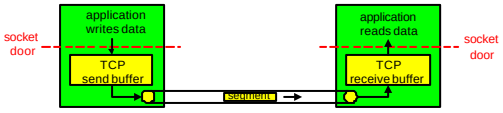
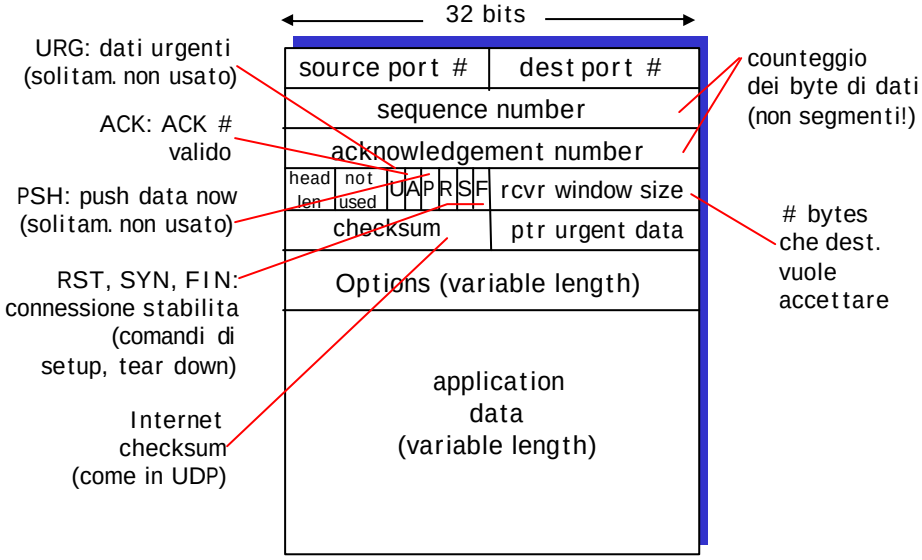
TCP: Panoramica RFC: 793, 1122, 1323, 2018, 2581

- punto-punto:
  - un mittente, un destinatario
- *flusso di byte affidabile e ordinato*
- protocollo pipeline:
  - il controllo di flusso e di congestione definisce la dimensione della window
- *buffer send & receive*



- dati full duplex :
  - Flusso bi-direzionale nella stessa connessione
  - MSS: maximum segment size
- connection-oriented:
  - handshaking per inizializzare lo stato del mittente e destinatario
- flusso controllato:
  - Il mittente non sovraccarica il destinatario

TCP: struttura del segmento



TCP: Connessione

Ricordate: nel TCP si stabilisce una "connessione" prima di scambiare segmenti dati

- inizializzare variabili TCP:
  - seq. #
  - info buffers, controllo
  - flusso (es., **RcvWindow**)
- *client:* avvia connessione

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

- *server:* contattato da client

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

Three way handshake:

Passo 1: il client invia un SYN al server
- SYN=1, specifica il seq # iniziale

Passo 2: il server riceve SYN, risponde SYNACK
- alloca buffers
- ACK del SYN, specifica server-> seq. # iniziale

Passo 3: client ric. SYNACK
- alloca buffers
- invia riscontro (SYN = 0, seq# =iniziale+1, ACK del seq# server+1)

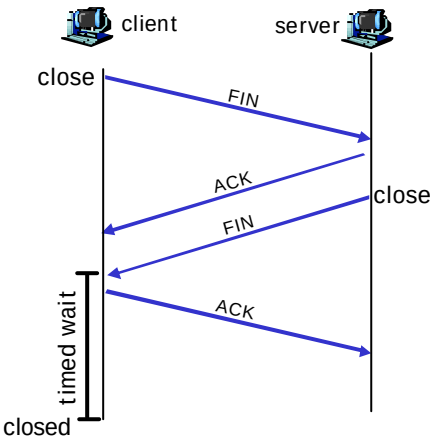
TCP: Connessione (cont.)

Chiusura connessione:

client closes socket:
clientSocket.close();

Passo 1: il client invia FIN al server

Passo 2: il server riceve FIN, replica con ACK. Chiude la connessione, invia FIN.



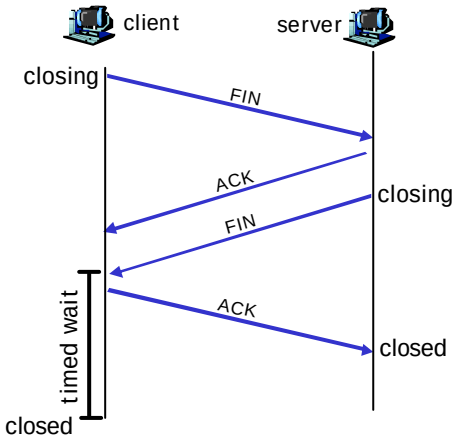
TCP: Connessione (cont.)

Passo 3: il client riceve FIN, replica con ACK.

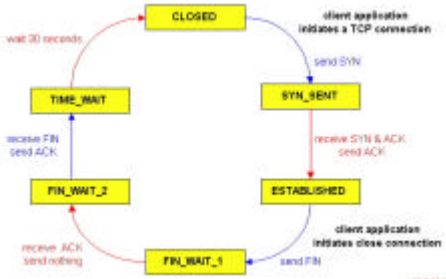
- Entra in "attesa" risponde con ACK ai FIN ricevuti

Passo 4: server, riceve ACK. chiude la connessione

Nota: con poche modifiche, può gestire FIN simultanei

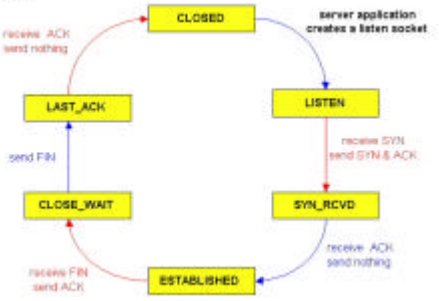


TCP: Connessione (cont)

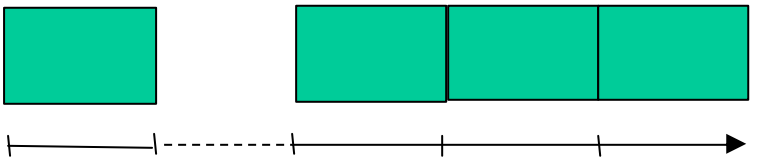


Ciclo di vita client TCP

Ciclo di vita server TCP



TCP seq. # e ACK



TCP: fornisce riscontri cumulativi

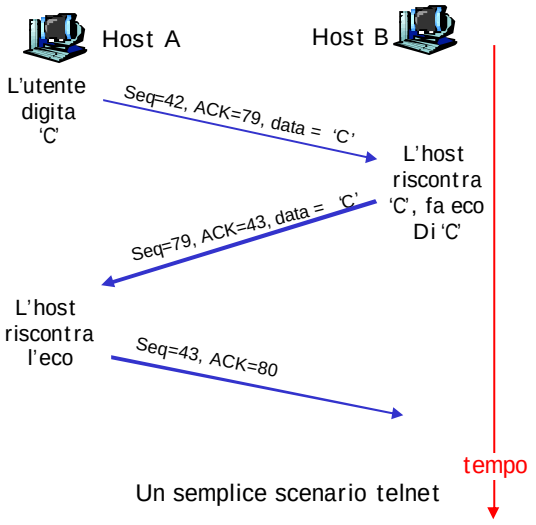
TCP: non ci sono regole fisse nello standard per i segmenti fuori ordine.

TCP seq. # e ACK

Seq. #:
- numero del primo byte nei dati del segmento (primo # scelto a caso)

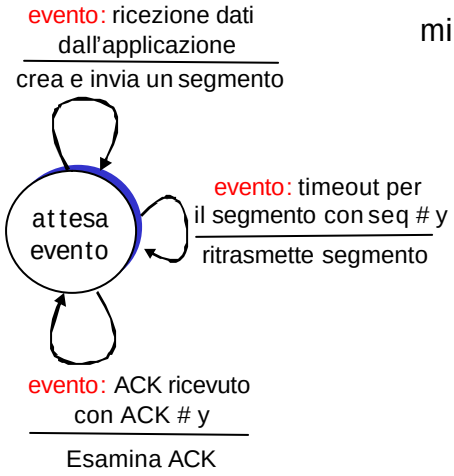
ACK:
- seq # del prossimo byte atteso
- ACK cumulativi

D: come il destinatario tratta i segmenti fuori ordine ?
- R: le specifiche del TCP non lo dicono, dipende dalla implementazione



Un semplice scenario telnet

TCP: trasferimento affidabile



mittente semplice, assumendo

- trasferimento one way
- nessun controllo di flusso o congestione

TCP: trasferim affidabile

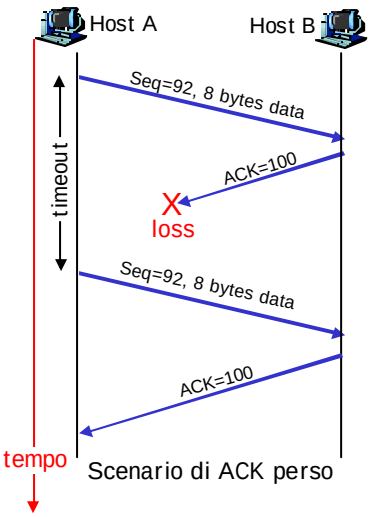
mittente TCP semplificato

```
00  sendbase = initial_sequence number
01   nextseqnum = initial_sequence number
02
03    loop (forever) {
04     switch(event)
05     event: data received from application above
06          create TCP segment with sequence number nextseqnum
07          start timer for segment nextseqnum
08          pass segment to IP
09          nextseqnum = nextseqnum + length(data)
10      event: timer timeout for segment with sequence number y
11          retransmit segment with sequence number y
12          compue new timeout interval for segment y
13          restart timer for sequence number y
14      event: ACK received, with ACK field value of y
15          if (y > sendbase) { /* cumulative ACK of all data up to y */
16             cancel all timers for segments with sequence numbers < y
17              sendbase = y
18             }
19          else { /* a duplicate ACK for already ACKed segment */
20              increment number of duplicate ACKs received for y
21              if (number of duplicate ACKS received for y == 3) {
22                 /* TCP fast retransmit */
23                 resend segment with sequence number y
24                 restart timer for segment y
25              }
26     } /* end of loop forever */
```

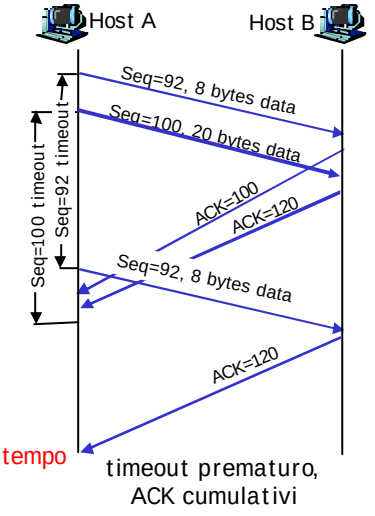
TCP: generaz. ACK [RFC 1122, RFC 2581]

Evento	azioni del destinatario TCP
arrivo ordinato segmenti, non ci sono “buchi”, tutto il resto già riscontrato	ACK ritardato. Attendi 500ms per prossimo segmento. Se non arriva, Invia ACK
arrivo ordinato segmenti, non ci sono “buchi”, un ACK ritardato sospeso	Invia immediatamente un singolo ACK cumulativo
arrivo segmento fuori ordine seq. # maggiore di quello atteso rilevato un “buco” (gap)	invia ACK duplicato che indica il seq. # del prossimo byte atteso
arrivo di un segmento che colma partialmente o completamente il gap	ACK immediato se il segmento inizia all’estremo inferiore del gap

TCP: ritrasmissione



Scenario di ACK perso



timeout prematuro, ACK cumulativi

TCP: Controllo Flusso

flow control

il mittente non sovrac. il buffer del destin. transmettendo troppo, troppo in fretta

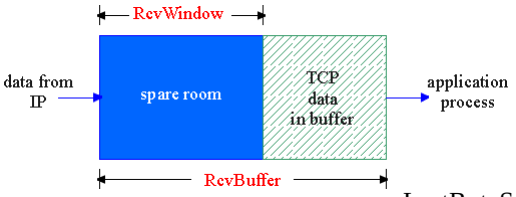
Destinat.: informa esplicitam. mittente dello spazio disponibile (cambia dinamicam.)

- **campo RcvWindow** nel segmento TCP

RcvBuffer = dimensione TCP Receive Buffer
RcvWindow = quantità di spazio rimasto nel Buffer

LastByteRcvd – LastByteRead <= RcvBuffer

RcvWindow = RcvBuffer – [LastByteRcvd – LastByteRead]



Buffer destinatario

Mittente: fa si che i dati trasmessi ma non riscontrati < del più recente **RcvWindow** ricevuto

LastByteSent – LastByteAcked <= RcvWindow

TCP Round Trip Time e Timeout

D: come si stabilisce il valore del timeout?

- Maggiore di RTT
  - nota: RTT varia
- Troppo breve: timeout prematuro
  - Ritrasmissioni inutili
- Troppo lungo: reazione lenta allo smarrimento di segmenti

D: Come stimare RTT?

- **SampleRTT**: misura del tempo dalla trasmissione del segmento fino alla ricezione dell'ACK
  - ignora ritrasmissioni, ACK cumulativi
- **SampleRTT** varia, occorre stimare RTT in maniera opportuna
  - media di molte misure recenti e non solo del **SampleRTT** corrente

TCP Round Trip Time e Timeout

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

- media variabile pesata esponenzialmente
- influenza di un dato campione diminuisce esponenzialm.
- Valore tipico di x: 0.1

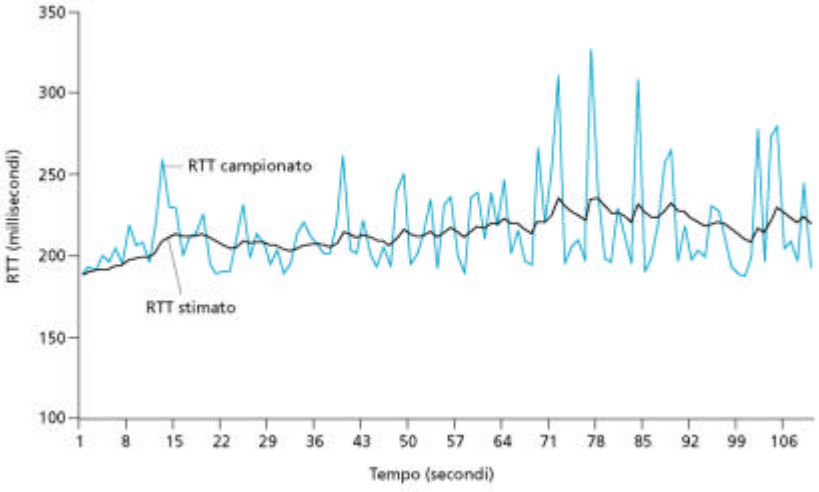
Impostazione del timeout

- **EstimatedRTT** più "margine di sicurezza"
- Ampie variazioni **EstimatedRTT ->** margine di sicurezza maggiore

Timeout = EstimatedRTT + 4*Deviation

Deviation = (1-x)*Deviation +
x*|SampleRTT-EstimatedRTT|

RTT campionati e RTT stimani



Principi di Controllo Congestione

Congestione:

- informalmente: “troppe sorgenti che inviano troppi dati troppo in fretta perché la *rete* sia in grado di gestirli”
- È diverso dal controllo di flusso!
- effetti:
  - Pacchetti persi (a causa dell’ overflow del buffer ai router)
  - Ritardi lunghi (a causa delle code nei buffer dei router)
- Un problema nella top-10!

Approcci per il controllo congestione

Due approcci principali:

Controllo end-to-end :

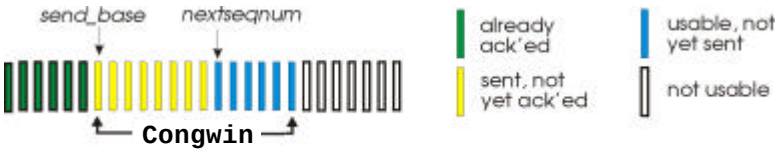
- Non c’è feedback esplicito dalla rete
- stato congestione ricavato dai livelli di perdita e ritardo osservati agli end-system
- L’approccio del TCP

Controllo network-assisted:

- I router forniscono feedback agli end system
  - Un bit indica la congestione (SNA, DECbit, TCP/IP ECN, ATM)
  - Viene specificato esplicitamente a quale velocità il mittente deve trasmettere

Controllo Congestione del TCP

- controllo end-to-end (niente feedback da network)
- La velocità transmissiva è limitata dalla dimensione della finestra di congestione, **Congwin**, sui segmenti:

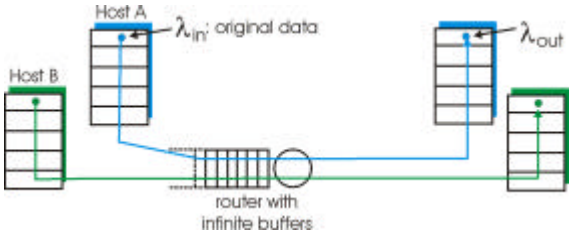


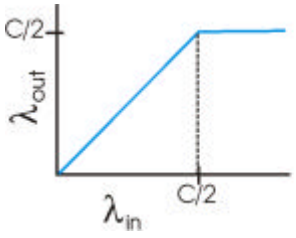
- w segmenti, ciascuno invia MSS bytes in un RTT:

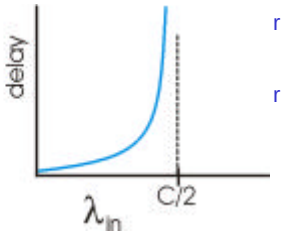
$$\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

Cause/costi congestione: scenario 1

- Due mittenti, due destinatari
- un router, buffer infinito
- No ritrasmissioni



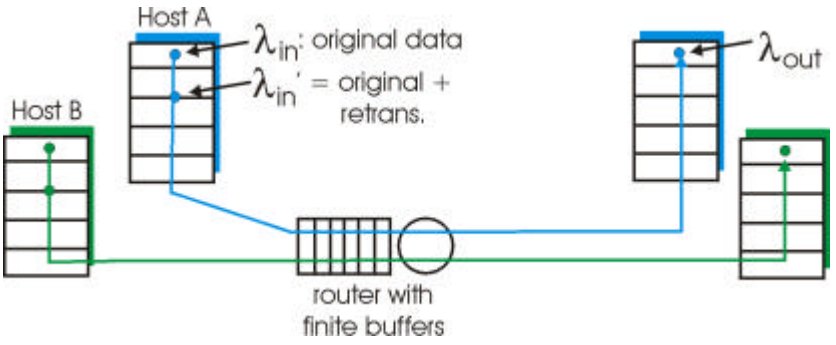




- grossi ritardi in caso congestione
- massimo throughput ottenibile

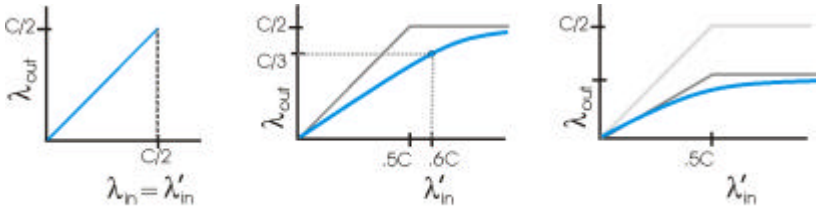
Cause/costi congestione: scenario 2

- un router, buffer *finito*
- Ritrasmissione dei pacchetti persi



Cause/costi congestione: scenario 2

- sempre: $\lambda_{in} = \lambda_{out}$ (goodput)
- ritrasmissione "perfetta" solo quando: $\lambda'_{in} > \lambda_{out}$
- ritrasmissione dei pacchetti ritardati (non persi) rende λ'_{in} più lungo (del caso perfetto) per alcuni λ_{out}



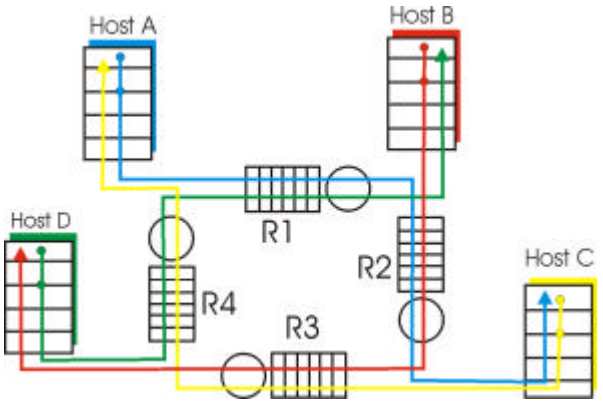
"costi" della congestione:

- più lavoro (ritrasm) per un dato "goodput"
- ritrasmissioni non necessarie: più copie del pkt sul link

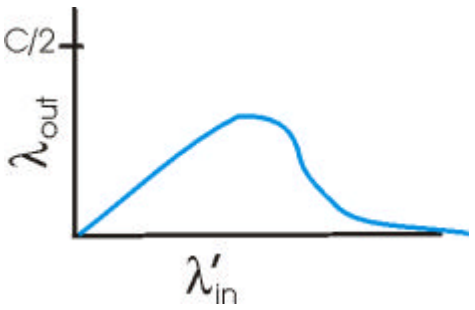
Cause/costi congestione: scenario 3

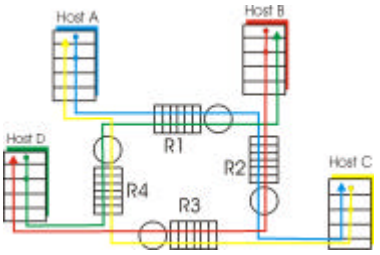
- quattro mittenti
- Percorsi multihop
- timeout/retransmit

D: cosa succede se λ_{in} e λ'_{in} aumentano ?



Cause/costi congestione: scenario 3





Un altro "costo" della congestione:

- Quando un pacchetto viene scartato, ogni capacità di trasmissione "upstream" usata per il pacchetto viene sprecata!

Studio di un Caso: controllo congestione dell' ATM (ABR)

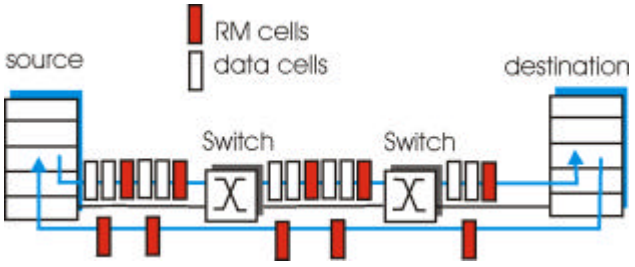
ABR: available bit rate:

- "servizio elastico"
- se il percorso del mittente è "scarico":
  - Il mittende deve usare la banda disponibile
- se il percorso mittente è "congestionato":
  - Il mittente viene riportata ad una velocità minima garantita

celle di RM (resource management) :

- Il mittente le "mischia" con le celle dati
- bits nella cella RM scritti dagli switch ("*network-assisted*")
  - NI bit: non aumentare la velocità (mild congestion)
  - CI bit: congestion indication
- Le celle RM vengono restituite dal destinatario con i bit inalterati

Studio di un Caso: controllo congestione dell' ATM (ABR)



- campo di due-byte ER (explicit rate) nella cella RM
  - uno switch in congestione può diminuire il valore di ER
- nelle celle dati c'è il bit EFCI: posto a 1 negli switch congestionati
  - se la cella dati che precede la cella RM ha EFCI, il mittente pone il bit CI a 1 nella cella RM di ritorno

Controllo Congestione del TCP:

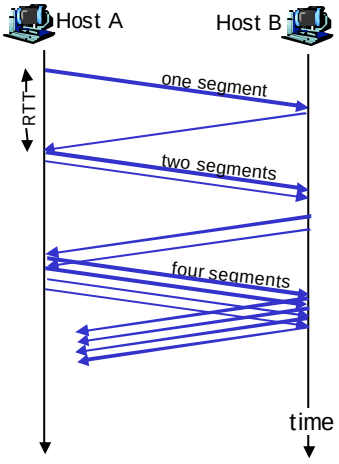
- "sondando" la disponibilità di banda:
  - idealmente: trasmetti il più velocmente possibile (**Congwin** il più ampia possibile) senza perdite
  - *incrementa* **Congwin** finché iniziano le perdite(congestione)
  - congestione: *decrementa* **Congwin**, poi inizia a sondare di nuovo
- due "fasi"
  - slow start
  - congestion avoidance
- variabili importanti:
  - **Congwin**
  - **threshold:** definisce la soglia tra le due fasi di slow start e congestion control

TCP Slowstart

algoritmo Slowstart

initialize: Congwin = 1
for (each segment ACKed)
    Congwin++
until (loss event OR
    CongWin > threshold)

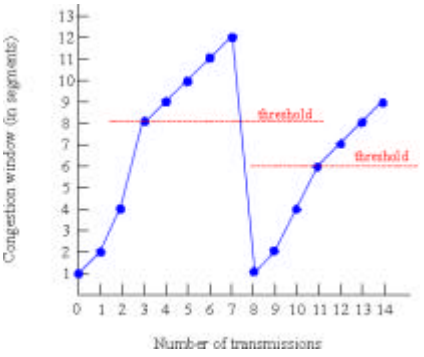
- Incremento esponenziale (per RTT) nel window size (non è poi così lento!)
- evento loss : timeout (Tahoe TCP) and/or tre ACK duplicati (Reno TCP)



TCP Congestion Avoidance

Congestion avoidance

```
/* slowstart is over        */
/* Congwin > threshold */
Until (loss event) {
  every w segments ACKed:
      Congwin++
  }
threshold = Congwin/2
Congwin = 1
perform slowstart[1]
```



1: TCP Reno salta lo slowstart (fast recovery) dopo tre ACK duplicati

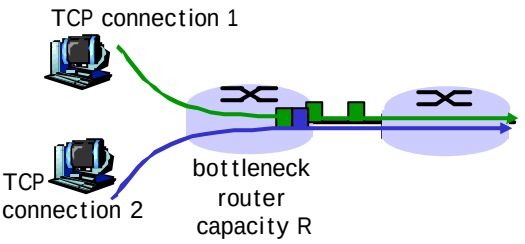
AIMD

TCP congestion avoidance:

- AIMD: *additive increase, multiplicative decrease*
  - Aumenta la window di 1 per RTT
  - Decrementa la window di un fattore 2 se c'è congestione

TCP: Equità (Fairness)

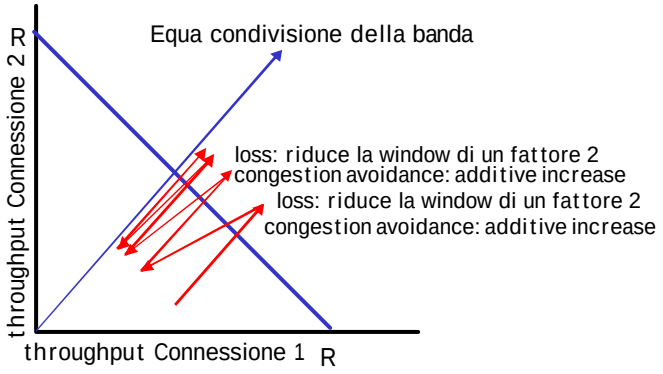
obiettivi della Fairness:
se N sessioni TCP condividono lo stesso link, ciascuna deve ottenere 1/N della capacità del link



Perchè il TCP è equo?

Due sessioni in competizione:

- L'additive increase da pendenza 1, se il throughput cresce
- Il multiplicative decrease riduce il throughput proporzionalmente



TCP: modello di latenza

D: Quanto tempo occorre per ricevere un oggetto da un Web server dopo aver inviato una richiesta?

- Stabilire una connessione TCP
- Ritardo trasferimento dati

Notazioni, assunzioni:

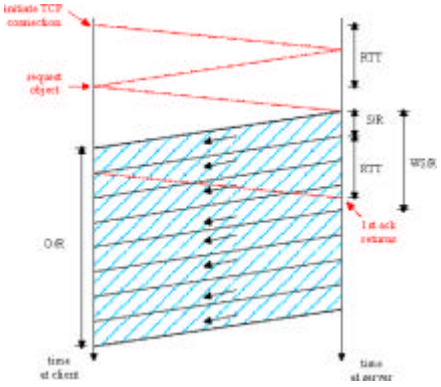
- Un solo link tra client e server di velocità R
- Window di congestione fissa, W segmenti
- S: MSS (bits)
- O: object size (bits)
- no ritrasmissioni (no loss, no corruption)

due casi da considerare:

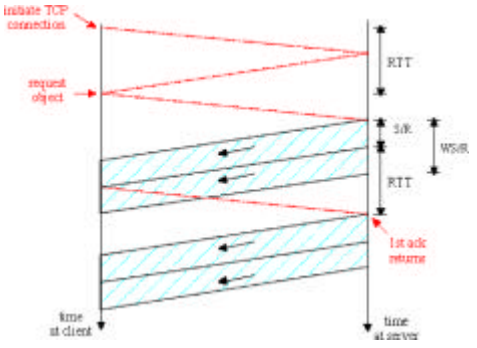
- WS/R > RTT + S/R: ACK del primo segmento nella window torna prima
- WS/R < RTT + S/R: aspetta ACK dopo aver spedito il valore della finestra dati inviata

TCP latency Modeling

K:= O/WS



Case 1: latency = 2RTT + O/R



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

TCP Latency Modeling: Slow Start

- Now suppose window grows according to slow start.
- Will show that 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 of infinite size.

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

TCP Latency Modeling: Slow Start (cont.)

Example:

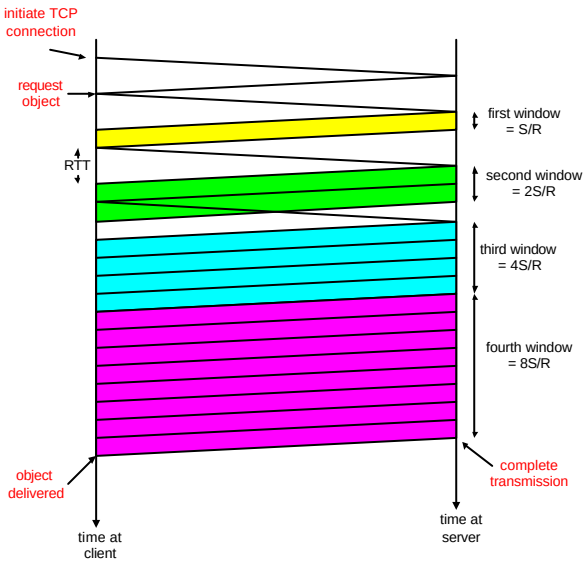
O/S = 15 segments

K = 4 windows

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

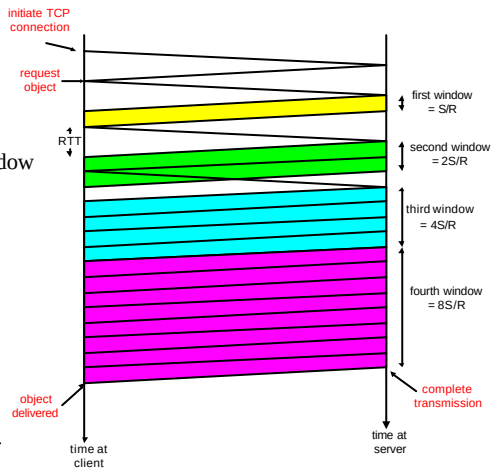
$2^{k-1}\frac{S}{R} =$ time to transmit the kth window

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

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

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

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



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"