PARTE 5c

LIVELLO TRASPORTO

Modulo 9:
Sliding window

Sliding window - mittente

- **Il mittente assegna a ciascun segmento un numero di sequenza Num_segm**
- **Per il momento, si ipotizza che questo numero possa crescere a piacere e non che appartenga ad un intervallo finito. Nella realtà il range è: 0 a 2^31-1**

PRINCIPI:

- Ad ogni istante, il mittente mantiene una *finestra scorrevole* sugli indici dei segmenti, e solo quelli all'interno della finestra possono essere trasmessi

  (o sono stati spediti o sono da spedire)
- La dimensione della finestra utile del mittente è controllata dal destinatario

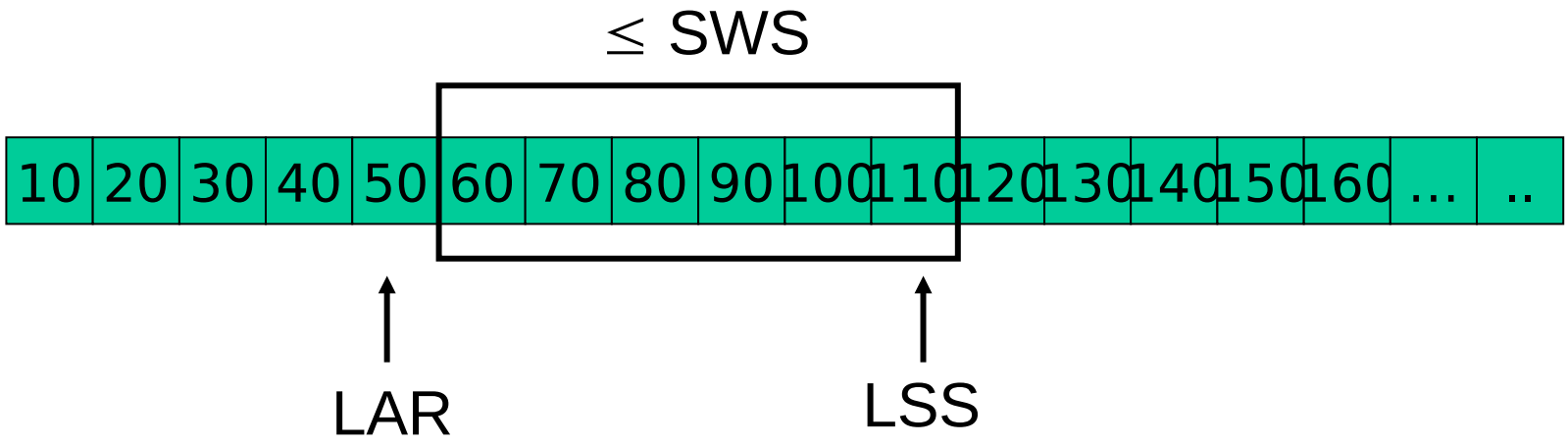
Sliding window – mittente

- **Per gestire la sliding window (finestra scorrevole), il mittente utilizza tre variabili:**
  - Dimensione della finestra di invio SWS (Sender Window Size): indica il limite superiore per il numero di segmenti che il mittente può inviare senza aver ricevuto un ACK
  - Numero di sequenza dell'ultima conferma ricevuta LAR (Last Acknowledgement Received)
  - Numero di sequenza dell'ultimo segmento inviato LSS (Last Segment Sent)

Sliding window - mittente

- **Le tre variabili del mittente devono soddisfare la seguente relazione:**

$$LSS - LAR <= SWS$$



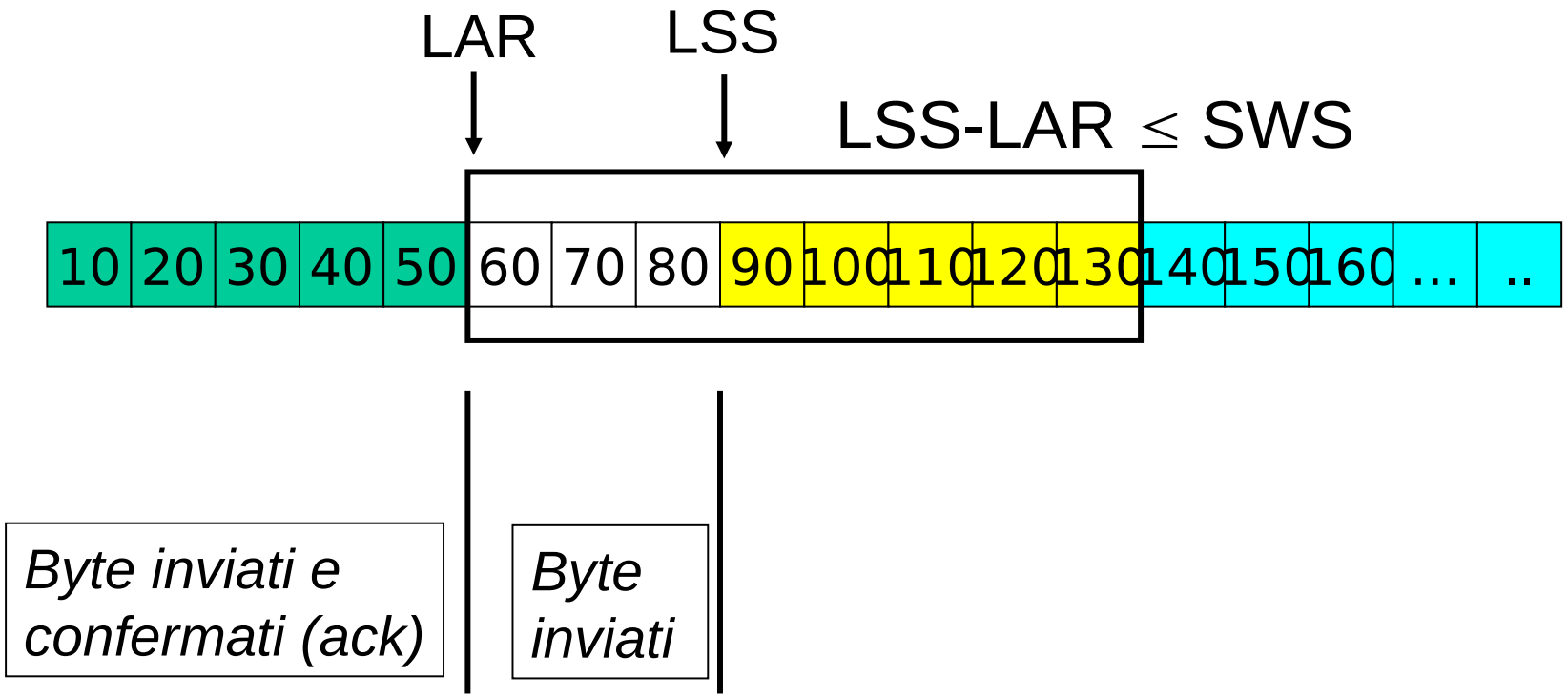
SWS: *Sender Window Size*

LAR: *Last Acknowledgement Received*

LSS: *Last Segment Sent*

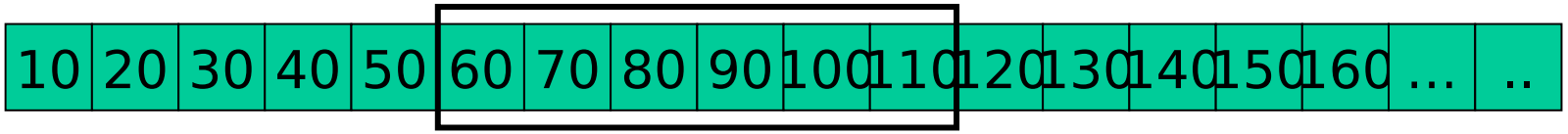
Esempio: Sliding window - mittente

- **IPOTESI: segment size di 10 byte**

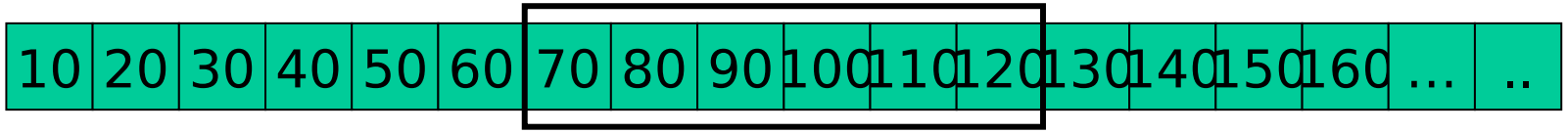


Sliding window - mittente

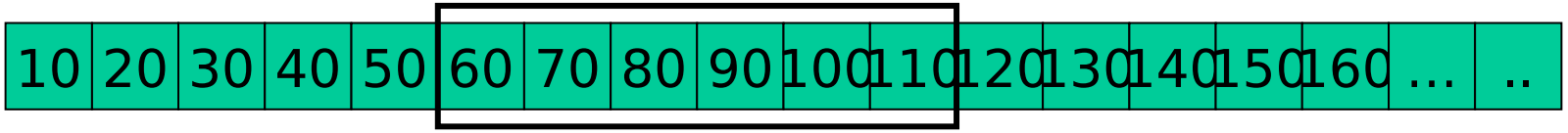
- **Quando arriva un ACK, il mittente sposta LAR verso destra, consentendo così l'invio di un altro segmento**
- **Inoltre, il mittente associa un time-out a ciascun segmento che trasmette, con conseguente ritrasmissione del segmento se il time-out scade prima di aver ricevuto il relativo ACK**



Se l'ack del segmento 60 arriva entro il time-out → trasmissione di 120



Se l'ack del segmento 60 non arriva entro il time-out → ritrasmissione di 60



Sliding window - destinatario

- **PRINCIPIO: ad ogni istante, il destinatario mantiene una finestra scorrevole sugli indici dei segmenti ricevuti**

- **La dimensione della finestra e le modalità di gestione dipendono dall'algoritmo utilizzato**

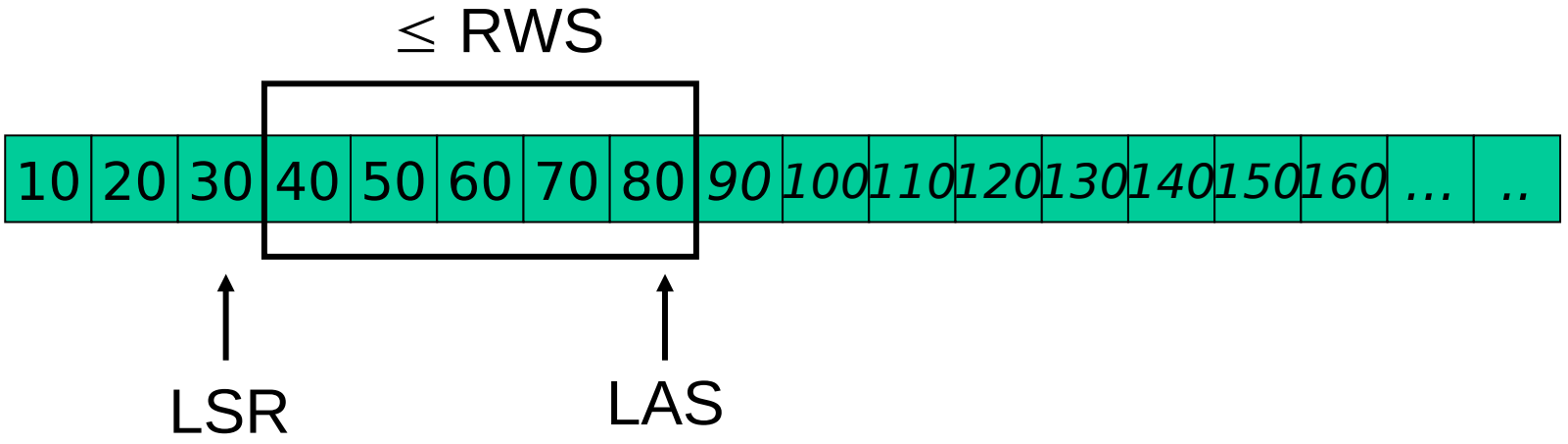
Sliding window – destinatario

- **Per gestire la sliding window, il destinatario utilizza tre variabili:**
  - Dimensione della finestra di ricezione RWS (Receive Windows Size): indica il limite superiore per il numero di segmenti “fuori sequenza” che il destinatario può accettare
  - Numero di sequenza del segmento accettabile più elevato LAS (Largest Acceptable Segment)
  - Numero di sequenza dell’ultimo segmento ricevuto “in sequenza” LSR (Last Segment Received)

Sliding window - destinatario

- **Le tre variabili del destinatario devono soddisfare la seguente relazione:**

LAS – LSR <= RWS



RWS: *Receive Window Size*

LAS: *Largest Acceptable Segment*

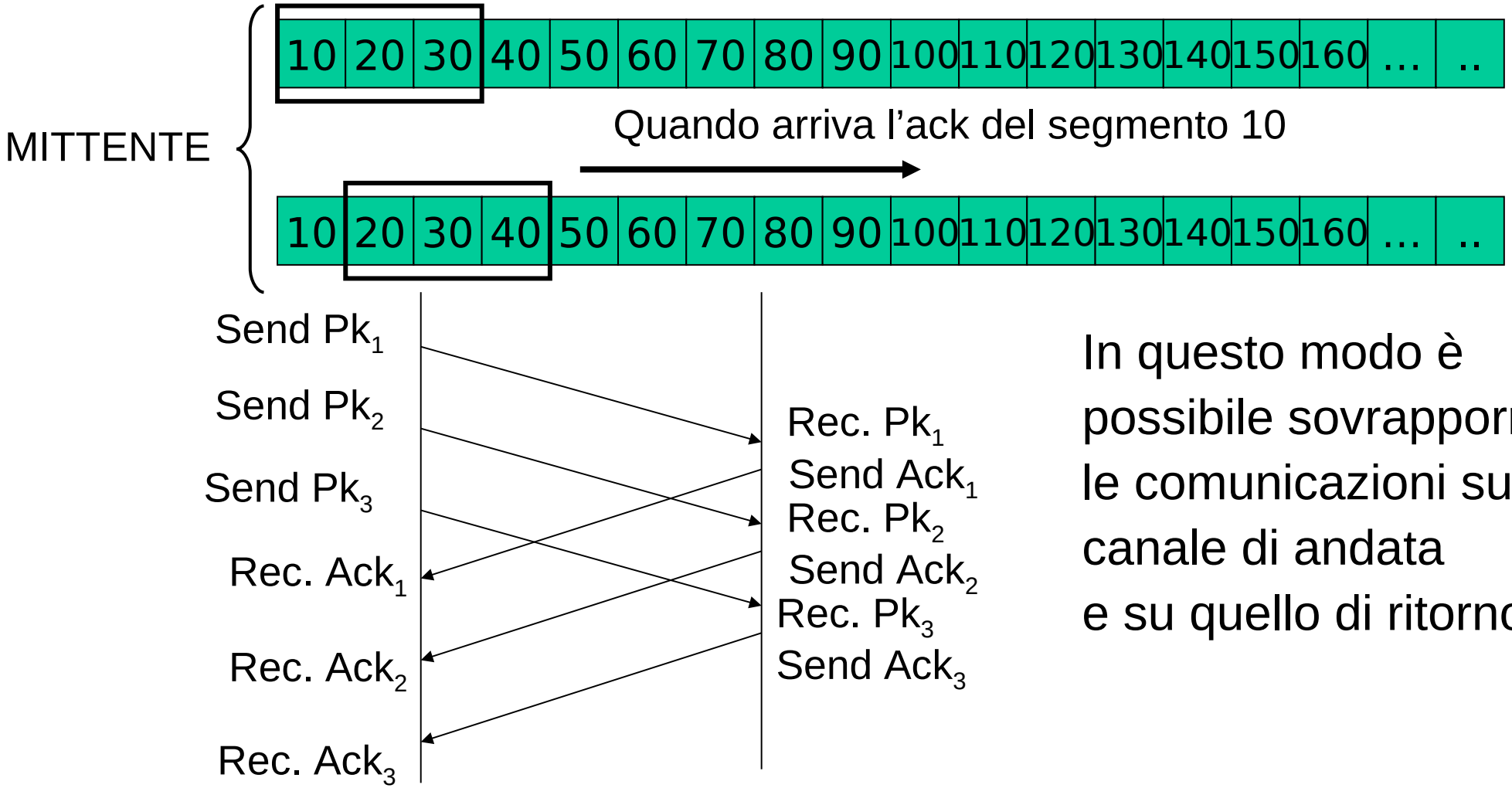
LSR: *Last Segment Received*

Sliding window - destinatario

- **Quando arriva un segmento con numero di sequenza Num_segm, il destinatario agisce come segue:**
  - Se **Num_segm≤LSR** oppure **Num_segm>LAS**, significa che il segmento si trova al di fuori della finestra del destinatario e viene scartato
  - Se **LSR<Num_segm≤LAS** , il segmento si trova all'interno della finestra del destinatario e viene inserito nel buffer

DOMANDA: Quando bisogna inviare un ACK al mittente?

Es.: finestra a scorrimento (size 3)



In questo modo è possibile sovrapporre le comunicazioni sul canale di andata e su quello di ritorno

Tutto abbastanza semplice fin quando le "cose vanno bene": cioè le trasmissioni vanno a buon fine e avvengono in modo ordinato.

Ma cosa succede se qualcosa va male …

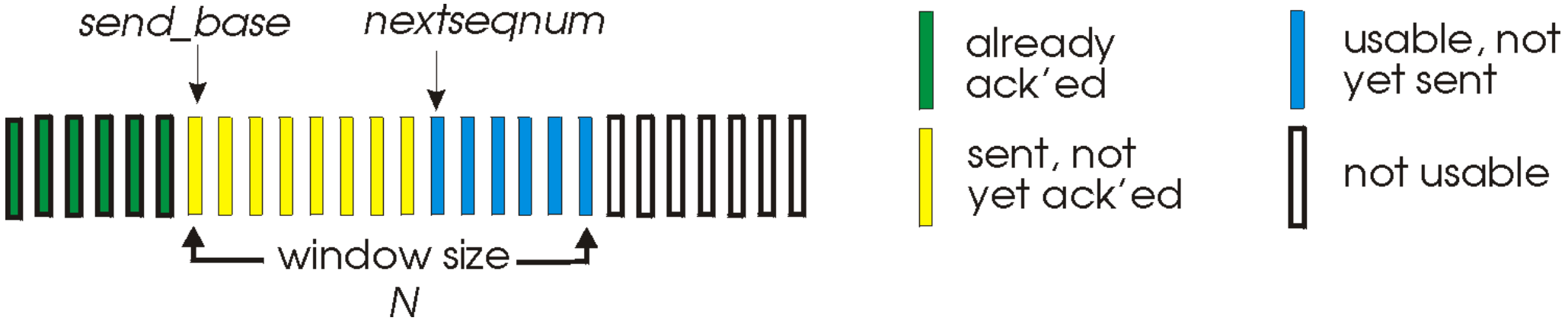
Algoritmi per l'affidabilità del pipelining

- **"Andar male" = mancato arrivo di un pacchetto ACK entro il timeout?**
- **Vi sono due "filosofie" alternative per affrontare il problema dell'affidabilità della comunicazione nel caso di un protocollo con pipelining:**
  - Go-Back-N
  - Ritrasmissione selettiva
- **NOTA: Sono algoritmi generali, non riferibili in modo specifico al protocollo TCP**

Algoritmo 1: Go-Back-N

MITTENTE

IPOTESI: Finestra (*window size*) di max *N* segmenti consecutivi, inviabili senza ACK



- Timeout per singolo segmento
- In caso di *timeout*(*i*) → il mittente deve ritrasmettere il segmento *i* e tutti i segmenti che hanno un numero di sequenza superiore ad *i*

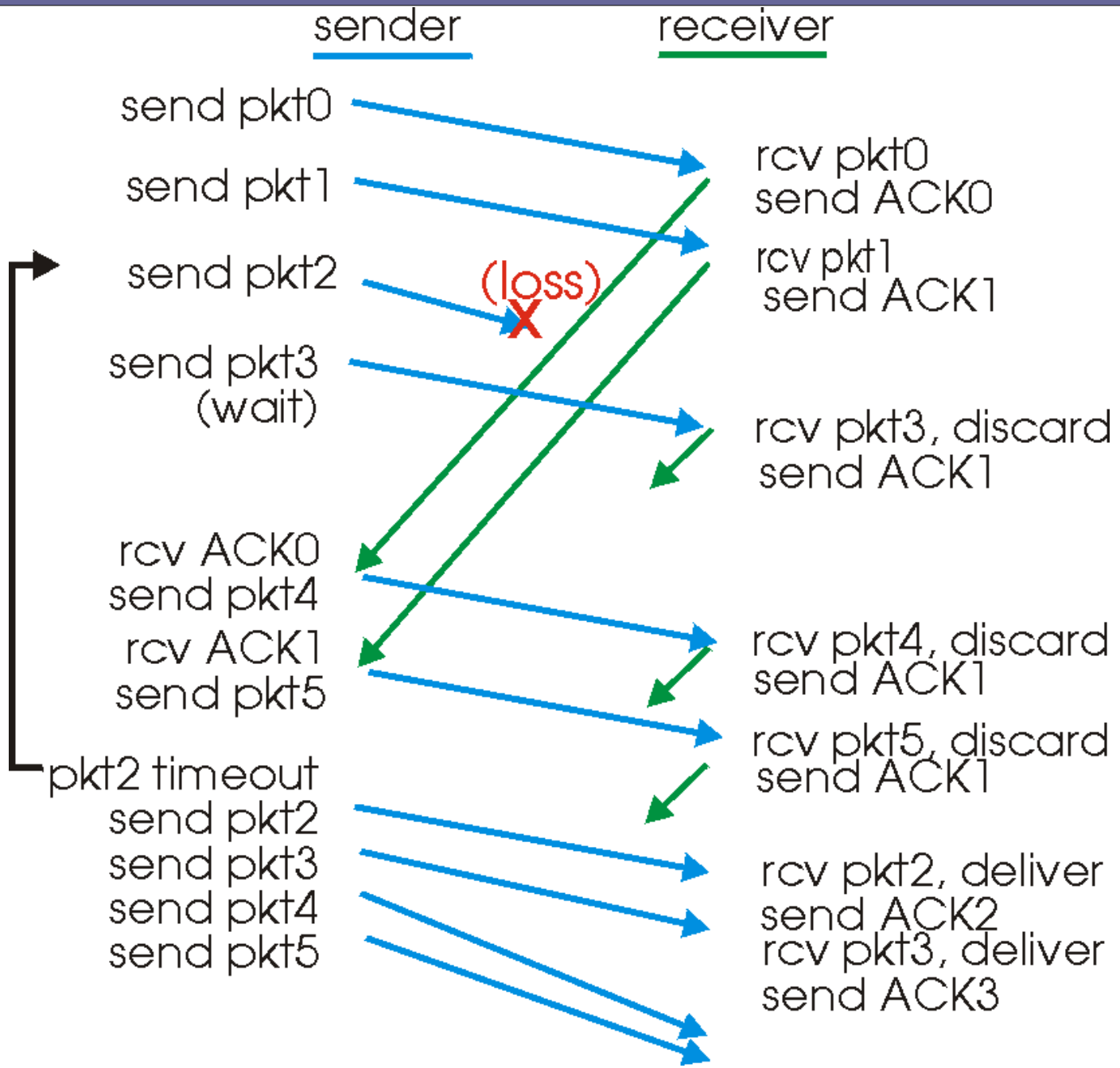
Algoritmo 1: Go-Back-N

- **Visione lato destinatario**
- **“ACK cumulativi” da parte del destinatario → ACK(n) conferma che sono arrivati correttamente i primi n segmenti**
- **Segmenti che arrivano fuori sequenza vengono scartati senza essere inseriti nel buffer**
- **Non c’è bisogno di buffer di ricezione a lato destinatario per gestire il pipeling**
- **Serve solo per gestire l’asincronia tra l’arrivo dei dati a livello di TCP (sistema operativo) e il loro consumo da parte del processo applicativo del destinatario**

Funzionamento algoritmo Go-Back-N

Ipotesi:

Finestra *N*=4

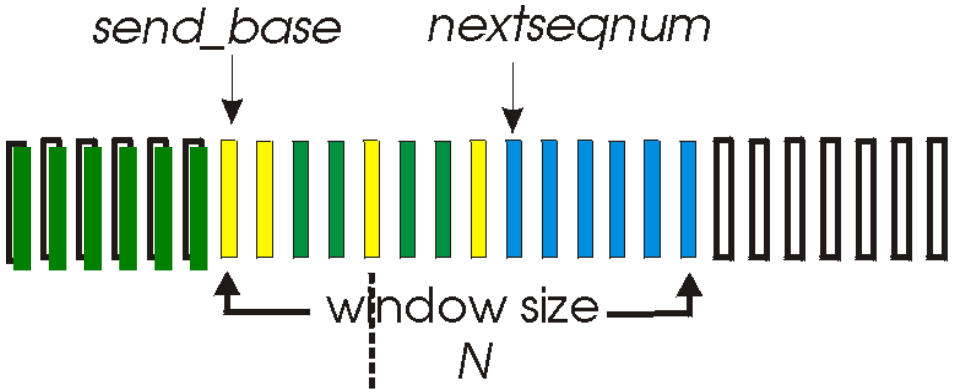


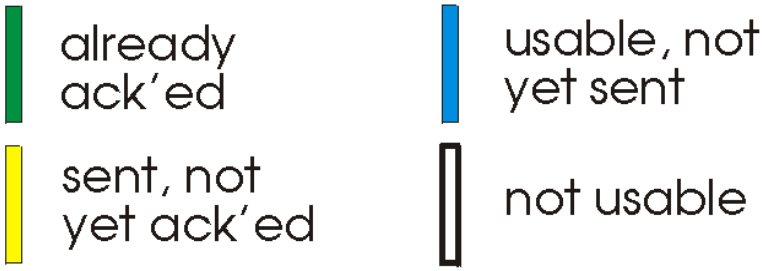
Algoritmo 2: Ritrasmissione selettiva

- **Il destinatario invia ACK di tutti i segmenti ricevuti correttamente**
  - Bufferizzazione dei pacchetti, per consegnarli secondo la sequenza ordinata al processo applicativo
- **Il mittente ritrasmette soltanto i segmenti per i quali non ha ricevuto ACK dal destinatario entro il time-out**
  - Gestisce un timeout per ciascun pacchetto
- **Mittente e destinatario gestiscono due finestre di sequenze di N segmenti consecutivi**
  - La finestra del destinatario avanza di 1 quando il segmento “base_destinatario” è ricevuto correttamente
  - La finestra del mittente avanza di 1 quando riceve un ACK entro il timeout sul segmento “base_mittente”

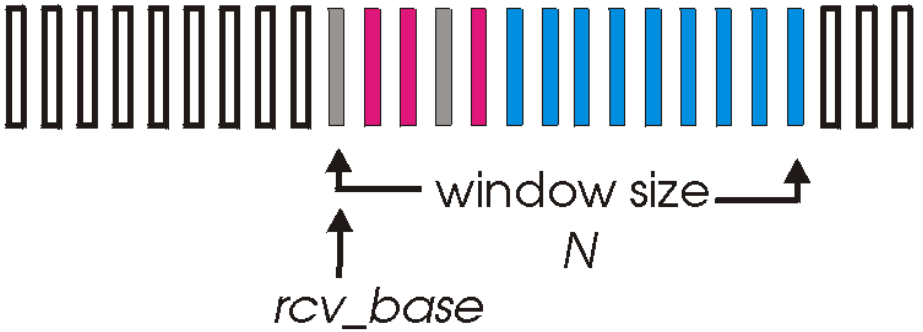
Finestra mittente e destinatario

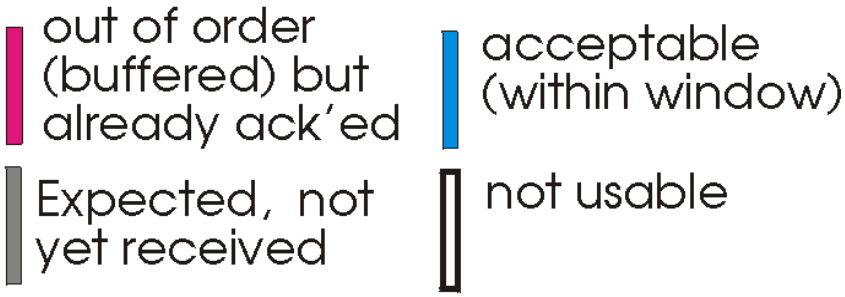
Ipotesi: finestra di dimensione *N* =14





Window del mittente





Window del destinatario

Algoritmo di Ritrasmissione selettiva

mittente

Dati dall'applicazione:

- se c'è un numero di sequenza disponibile nella finestra, invia segmento

Timeout(*i*):

- ritrasmetti segmento *i*, inizializza nuovo timer per *i*

ACK(*i*) nella finestra [send_base, send_base+*N*]:

- segna segmento *i* ricevuto
- se *i* è il segmento "base" non ancora ACK, incrementa la finestra fino al successivo pacchetto non ACK

destinatario

Segmento *i* ricevuto in [rcv_base, rcv_base+*N*-1]

- invia ACK(*i*)
- *non ordinato*: metti in buffer
- *ordinato*: consegna al processo applicativo; avanza la finestra al segmento successivo non ancora ricevuto

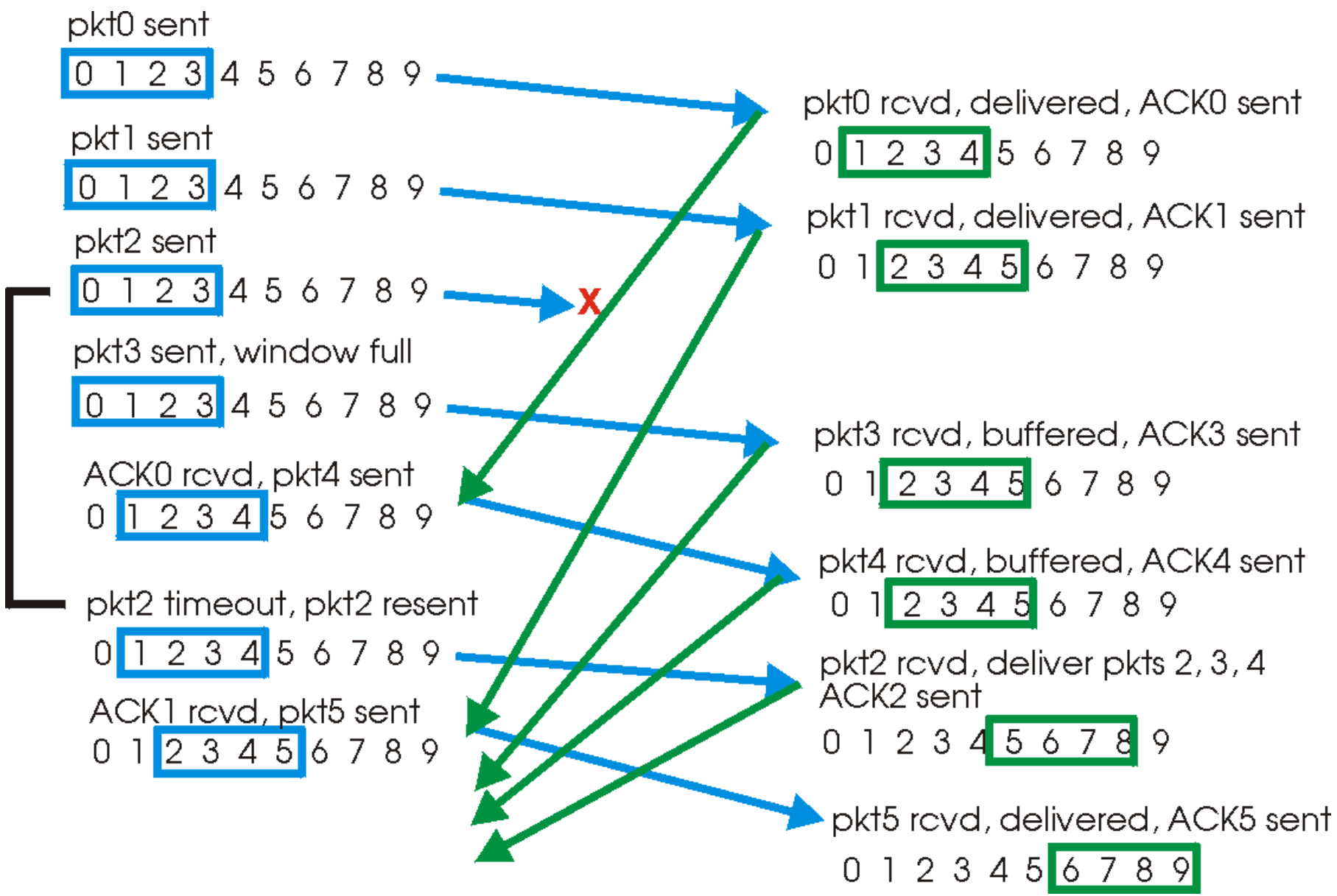
Segmento *i* ricevuto in [rcv_base-*N*, rcv_base-1]

- ACK(*i*)

Altrimenti:

- ignora

Funzionamento Ritrasmissione selettiva



Algoritmo del protocollo TCP

- **Sebbene sia stata proposta una modifica al protocollo TCP [RFC 2018] per andare verso un meccanismo di ritrasmissione selettiva, attualmente il TCP non segue nessuna delle due versioni (Go-Back-N e Ritrasmissione Selettiva) in modo puro, in quanto utilizza**

Algoritmo del protocollo TCP

- **“ACK cumulativi” da parte del destinatario**
  - ACK(n) conferma che sono arrivati correttamente i primi n byte dei segmenti dati inviati
  - come Go-Back-N
- **Segmenti arrivati fuori ordine, vengono salvati nel buffer di ricezione**
  - come Ritrasmissione Selettiva

Algoritmo del protocollo TCP

- **Mittente e destinatario gestiscono due finestre di sequenze di segmenti da inviare e ricevuti**
- **Il destinatario invia un ACK cumulativo relativo all'ultimo byte dell'ultimo segmento ricevuto senza errori ed "in sequenza"**
- **Il mittente ritrasmette soltanto i segmenti per i quali non ha ricevuto ACK dal destinatario entro il time-out**
- **Sia la window del destinatario sia quella del mittente possono avanzare di d posizioni (d>=1)**

Algoritmo del protocollo TCP (es.)

- **LSR=5, cioè l'ultimo ACK inviato dal destinatario è relativo al segmento 5**
- **RWS=4 (receiver window size) e quindi LAS=9**
- **Se dovessero arrivare i segmenti 7 e 8, verrebbero memorizzati nel buffer perché si trovano all'interno della finestra del destinatario, ma poiché non è arrivato ancora il segmento 6, a seconda dell'implementazione del TCP:**
  - non verrebbe inviato alcun ACK oppure
  - verrebbe inviato nuovamente ACK(5)

Algoritmo del protocollo TCP (es.)

- **Quando arriva il segmento 6, il destinatario può inviare ACK(8), confermando la ricezione corretta di 6, 7 e 8 (cioè d=3), consentendo quindi di settare LFS=8 e LAS=12**

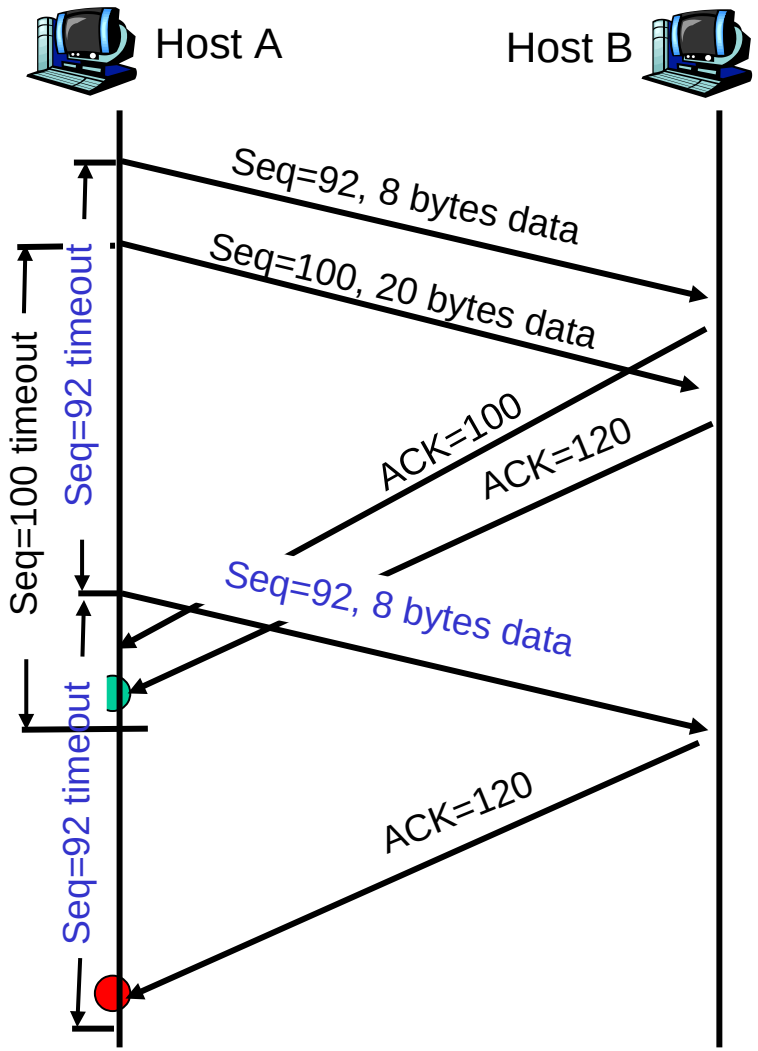
Algoritmo del protocollo TCP (conseguenze)

- **Si evita la ritrasmissione di segmenti ricevuti correttamente che si verificava nel caso di Go-Back-N**
- **Si sfruttano tutti gli ACK per comprendere che i segmenti sono stati ricevuti correttamente e, quindi, in caso di trasmissioni corrette, si velocizza l'avanzamento della finestra di spedizione (vedi esempi successivi)**

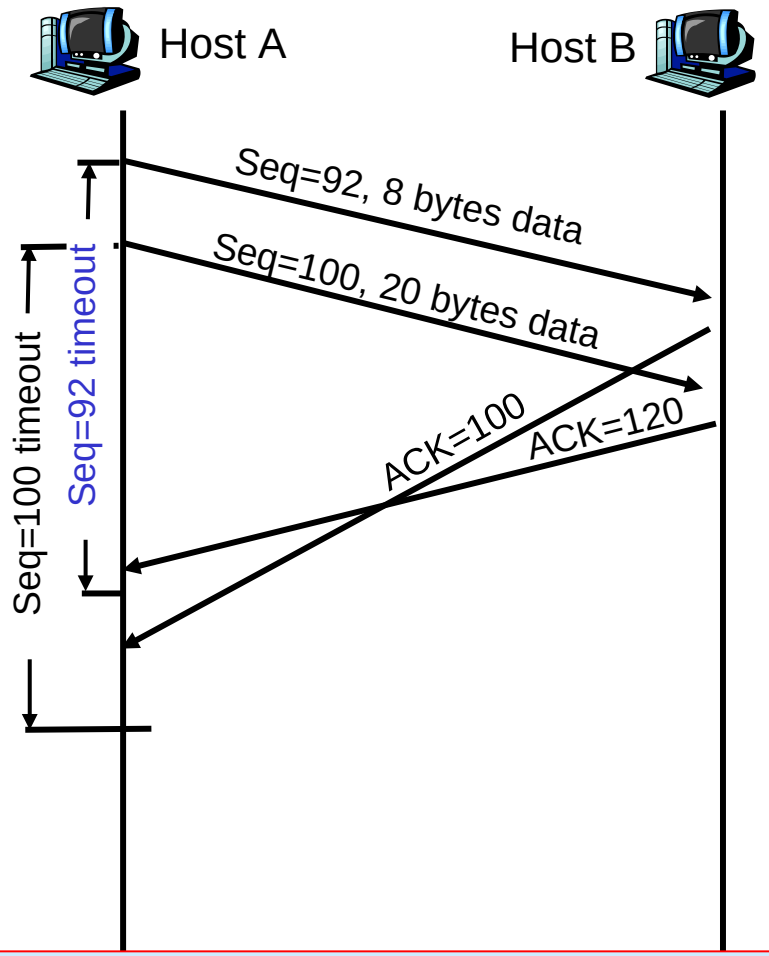
Algoritmo del protocollo TCP (conseguenze)

- **In caso di time-out, la quantità di dati trasmettibile diminuisce perché il mittente non è in grado di far avanzare la finestra ed il buffer risulta occupato da altri segmenti. Più tempo occorre per accorgersi che un segmento è andato perduto, più si limita la capacità della banda di trasmissione**
  - → necessità di comprendere problemi al più presto

Scenari di ritrasmissione per ritardi (protocol pipelining)



L'ack=100 arriva, ma oltre il timeout prestabilito. L'host A deve rinviare il segmento. Doppia duplicazione: segmento a host B, ack=120 a host A



Caso molto particolare in cui ack=100 non arriva entro il timeout prestabilito, mentre ack=120 arriva addirittura entro il timeout del segmento 92.
NON C'E' RITRASMISSIONE! Perché?

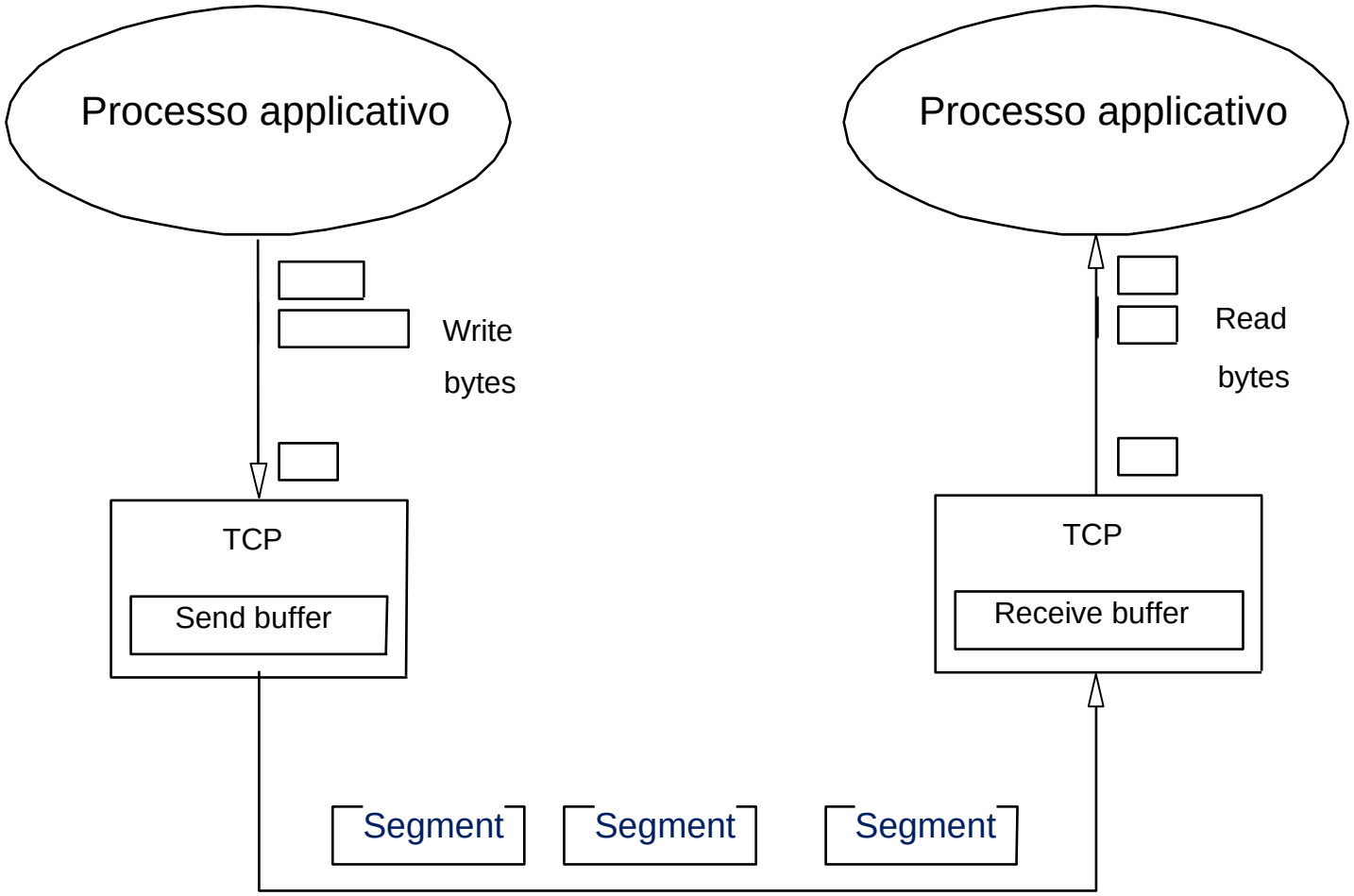
Altri due problemi da risolvere nel TCP

- **Il TCP è il livello che deve evitare di spedire più segmenti di quanti il destinatario sia in grado di riceverne**
  - "Controllo di flusso"

- **Il TCP è il livello che deve evitare di spedire più segmenti di quanti la rete tra mittente e destinatario sia in grado di assorbirne**
  - "Controllo di congestione"

Modulo 10:
Controllo di flusso

Trasmissione dati nel TCP

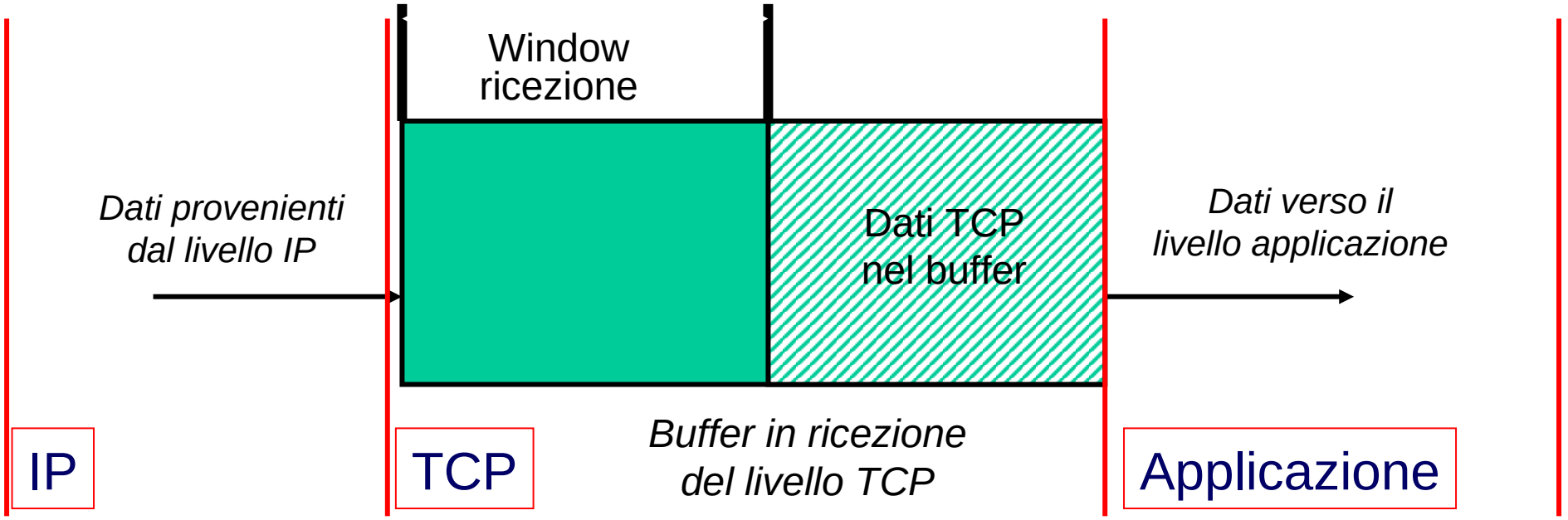


Trasmissione segmenti

Controllo di flusso

Obiettivo: il mittente non deve riempire il buffer del destinatario inviando una quantità eccessiva di dati ad un tasso di trasmissione troppo elevato

Il destinatario informa esplicitamente il mittente della quantità di spazio libero nel buffer ricezione TCP. La dimensione della finestra nel segmento TCP varia dinamicamente per cui ogni segmento ACK informa il mittente del numero massimo di byte ricevibili



Controllo di flusso a livello di end-point

- **L’ack è inviato a livello di segmento, per cui un solo ack**
  - vale per molti byte
- **L’ack porta due informazioni:**
  - Quanti byte sono stati ricevuti dal destinatario
  - Quanti byte il destinatario può ancora ricevere (in quel momento)

Controllo di flusso a livello di end-point

- **Il mittente regola la sua finestra in base alla disponibilità che gli è stata indicata dal destinatario**
  - Se il destinatario ha esaurito il buffer, indica una disponibilità pari a 0
- **In tal caso, la trasmissione si sospende e riprende solo quando il destinatario segnala di essere nuovamente in grado di ricevere byte**

Dimensione della finestra effettiva

- **AdvertisedWindow: dimensione della finestra massima di ricezione comunicata dal destinatario (=quantità di spazio libero nel buffer di ricezione)**

AdvertisedWindow = MaxRcvBuffer –

((NextByteExpected-1)-LastByteRead)

- **EffectiveWindow: il mittente calcola la finestra effettiva che limita la massima quantità di dati che possono essere inviati (il mittente sa che ha già spedito dei segmenti):**

EffectiveWindow = AdvertisedWindow –
(LastByteSent - LastByteAcked)

Esempio: blocco processo

- **Il mittente può spedire dati solo se EffectiveWindow>0**
- **Quindi, è possibile che arrivi un segmento che conferma x byte, consentendo al mittente di aumentare LastByteAcked di x, ma nell'ipotesi che il processo applicativo ricevente non stia leggendo dati dal buffer, viene anche comunicata una AdvertisedWindow di x byte più piccola di prima**
- **→ Il mittente può liberare un po' di spazio nel suo buffer, ma non può spedire altri dati.**

Esempio: blocco processo

- **Se la situazione persiste, può capitare che il buffer mittente si riempia ed il TCP deve bloccare la generazione di nuovi dati da parte del processo**
- **→ CONSEGUENZA: Un processo applicativo lento sul computer destinatario può provocare il blocco di un processo mittente veloce!**

Come riprendere da una windows=0

- **Una volta che il destinatario ha comunicato una finestra di dimensione 0**

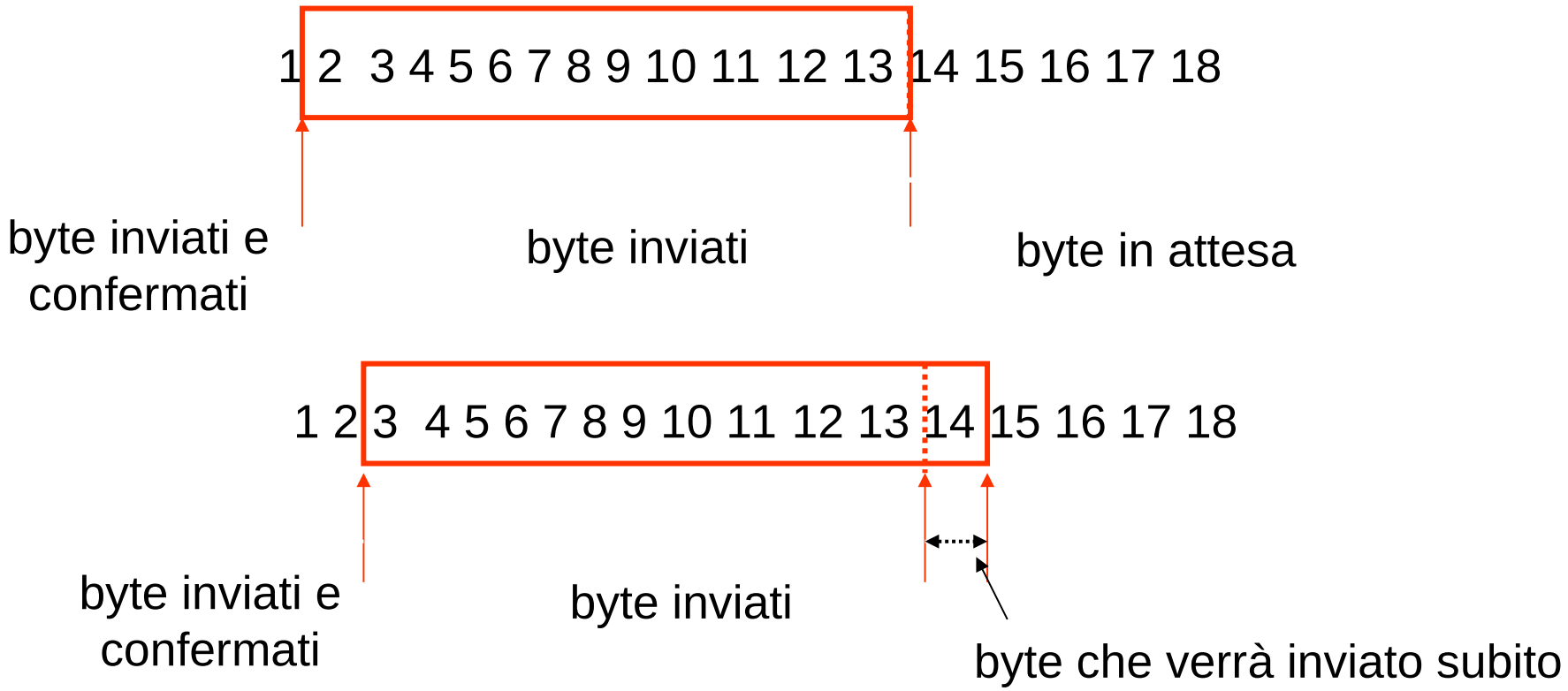
  → Il mittente non può più inviare dati

  → Il mittente non può più conoscere se la finestra di ricezione è aumentata perché il destinatario non invia messaggi spontaneamente, ma solo in risposta a segmenti in arrivo

SOLUZIONE TCP: Quando il mittente riceve una AdvertisedWindow=0, continua ad inviare periodicamente un segmento con 1 byte di dati

Esempio: Sindrome della finestra cattiva

Che cosa succede se il ricevitore dà l'ACK appena ha un po' di spazio libero (ad esempio un byte)?



Possibili contromisure

LATO DESTINATARIO

- **Il destinatario aspetta di avere svuotato almeno il 50% del “buffer” prima di inviare un ACK di restart**
- **In alternativa, il destinatario ritarda l’invio dell’ACK sistematicamente quando il buffer disponibile si riduce troppo (raccomandato dagli standard)**
- **Il rischio è di attendere troppo e che il mittente ri-invii i dati che vanno in time-out**
- **Inoltre, questo approccio altera la stima di RTT**

LATO MITTENTE

- **Il mittente aspetta di avere abbastanza dati per riempire un segmento di dimensione MSS**
- **Se però, mentre attende, arriva un ACK, invia comunque un segmento (più corto)**

Ritrasmissione rapida

- **Nel caso in cui il time-out sia abbastanza lungo, si può ritardare molto la necessaria ritrasmissione di un segmento, con conseguenti limitazioni a livello di buffer mittente e destinatario**
- **In alcune versioni di TCP si implementa un meccanismo aggiuntivo per rilevare la perdita dei pacchetti prima che si verifichi un time-out:**
  - → ACK duplicato

Ritrasmissione rapida

- **Quando il destinatario riceve segmenti fuori ordine, continua a spedire ACK relativi all'ultimo byte del segmento dati che ha ricevuto correttamente ed in sequenza ordinata**
- **Poiché il mittente invia, in genere, molti segmenti con una certa continuità, la perdita di un segmento causerà l'invio di molti ACK duplicati**
- **Triplo ACK replicato dello stesso segmento → del tutto simile ad un "not ack" (NACK) sul segmento successivo**

Modulo 11:
Controllo di congestione

Controllo di congestione

- **Congestione (def.):**
  - Informalmente: “un numero elevato di sorgenti inviano contemporaneamente troppi dati generando un traffico che la rete (Internet) non è in grado di gestire”
- **Congestione diverso da Controllo del flusso!**
- **Effetti della congestione:**
  - perdita di pacchetti (overflow dei buffer nei router)
  - lunghi ritardi (tempi di attesa nei buffer dei router)

Due approcci per il controllo congestione

- **Controllo di congestione end-to-end**
  - il livello di rete non fornisce un supporto esplicito al livello di trasporto
  - la situazione di congestione è determinata analizzando le perdite di pacchetti ed i ritardi nei nodi terminali
  - approccio utilizzato dal TCP

Due approcci per il controllo congestione

- **Controllo di congestione assistito dalla rete**
  - i router forniscono un feedback esplicito ai nodi terminali riguardante lo stato di congestione nella rete
  - misura della congestione nei router: lunghezza della coda dei buffer
  - singolo bit che indica la congestione di un link (es., controllo di congestione in reti ATM ABR)
  - feedback diretto oppure aggiornando un campo del pacchetto che viaggia tra i nodi terminali

Soluzioni TCP

- **CONTROLLO DI CONGESTIONE “END-TO-END”**
  - (cioè, regolato da mittente e destinatario, non dalla rete)
- **“SLOW START”**
- **AIMD (sulla congestion window)**
  - Additive Increase - Multiplicative decrease

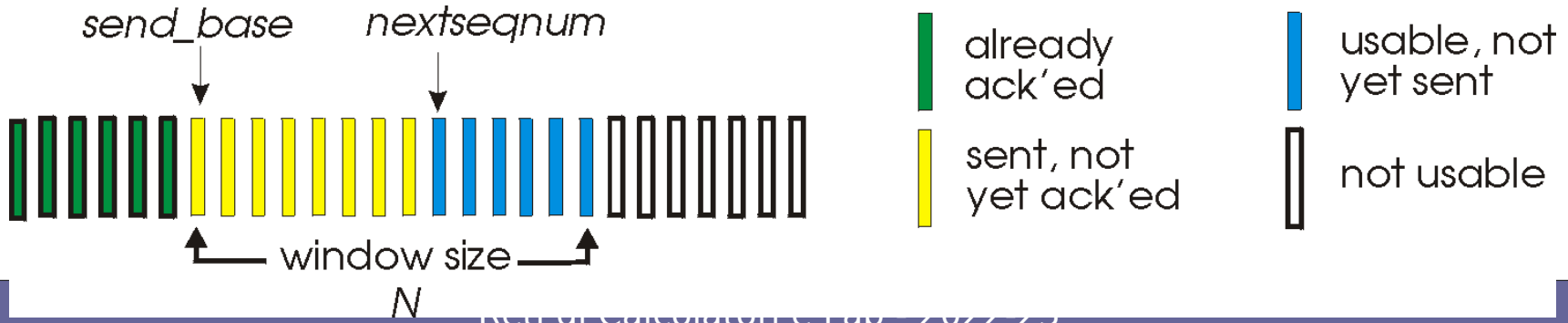
Esempio

- Aumenta la window size linearmente per ACK arrivato entro timeout (oppure differenzia l’inizio dai passi successivi)
- Diminuisce la window di un fattore moltiplicativo in caso di perdita (es., dividi la window size di 2)

Controllo di congestione in TCP

- SCELTA: Congestione end-to-end
  - nessuna informazione proveniente dalla rete
- Timeout e ritrasmissione contribuiscono ad aumentare la congestione
  - collasso della congestione
- TCP riduce il tasso di trasmissione in caso di congestione
  - CongestionWindow (CW): dimensione della finestra di congestione
  - AdvertisedWindow (AW): dimensione della finestra di ricezione

Advertised window* = min(AW, CW)



Controllo di congestione in TCP

- Misura delle prestazioni di una connessione TCP:

  - Throughput (tasso di trasmissione)

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

w = numero di segmenti nella finestra

MSS = dimensione massima del segmento

RTT = Round Trip Time

- Valutazione della banda disponibile:
  - idealmente: trasmettere il più velocemente possibile senza perdita di segmenti (valore massimo di CongestionWindow)
  - incrementare CongestionWindow il più possibile, finché non si verifica un episodio di congestione
  - diminuire CongestionWindow, ricominciando poi a incrementarla nuovamente

Controllo di congestione in TCP

Tecniche per il controllo della congestione nel TCP

- **Evitare la congestione (congestion avoidance)**
  - stato stazionario (non di congestione): la dimensione della finestra è pari a quella indicata dal ricevente (per il controllo di flusso)
  - stato di congestione: riduzione della dimensione della finestra

Controllo di congestione in TCP

Tecniche per il controllo della congestione nel TCP

- **Slow-start**
  - all'inizio dell'utilizzo di una nuova connessione o in seguito a congestione di una connessione la dimensione della finestra è pari a 1
  - incremento progressivo (esponenziale) della dimensione della finestra
  - threshold: valore della dimensione della finestra, raggiunto il quale la fase di incremento esponenziale termina e si raggiunge lo stato stazionario di incremento lineare

Slow start del TCP

Algoritmo di slow-start

Inizializza: CW = 1

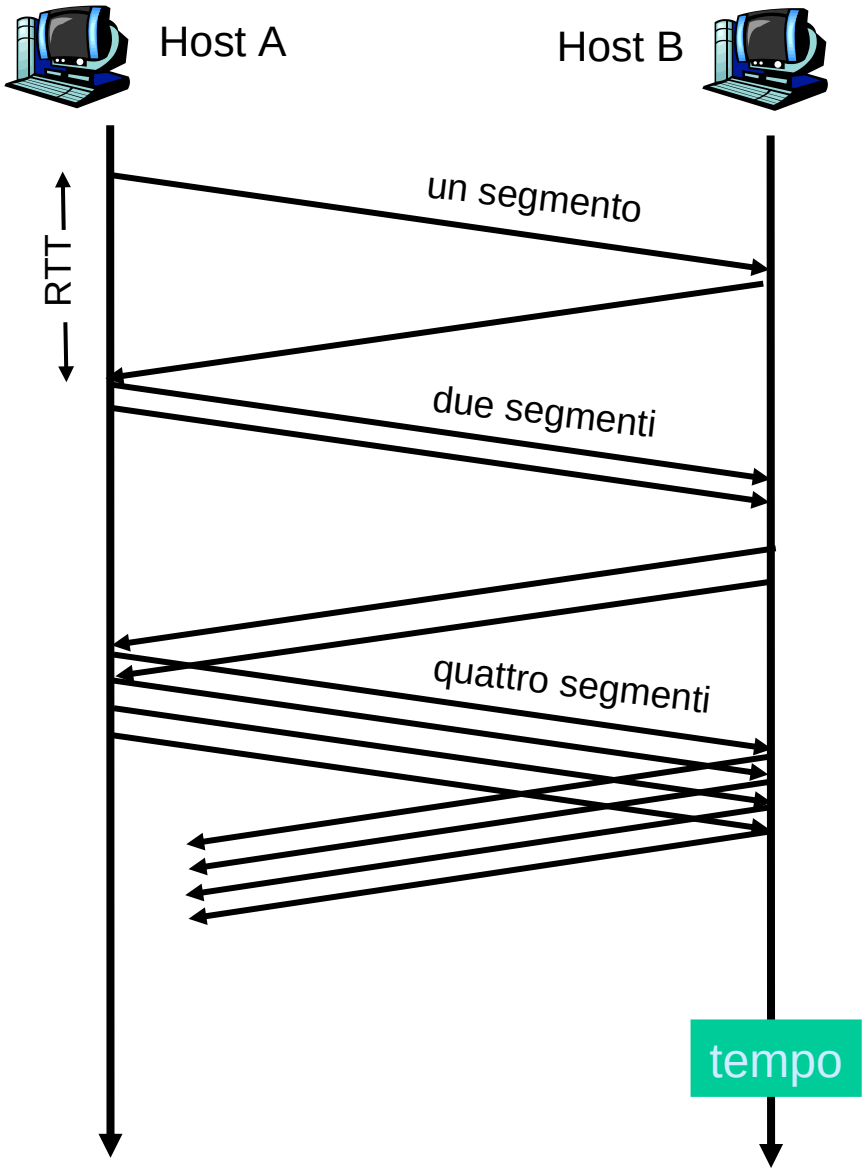
for (ogni segmento ACK)

CW++

until ((timeout OR

(CW > threshold))

$\log_2 N$ RTT prima di usare finestra di dimensione N → aumento esponenziale

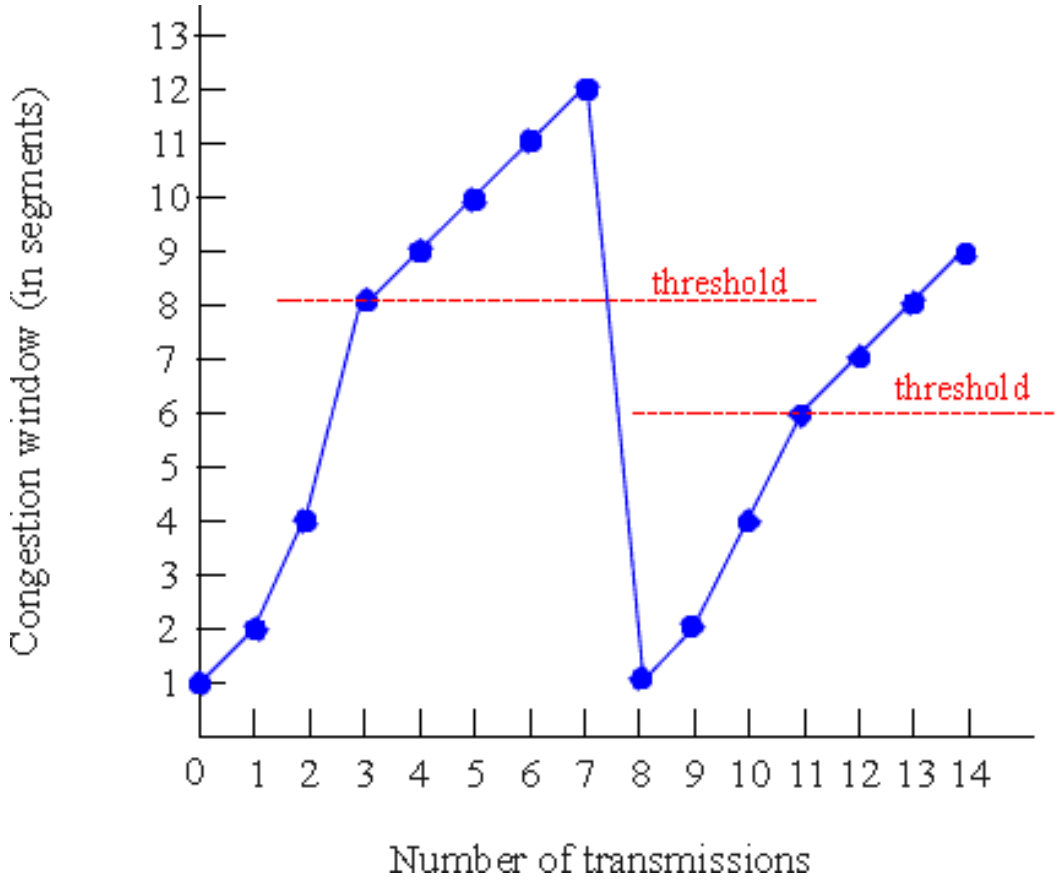


Controllo congestione: Algoritmo Tahoe

Evitare la congestione (algoritmo dell'incremento additivo e decremento moltiplicativo): si incrementa esponenzialmente fino a threshold, e poi si incrementa di 1

Tahoe

```
  /* slow-start terminato */
 /* CW > threshold */
if not(perdita) {
   ogni w segmenti ACK:
   CW++
    }
else { threshold = congwin/2;
          CW = 1;
          esegui slow-start }
```



Controllo congestione: Algoritmo Reno

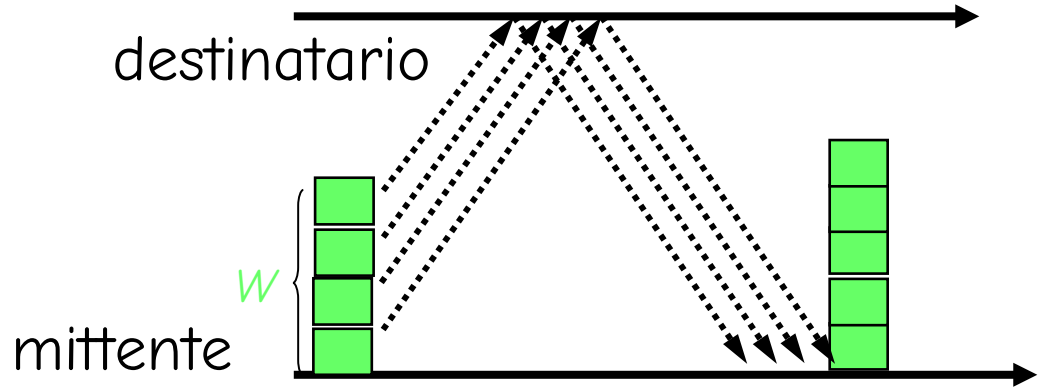
Si rilevano due tipi di errore

- **Timeout**
- → Riduci la window a 1
- **Tre ACK duplicati**
- → Non “esagera” come il Tahoe riducendo la window a 1, ma diminuisce la finestra di metà
- (MOTIVAZIONE: 1 segmento si è perso, ma la rete funziona perché almeno 3 segmenti sono stati ricevuti dal destinatario)

```
/* slow-start terminato  */
/* CW > threshold   */
Until (loss event) {
  every w segments ACKed:
     CW++
  }
threshold = CW/2
if (loss detected by timeout) {
    CW = 1
    perform slowstart }
if (loss detected by triple
duplicate ACK)
    CW = CW/2
```

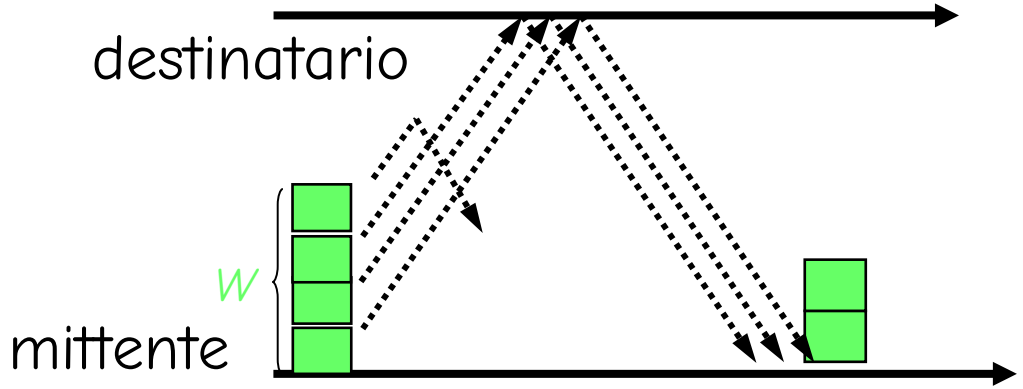
Es., controllo congestione: Reno

- **Aumenta la window di 1 per RTT se non c'è perdita: CW++**



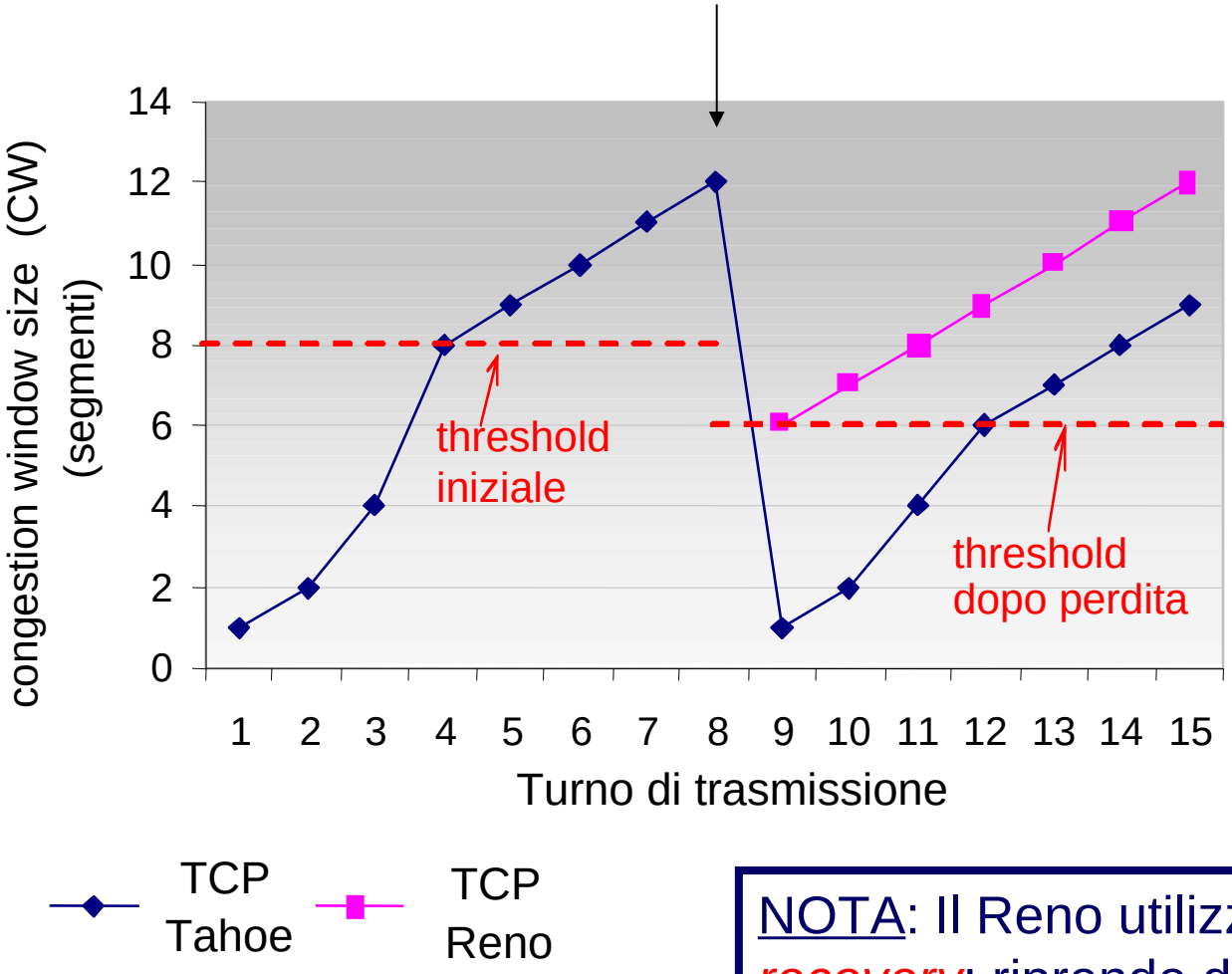
- **Riduci la window di metà se viene evidenziata una perdita con un triplo ACK duplicato:**

  CW = CW/2



TCP Reno e TCP Tahoe a confronto

Al turno 8, si verifica una perdita di pacchetto (rilevata dal time-out nel *Tahoe* e da un triplo ACK nel *Reno*)



TAHOE:
- threshold=CW/2
- CW=1

RENO:
- threshold=CW/2
- CW=threshold

NOTA: Il Reno utilizza il cosiddetto *fast recovery*: riprende da threshold e non da 1, ma con crescita lineare e non esponenziale

Controllo di congestione TCP

- **Non c’è una scelta migliore assodata**
- **All’inizio si è usata la versione Tahoe**
- **Successivamente si è passati a Reno**
- **Oggi entrambi sono obsoleti**
- **Sono state proposte tante varianti. Es.,**
  - Vegas
  - BIC
  - CUBIC

} Usate in Linux a partire da 2.6.x, Adatte a LFN (Long Fat Networks)

Controllo di congestione “Vegas”

- **Cerca di comprendere problemi di trasmissione, prima che si verifichino**
  - Approccio proattivo invece che reattivo
- **Elementi di modifica rispetto a TCP Reno**
  - Meccanismo di ritrasmissione per individuare più velocemente i pacchetti persi
  - Congestion avoidance basata su osservazione dei RTT
  - Modifica del meccanismo slow start
- **Principio: diminuzione (lineare) della frequenza di invio dei segmenti nel momento in cui si osserva un continuo aumento del RTT**

TCP Reno vs. TCP Vegas

- **TCP RENO**
- **Individuazione della perdita di pacchetti**
  - Timeout
  - 3 ACK replicati
  - Verifiche basate su timer a grana grossa (timer globale, 500 ms)

- **TCP VEGAS**
- **Individuazione della perdita di pacchetti**
  - Timeout basato su RTT
  - Verifiche basate su timer a grana fine per ogni pacchetto

TCP Reno vs. TCP Vegas

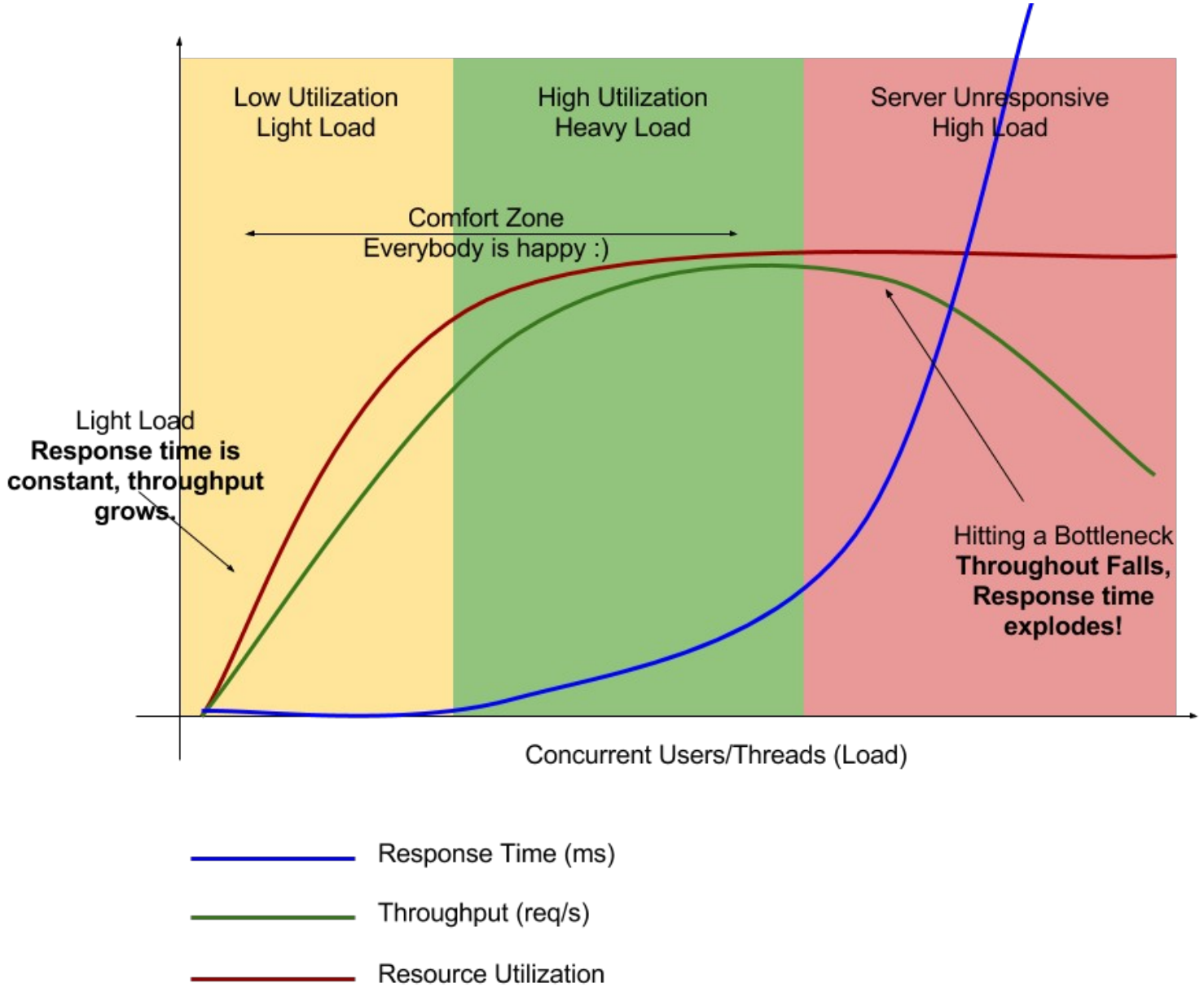
- **TCP Reno**
- **Congestion detection**
  - Perdita di pacchetti
- **In caso di timeout o 3 ACK replicati**
  - Si assume congestione
  - Si esegue slow start o fast recovery

- **TCP VEGAS**
- **Congestion detection**
  - Aumento di RTT
  - Il throughput diventa insensibile al send rate
- **Gestione della congestione separata da slow start**
  - Si considera:

$$X = \frac{WndSize\,(now) - WndSize\,(old)}{RTT\,(now) - RTT\,(old)}$$

  - Se X>0 → diminuisci congestion wnd di 1/8
  - Se X≤0 → aumenta di 1 MSS

Curve di carico tipiche di un sistema



Congestion detection in TCP Vegas

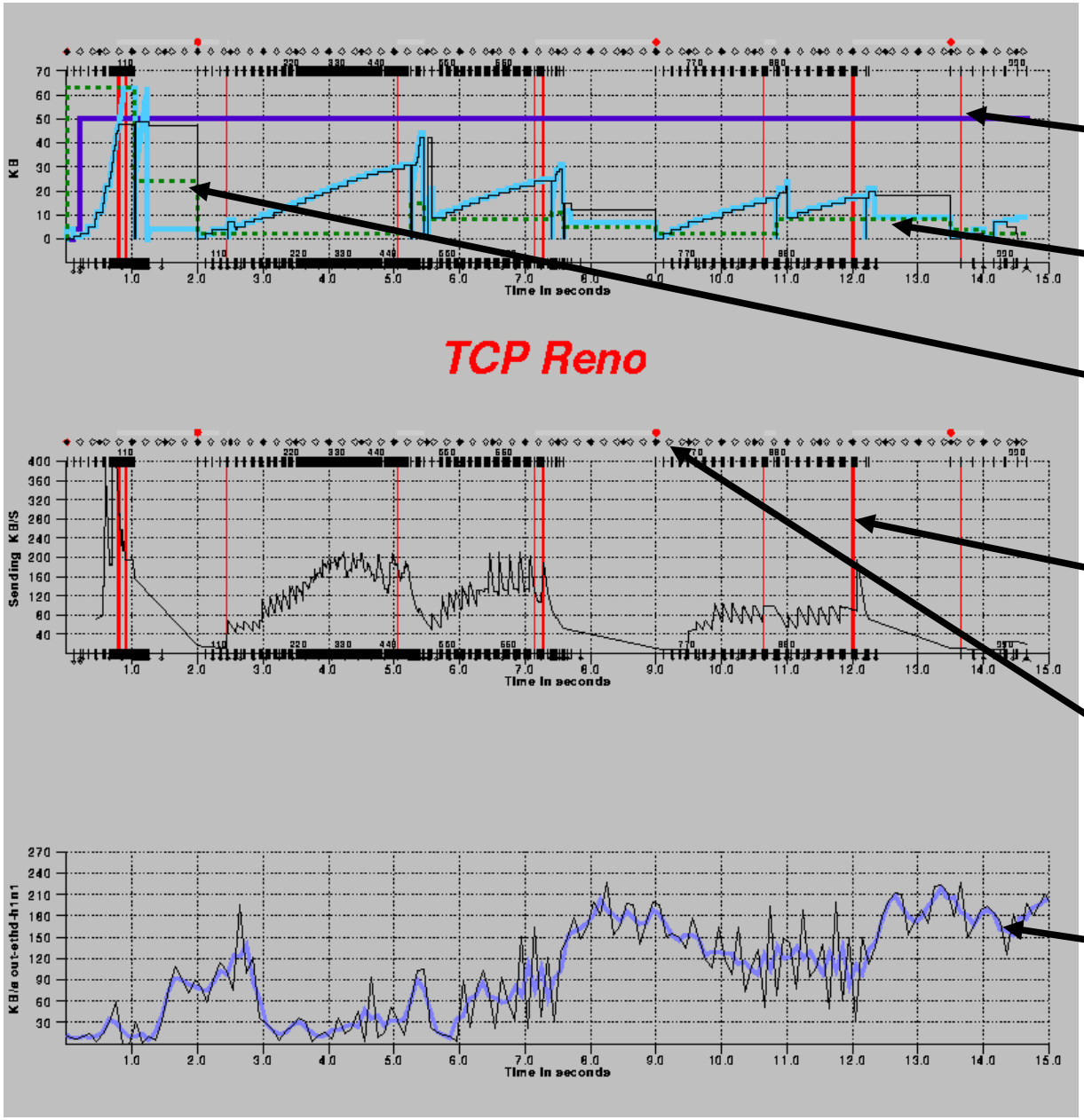
- **Se X>0**
  - All'aumento della congestion wnd corrisponde aumento RTT
  - Alla diminuzione della congestion wnd corrisponde aumento RTT
  - Congestion wnd e RTT sono direttamente correlati,
  - **→ C'è congestione**
- **Se X≤0**
  - Non si evidenzia chiara connessione tra Congestion wnd e RTT
  - **→ Non c’è congestione**

TCP Reno vs. TCP Vegas

- **Slow start**
  - Aumento di 1 Congestion wnd per ogni ACK positivo
- **Crescita esponenziale della congestion wnd ad ogni RTT senza errori**

- **Slow start**
  - Se cambia sempre la dimensione della finestra la congestion avoidance non funziona bene
- **La congestion wnd cambia ad intervalli regolari pari a 2*RTT**

Analisi del protocollo TCP Reno



Send window (buffer size)

Congestion window

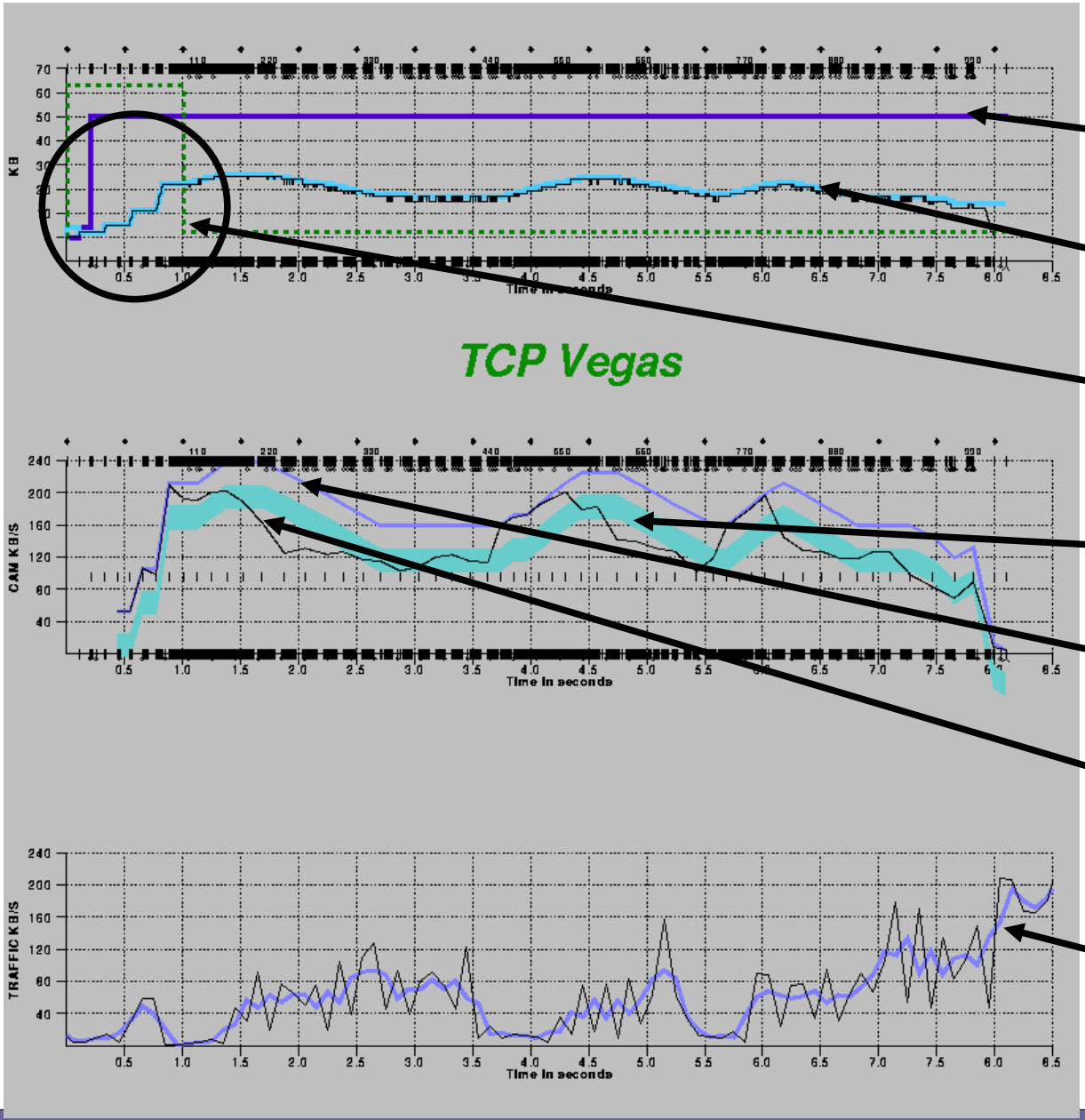
Threshold slow start

Pacchetti persi (invio)

Pacchetti persi (rilevazione)

Traffico

Analisi del protocollo TCP Vegas



Send window (buffer size)

Congestion window

Slow start

RTT

Stima throughput

Misurazione throughput

Traffico

Fairness: Vegas vs. Reno

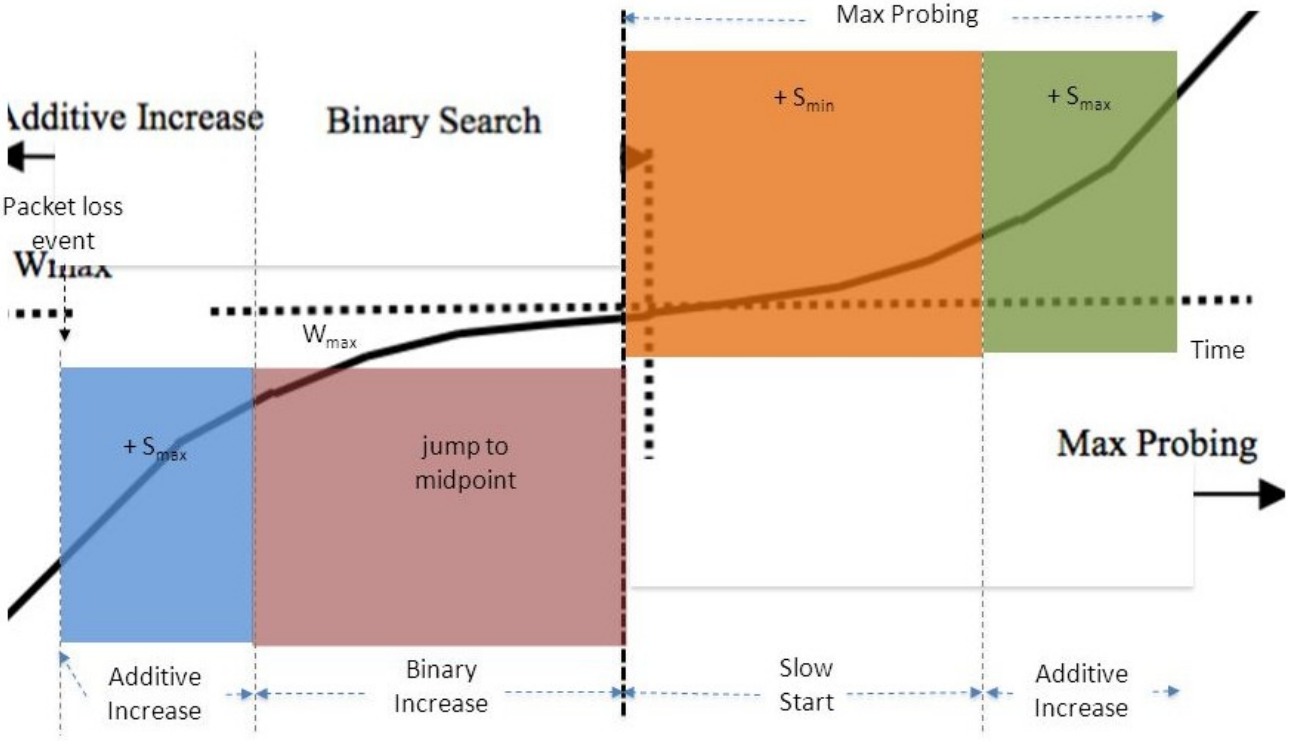
- **Problema della fairness: se flussi TCP Vegas e Reno competono per la banda**
  - Vegas sente la congestione prima e riduce il transmission rate
  - Reno continua ad aumentare la congestion window
- **Limiti nella diffusione di TCP Vegas**
  - Supportato da diversi Sistemi operativi (disponibile già nel kernel Linux dal 1999, v2.2)
  - Accettato dalla comunità scientifica, tuttavia “the TCP Vegas variant was not widely deployed outside Peterson's laboratory”

TCP per Long Fat Network

- **LFN: Long Fat Network**
- **Caratteristiche peculiari**
  - Alta Banda passante
    → Slow start può essere troppo lento per arrivare a convergenza in tempo utile
  - Alta latenza
    → i ritardi possono interferire con la stima della finestra
- **Le versioni di TCP oggi comunemente utilizzate non riescono a trarre vantaggio dalle nuove reti**
  → si propongono varianti diverse di TCP
  → implementate nel kernel di Linux
  - TCP BIC (fino a Linux 2.6.18)
  - TCP CUBIC (dopo Linux 2.6.18)

TCP per Long Fat Network: TCP BIC

- **BIC = Binary Increase Congestion control**
- **Diverse fasi:**
  - Additive Increase
  - Binary search
  - Max probing

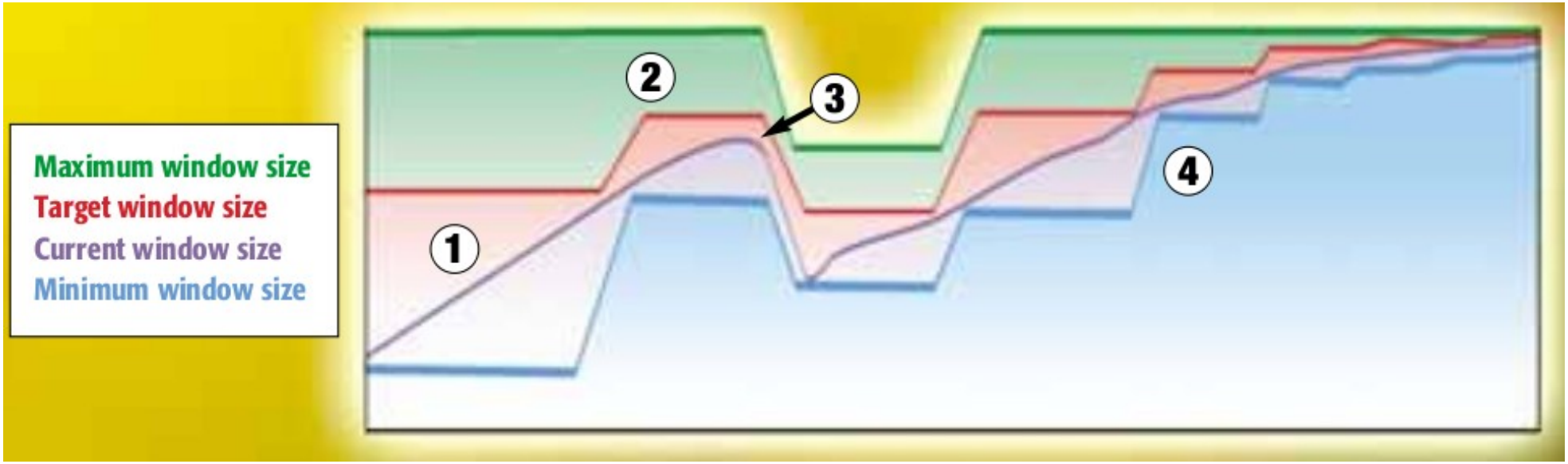


TCP per Long Fat Network: TCP BIC

- **Ci si concentra su Binary search**
- **Mantiene quattro valori di finestra di congestione:**
  - Corrente
  - Minimo
  - Massimo
  - Atteso (valore medio tra massimo e minimo)
- **Usa tecniche di aggiustamento della finestra differenti in funzione della differenza tra finestra di congestione minima e massima**

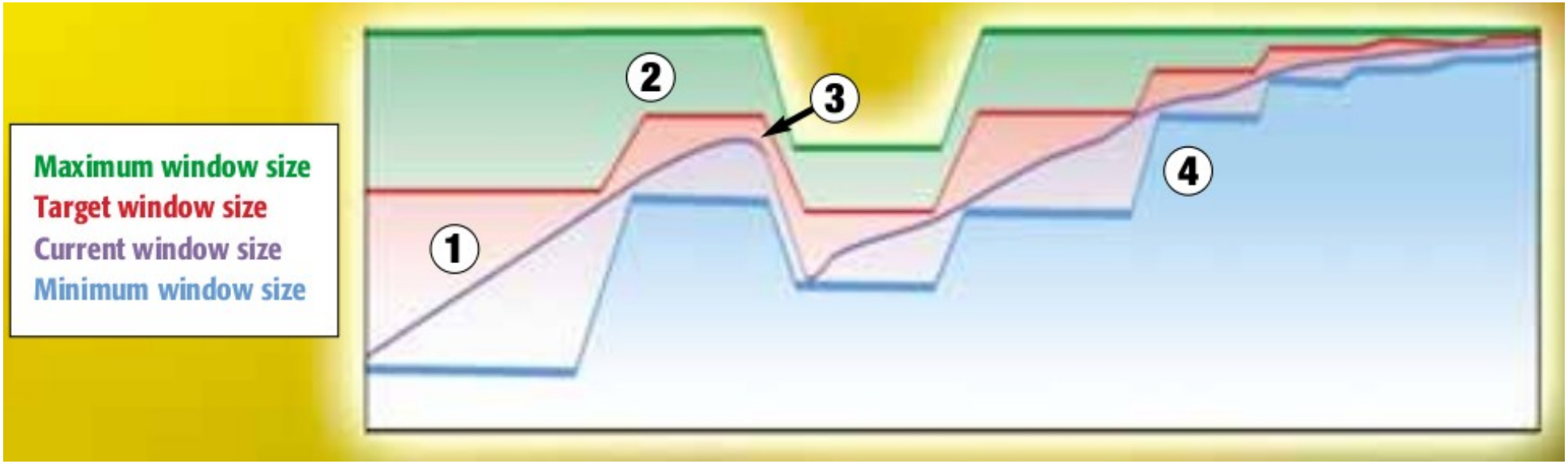
TCP BIC

- **1) Quando la differenza tra finestra minima e massima è grande si usa crescita aggressiva della finestra (incremento additivo)**
- **2) Quando la finestra corrente raggiunge quella attesa senza errori la finestra minima diventa quella attesa**



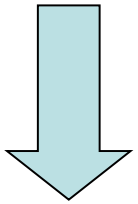
TCP BIC

- **3) In caso di errore la finestra massima assume il valore della finestra corrente. La finestra corrente si riduce al nuovo valore minimo**
- **4) Mano a mano che le finestre minime e massime si restringono la ricerca si fa meno aggressiva**

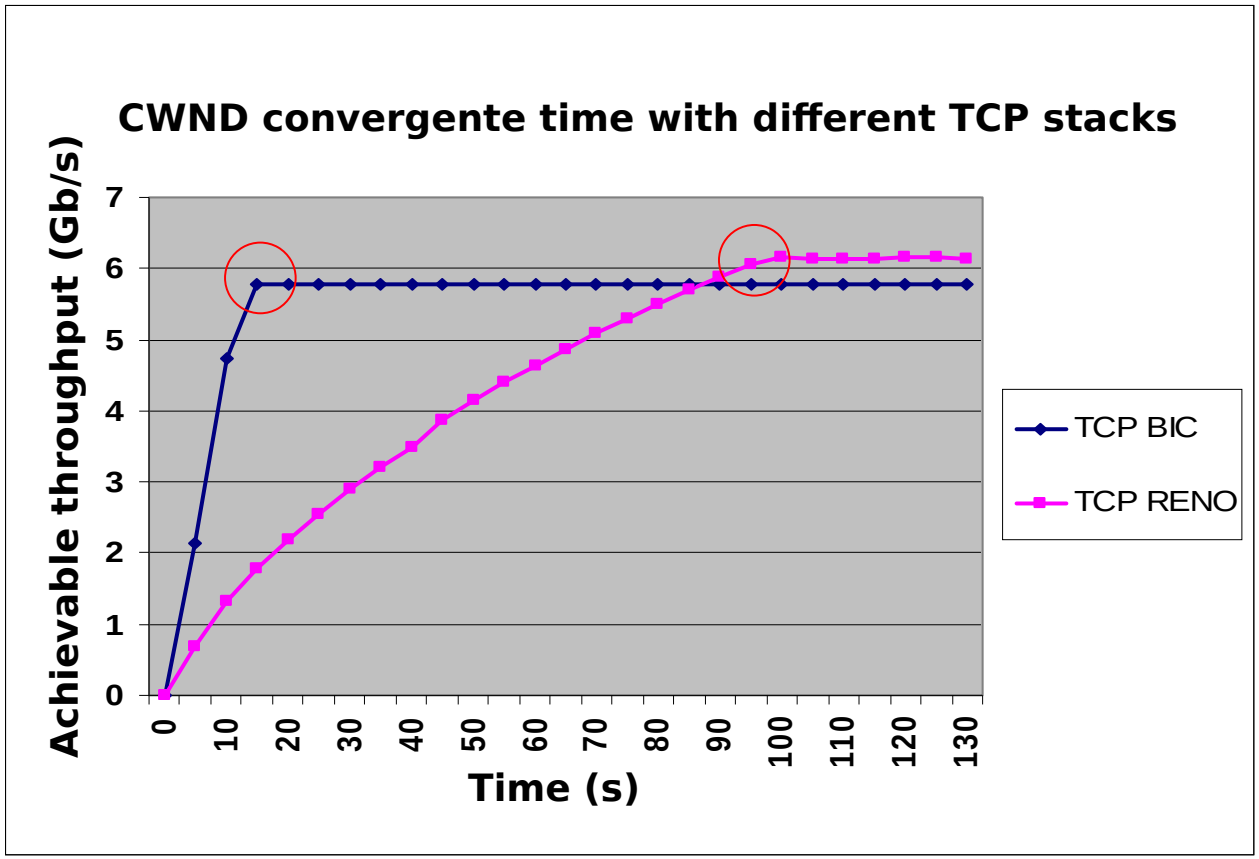


TCP Reno vs TCP BIC su reti geografiche

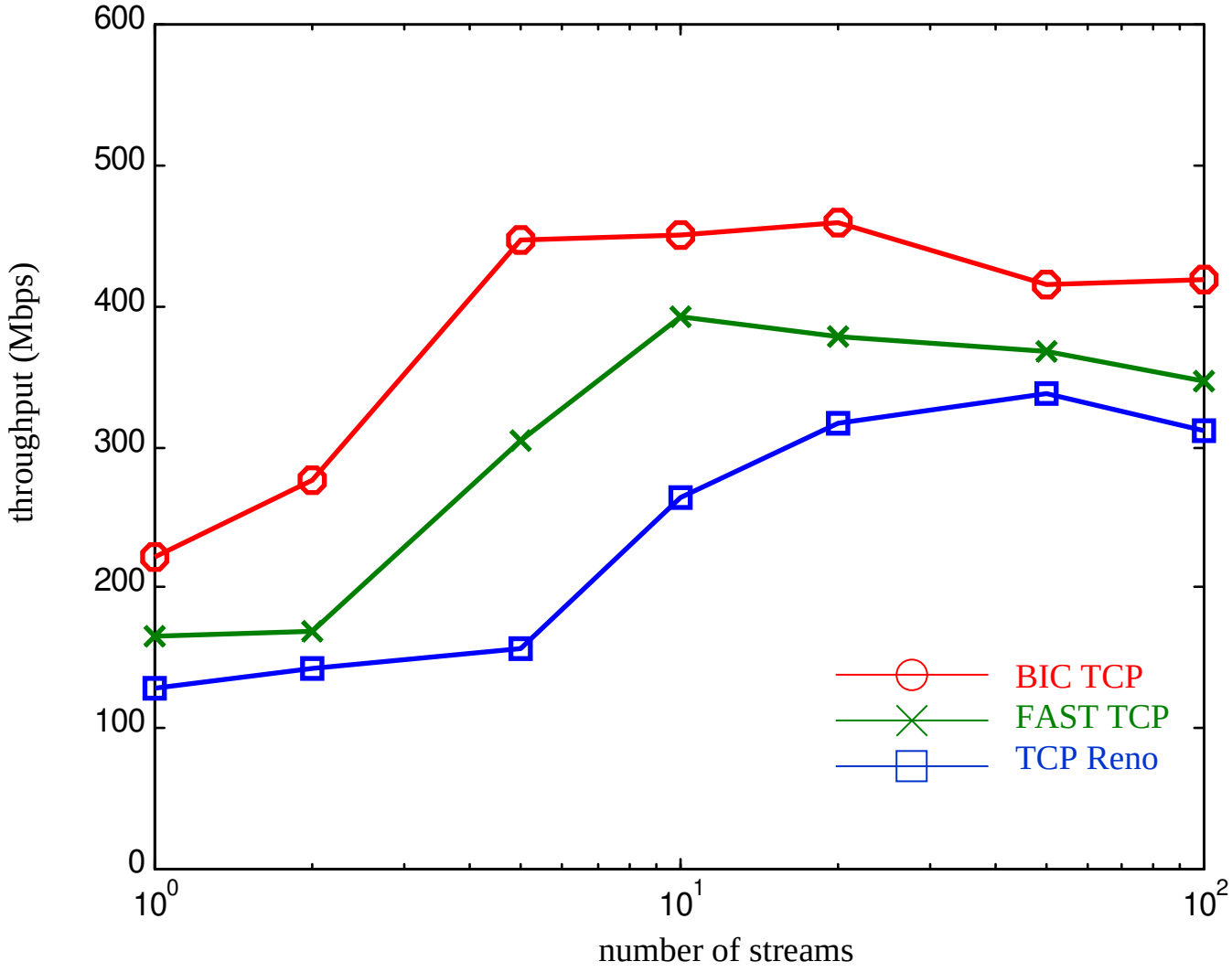
- **TCP Reno → default nel kernel 2.4**
- **TCP (CU)BIC → default nel kernel 2.6.x**



- **TCP (CU)BIC usa incremento rapido della finestra:**
  - Accelera convergenza verso equilibrio
  - TCP rimane fair per qualsiasi dimensione della finestra



Throughput a confronto su tratta di 10000 KM



RTT : 203 ms (7hops)

Effective Bandwidth:
622 Mbps

TCP CUBIC - Motivazioni

- **Miglioramento di TCP BIC**
  - Preserva scalabilità e stabilità di BIC
  - Semplifica il controllo della finestra di congestione
  - Migliora la fairness
  - Aggiorna le finestre in tempo reale invece che basarsi su messaggi di ACK → non sensibile a RTT
- **Migliora l'individuazione della regione TCP**
  - Migliora capacità di decidere se essere aggressivo o meno

CUBIC - Progetto

- **Basato su TCP BIC**

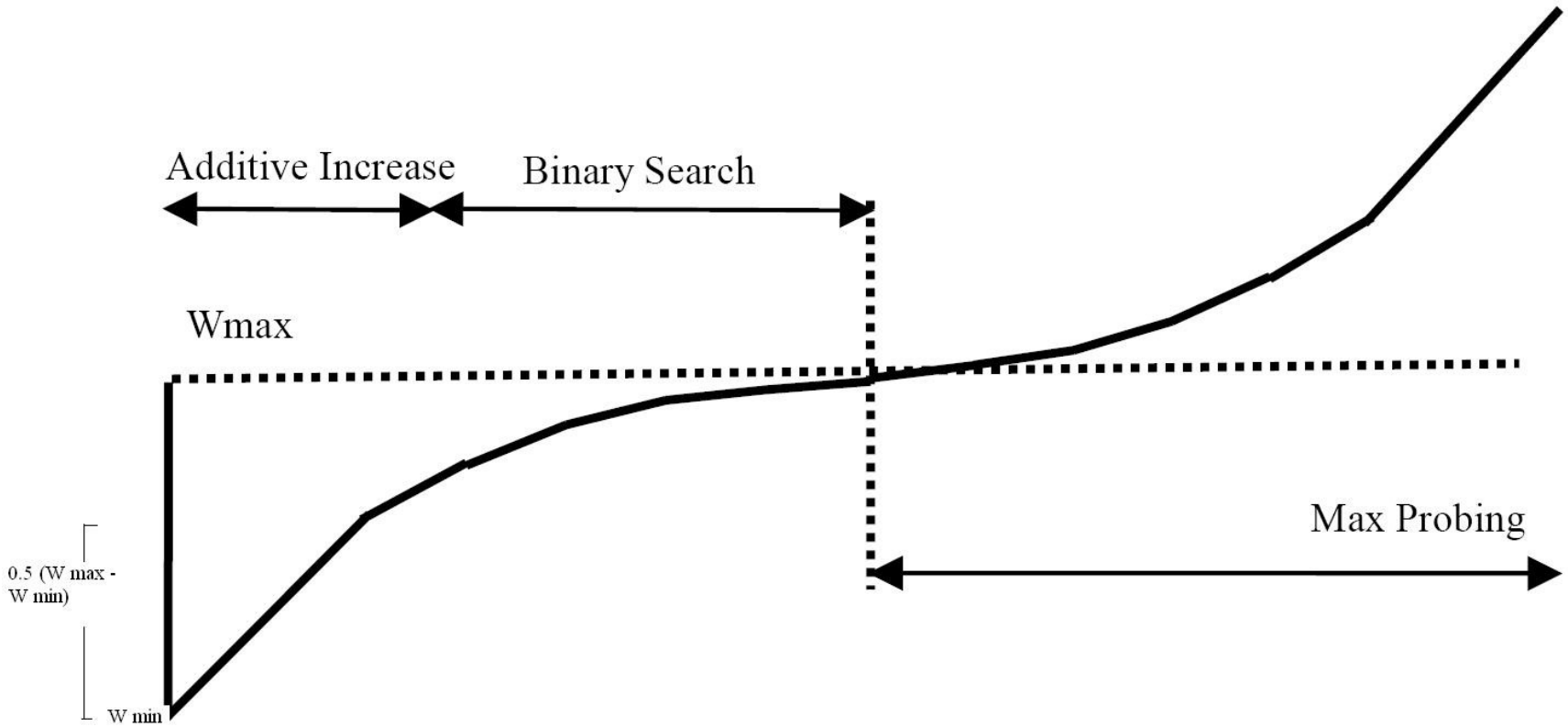


Fig. 1: The Window Growth Function of BIC

CUBIC - Progetto

- **Funzione di crescita della finestra: f((now-t_last_congestion)^3)**

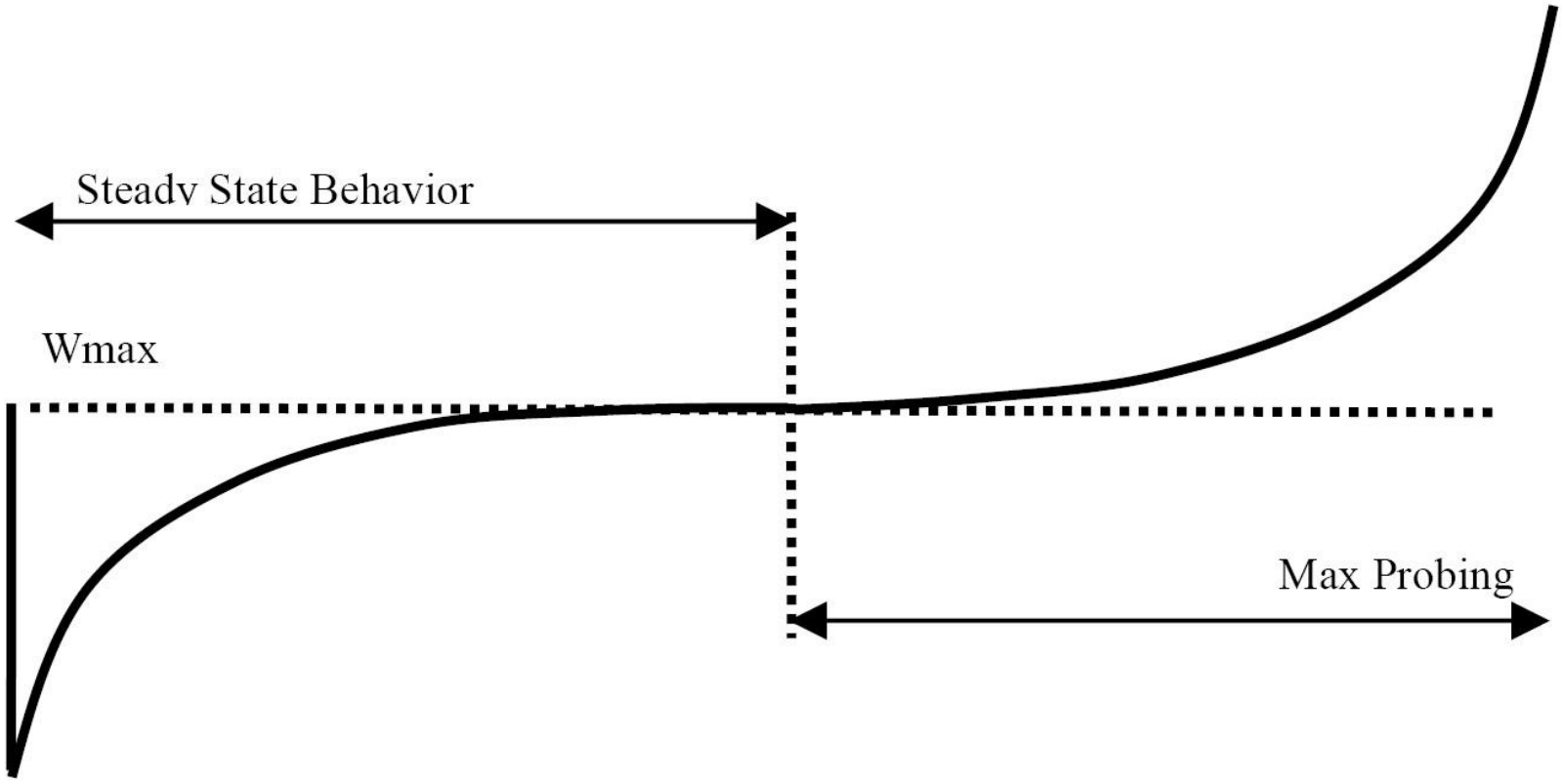


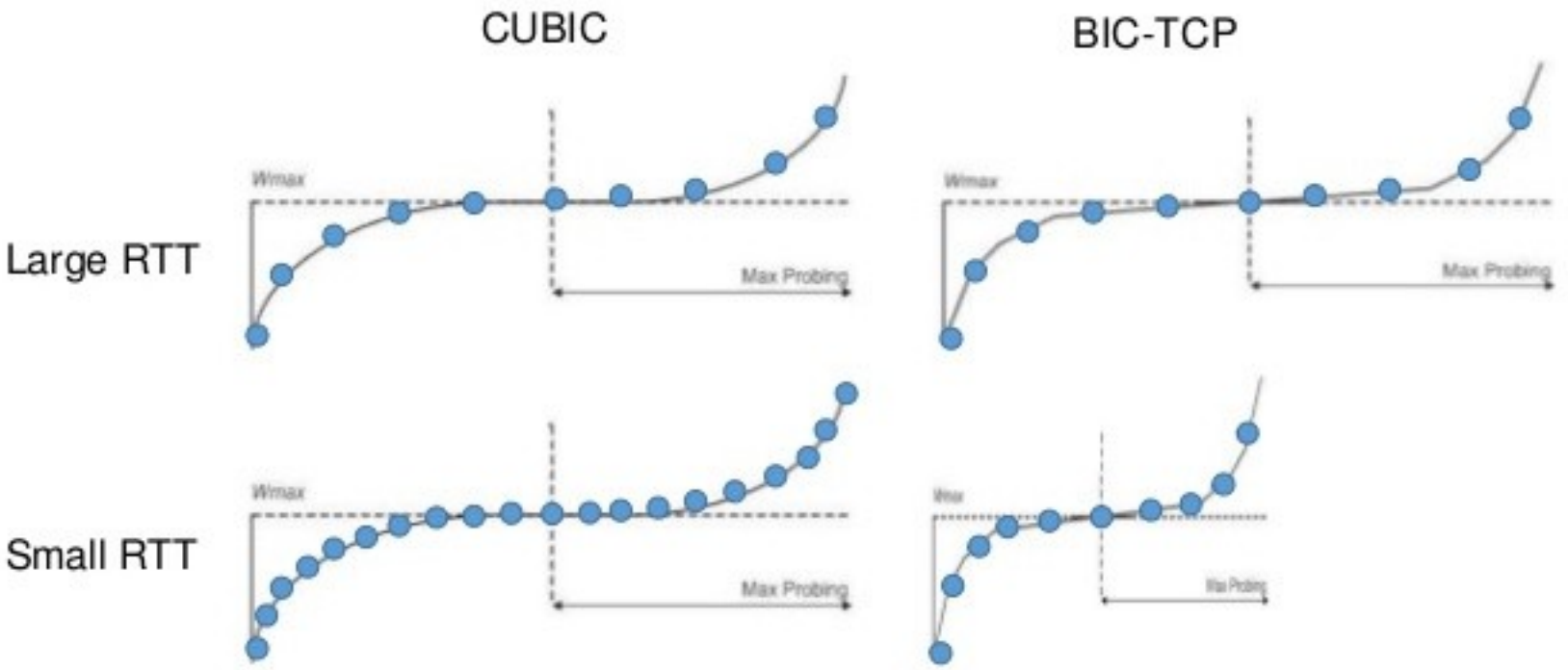
Fig. 2: The Window Growth Function of CUBIC

CUBIC – Criteri progettuali

- **Stabilità – La finesta aumenta lentamente nei dintorni di *Wmax.***
- **Scalabilità – Esegue un probing molto veloce lontano da *Wmax.***
- **Fairness tra flussi – Due flussi CUBIC che competono, convergono verso una dimensione della finestra “giusta”**

CUBIC – Criteri progettuali

- **Fairness rispetto a RTT – La crescita della finestra dipende dal tempo e non dal RTT. Questo migliora la fairness del procollo.**

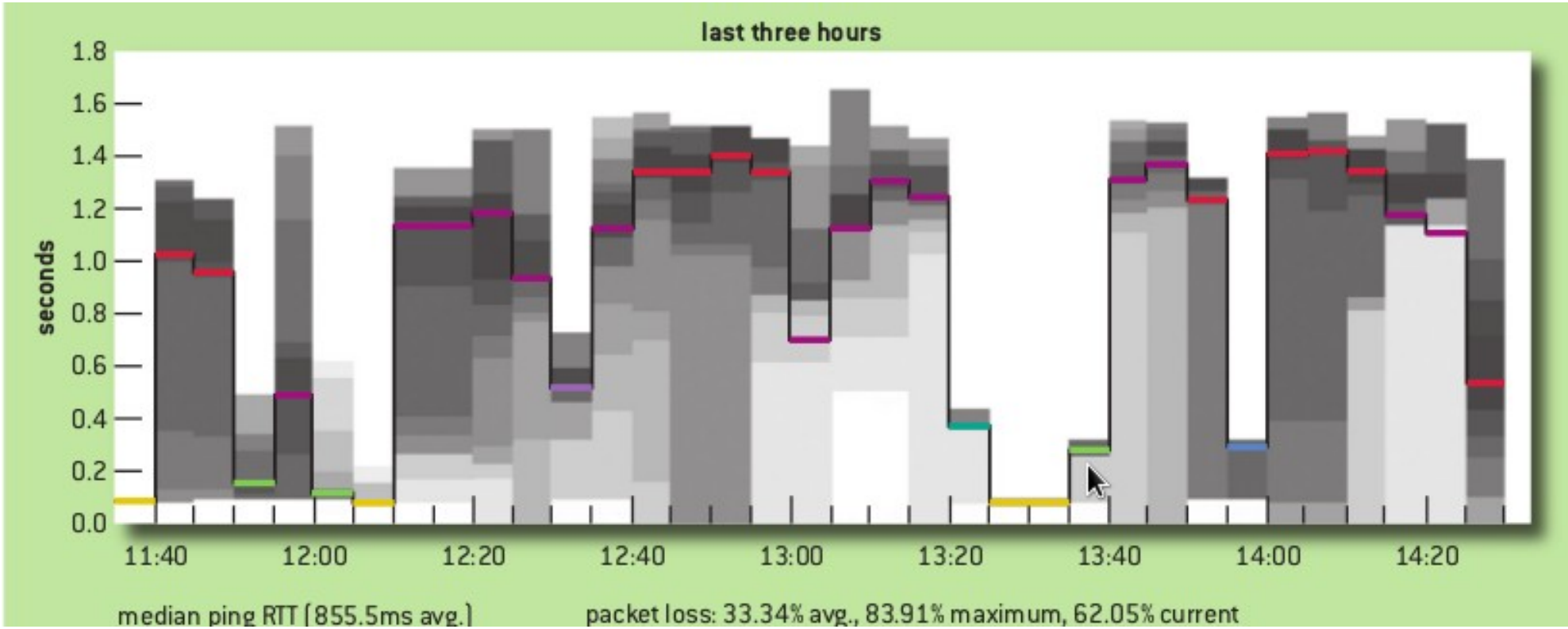


Modulo 12:
Buffer bloating

Buffer bloating

- **Problema legato a presenza di buffer molto grandi nella rete, tipicamente di dimensione variabile**
- **Esaspera l'effetto Long Fat Network**
- **Problema legato al ritardo introdotto nella identificazione della congestione**
- **Si verifica in presenza di lunghi data transfer**
- **E' aggravato dalla presenza di più flussi TCP concorrenti (usati spesso anche nel Web)**
- **Effetto finale critico per:**
  - Applicazioni interattive
  - Trading online (molto sentito dai professionisti)

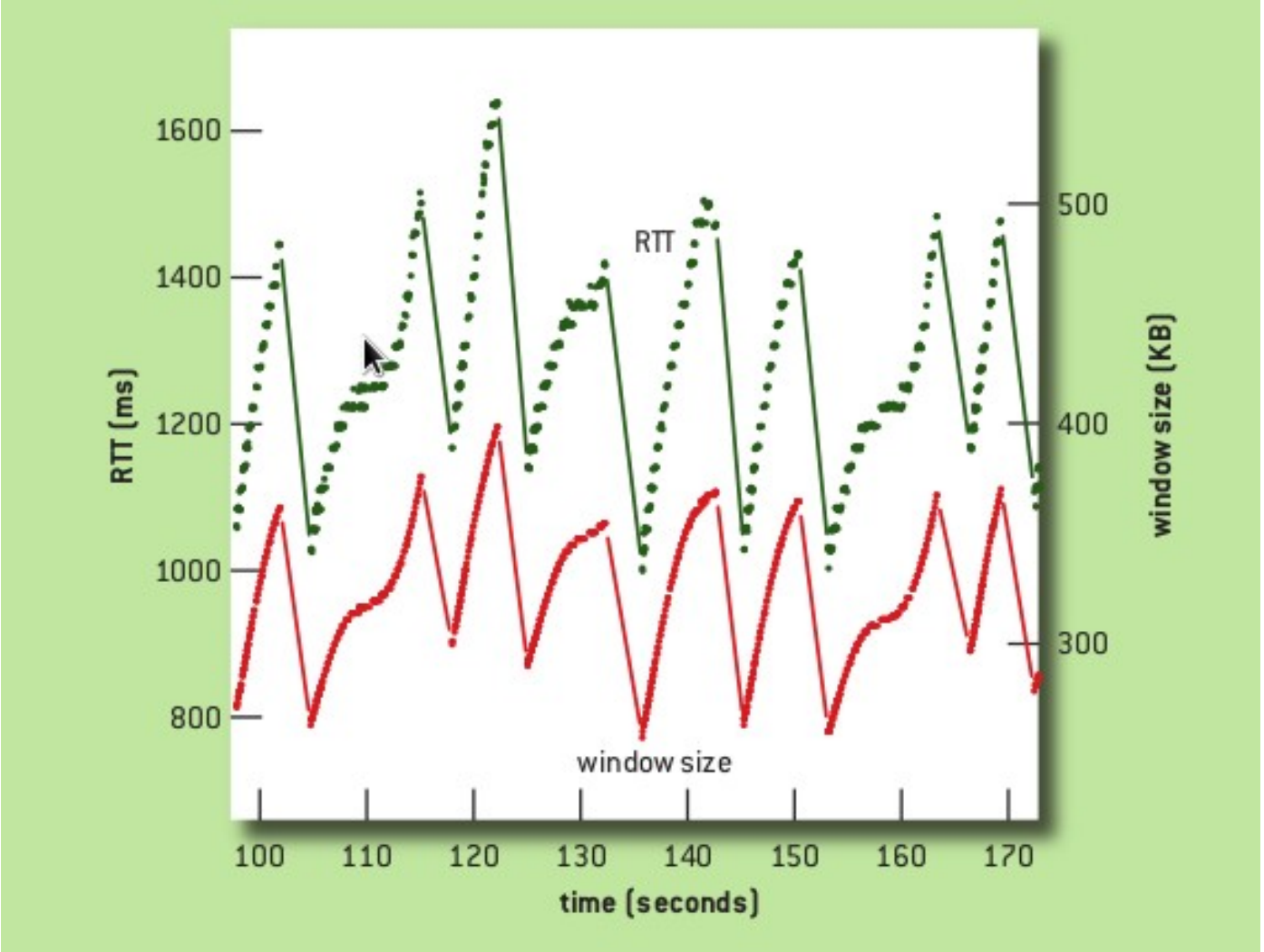
Esempio di buffer bloating



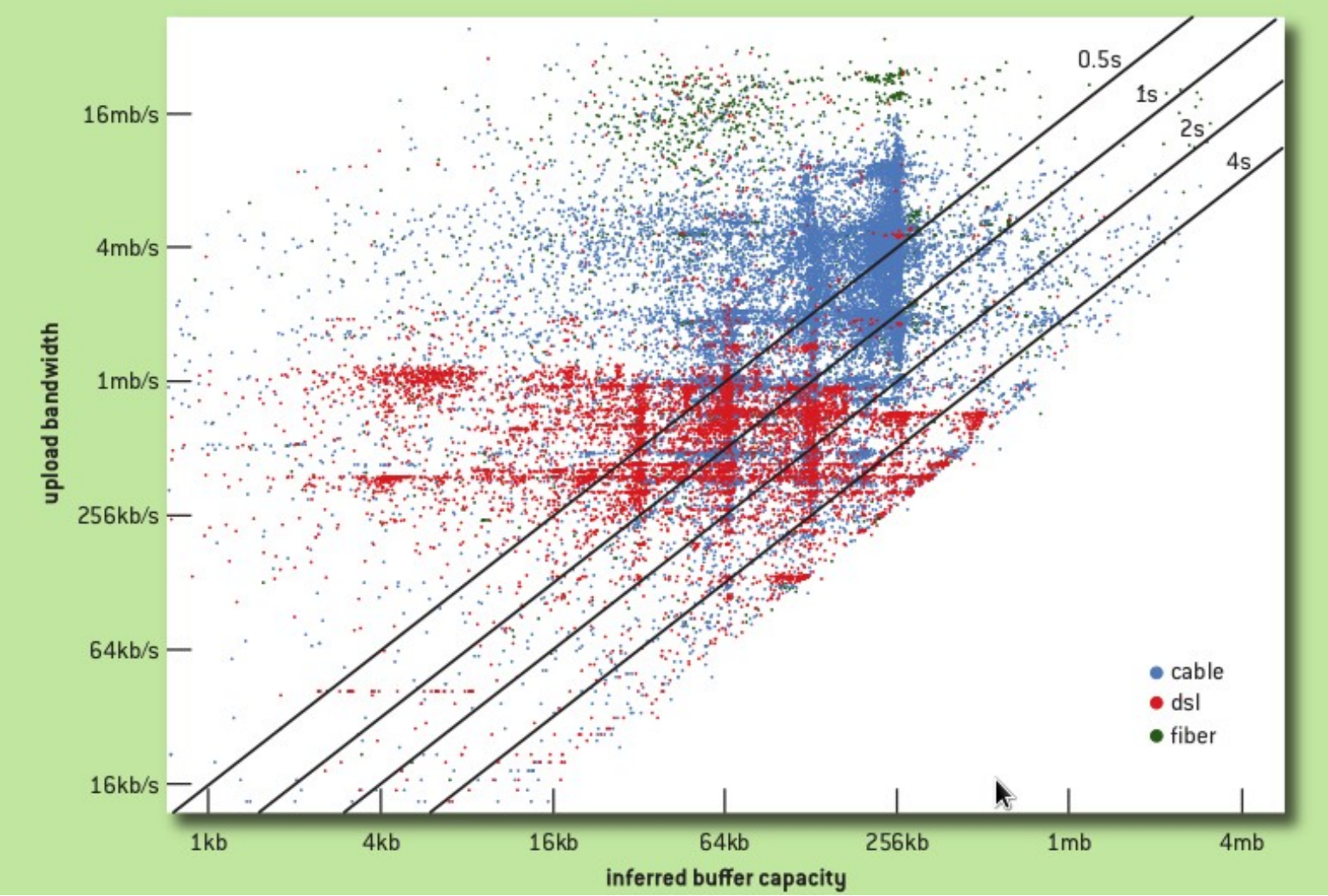
Spiegazione dell'effetto

- **La presenza di larghi buffer non determina perdita di pacchetti anche in presenza di congestione**
- **L'aumento nel RTT determina scarsa responsiveness rispetto alla congestione**
- **Quando i buffer sono pieni il RTT è così alto che il timeout per la perdita di pacchetti rallenta l'intervento del controllo di congestione**
- **Presenza di flussi concorrenti crea ulteriori problemi**
- **Risultato**
  - Crollo del throughput per l'effetto dei ritardi
  - Alto tasso di perdita dei pacchetti

Dimostrazione di buffer bloat



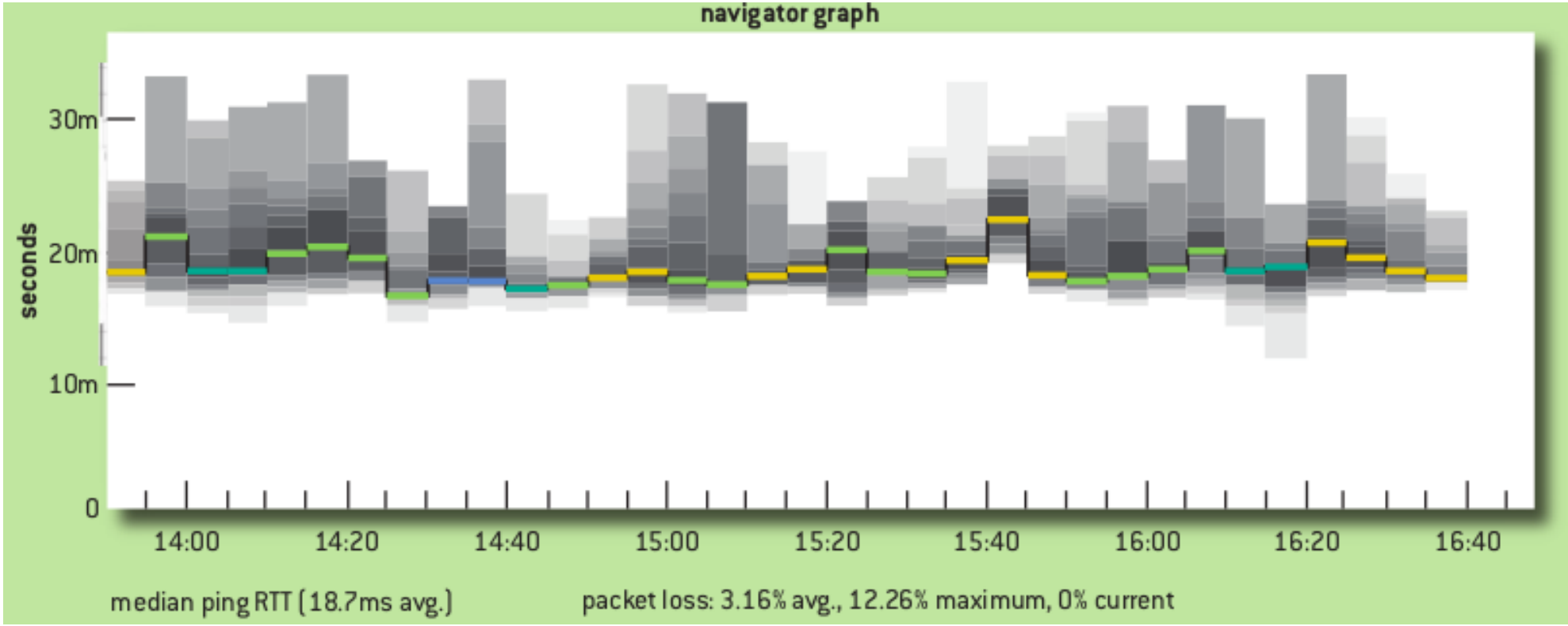
Stima dei buffer presenti in rete



Possibili contromisure

- **Proposta di nuovi algoritmi di AQM (adaptive queue management)**
  - Gestione dinamica della dimensione dei buffer
  - Attenzione al caso di coesistenza di tanti flussi TCP
  - Linux: algoritmo CoDel (Controlled Delay)
- **Proposta di nuovi algoritmi che accorpano più flussi applicativi in un unico stream TCP**
  - es. google SPDY, HTTP/2

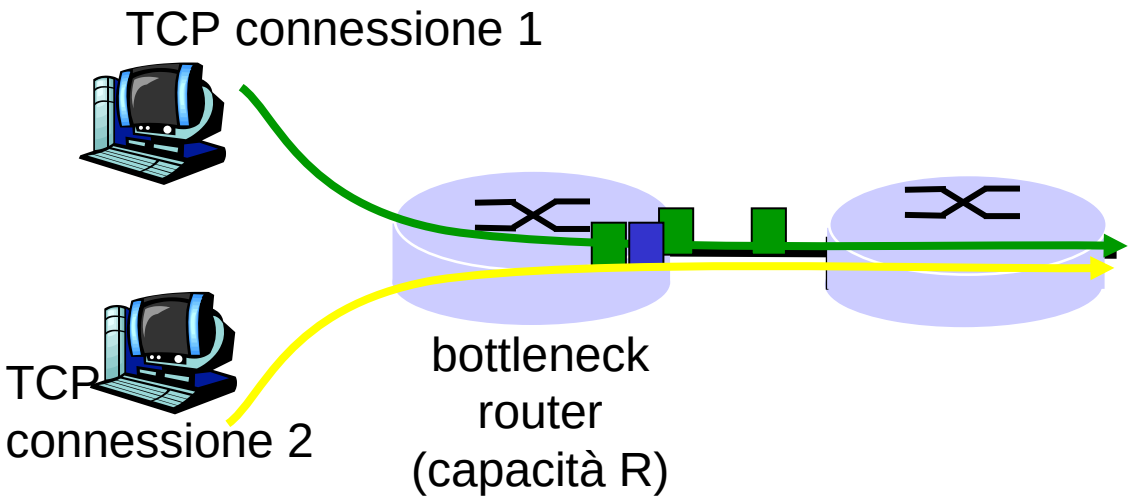
Conclusione



Modulo 13:
Altre caratteristiche del TCP

Fairness del TCP

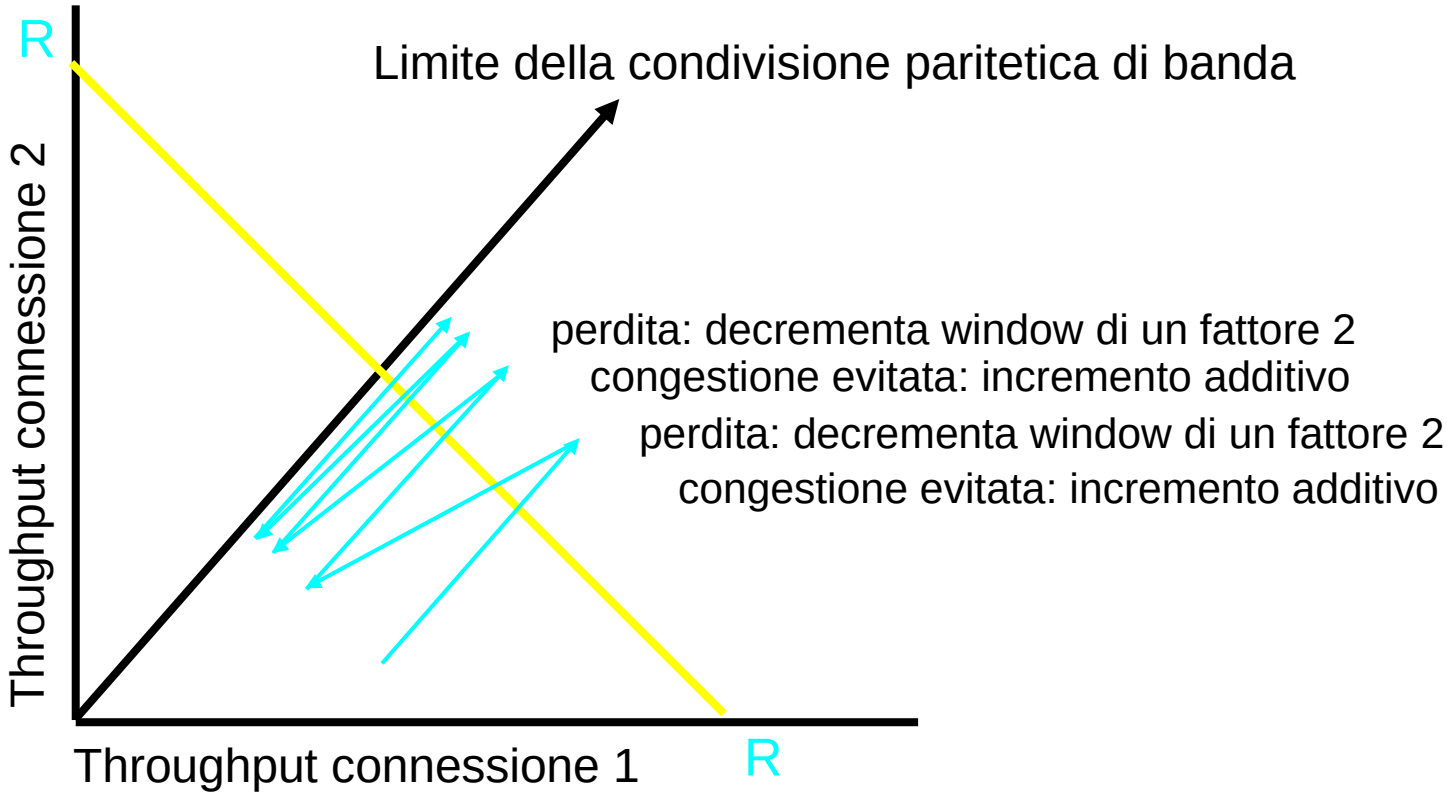
- **Scopo della *fairness*: se vi sono *N* sessioni TCP che condividono lo stesso link, ciascuna deve ottenere 1/*N*-mo della capacità del link**



Perché TCP è fair?

Due sessioni in competizione su un link a capacità limitata

- **Incremento additivo dà la direzione di 1, facendo crescere il throughput**
- **Decremento moltiplicativo diminuisce il throughput proporzionalmente**



Modulo 14:
UDP vs. TCP

Perché usare UDP anziché TCP

- **Non c’è instaurazione della connessione**
  - TCP usa un meccanismo di three-way handshaking prima di iniziare il trasferimento dei dati
  - UDP non introduce un ritardo per instaurare la connessione
- **Non c’è stato di connessione**
  - TCP mantiene lo stato della connessione tra i nodi terminali (buffer di invio/ricezione, parametri per il controllo della congestione, numeri di sequenza ed acknowledgment)
  - un server può supportare un maggior numero di connessioni attive se usa UDP piuttosto che TCP

Perché usare UDP anziché TCP (cont.)

- **Overhead minore dovuto all'header del segmento più piccolo**
  - segmento TCP: **20 byte**
  - segmento UDP: **8 byte**

- **Flusso di invio non regolato**
  - il controllo di congestione del TCP può avere un forte impatto (negativo) sulle applicazioni di rete in real-time

Quando si può usare UDP

- **Si opera su rete locale**
- **L'applicazione mette tutti i dati in un singolo pacchetto**
- **Non è importante che tutti i pacchetti arrivino a destinazione (es., alcune applicazioni multimediali)**
- **L'applicazione gestisce meccanismi di ritrasmissione**

Applicazioni e protocollo trasporto

Applicazione	Prot. strato applicativo	Prot. trasporto sottostante
posta elettronica	SMTP	TCP
accesso terminale remoto	Telnet	TCP
Web	HTTP	TCP
trasferimento file	FTP	TCP
file server remote	NFS	solitamente UDP
multimedia streaming	RTP, Real Media, ...	solitamente UDP
Telefonia Internet	SIP, H.323, ...	solitamente UDP
Gestione della rete	SNMP	solitamente UDP
Protocollo di routing	RIP	solitamente UDP
Traduzione dei nomi	DNS	solitamente UDP

Applicazioni e protocollo trasporto

1) Posta elettronica, accesso da terminale remoto, Web, trasferimento di file

→ TCP

Necessario il servizio di trasferimento dati affidabile fornito da TCP

2) Aggiornamento tabelle di routing in RIP

→ UDP

Aggiornamenti periodici delle tabelle: eventuali informazioni perse sostituite da informazioni più aggiornate

Applicazioni di rete per UDP

4) DNS query → UDP

Si evitano i ritardi dovuti all'instaurazione della connessione

5) Telefonia Internet → UDP

Tolleranza alla perdita di dati (no servizio affidabile)

Tasso di trasmissione costante (no controllo di congestione)

6) Applicazioni multimediali → UDP

TCP non può essere impiegato con multicast

Non serve controllo di congestione

Serve protocollo molto veloce (requisiti real time)

Alcune domande sul TCP

- **Perché è necessario il 3-way handshaking?**
- **Chi invia il primo segmento FIN: il server o il client?**
- **Una volta che la connessione TCP è stata stabilita, qual è la differenza tra le operazioni del livello TCP del server e del livello TCP del client?**