connessione

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

- *server:* chi riceve la richiesta

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



Handshake a 3 vie:

passo 1: client spedisce segmento TCP con SYN=1 al server e specificando un num.seq. iniziale scelto a caso

passo 2: appena il server riceve SYN, risponde con SYN/ACK

- Conferma ricezione con suo SYN a 1
- mette ACK= num.seq(r) +1 del client
- Alloca spazio per buffer
- Sceglie casualmente il suo seq.num.

passo 3: client riscontra il segmento di conferma del server, alloca il suo buffer, mette il suo ACK = num. seq(s) +1, e mette SYN =0 perche' l'handshake e' terminato

TCP: gestione della connessione

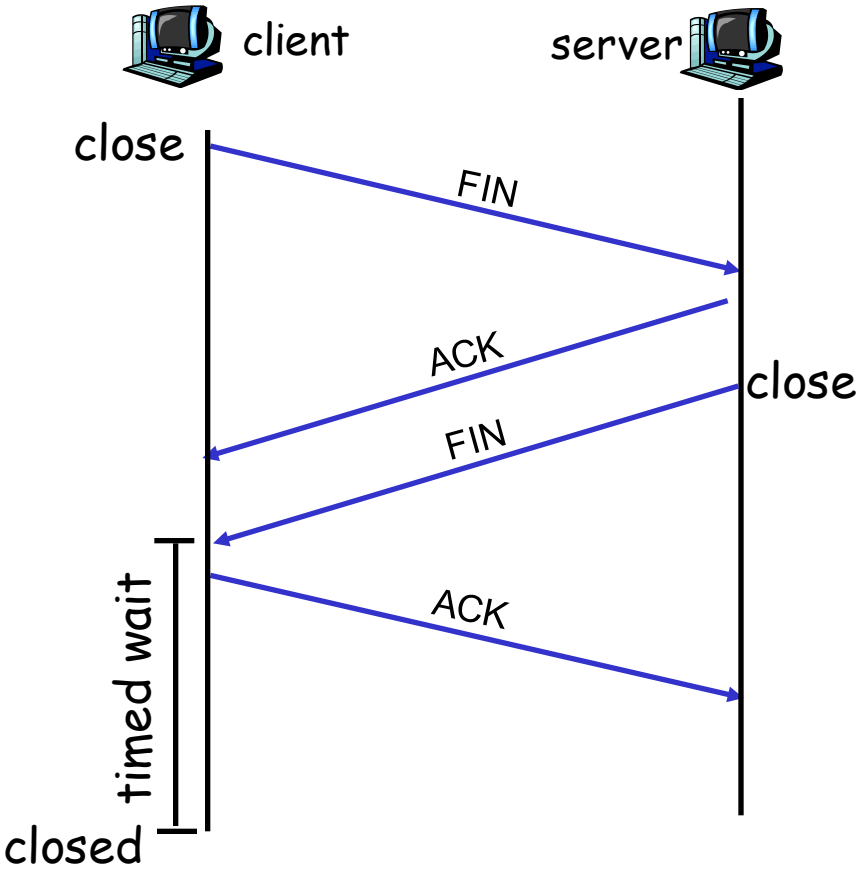
Chiusura della connessione:

client closes socket:

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

Passo 1: client spedisce segmento TCP FIN al server

passo 2: server riceve FIN e conferma con ACK, poi chiude la connessione e spedisce segmento FIN a sua volta.

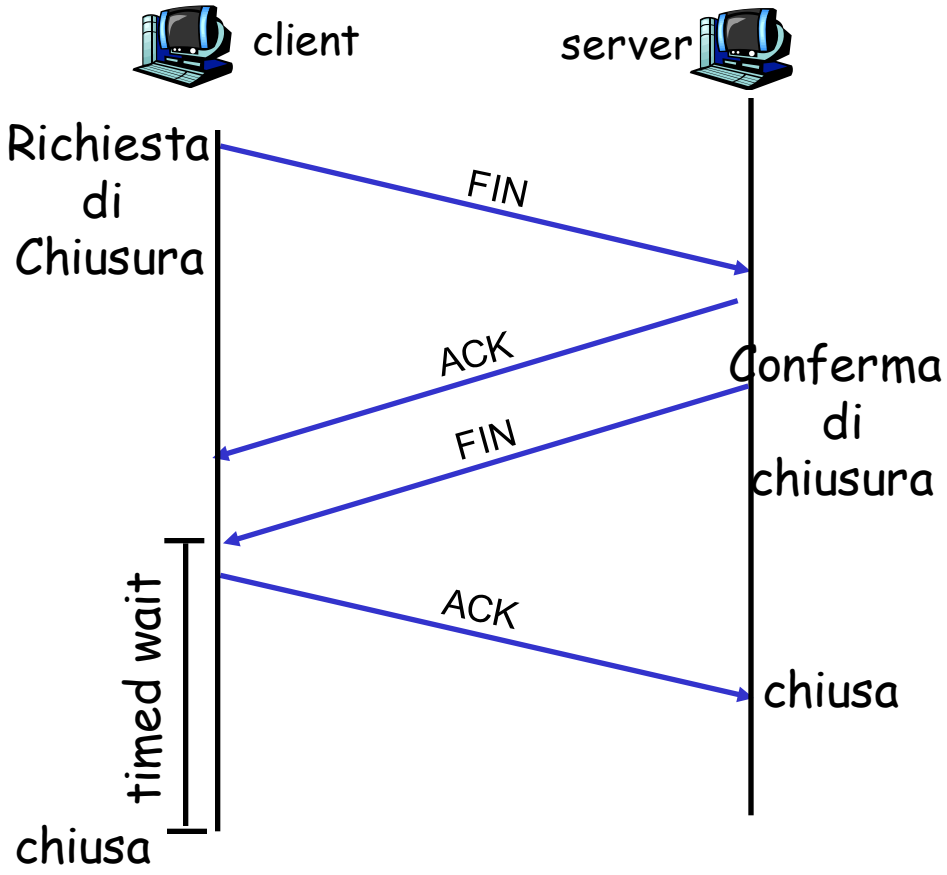


TCP: gestione della connessione

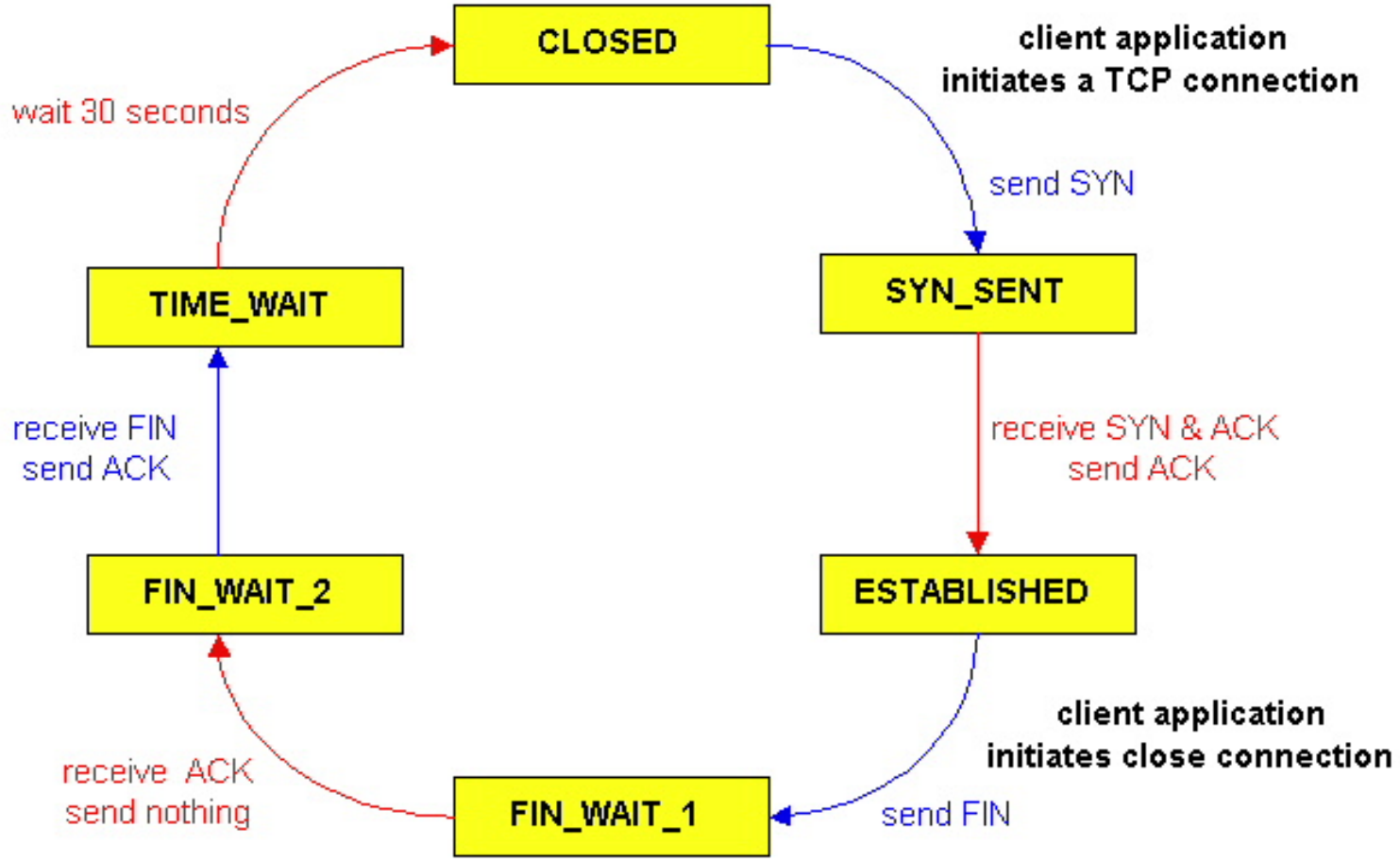
passo 3: client riceve FIN e risponde con ACK.

- ...poi entra in fase di attesa limitata nella quale risponde con ACK a eventuali FIN replicati

passo 4: server, riceve ACK e la connessione è chiusa.



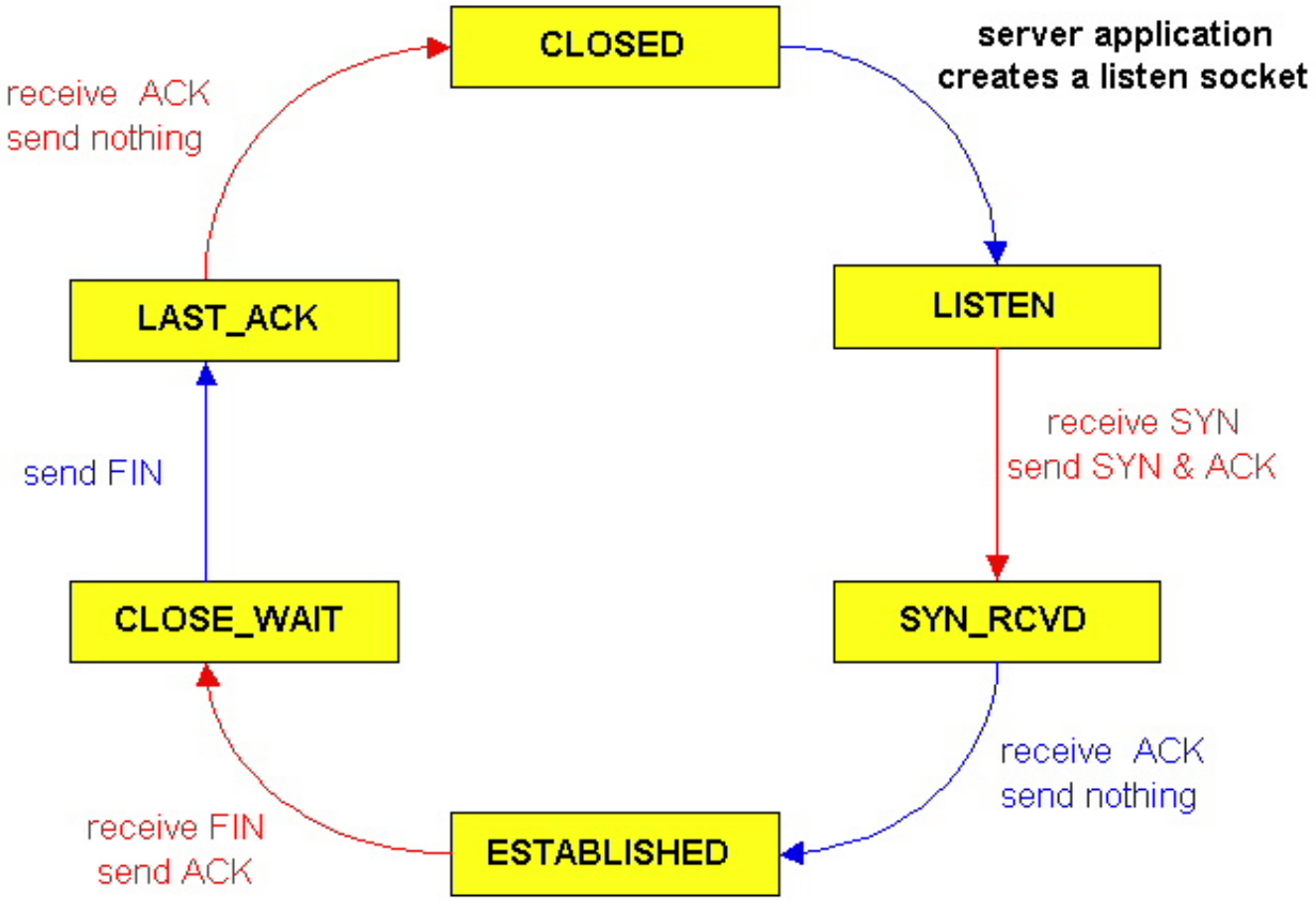
TCP: gestione della connessione



Ciclo di vita degli stati del client TCP

TCP: gestione della connessione

Ciclo di vita degli stati del server TCP



Il controllo della congestione

Congestione:

- Troppi host spediscono troppi dati e la rete non è in grado di inoltrarli tutti verso le destinazioni
- È un problema dei router intermedi del cammino (e non solo del receiver, come per il controllo di flusso)
- Cosa causa:
  - Pacchetti perduti (buffer dei router saturi)
  - Lunghi ritardi (lunghe code nei buffer)

Come si realizza il controllo di congestione

Due approcci possibili:

Controllo di congestione End-end:

Non c'è indicazione esplicita dalla rete

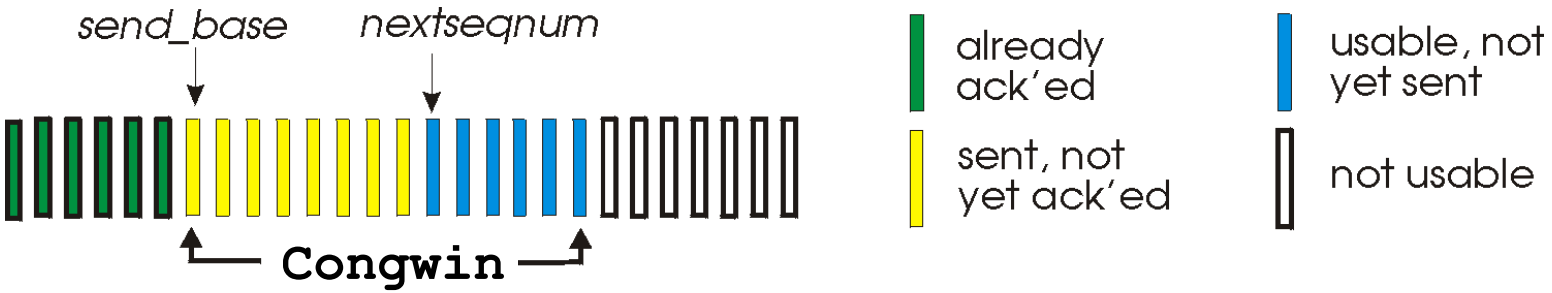
- La congestione si interpreta sulla base dei pacchetti perduti e dei ritardi stimati
- TCP usa questa tecnica

Controllo di congestione assistito dalla rete

- I router forniscono segnali di rischio
  - Un bit nell'header indica presenza o rischio di congestione (explicit congestion notification) TCP/IP ECN
  - Si indica anche il ritmo di invio che il sender dovrebbe assumere

TCP: controllo della congestione

- Controllo end-end (non assistito dai router intermedi della rete)
- Il ritmo di invio è limitato dalla dimensione della finestra di congestione **Congwin**:



- Dati w segmenti, di MSS bytes spediti ogni RTT, posso raggiungere un throughput massimo di:

$$\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

TCP: controllo della congestione e del flusso

- Algoritmi per il controllo della congestione e del flusso vanno armonizzati
  - idea: si usa il minimo tra I valori di **Congwin (congestione)** e di RcvWindow (flusso)evitando perdita segmenti
  - Cio' e' fatto dal sender nell'inviare segmenti al receiver nella seguente maniera:

LastByteSent – LastByteAcked <= min{Congwin, RcvWindow)

TCP: controllo della congestione

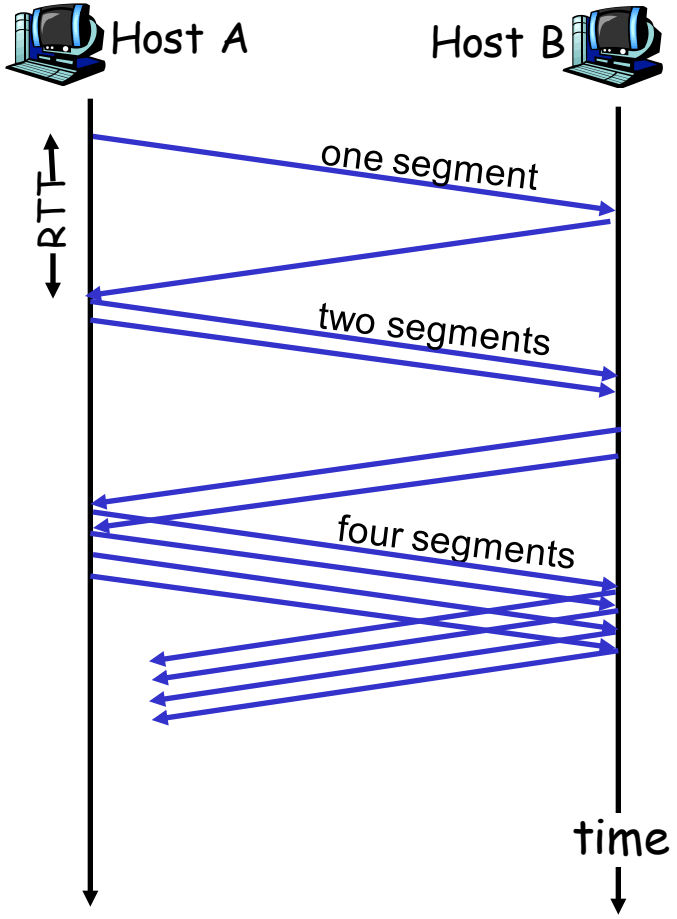
- "probing" della banda disponibile
  - idea: inviare al ritmo massimo (**Congwin** più grande possibile) evitando perdita segmenti
  - *incrementa* **Congwin** finchè non c'è perdita
  - *decrementa* **Congwin** se c'è perdita e poi riparti con la fase di incremento
- Due fasi
  - slow start
  - congestion avoidance
- variabili usate:
  - **Congwin (dimensione della finestra)**
  - **threshold**: definisce la soglia di dimensione della finestra oltre la quale termina la fase "slow start" e inizia "congestion avoidance"

TCP Slow(!?)start

Algoritmo Slowstart

inizializza: Congwin = 1
Per ogni ACK ricevuto
    Congwin*2
finchè (ACK non ricevuto
oppure
    CongWin > threshold)

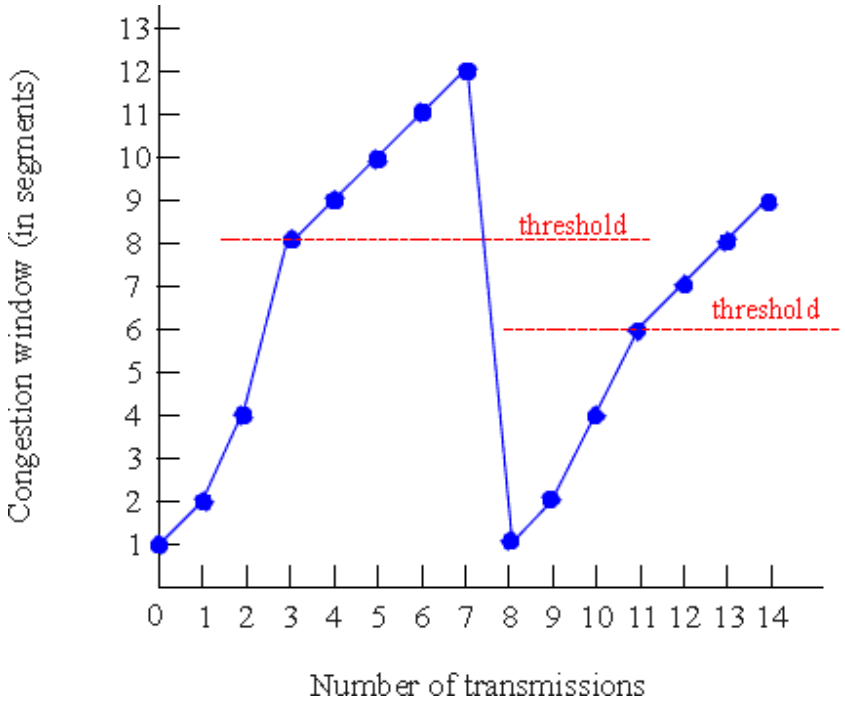
- Crescita esponenziale (per RTT) della finestra (quindi non troppo LENTA!! Aumenta di 1 MSS ad ogni ACK all'interno di un RTT)



TCP "Congestion Avoidance"

Congestion avoidance

```
/* slowstart terminata        */
/* Congwin > threshold */
finchè (non perde segm.) {
  ogni w ACK ricevuti:
      Congwin++ (cioe'
aumenta di 1 MSS)
  }
/* se perde ACKs (timeout)
*/
threshold = Congwin/2
Congwin = 1
Riparti da fase "slowstart"
```



TCP "Congestion Avoidance" con 3 ACK duplicati -> fast recovery

Congestion avoidance

2 alternative:

/* se perde ACKs (timeout) */
threshold = Congwin/2
Congwin = 1

/* se perde ACKs (3 ack duplicati)*/
threshold = Congwin/2
Congwin = threshold
Non ri-inizia da 1MSS, i 3 ack duplicati dicono che rete puo' fare di piu'

AIMD

La fase "congestion avoidance" di TCP si dice AIMD: *additive increase, multiplicative decrease*

- Xchè incrementa finestra di 1 per RTT
  - In realta' ad ogni ACK entro RTT aumento Congwin con la seguente formula: MSS x (MSS/Congwin) byte
  - Per esempio se MSS=1460 byte e Congwin 14600 byte in un RTT sono spediti 10 segmenti. Ad ogni ACK per ciascuno di questi aumento la dimensione di Congwin e quindi la spedizione di un 1/10. Se mi arrivano tutto, ho incrementato di un intero MSS la spedizione.
- Divide la finestra per 2 in caso di perdita segmento

throughput TCP : modello semplificato(!!)

Escludiamo le fasi di slow-start: percentualmente durano poco e poco pesano

Detta w in bytes la finestra di congestione all'interno di un RTT, la frequenza trasmissiva e': w/RTT

A ogni RTT, si aumenta di 1 MSS fino al primo timeout, diciamo W il valore della finestra w al timeout, supponiamo W e RTT costanti, allora lo throughput in RTT varia linearmente in: [W/(2RTT), W/RTT], cresco di 1 fino a W e poi dimezzo al timeout (TCP Reno)

Ovvero mediamente e': TCP-throughput = 0.75xW/RTT

throughput TCP : modello meno semplificato e futuro

Col modello semplificato di prima, se ho connessione TCP a larga banda (10 Gbps) con MSS di 1500 byte, RTT=100 msec, dovrei avere una finestra media di dimensione pari a: 83333 segmenti

Sono tantissimi, che succede se li perdo? Come rimedio? Quanto buffer al receiver mi occorre?

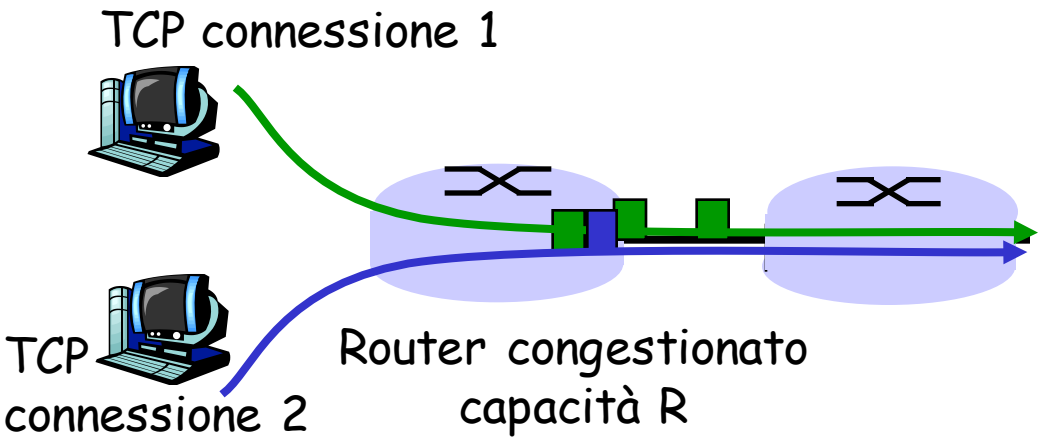
Un modello piu' sofisticato, con L frequenza di perdite, da' TCP-throughput = 1.22 x MSS/(RTT x SQRT(L))

Se chiedo 10 Gbps di throughput dovrei avere probabilita' di perdite una su 5 miliardi di segmenti...!!!

TCP va ripensato: lo stanno facendo in molti

TCP Fairness

Fairness: se N connessioni condividono lo stesso router che e' collo di bottiglia del sistema con capacita' R bps, ogni connessione dovrebbe ricevere R/N della capacità del router (e del link) TCP e' fair? Vediamolo nella prossima slide



Is TCP fair?

Situazione ideale con 2 connessioni TCP: senza traffico UDP, e medesimi valori di MSS e RTT

Ci si aspetta che:

- Le due connessioni si dividano a mezzo la banda

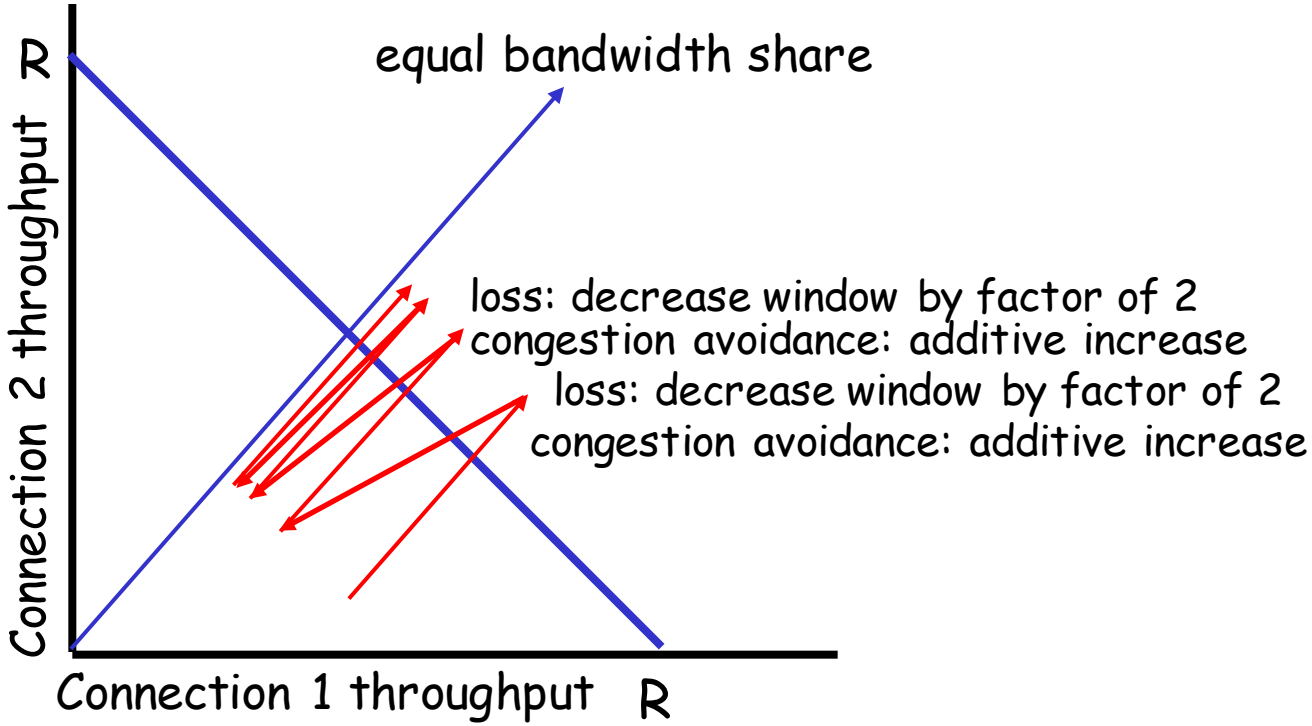
- La usino al massimo: ovvero la somma di quanta ne usa una e di quanta ne usa l'altra dovrebbe essere uguale a R

E infatti… cosi' succede:

Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



TCP fairness?

Nella realta' pero':

- Piu' di 2 connessioni TCP
- I valori di MSS e RTT possono essere molto differenti

Dunque accade che:

  - Le N connessioni non si dividano equamente la banda
  - Bensì chi ha RTT piu' piccolo apre piu' velocemente la propria finestra di congestione e acquisisce piu' banda: unfairness

- Inoltre, il traffico UDP non si lascia diminuire, e' unfair in natura e quindi alla lunga estromette il traffico TCP
- In piu' se in un collegamento uso connessioni TCP multiple, peggioro ulteriormente il problema

TCP latency modeling

Q: How long does it take to receive an object from a Web server after sending a request?

- TCP connection establishment
- data transfer delay (including slow start)

Notation, assumptions:

- Assume one link between client and server of rate R
- Assume: fixed congestion window, W segments
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

Without slow-start (minimal latency):

- Latency = 2 RTT + O/R
- Obtained summing: 1) RTT for SYN + ACK between client& server, 2) RTT for final ACK plus request for object, 3) O/R for the object

TCP latency modeling

Piu' in dettaglio, con finestra di W segmenti, dopo la richiesta del client, server trasferisce W segmenti e poi aumenta di 1 S ogni ACK.

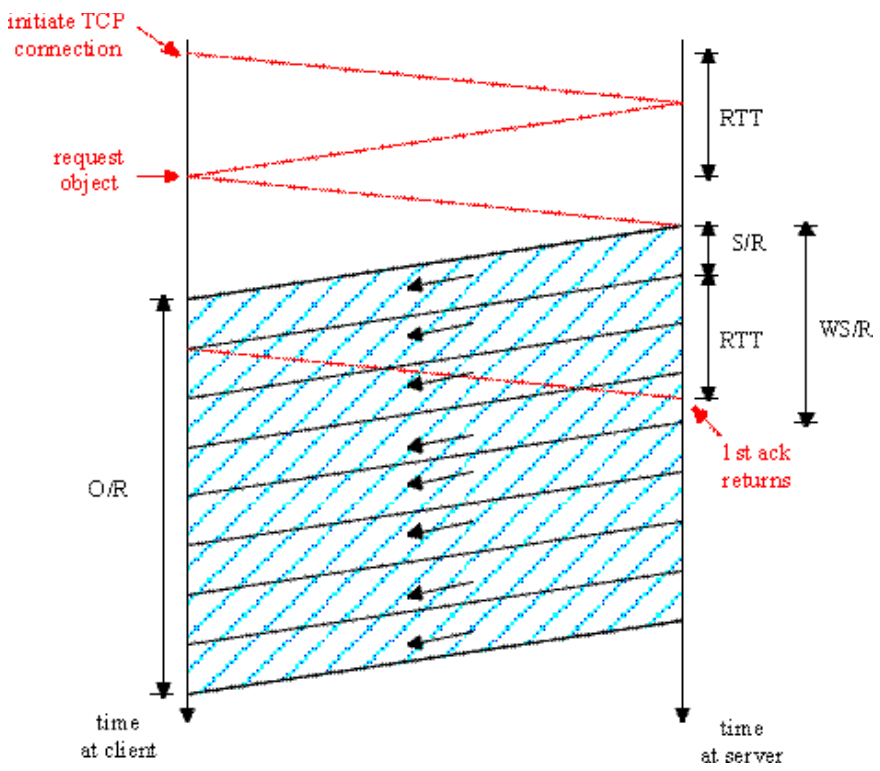
Sono dati due casi: 1) server finisce di mandare la prima finestra W (in tempo WS/R), in tempo per ricevere il primo ACK prima della scadenza di WS/R, mentre 2) il server finisce di mandare la prima finestra W, ma riceve il primo ACK quando gia' e' trascorso WS/R

Two cases to consider:

- WS/R > RTT + S/R: ACK for first segment in window returns before window's worth of data sent
- WS/R < RTT + S/R: wait for ACK after sending window's worth of data sent

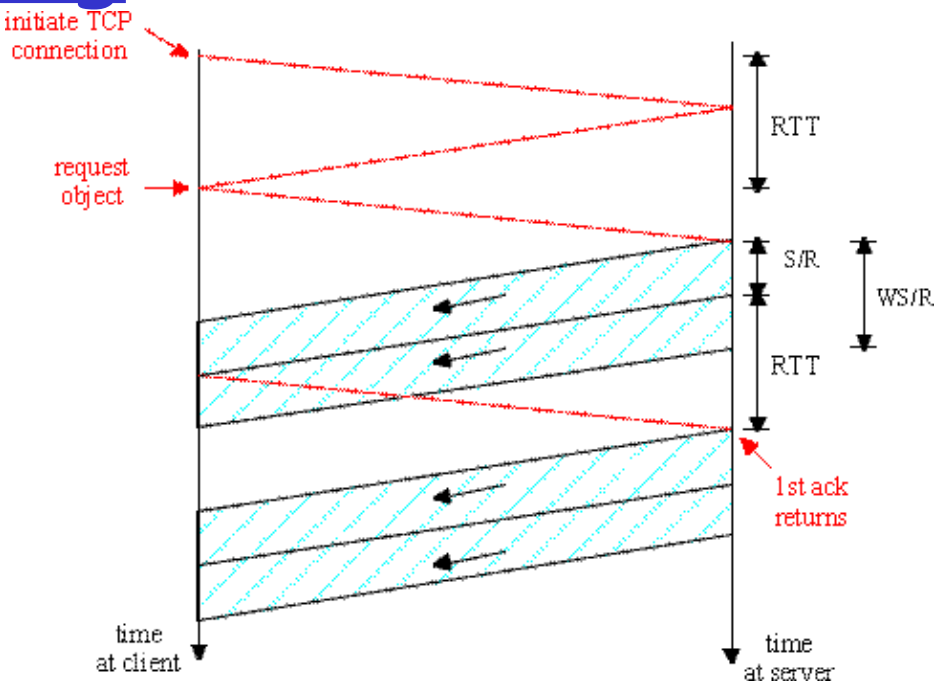
TCP latency Modeling

K:= O/WS



Case 1: Riesce a mandare finestre una dietro l'altra senza interruzioni e quindi

latency = 2RTT + O/R



Case 2: a ogni finestra mandata, si ferma x aspettare ACK, stalla per il numero K di finestre – 1, con periodo uguale a (S/R + RTT) – WS/R, quindi

latency = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]

TCP Latency Modeling: with Slow Start

Example:

O/S = 15 segments

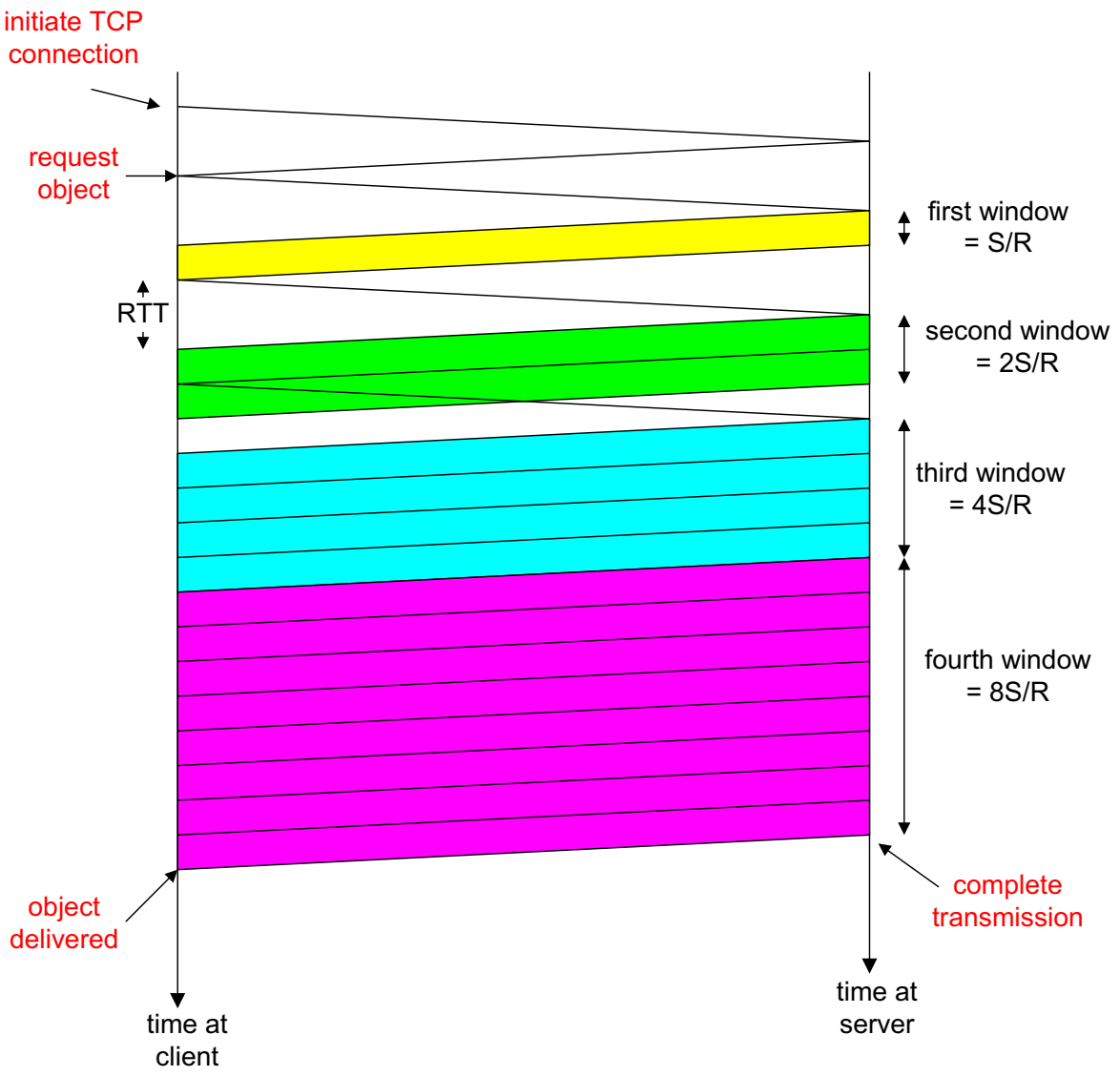
K = 4 windows

Kth windows has $2^{(k-1)}$ segments

Q = 2

P = min{K-1,Q} = 2

Server stalls P=2 times.



TCP Latency Modeling: Slow Start (cont.)

$\frac{S}{R} + RTT =$ time from when server starts to send segment

until server receives acknowledgement

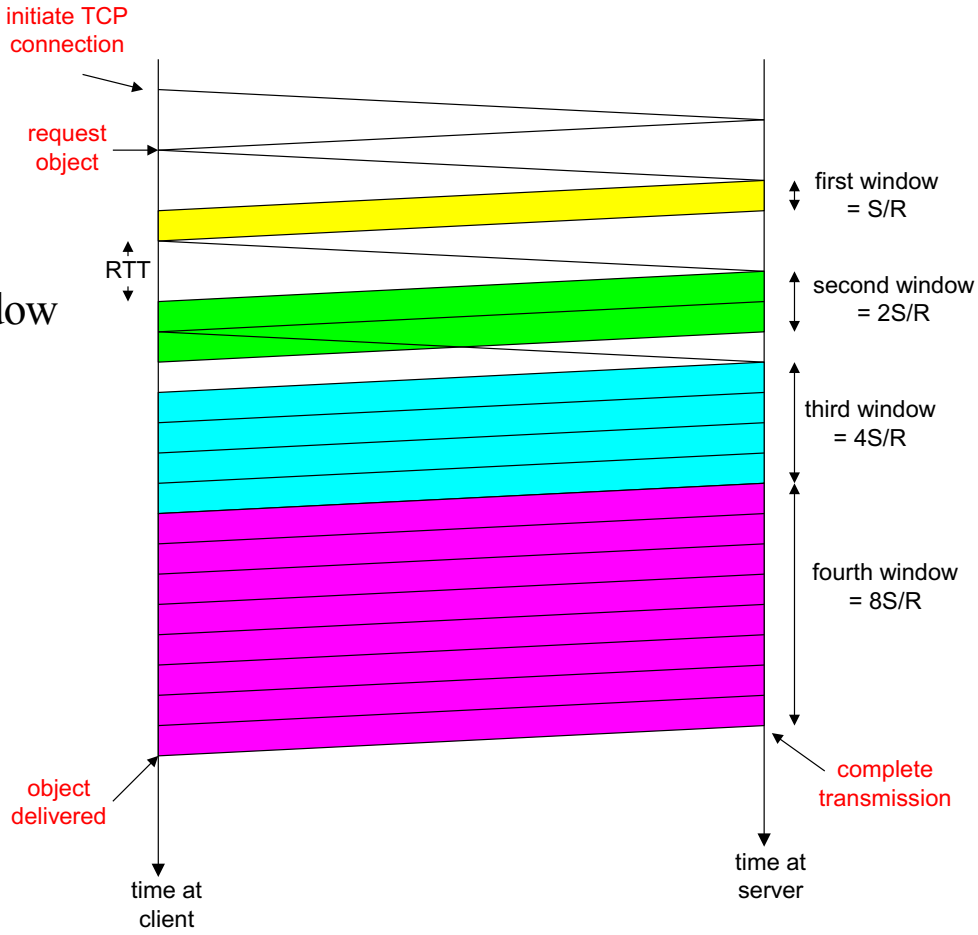
$2^{k-1}\frac{S}{R} =$ time to transmit the kth window

$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} =$ stall time after the *k*th window

$$\text{latency} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} stallTime_p$$

$$= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]$$

$$= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^{P} - 1)\frac{S}{R}$$



TCP Latency Modeling: Slow Start

- window grows according to slow start
- the latency of one object of size O is:

$$Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

where P is the number of times TCP stalls at server:

$$P = \min\{Q, K-1\}$$

- where Q is the number of times the server would stall if the object were comprised of a infinite number of segments
  Q = parte_intera(log (1 + (RTT/(S/R)))) +1

- and K is the number of windows that cover the object.

TCP Latency Modeling: Conclusioni

- Oltre al valore del ritardo TCP dato dalla formula precedente, e' interessante conforontare la formula con controllo di congestione con quella senza controllo di congestione a latenza minima (2RTT + O/R). Si ha:

- latenza/latenza minima <= 1 + (P/ ([(O/R)/RTT] +2)

Ovvero slow-start non aumenta (peggiora) di molto il valore della latenza se RTT << O/R, ossia se il tempo di andata e ritorno e' molto inferiore al tempo di trasmissione dell'oggetto: latenza/latenza minima quasi uguale a 1

TCP Latency: Alcuni casi

- S=536 bytes, RTT=100 msec, O=100 Kbyte, K=8:

R	O/R	P	latenza minima	latenza
28Kbps	28,6s	1	28,8s	28,9s
100Kbps	8s	2	8,2s	8,4s
1Mbps	800ms	5	1s	1,5s
10Mbps	80ms	7	0,28s	0,98s

- S=536 bytes, RTT=100 msec, O=5 Kbyte, K=4:

R	O/R	P	latenza minima	latenza
28Kbps	1,43s	1	1,63s	1,73s
100Kbps	0,4s	2	0,6s	0,76s
1Mbps	40ms	3	0,24s	0,52s
10Mbps	4ms	3	0,20s	0,50s

TCP Latency: Alcuni casi commentati

- Slow start genera ritardi aggiuntivi apprezzabili solo quando il tasso trasmissivo e' piu' alto (grande banda!!)

- Ovvero, con poca banda TCP va prima a regime, e server rimane in stallo poche volte, a 10 Mbps invece va in stallo sempre

- Il fenomeno si nota meno se il file e' piccolo, ho bisogno di meno finestre

TCP Latency: un altro caso, con RTT piu' alto

- S=536 bytes, RTT=1 sec, O=5 Kbyte, K=4:

R	O/R	P	latenza minima	latenza
28Kbps	1,43s	3	3,4s	5,8s
100Kbps	0,4s	3	2,4s	5,2s
1Mbps	40ms	3	2,0s	5,0s
10Mbps	4ms	3	2,0s	5,0s

Il fenomeno peggiora anche con file piccoli e banda modesta se aumenta RTT!!!!!

Chapter 3: Summary

- principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

- leaving the network "edge" (application transport layer)
- into the network "core"