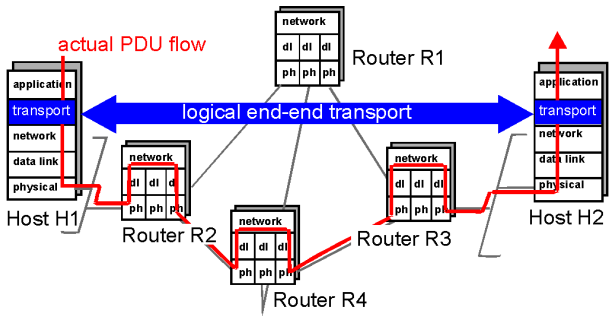
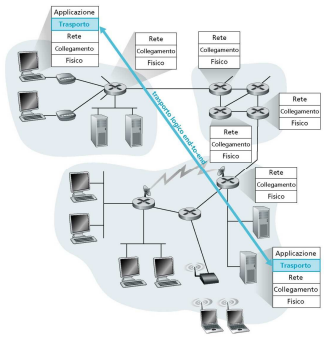
## Il livello di trasporto






## Il livello di trasporto

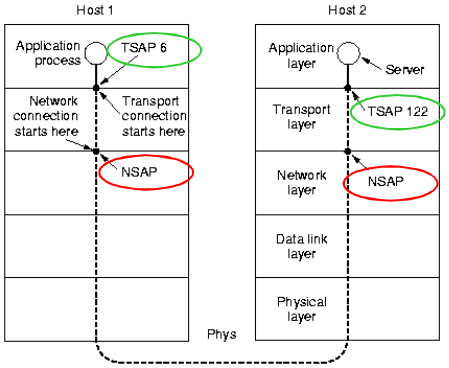
Il livello di trasporto fornisce una comunicazione logica tra gli host, saltando tutte le problematiche della connessione di rete.




Il livello di trasporto separa la parte alta della pila protocollare da quella bassa.

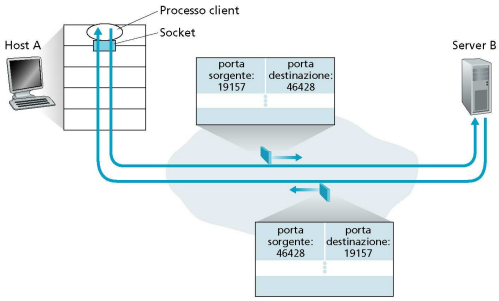


## Indirizzamento nel TL (mux - demux)





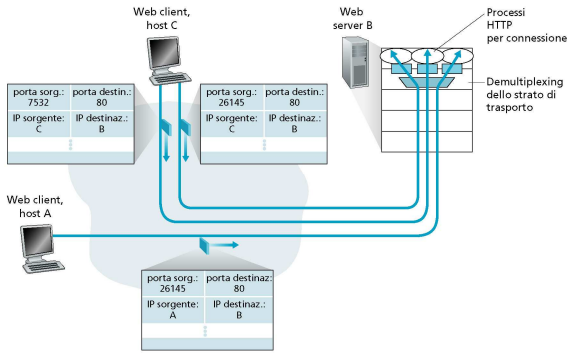

## Connectionless mux - demux




Nel segmento spedito al livello di trasporto sono presenti sia la porta di destinazione che quella mittente, per poi poter ricevere la risposta.



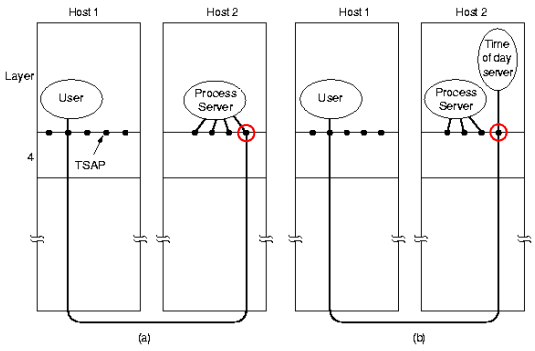
## Connection oriented mux - demux




Il socket è individuato da 5 parametri



## Indirizzamento con un Name Server



## UDP

UDP fornisce un servizio di Datagram per le applicazioni.

I segmenti dati vengono passati immediatamente al livello di rete, aggiungendo un piccolo preambolo.

UDP non ha un protocollo end-to-end (a parte il mux-demux delle porte)



## Vantaggi di UDP

- Controllo sul flusso di dati da parte dell'applicazione
- Nessuna connessione
- Nessuno stato
- Piccolo overhead di gestione

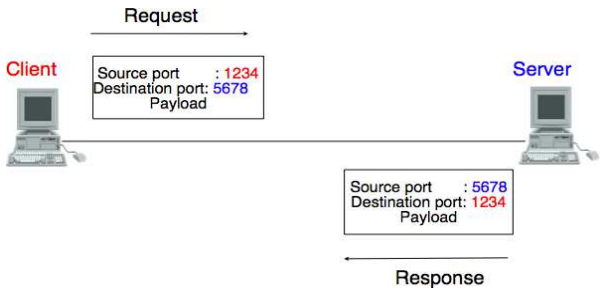


## UDP

| 0 | 15 | 16 | 31 |
|---|---|---|---|
| Source Port Number(16 bits) | | Destination Port Number(16 bits) | |
| Length(UDP Header + Data)16 bits | | UDP Checksum(16 bits) | |
| Application Data (Message) | | | |

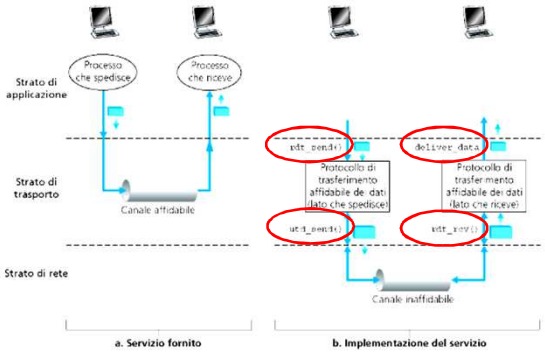


## UDP





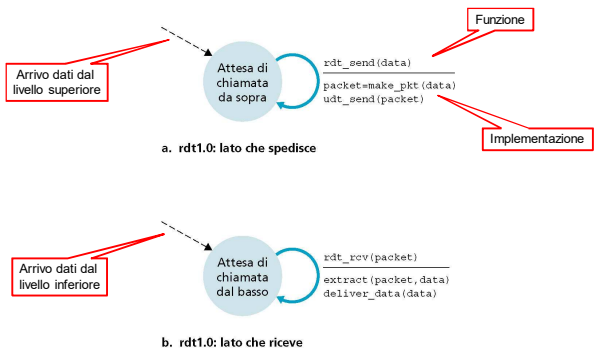

## Reliable data transfer






## RDT 1.0

Protocollo 1.0 : canale affidabile senza perdite



## Problemi

Il canale può inserire errori nei pacchetti:

⇨ gestione dei pacchetti errati

Il receiver manda:

• un ACK se il pacchetto ricevuto è corretto

• un NAK se il pacchetto ricevuto è errato



## RDT 2.0

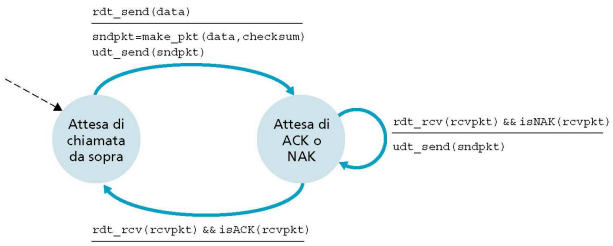
Canale con errori ma senza perdite

- I pacchetti arrivano sempre ma possono arrivare errati.
- Serve una funzione per calcolare o controllare il *checksum* (verifica di correttezza del pacchetto)



## RDT 2.0 sender

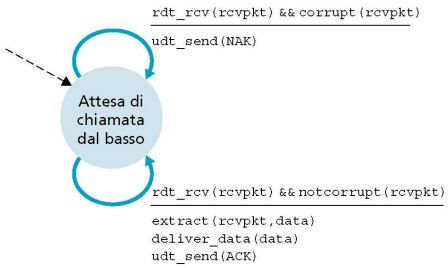
Canale con errori ma senza perdite






## RDT 2.0 receiver

Canale con errori ma senza perdite






## Problemi di RDT 2.0

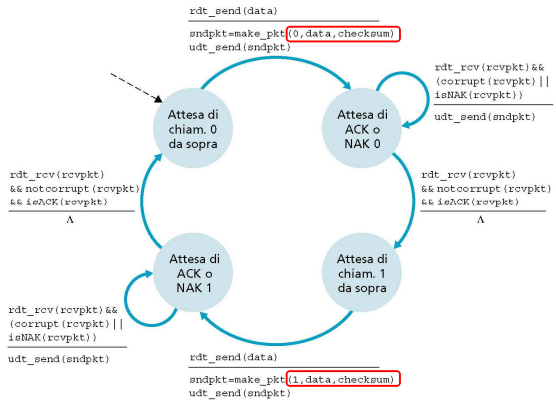
Se l'errore è nella fase di ritorno, il sender rimane bloccato, così come il receiver.

Il sistema è bloccato!

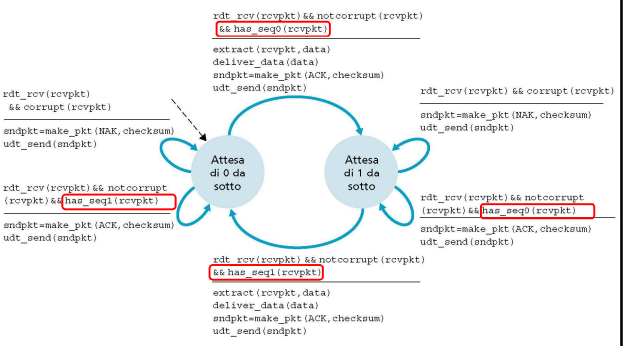
⇨ Gestione dei pacchetti di controllo errati (ACK - NAK)



## RDT 2.1 sender



## RDT 2.1 receiver



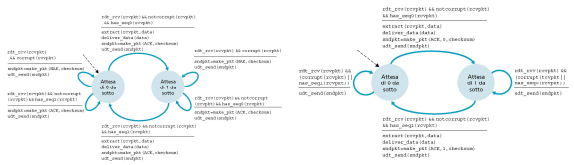



## Problemi

Duplicazione dei pacchetti di controllo (ACK - NAK)

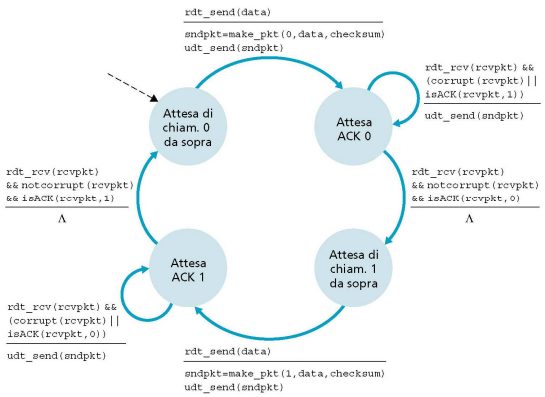
I NAK possono essere evitati rispedendo l'ACK relativo all'ultimo pacchetto ricevuto correttamente.

Questo semplifica il funzionamento del receiver.



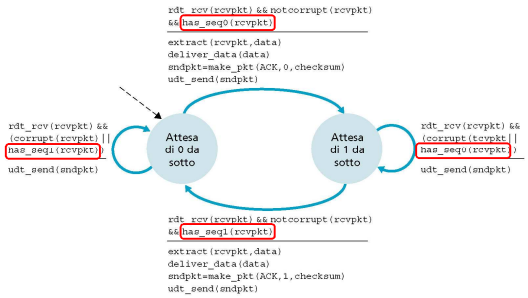


## RDT 2.2 sender






## RDT 2.2 receiver






## Problemi

Nei protocolli precedenti sia il Sender che il Receiver, dopo aver spedito un pacchetto, si mettono in attesa (*bloccati*) di una risposta.

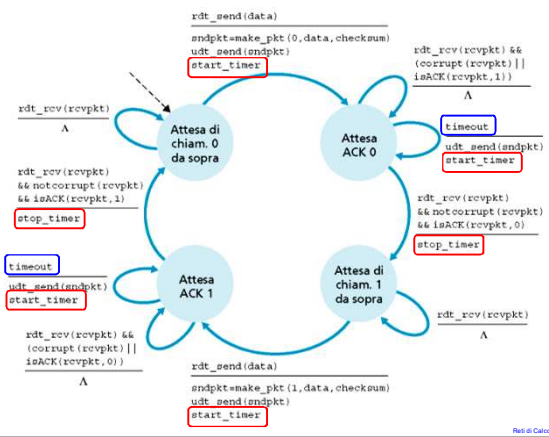
Se entrambi sono in attesa di un messaggio e non vi sono pacchetti in transito, il sistema è bloccato!

A volte il canale non consegna tutti i pacchetti: in questo caso abbiamo un:

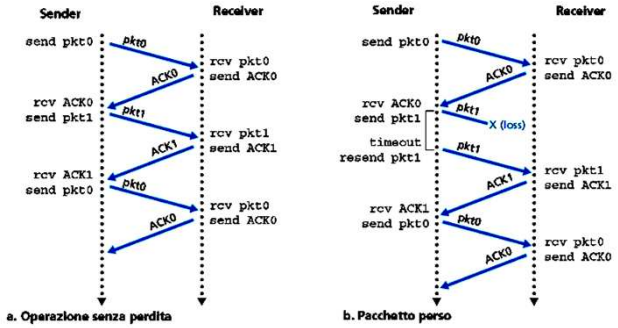
Lossy Channel with Bit Errors



## RDT 3.0 sender

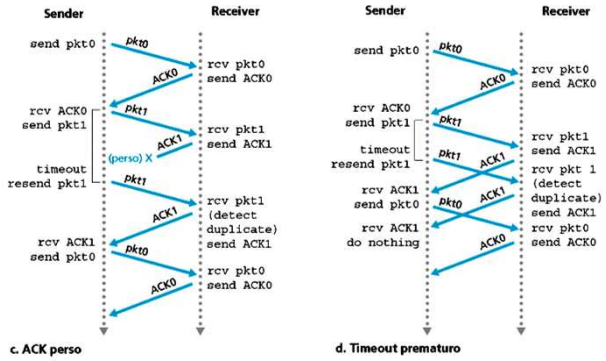


## RDT 3.0

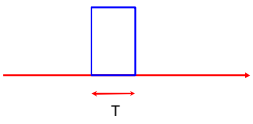





## RDT 3.0



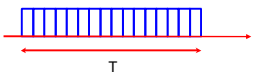



10 Mbps

1 s => 10.000.000 bit

1 bit => 0.1 μs




10 Mbps

Frame da 1500 byte

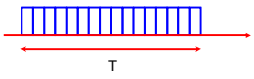
T = 1200 μs




1 km

v ≃ 200.000 km/s

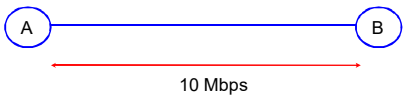
$T_{propagazione} \simeq 5\ \mu s$




10 Mbps

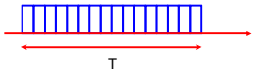
5 μs => 50 bit






10 Mbps

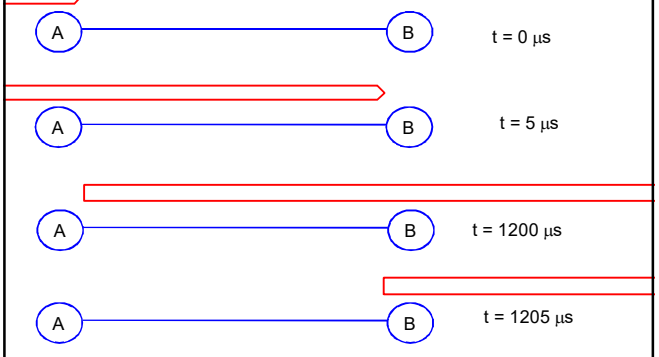
1 km T ≃ 5 μs

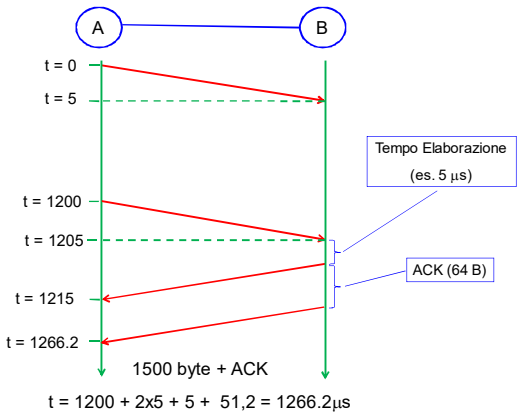


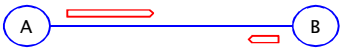

1500 byte

t = 1200 μs + 5 μs








1500 byte spediti in 1266.2 µs

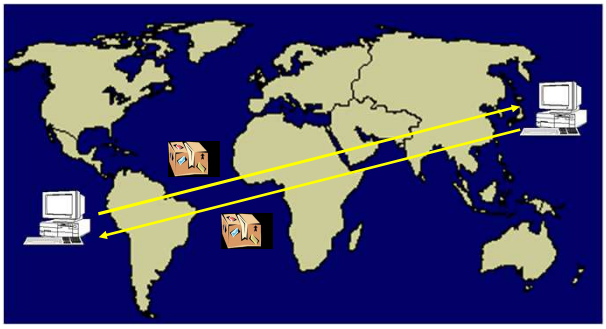
Throughput effettivo = 1500 x 8 / 1266.2 $10^{-6}$

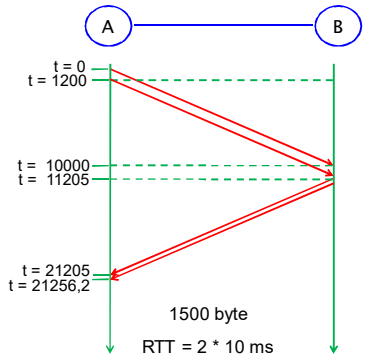
= 9,477 $10^{6}$ bps

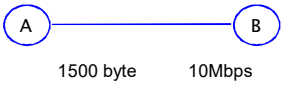
= 9,477 Mbps

## Problemi

Protocollo stop and wait





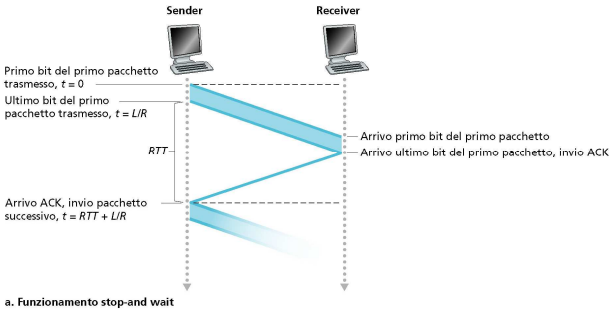




| | | | |
|---|---|---|---|
| RTT 10 µ s | t = 1266,2 µs | 9,477 Mbps | |
| RTT 20 ms | t = 21256,2 µs | 0,564 Mbps | 564 Kbps |

## Ritardi su Internet

```
Esecuzione di Ping www.l.google.com [74.125.39.106] con
32 byte di dati:
Risposta da 74.125.39.106: byte=32 durata=298ms TTL=238
Risposta da 74.125.39.106: byte=32 durata=268ms TTL=238
Risposta da 74.125.39.106: byte=32 durata=310ms TTL=238
Risposta da 74.125.39.106: byte=32 durata=267ms TTL=238

Statistiche Ping per 74.125.39.106:
    Pacchetti: Trasmessi = 4, Ricevuti = 4,
    Persi = 0 (0% persi),
Tempo approssimativo percorsi andata/ritorno in
millisecondi:
    Minimo = 267ms, Massimo =  310ms, Medio =  285ms
```

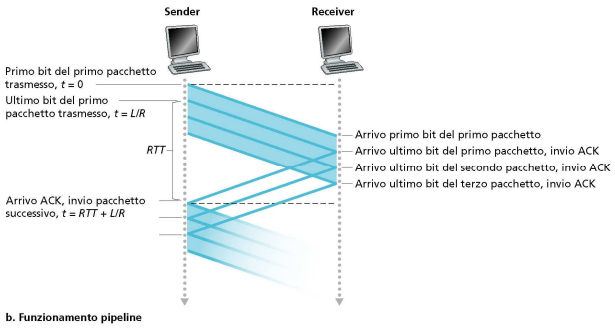
## Stop and wait




a. Funzionamento stop-and wait



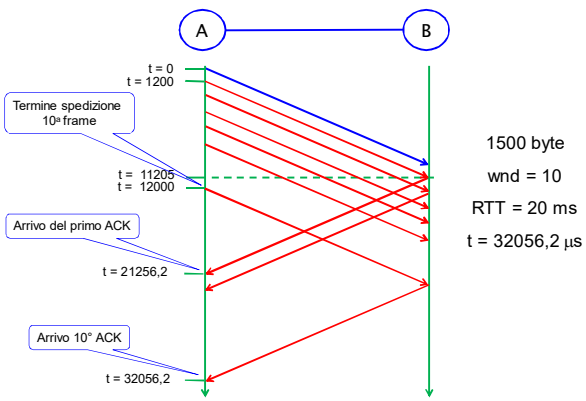
## Pipeline




b. Funzionamento pipeline



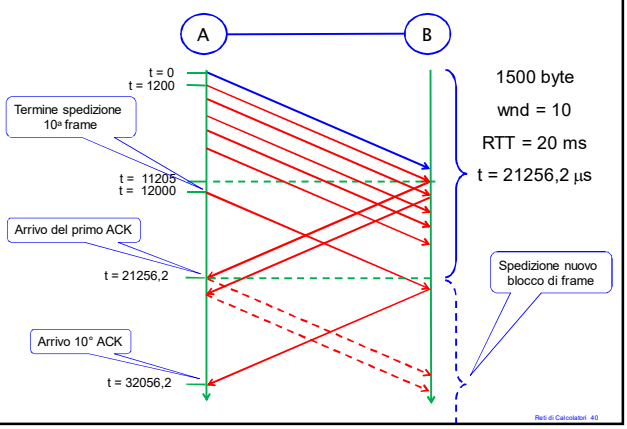
## Pipeline




1500 byte
wnd = 10
RTT = 20 ms
t = 32056,2 µs



## Pipeline





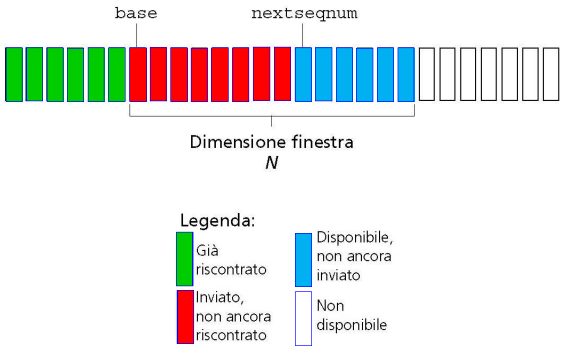

## Pipeline


Frame da 1500 byte  Bw = 10Mbps

| | | | |
|---|---|---|---|
| RTT 10 µs | wnd=1 | t = 1266,2 µs | 9,477 Mbps |
| RTT 20 ms | wnd= 1 | t = 21256,2 µs | 0,564 Mbps |
| RTT 20 ms | wnd=10 | t = 21256,2 µs | 5,645 Mbps |



## Go back N



## Go back N sender

```
rdt_send(data)
if(nextseqnum<base+N){
   sndpkt[nextseqnum]=make_pkt(nextseqnum,data,checksum)
   udt_send(sndpkt[nextseqnum])
   if(base==nextseqnum)
      start_timer
   nextseqnum++
   }
else
   refuse_data(data)

Λ
base=1
nextseqnum=1

timeout
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
...
udt_send(sndpkt[nextseqnum-1])

rdt_rcv(rcvpkt) && corrupt(rcvpkt)
Λ

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
base=getacknum(rcvpkt)+1
If(base==nextseqnum)
   stop_timer
else
   start_timer
```

Attesa



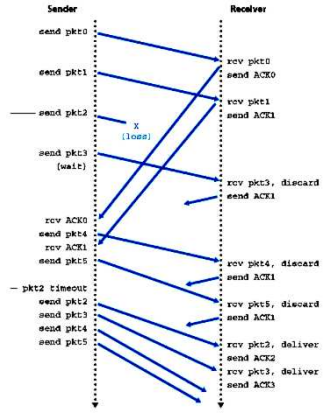
## Go back N receiver

```
rdt_rcv(rcvpkt)
  && notcorrupt(rcvpkt)
  && hasseqnum(rcvpkt,expectedseqnum)

extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(expectedseqnum,ACK,checksum)
udt_send(sndpkt)
expectedseqnum++

default
udt_send(sndpkt)

Λ
expectedseqnum=1
sndpkt=make_pkt(0,ACK,checksum)
```

Attesa

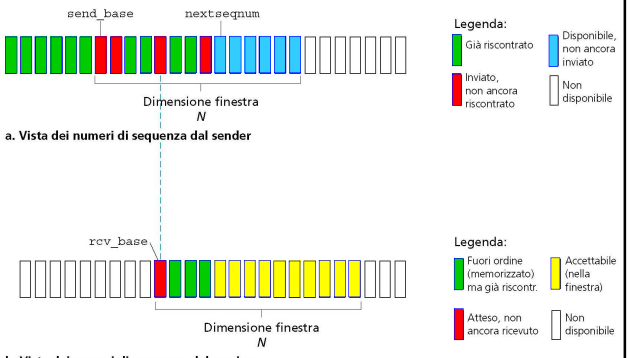


## Go back N



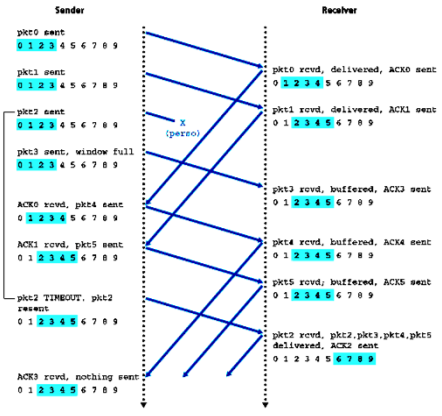



## Ripetizione selettiva



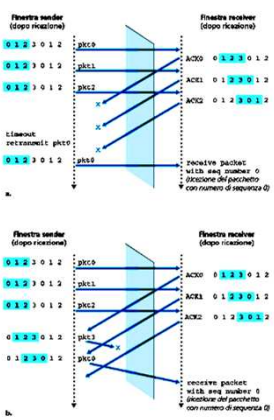



## Ripetizione selettiva






## Ripetizione selettiva



## TCP

- Dicembre 1970: NCP (Network Control Program)
- Dicembre 1974: TCP v1 (Transmission Control Program, RFC 675)
- Marzo 1977: TCPv2
- Primavera 1978: TCPv3 / IPv3 (Transmission Control Protocol)
- Settembre 1981: TCPv4 / IPv4 (RFC 793)



## Funzionalità di TCP

- Addressing/Multiplexing
- Connection Establishment, Management and Termination
- Data Handling and Packaging
- Data Transfer
- Providing Reliability and Transmission Quality Services
- Providing Flow Control and Congestion Avoidance Features



## Cosa non fa TCP

- Specifying Application Use
- Providing Security
- Maintaining Message Boundaries
- Guaranteeing Communication

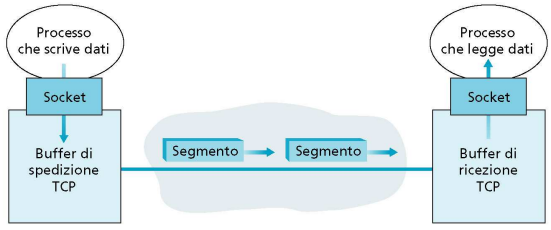


## Caratteristiche di TCP

- Connection-Oriented
- Bidirectional
- Multiply-Connected and Endpoint-Identified
- Reliable
- Acknowledged
- Stream-Oriented
- Data-Unstructured
- Data-Flow-Managed



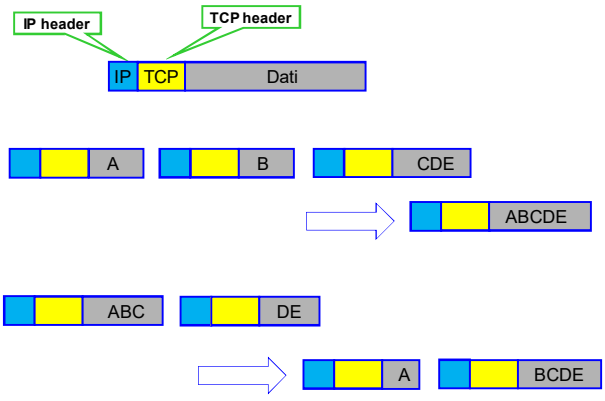
## Il Protocollo TCP



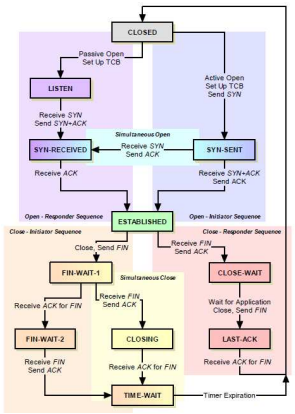

A differenza di UDP, TCP bufferizza i dati prima di inviarli sul canale.



## Il Protocollo TCP

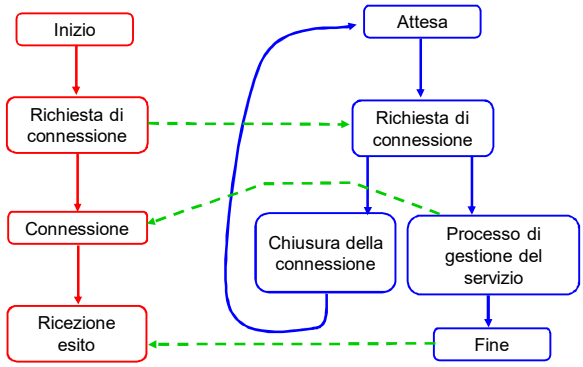


## Modello Client-Server



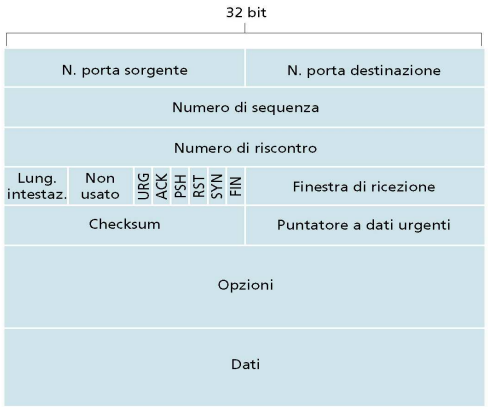



## Indirizzamento con Server Multipli



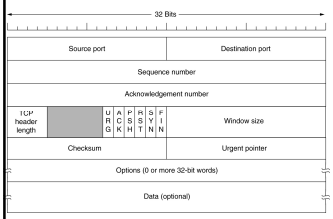



## Formato pacchetto TCP






## Formato pacchetto TCP




TCP utilizza un numero di sequenza da 32 bit per ogni byte trasmesso. Ciò assicura un tempo elevato prima dell'eventuale riutilizzo.

TCP suddivide i dati in segmenti, composti da un preambolo fisso da 20 byte, eventuali preamboli opzionali ed eventuali dati.



## TCP Default Maximum Segment Size

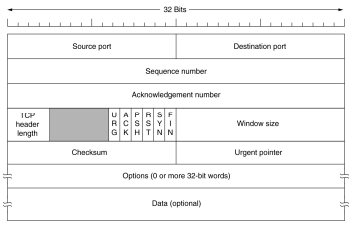
TCP richiede che tutte le reti siano in grado di trasferire pacchetti IP di almeno 576 byte.

- Header IP = 20 byte
- Header TCP = 20 byte
- Dati TCP = 536 byte

Il minimo MSS di TCP è 576.



## Formato pacchetto TCP

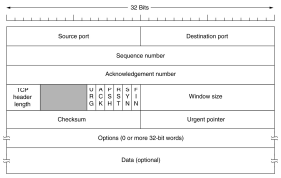



Il campo **Acknowledge** identifica il byte successivo atteso, e non l'ultimo arrivato.

**TCP HL** (4 bit) indica la lunghezza totale dell'header, in parole da 32 bit.

Seguono 6 bit inutilizzati (reserved).

## Formato pacchetto TCP




**URG** è il bit di dati urgenti (Out of Band Data)

**ACK** indica che il campo ACKNOWLEDGE è valido

**PSH** è il bit di PUSH

**RST** viene usato per resettare una connessione.

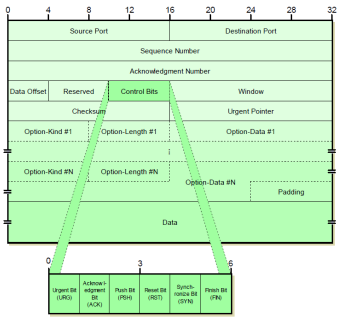
**SYN** si usa per creare una connessione

**FIN** serve per chiudere una connessione.



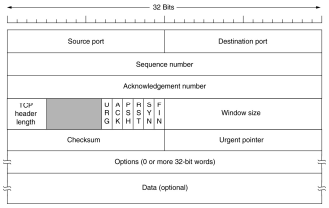
## Formato pacchetto TCP

Il campo **opzioni** viene, ad esempio, usato da sender e receiver per negoziare la massima dimensione del segmento (MSS).






## Formato pacchetto TCP

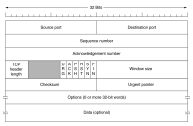



Il campo **window size** specifica la dimensione della finestra di ricezione.

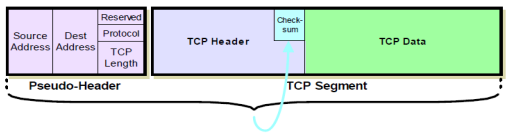
Normalmente non consente di spedire più di 64kB per volta.



## Formato pacchetto TCP

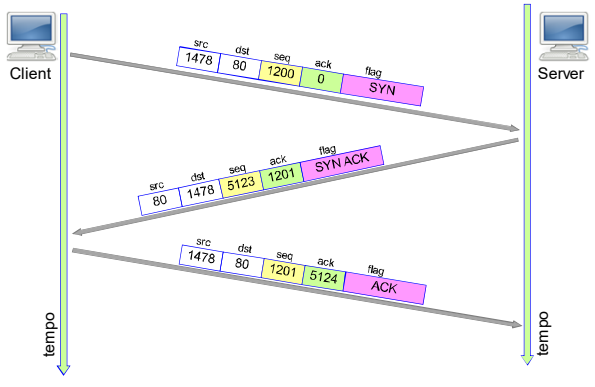


Il campo **checksum** serve per fornire un controllo sul preambolo, sui dati e sullo pseudopreambolo.



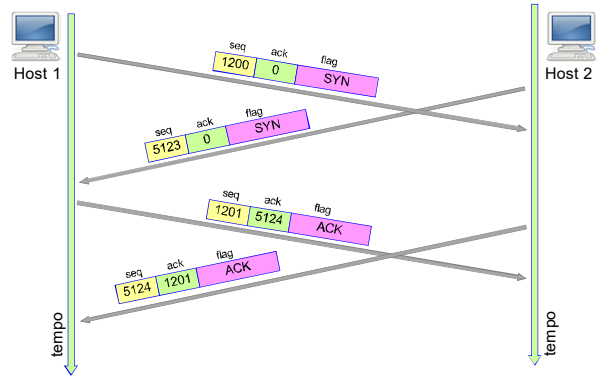



## Creazione della connessione TCP

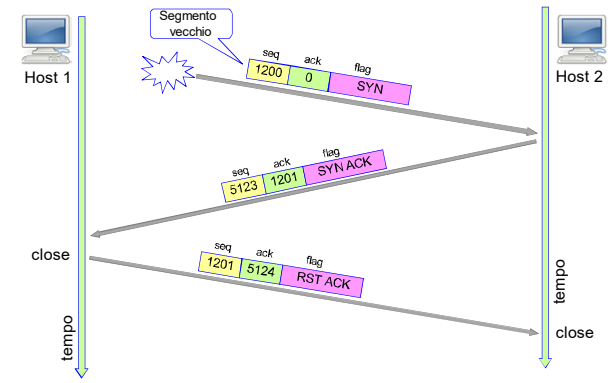





## Creazione della connessione TCP

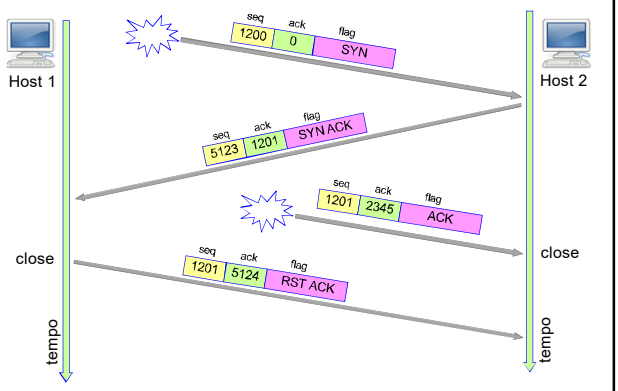


## Creazione della connessione TCP



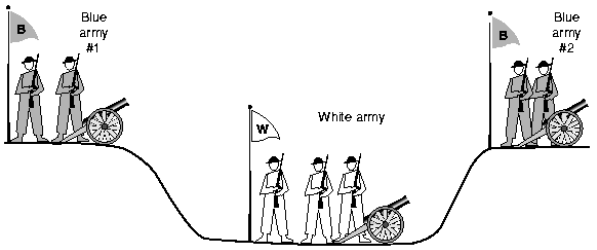



## Creazione della connessione TCP



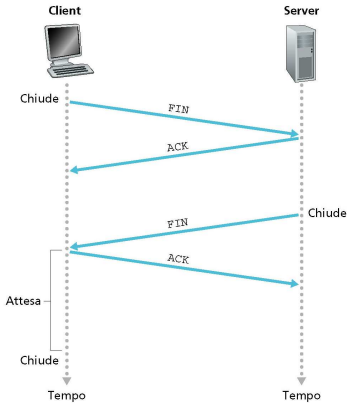



## Chiusura della connessione



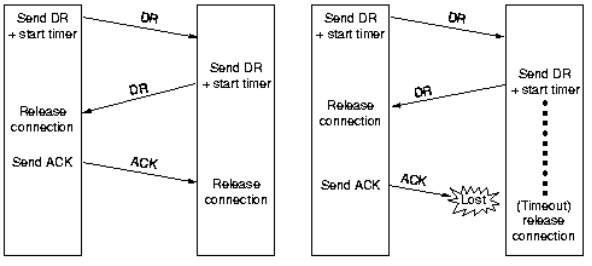



## Chiusura della connessione



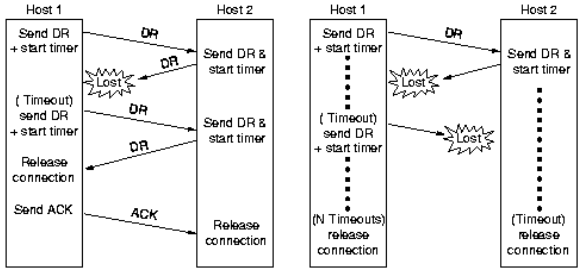



## Chiusura della connessione

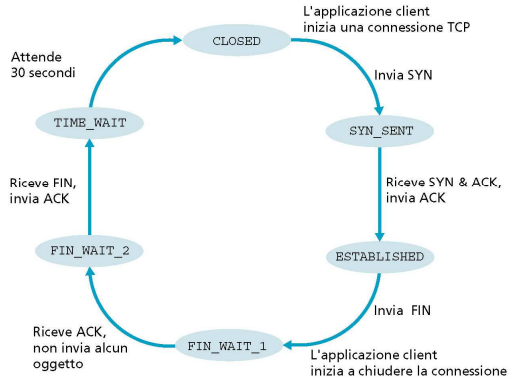





## Chiusura della connessione

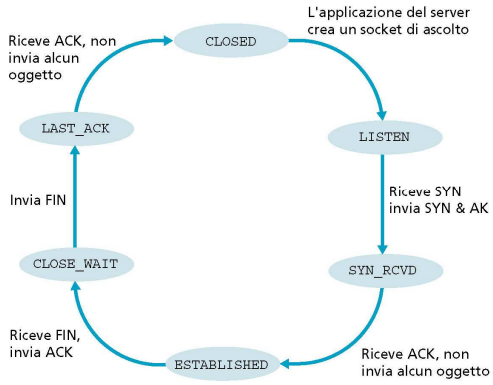


## Chiusura lato Client



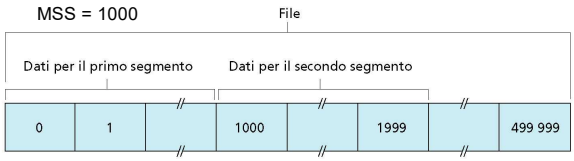



## Chiusura lato Server






## Numeri di sequenza



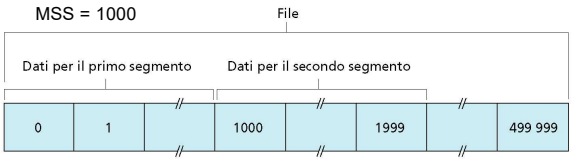

La dimensione massima dei segmenti è di 65535 byte (compreso il preambolo IP), ma normalmente si generano pacchetti rispettando il limite del MTU (Maximum Transfer Unit), di solito pari a 1500 byte.

Di norma MSS = MTU - 20 - 20

Eventualmente i router attraversati possono frammentare i segmenti, replicando però i preamboli. Ciò però comporta uno spreco nella banda utilizzata.



## Numeri di sequenza



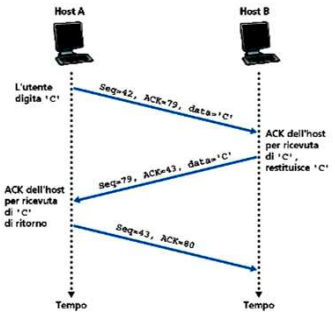

TCP utilizza un protocollo sliding window per la gestione del flusso dati.

Una eventuale frammentazione successiva complica la gestione, in quanto di un segmento unico ne possono arrivare alcune parti.

Il campo ACK contiene il valore del prossimo byte atteso, quindi fornisce riscontri cumulativi.



## Esempio: Telnet e TCP

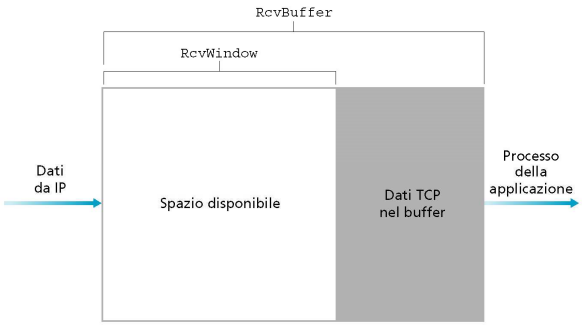



Ogni carattere generato dal client viene inviato al server che poi lo rimanda indietro per la visualizzazione.

Per non sprecare pacchetti, il TCP usa il meccanismo del Piggybacking per spedire gli ACK



## Finestra di ricezione



## Algoritmo di Nagle (RFC 896)

Il telnet produce pacchetti di dimensione minima (1 byte) con spreco di banda.

L'algoritmo di Nagle tenta di ridurre l'overhead, bufferizzando i dati da trasmettere.

```
if available_data > 0 then
  if window_size ≥ MSS & available_data ≥ MSS then
    send_a_MSS_segment
  else
    if waiting_for_an_ack == true then
      enqueue_data /* until an acknowledge is received */
    else
      send_data
    end if
  end if
end if
```



## Algoritmo di Nagle (RFC 896)

Nelle reti con basso RTT l'algoritmo di Nagle spedisce pacchetti piccoli con elevata frequenza.

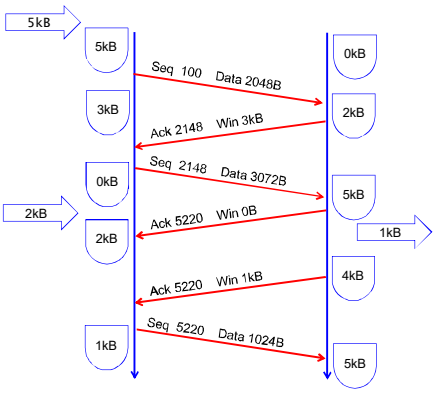
Nelle reti con RTT elevato i dati vengono bufferizzati ed i paccchetti sono molto grandi.

```
if available_data > 0 then
  if window_size ≥ MSS & available_data ≥ MSS then
    send_a_MSS_segment
  else
    if waiting_for_an_ack == true then
      enqueue_data /* until an acknowledge is received */
    else
      send_data
    end if
  end if
end if
```

A volte, per avere una elevata reattività, l'algoritmo viene disabilitato.

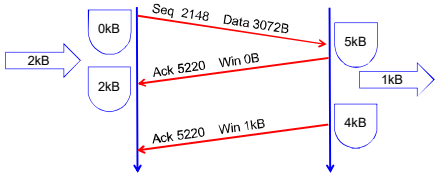


## Finestra di ricezione e ACK






## Finestra di ricezione e ACK

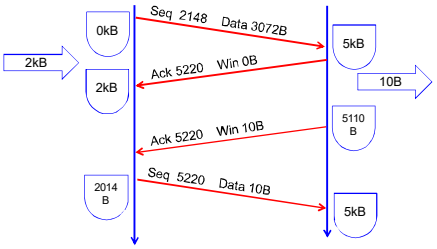



Se il messaggio di «sblocco» dovesse perdersi, il sistema rimarrebbe bloccato.

Il sender, ricevuto il pacchetto con rcv_wnd =0 avvia un timer (*Persistent Timer*), allo scadere del quale invia un pacchetto vuoto di test, per sollecitare una risposta.



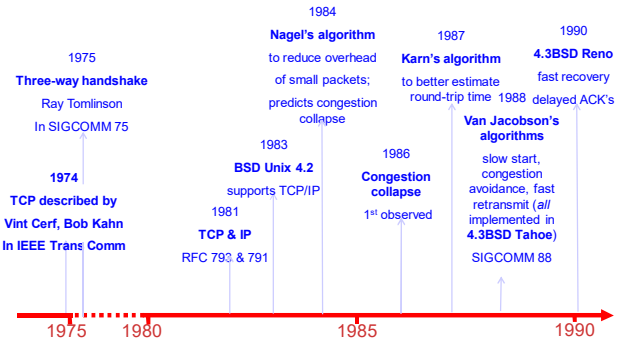
## Silly window syndrome



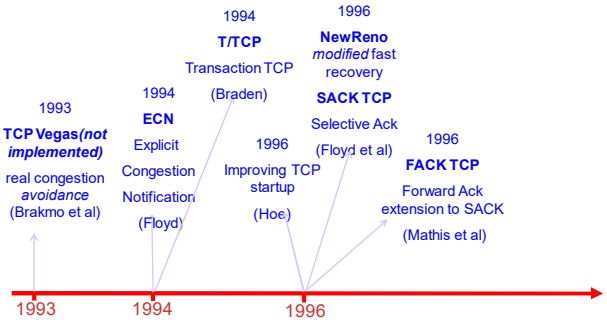

La «soluzione di Clark» prevede che il ricevente non aggiorni subito la sua *receive_windows* in caso di piccole variazioni.



## Evolution of TCP

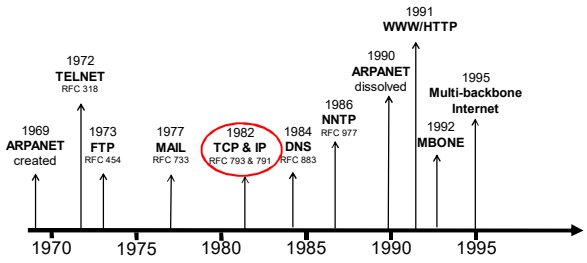


## Evolution of TCP



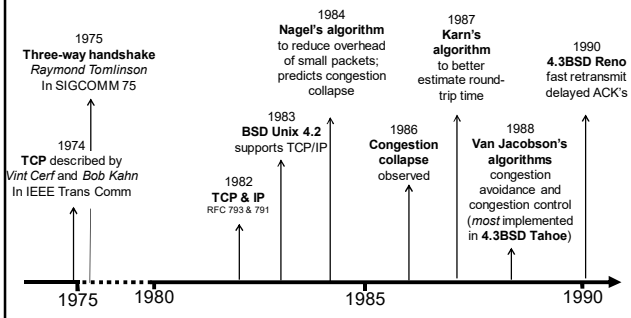



## A brief Internet history...



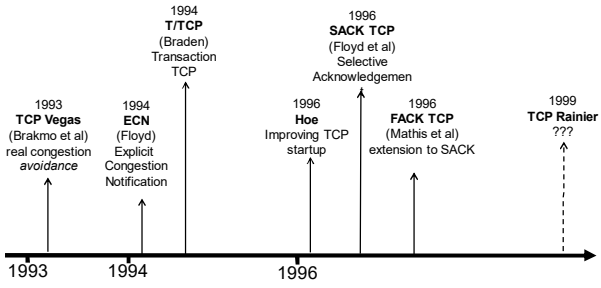



## TCP: This is your life...






## TCP: After 1990






## Tempi di Round-trip

Avere valori corretti per il timer consente di migliorare le prestazioni e diminuire la congestione di rete.

TCP effettua una stima dei tempi di Round Trip per fissare un valore di timeout.

Si usa una EWMA (Exponential weighted moving average)

$$Estimated\_RTT_n = (1-\alpha) \bullet Estimated\_RTT_{n-1} + \alpha \bullet Sample\_RTT_n$$

Tipicamente $\alpha = 0.125$



## EWMA

$E\_RTT_{n-2} = (1-\alpha)\ E\_RTT_{n-3} + \alpha\ S\_RTT_{n-2}$

$E\_RTT_{n-1} = (1-\alpha)\ E\_RTT_{n-2} + \alpha\ S\_RTT_{n-1}$

$E\_RTT_n = (1-\alpha)\ E\_RTT_{n-1} + \alpha\ S\_RTT_n$ $\quad \alpha = 0.125 \quad 1-\alpha = 0.875$

$E\_RTT_n = (1-\alpha)\ ((1-\alpha)\ E\_RTT_{n-2} + \alpha\ S\_RTT_{n-1}) + \alpha\ S\_RTT_n$

$E\_RTT_n = (1-\alpha)^2\ E\_RTT_{n-2} + \alpha\ (1-\alpha)\ S\_RTT_{n-1} + \alpha\ S\_RTT_n$

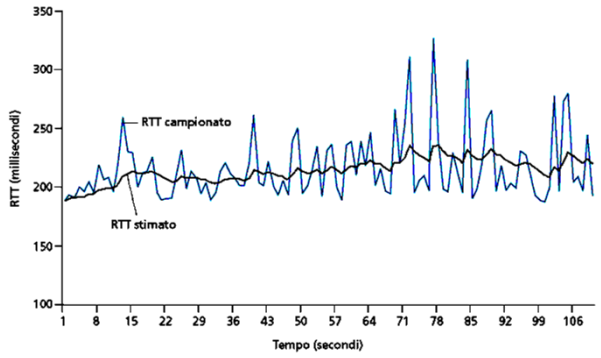
$E\_RTT_n = (1-\alpha)^2\ ((1-\alpha)\ E\_RTT_{n-3} + \alpha\ S\_RTT_{n-2}) + \alpha\ (1-\alpha) \bullet S\_RTT_{n-1} + \alpha \bullet S\_RTT_n$

$E\_RTT_n = (1-\alpha)^3\ E\_RTT_{n-3} + \alpha\ (1-\alpha)^2 \bullet S\_RTT_{n-2} + \alpha\ (1-\alpha) \bullet S\_RTT_{n-1} + \alpha \bullet S\_RTT_n$

$E\_RTT_n = \mathbf{0{,}6699}\ E\_RTT_{n-3} + \mathbf{0{,}0957}\ S\_RTT_{n-2} + \mathbf{0{,}1094}\ S\_RTT_{n-1} + \mathbf{0{,}125}\ S\_RTT_n$

## Tempi di Round-trip

Estimated RTT e Sample RTT






## Tempi di Round-trip

TCP effettua anche una stima sulla variabilità dei tempi di Round Trip.

$$DevRTT = (1-\beta) \cdot DevRTT + \beta \cdot |SampleRTT - EstimatedRTT|$$

Tipicamente $\beta = 0.25$

$$RTO = EstimatedRTT + 4 \cdot DevRTT$$

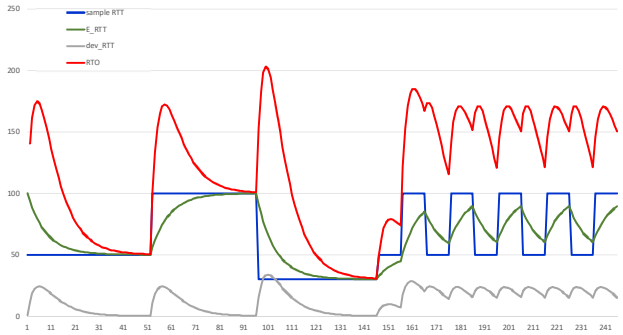
Il valore iniziale raccomandato per RTO è 1 secondo (RFC 6298)

In caso di timeout, il nuovo timer viene settato al doppio del valore precedente.

I timer crescono quindi esponenzialmente in caso di timeout.



## Timer TCP






## Timer

- TCP utilizza un solo timer per volta (raccomandazione RFC 6298), anche se ci sono più segmenti non ancora confermati.
- La ricezione di un ACK comporta l'avvio del timer per il segmento successivo non ancora confermato.

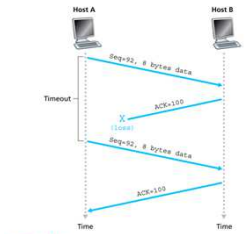



Figure 3.34 ♦ Retransmission due to a lost acknowledgment

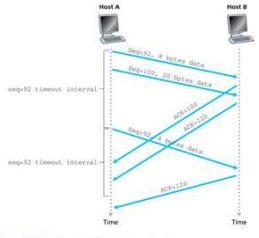



Figure 3.35 ♦ Segment 100 not retransmitted



## Timer

```
loop (forever) {
  switch(event)
    event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
        start timer
      pass segment to IP
      NextSeqNum=NextSeqNum+length(data)
      break;

    event: timer timeout
      retransmit not-yet-acknowledged segment with smallest sequence number
      start timer
      break;

    event: ACK received, with ACK field value of y
      if (y > SendBase) {
        SendBase=y
        if (there are currently any not-yet-acknowledged segments)
          start timer
      }
      break;
} /* end of loop forever */
```



## Timer

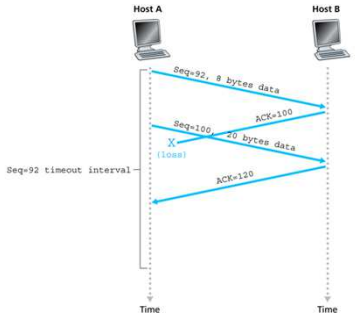


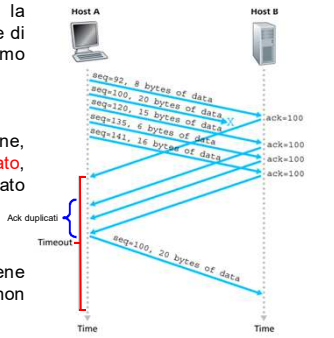

Figure 3.36 ♦ A cumulative acknowledgment avoids retransmission of the first segment.

## Fast Retransmit

Per evitare di rallentare troppo la comunicazione in caso di perdite di pacchetti, si utilizza il meccanismo detto " Fast Retransmit".

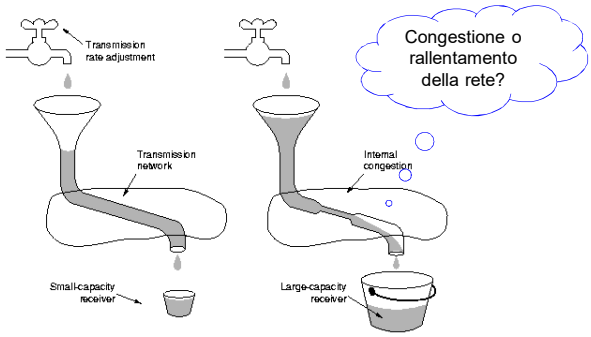
Se arriva un segmento fuori ordine, viene mandato un ACK duplicato, relativo all'ultimo segmento arrivato in ordine.

Dopo 3 ack duplicati, viene rispedito il primo segmento non confermato.



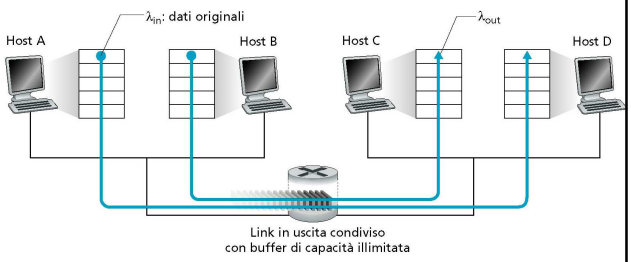



## Controllo del flusso vs Controllo della Congestione





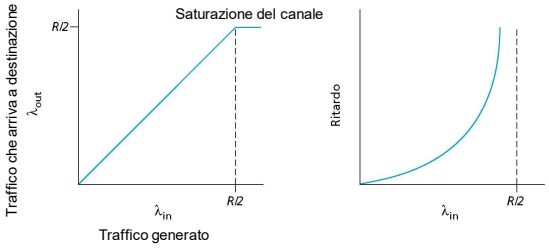

## Primo scenario: buffer illimitato





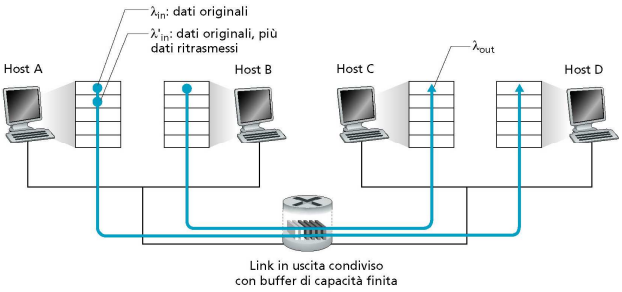

## Primo scenario: buffer illimitato

Immaginando che le due connessioni si suddividano equamente la banda a disposizione (R) avremo:






## Secondo scenario: buffer reale

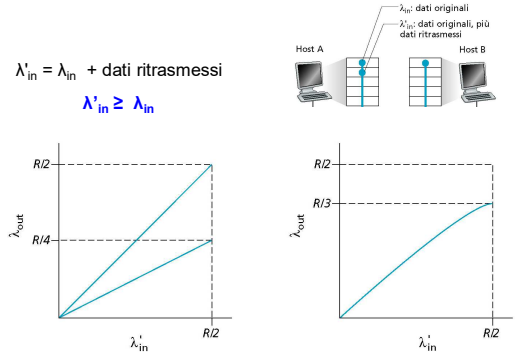




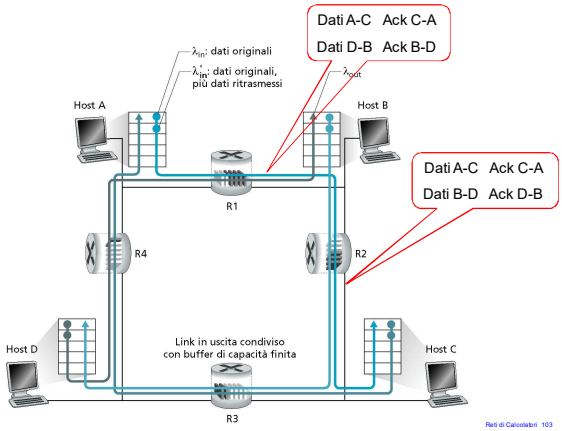

## Secondo scenario: buffer reale

$\lambda'_{in} = \lambda_{in}$ + dati ritrasmessi

$\lambda'_{in} \geq \lambda_{in}$

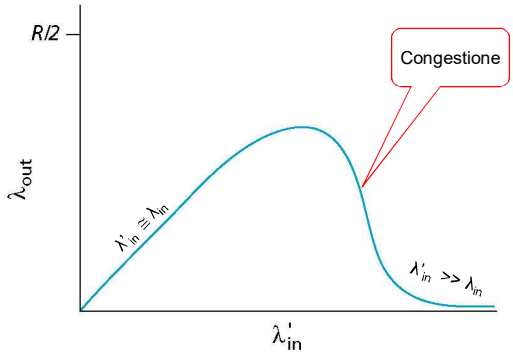


## Terzo scenario



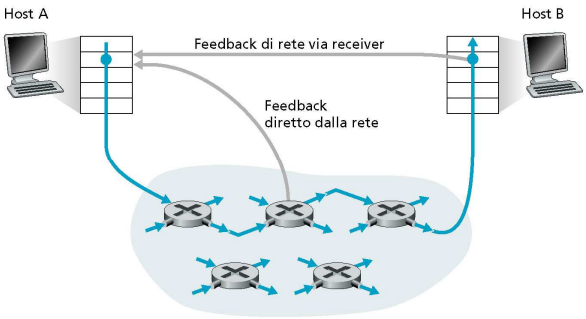



## Terzo scenario



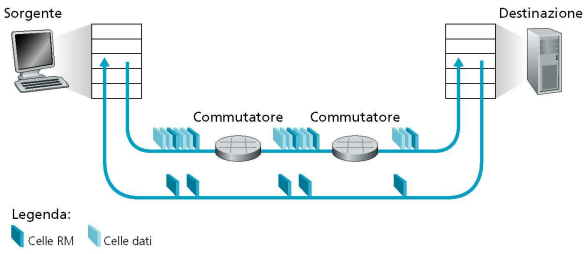



## Controllo della congestione





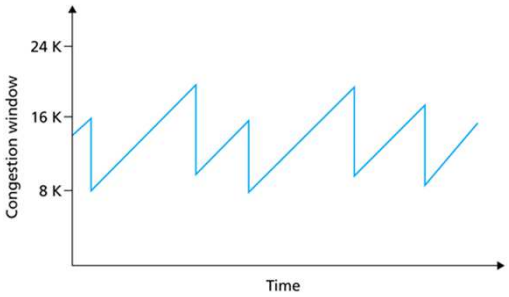

## Controllo della congestione




Controllo della congestione in ATM, affidato alla rete tramite i pkt di controllo.



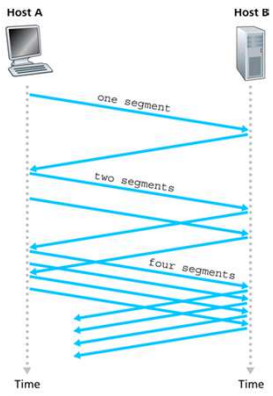
## Controllo della congestione - AIMD



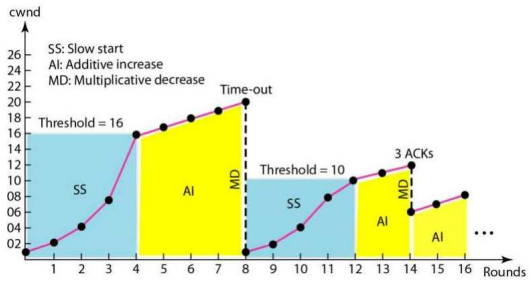

Incremento additivo – Decremento moltiplicativo



## Controllo della congestione – Slow start



## Controllo della congestione in TCP






## TCP Tahoe - TCP Reno

TCP-Tahoe: implements:

1. the slow start,
2. congestion avoidance,
3. fast retransmit algorithms.

TCP-Reno: implements:

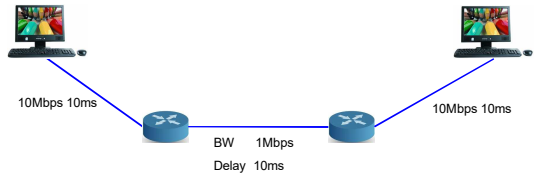
1. the slow start,
2. congestion avoidance,
3. fast retransmit,
4. fast recovery.



## Controllo del flusso e della congestione

Scenario 1

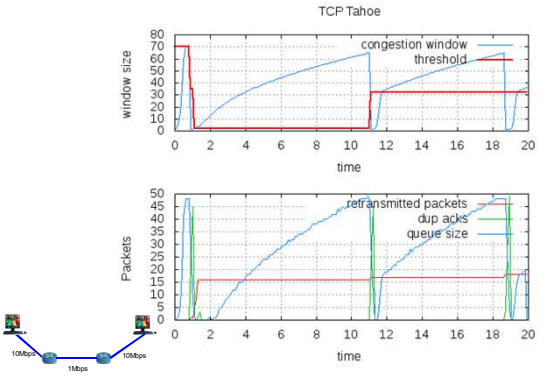
Un flusso TCP in un canale senza perdite e ritardo costante






## Controllo del flusso e della congestione

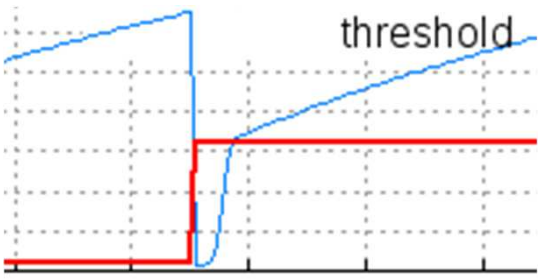
Scenario 1: Un flusso TCP in un canale senza perdite e ritardo costante






## Controllo del flusso e della congestione

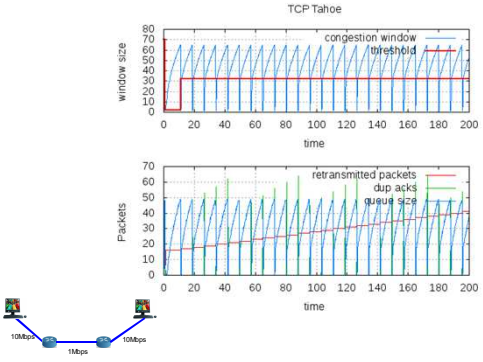
Scenario 1: Un flusso TCP in un canale senza perdite e ritardo costante





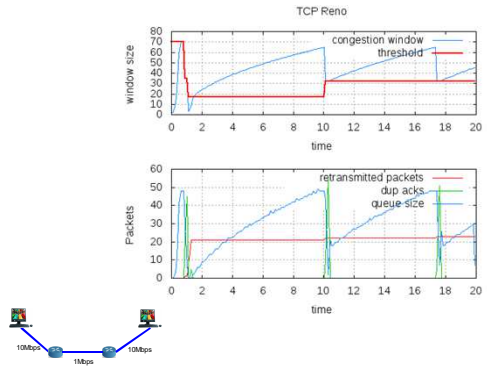

## Controllo del flusso e della congestione

Scenario 1: Un flusso TCP in un canale senza perdite e ritardo costante



## Controllo del flusso e della congestione

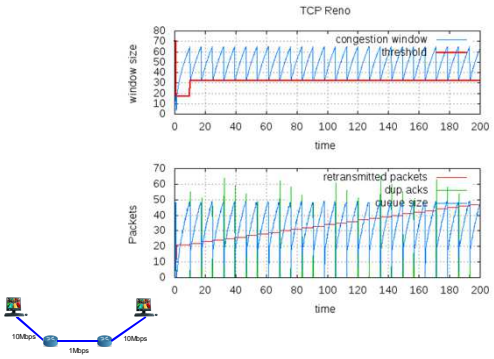
Scenario 1: Un flusso TCP in un canale senza perdite e ritardo costante






## Controllo del flusso e della congestione

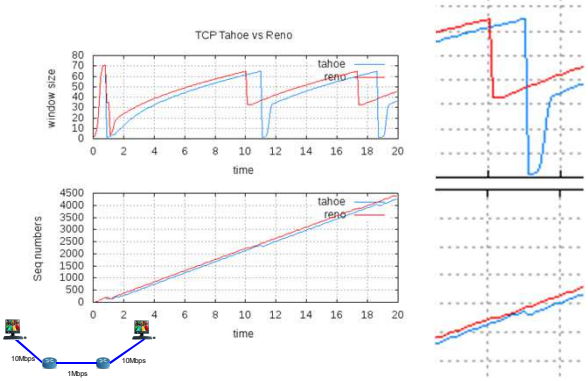
Scenario 1: Un flusso TCP in un canale senza perdite e ritardo costante






## Controllo del flusso e della congestione

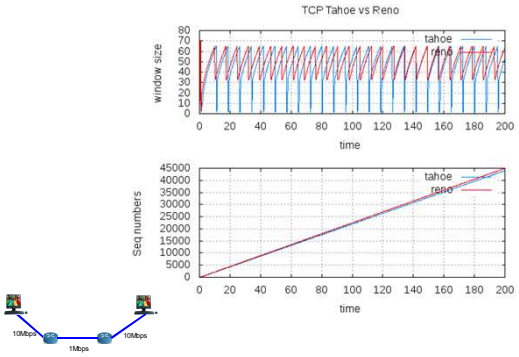
Scenario 1: Un flusso TCP in un canale senza perdite e ritardo costante






## Controllo del flusso e della congestione

Scenario 1: Un flusso TCP in un canale senza perdite e ritardo costante

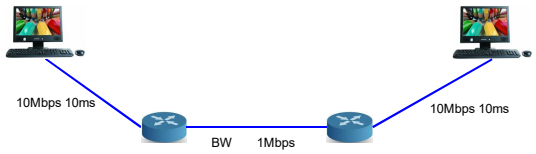





## Controllo del flusso e della congestione

Scenario 2

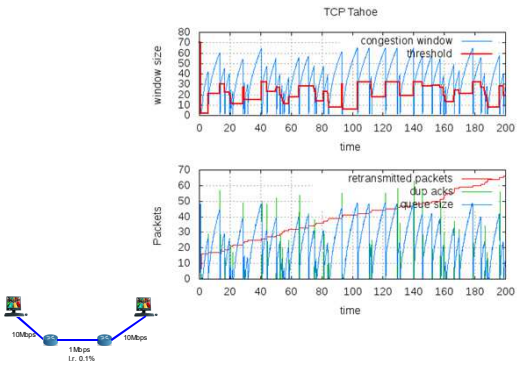
Un flusso TCP in un canale con perdite e ritardo costante





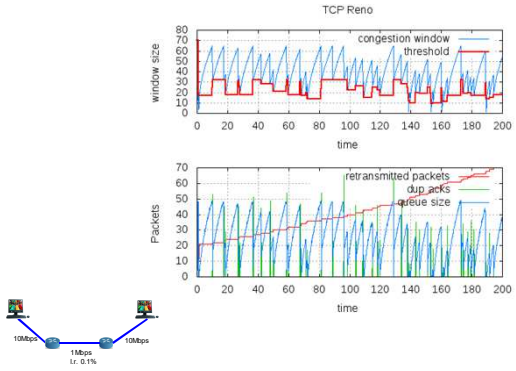

## Controllo del flusso e della congestione

Scenario 2: Un flusso TCP in un canale con perdite e ritardo costante



## Controllo del flusso e della congestione

Scenario 2: Un flusso TCP in un canale con perdite e ritardo costante

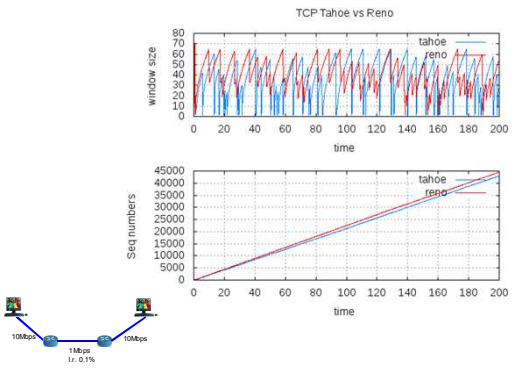







## Controllo del flusso e della congestione

Scenario 2: Un flusso TCP in un canale con perdite e ritardo costante



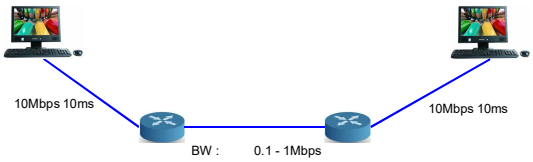





## Controllo del flusso e della congestione

Scenario 3

Un flusso TCP in un canale con perdite, bw variabile e ritardo variabile




BW : 0.1 - 1Mbps

Delay: 1 - 20ms

Loss rate: 0 - 1%

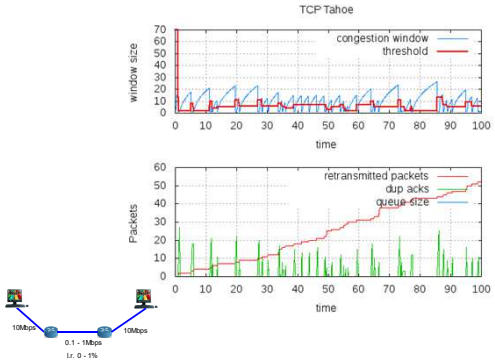
(Distribuzioni uniformi tra gli intervalli dati)





## Controllo del flusso e della congestione

Scenario 3: Un flusso TCP in un canale con perdite e ritardo variabile

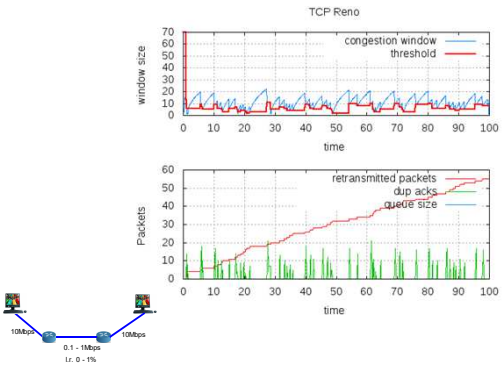







## Controllo del flusso e della congestione

Scenario 3: Un flusso TCP in un canale con perdite e ritardo variabile

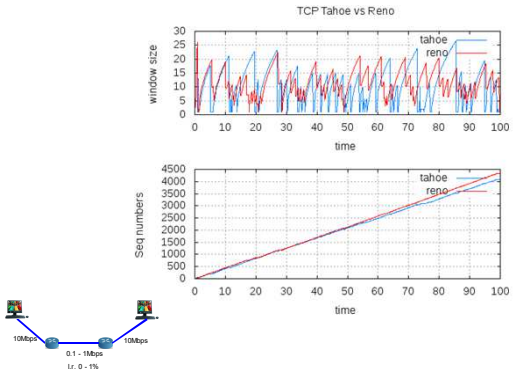







## Controllo del flusso e della congestione

Scenario 3: Un flusso TCP in un canale con perdite e ritardo variabile

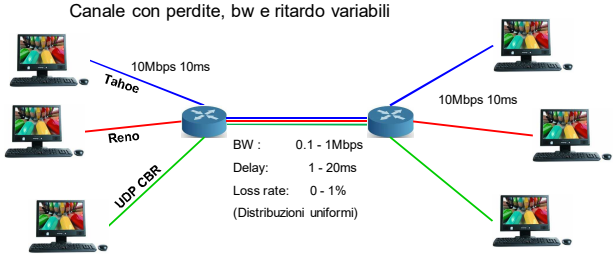


## Controllo del flusso e della congestione

Scenario 4: tre flussi concorrenti sullo stesso canale

- un flusso UDP CBR
- un flusso TCP Tahoe
- un flusso TCP Reno

Canale con perdite, bw e ritardo variabili

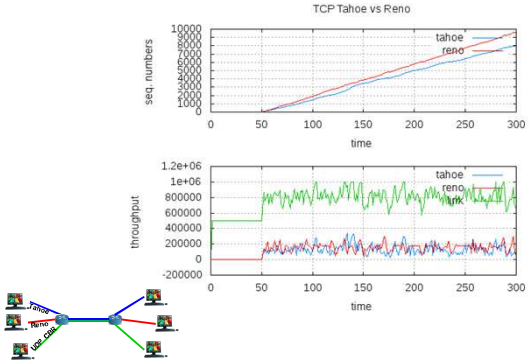





## Controllo del flusso e della congestione

Scenario 4: tre flussi concorrenti sullo stesso canale

UDP inizia a T=0s, TCP iniziano a T=50s

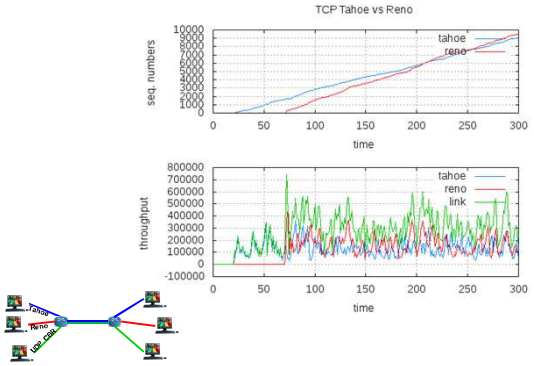





## Controllo del flusso e della congestione

Scenario 4b: 2 flussi TCP concorrenti sullo stesso canale

TCP Tahoe inizia a T=25s e TCP Reno a T=70s

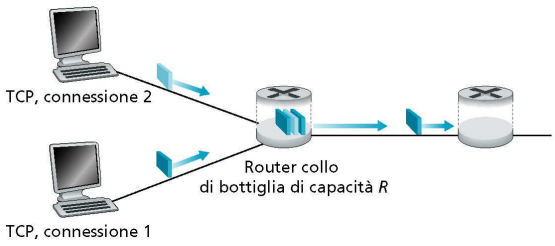





## Fairness

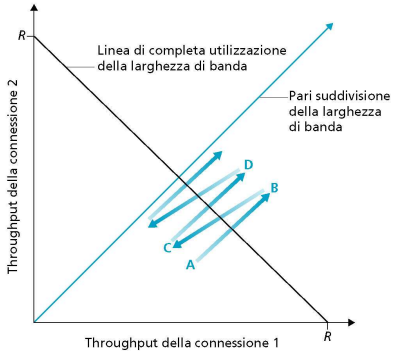
"Fairness is the quality of being reasonable, right, and just."

(Collins)






## Fairness






## Fairness