Principi di Sliding window

(*nelle ultime versioni il protocollo TCP reale usa varianti differenti*)

Sliding window - *mittente*

- Il mittente assegna a ciascun segmento un numero di sequenza **Num_segm**
- Si ipotizza che questo numero possa crescere a piacere, anche se nella realtà il range va da 0 a 2 -1
- Assunzione iniziale (poi rilassata): dimensione window fissa

PRINCIPI:

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

  (o sono stati spediti o stanno per essere spediti)
- **La dimensione massima della finestra del mittente è controllata dal destinatario**

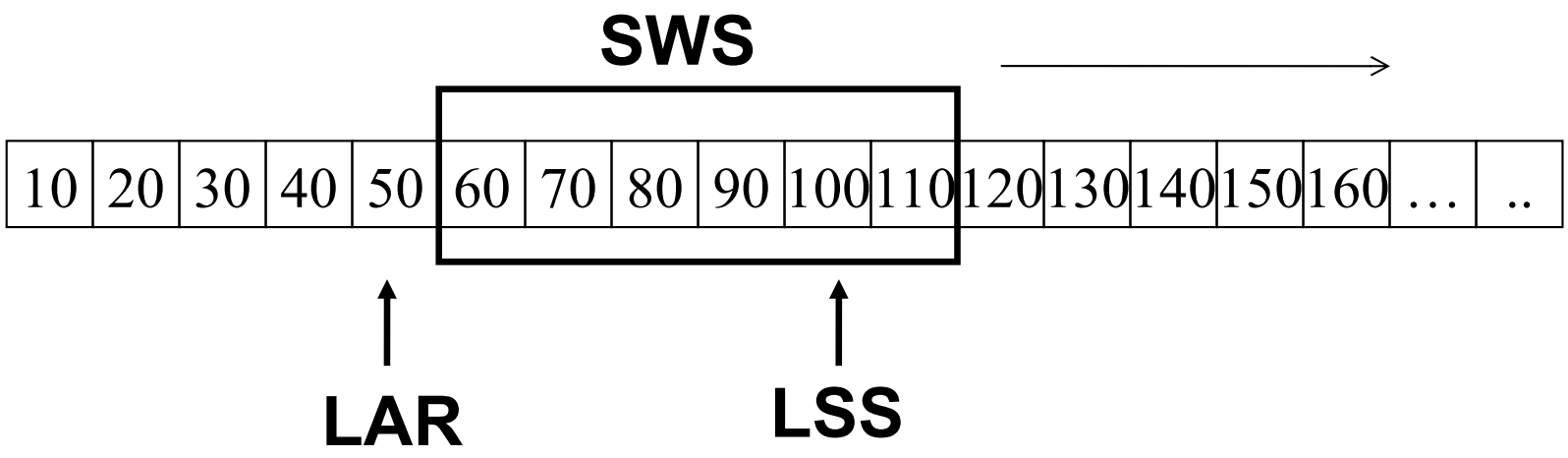
Sliding window – *mittente* (2)

- 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 in pipeline 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* (3)

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

$$\mathbf{LSS - LAR \leq SWS}$$



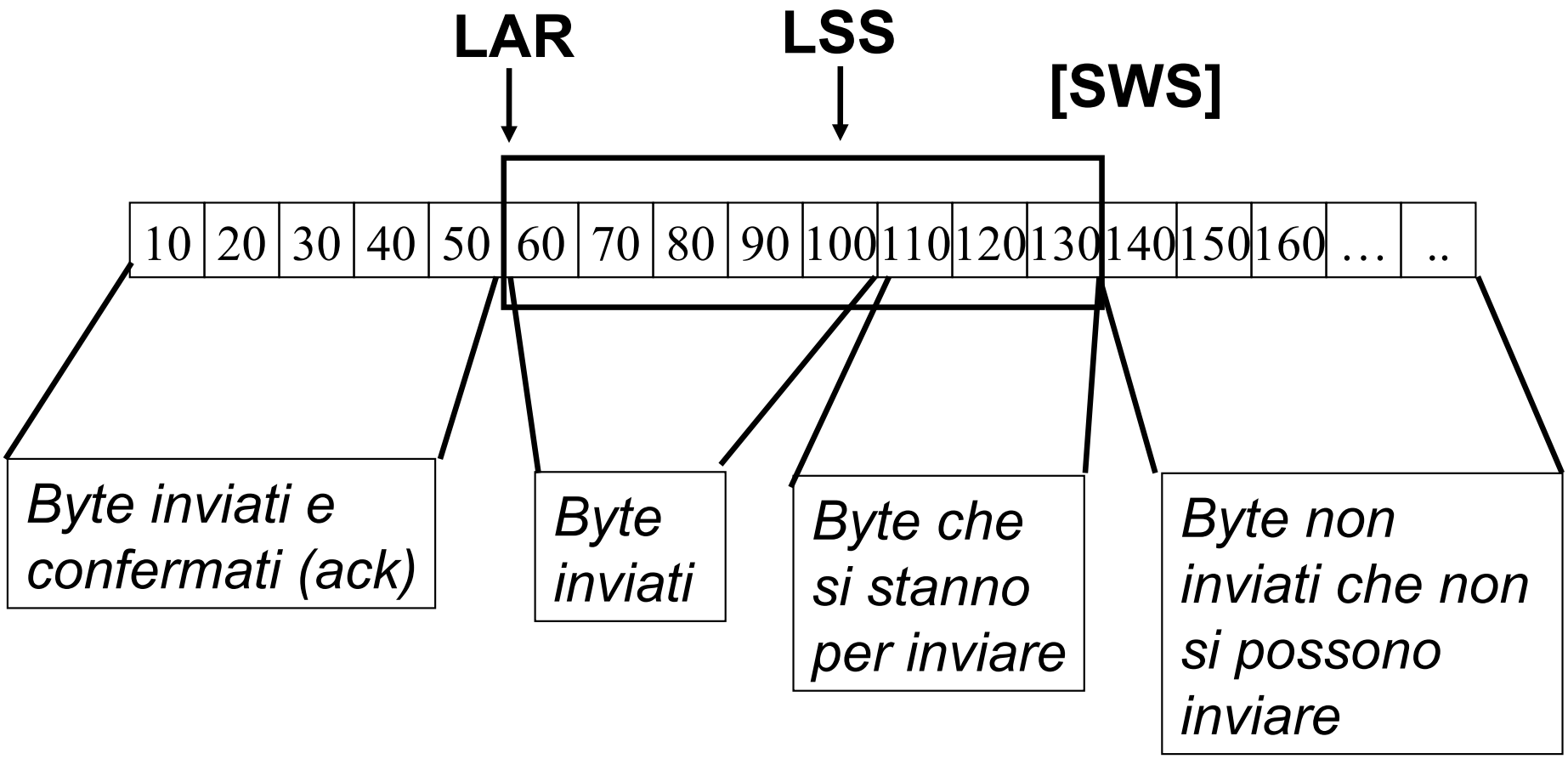
SWS: ***Sender Window Size***

LAR: ***Last Acknowledgement Received***

LSS: ***Last Segment Sent***

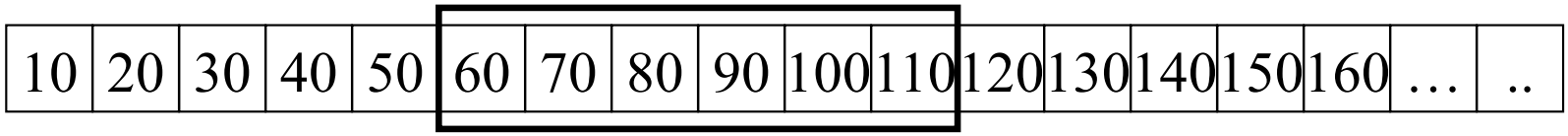
Esempio: Sliding window - *mittente* (4)

IPOTESI: segment size di 10 byte

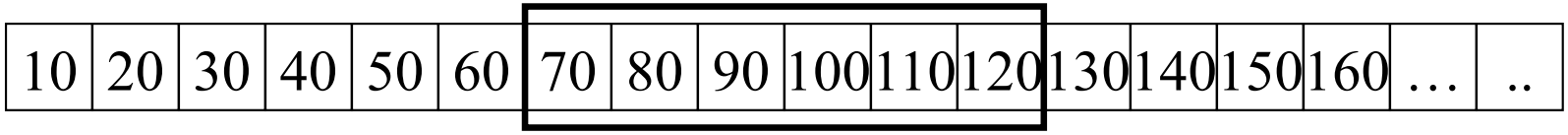


Sliding window - *mittente* (5)

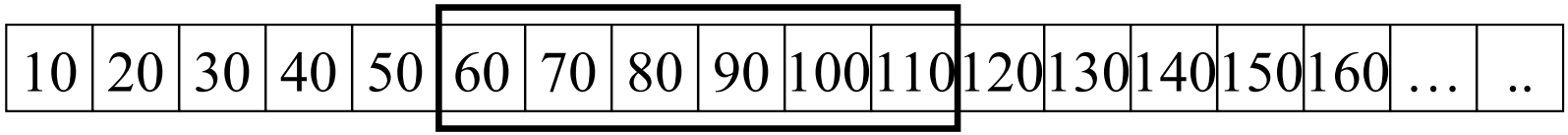
- Quando arriva un ACK, il mittente sposta LAR verso destra, consentendo così l'invio di un altro segmento
- Inoltre, il mittente associa un time-out a ciascun segmento che trasmette, con conseguente ritrasmissione del segmento se il time-out scade prima di aver ricevuto il relativo ACK



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



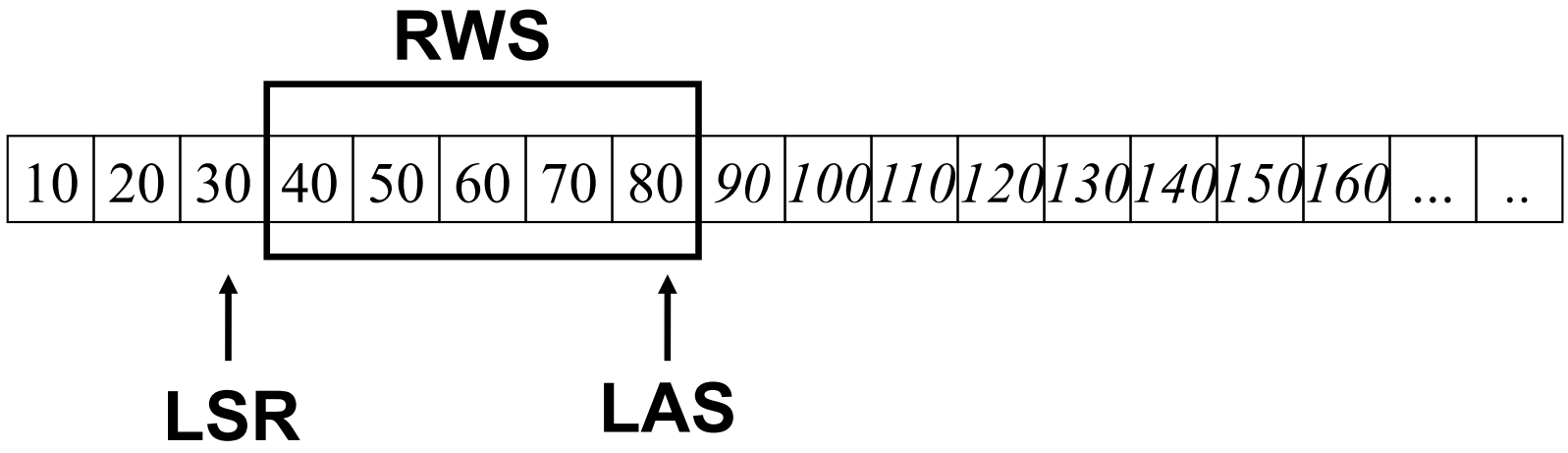
Sliding window – *destinatario* (1)

- I destinatari che gestiscono una ***sliding window*** utilizzano tre variabili:
  - Dimensione della finestra di ricezione **RWS** (***Receive Windows Size***): indica il limite superiore dei 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* (2)

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

$$\mathbf{LAS - LSR \leq RWS}$$



RWS: ***Receive Window Size***

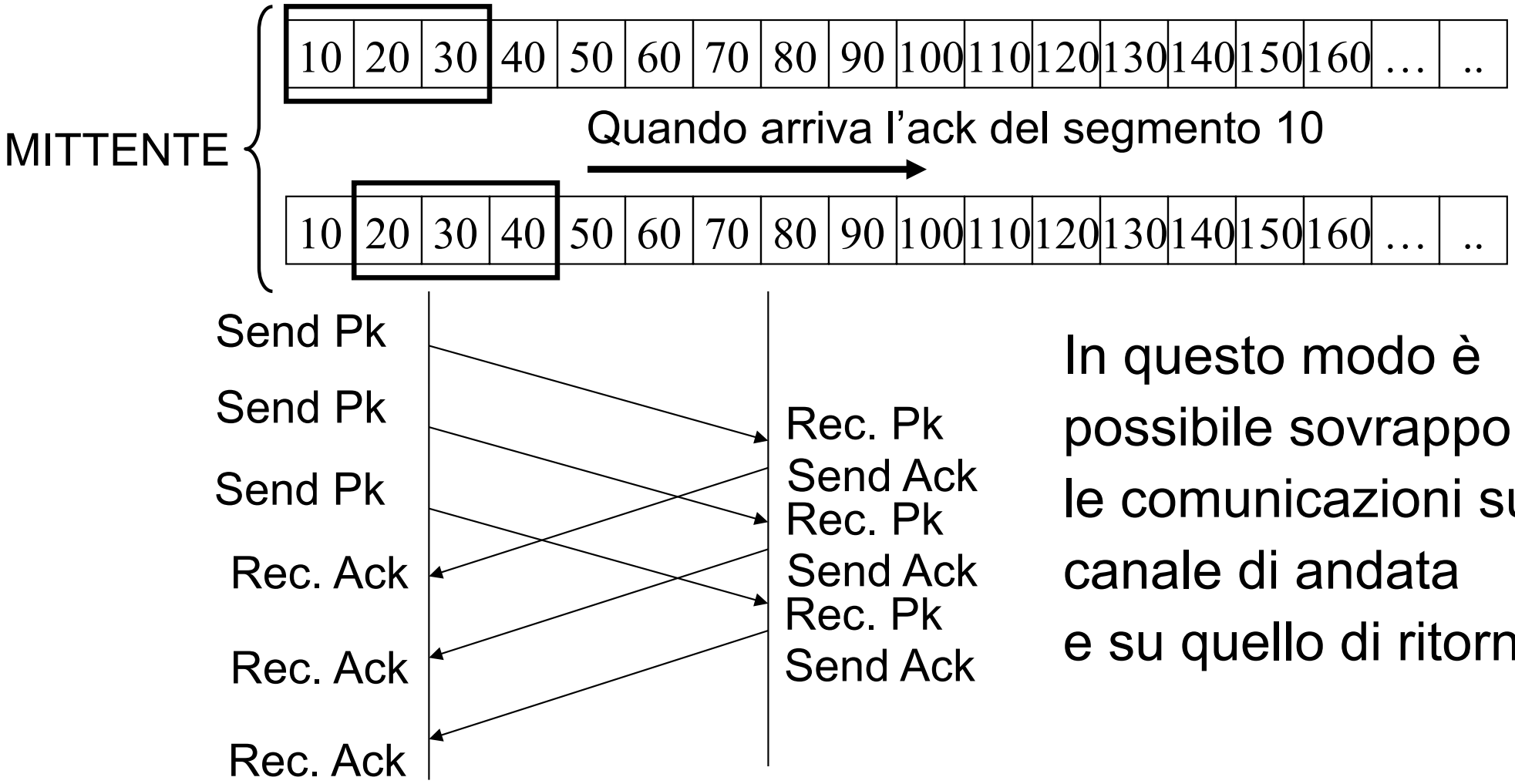
LAS: ***Largest Acceptable Segment***

LSR: ***Last Segment Received***

Sliding window - *destinatario* (3)

- Quando arriva un segmento con numero di sequenza **Num_segm**, il destinatario agisce come segue:
  - Se **Num_segm≤LSR** (pacchetto duplicato) oppure (per errore) **Num_segm>LAS**, significa che il segmento si trova al di fuori della finestra utile 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

Esempio: finestra a scorrimento (size 3)



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

NOTE

- Nel funzionamento (teorico) visto in precedenza, si è assunto che la DIMENSIONE della sliding window fosse di dimensione fissa ← ***Questa assunzione NON è vera***
- Tutto il funzionamento risulta abbastanza semplice fin quando le “cose vanno bene”: cioè le trasmissioni vanno a buon fine e i segmenti arrivano 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 approcci alternativi per affrontare il problema dell'affidabilità della comunicazione nel caso di un protocollo di tipo *pipelining* (che invia più segmenti prima dell'arrivo dell'ack)

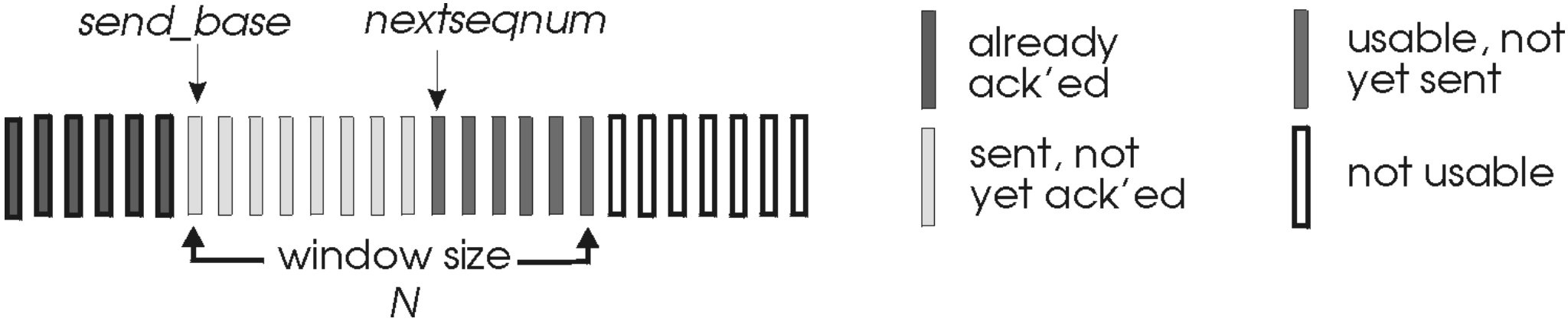
- **Go-Back-N**
- **Ritrasmissione selettiva**

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*. *Perché?***

Algoritmo 1: *Go-Back-N*

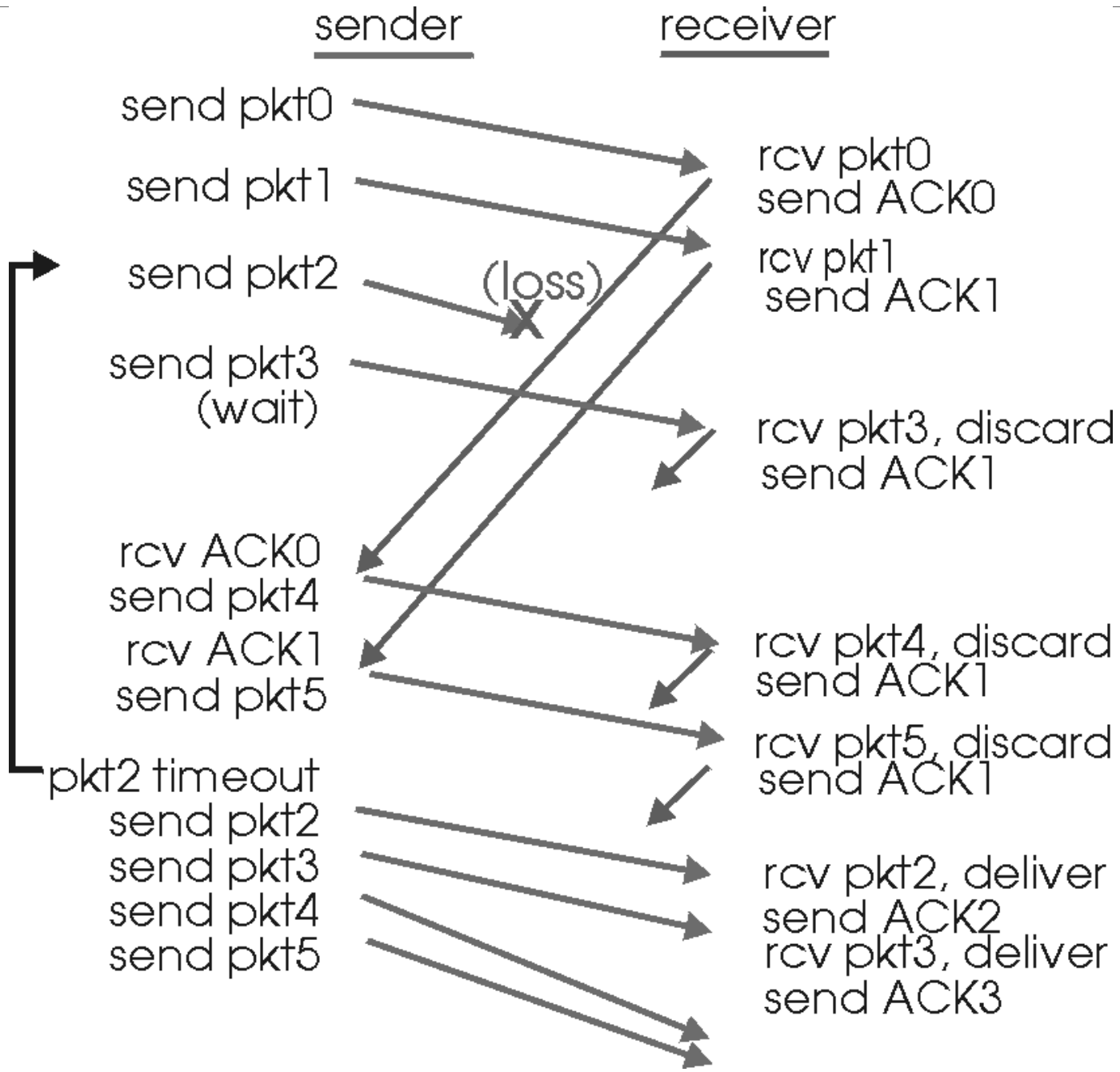
DESTINATARIO

- "ACK cumulativi" da parte del destinatario → ACK(*n*) conferma che sono arrivati correttamente i primi *n* segmenti
- I segmenti che arrivano fuori sequenza vengono scartati senza essere inseriti nel buffer
- Quindi;
  - C'è un buffer di ricezione, ma non c'è necessità di una sliding window lato destinatario per gestire il pipeling, ma 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

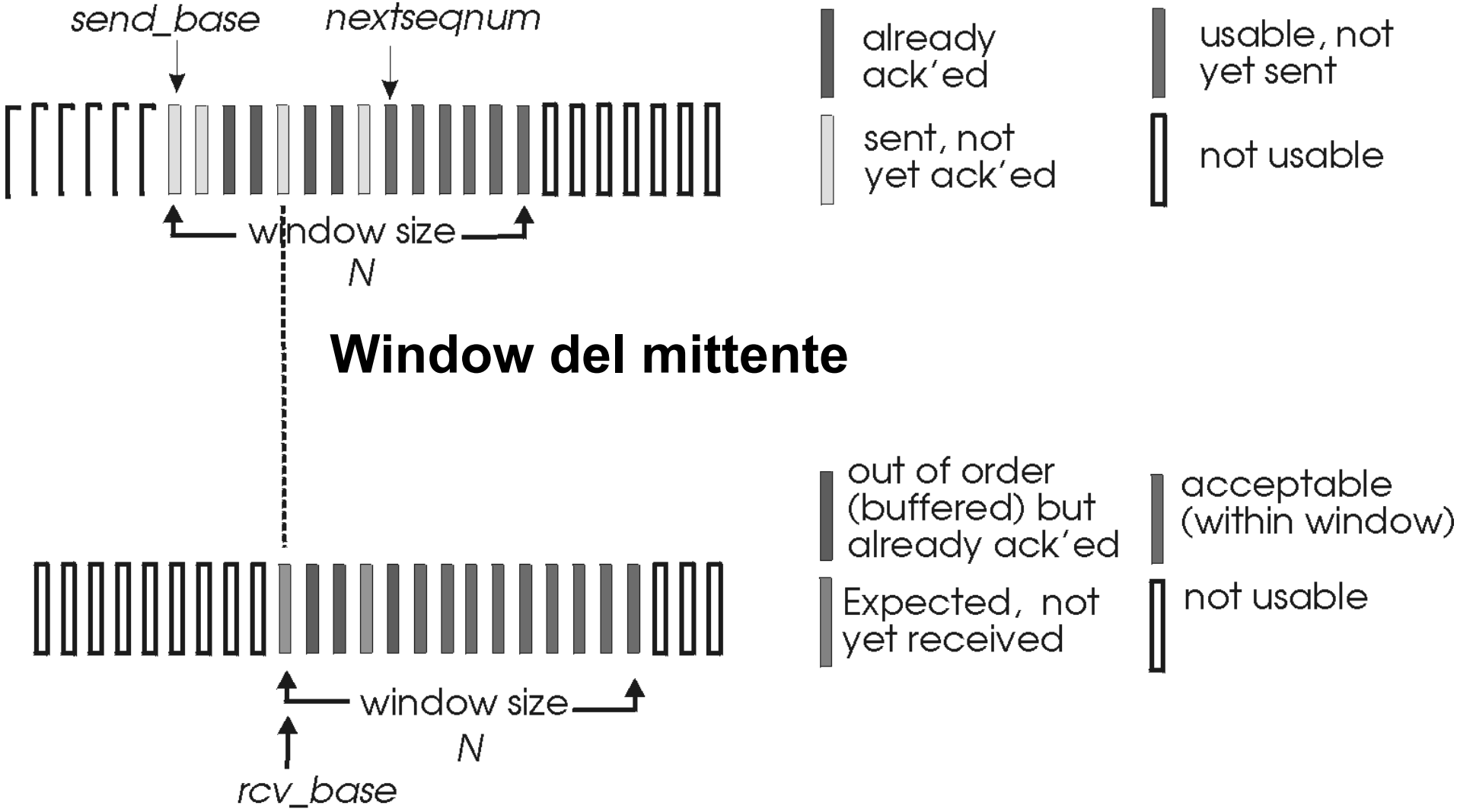


Algoritmo 2: *Ritrasmissione selettiva*

- **Il destinatario invia ACK relativo a ciascun segmento ricevuto 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
- **Sia mittente sia destinatario gestiscono la propria *sliding window***

Finestra mittente e destinatario

Ipotesi: finestra di dimensione N =14



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*: segnala come consegnabile al processo applicativo; avanza la finestra al segmento successivo non ancora ricevuto

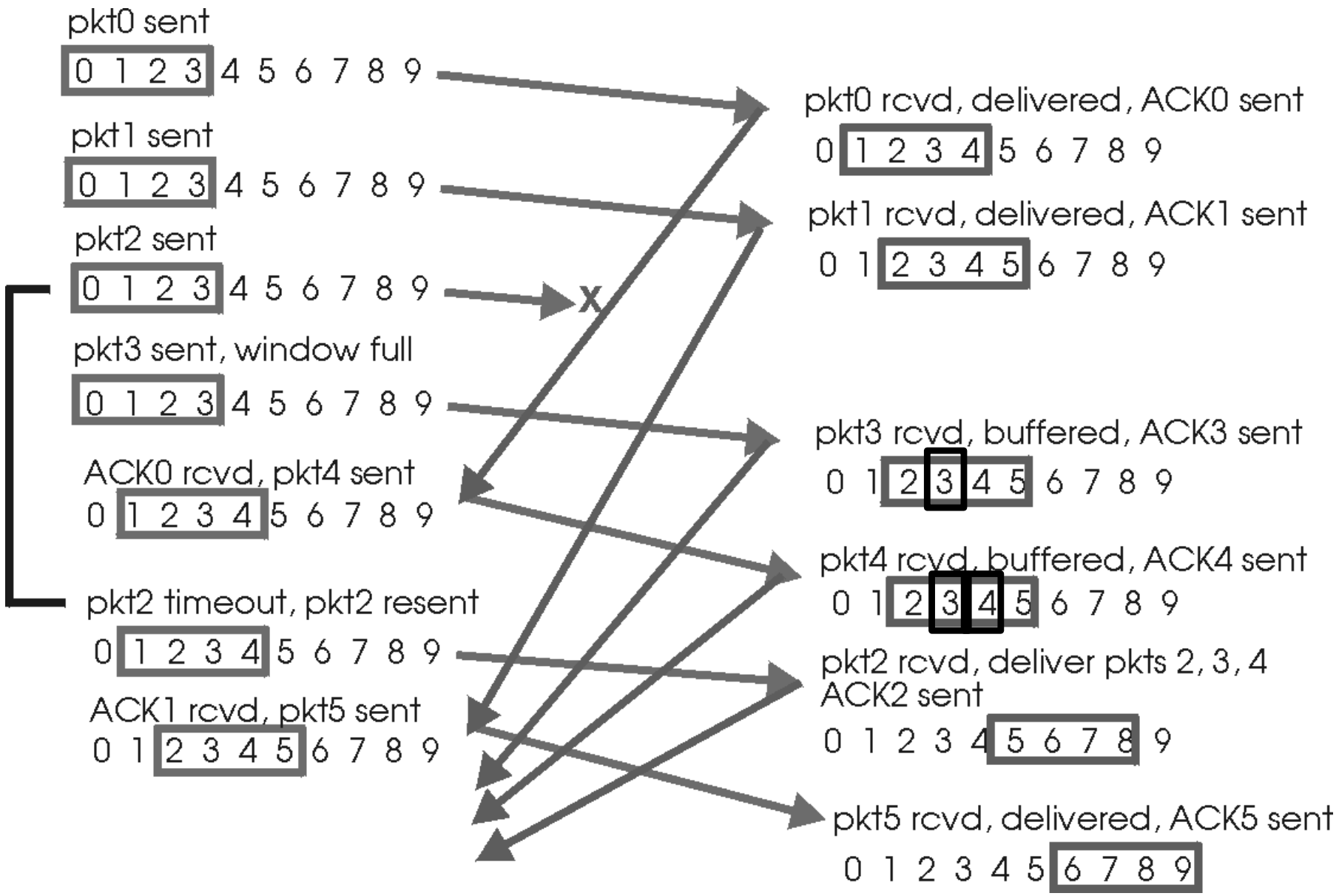
Segmento *i* ricevuto in [rcv_base-*N*, rcv_base-1]

- ACK(*i*)

Altrimenti:

- ignora

Funzionamento *Ritrasmissione selettiva*



Sliding window nel TCP

Protocollo TCP e sliding window

- La dimensione della sliding window del TCP non è fissata staticamente, ma al contrario il **calcolo continuo e dinamico della dimensione della sliding window è alla base delle capacità adattative del TCP e della sua fairness**
- Attualmente il TCP utilizza un approccio che possiamo considerare **ibrido** fra *Go-Back-N* e *Ritrasmissione Selettiva)*

Algoritmo del protocollo TCP

Il TCP non segue *Go-Back-N* e *Ritrasmissione Selettiva*, in quanto utilizza:

➔ **"ACK cumulativi"** da parte del destinatario → ACK(*n*) conferma che sono arrivati correttamente i primi *n* byte dei segmenti dati inviati (come nel *Go-Back-N*)

➔ Segmenti arrivati fuori ordine, vengono salvati nel buffer di ricezione (come nella *Ritrasmissione Selettiva*)

Algoritmo del protocollo TCP (2)

- 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 e "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 ($d \geq 1$)

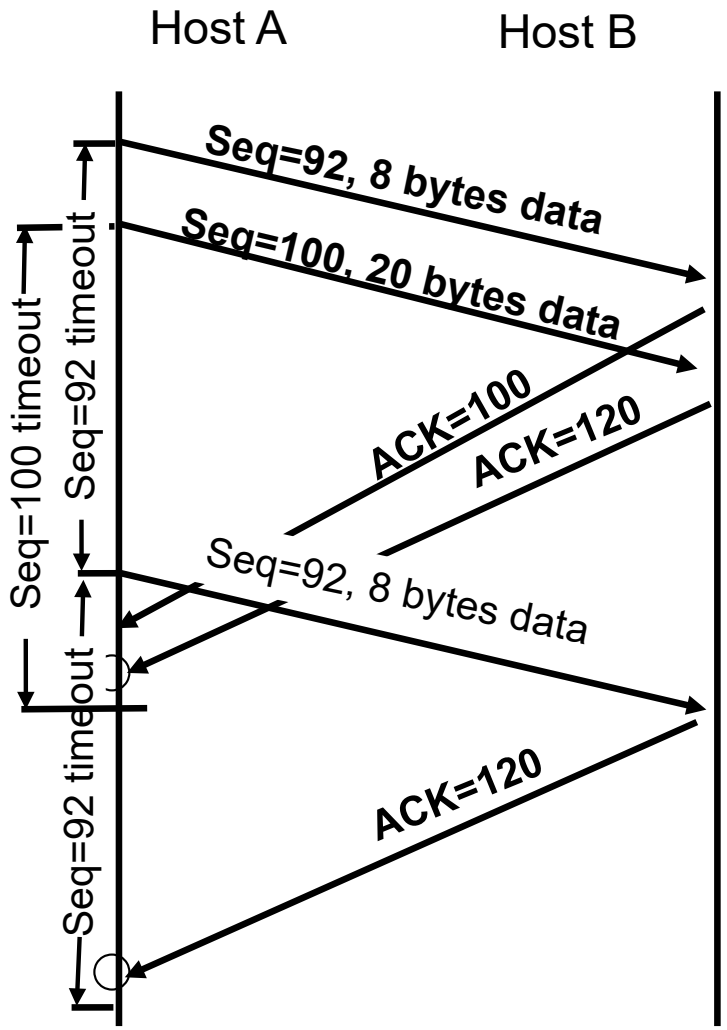
Algoritmo protocollo TCP (esempio)

- 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)
- 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

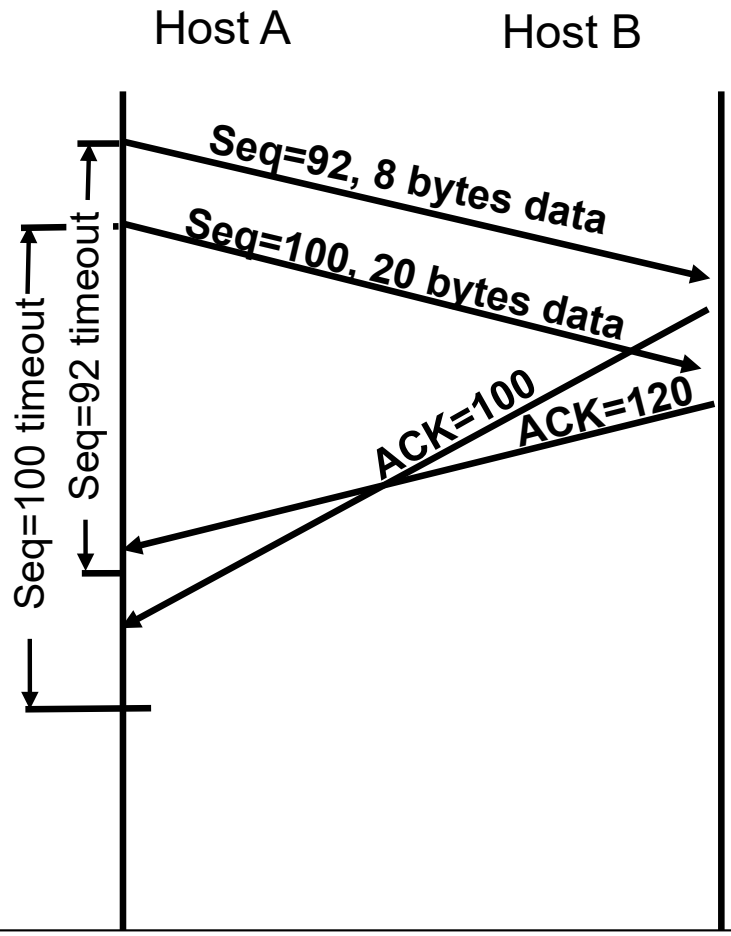
Algoritmo del protocollo TCP (*conseguenze*)

- 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*)
- In caso di time-out, la quantità di dati trasmettibile diminuisce perché il mittente non è in grado di far avanzare la finestra e 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 (protocollo *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?

Calcolo della dimensione

- 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"**

Entrambi gli obiettivi vengono raggiunti determinando come *adattare dinamicamente* la dimensione della *sliding window mittente*

Controllo della congestione nel TCP

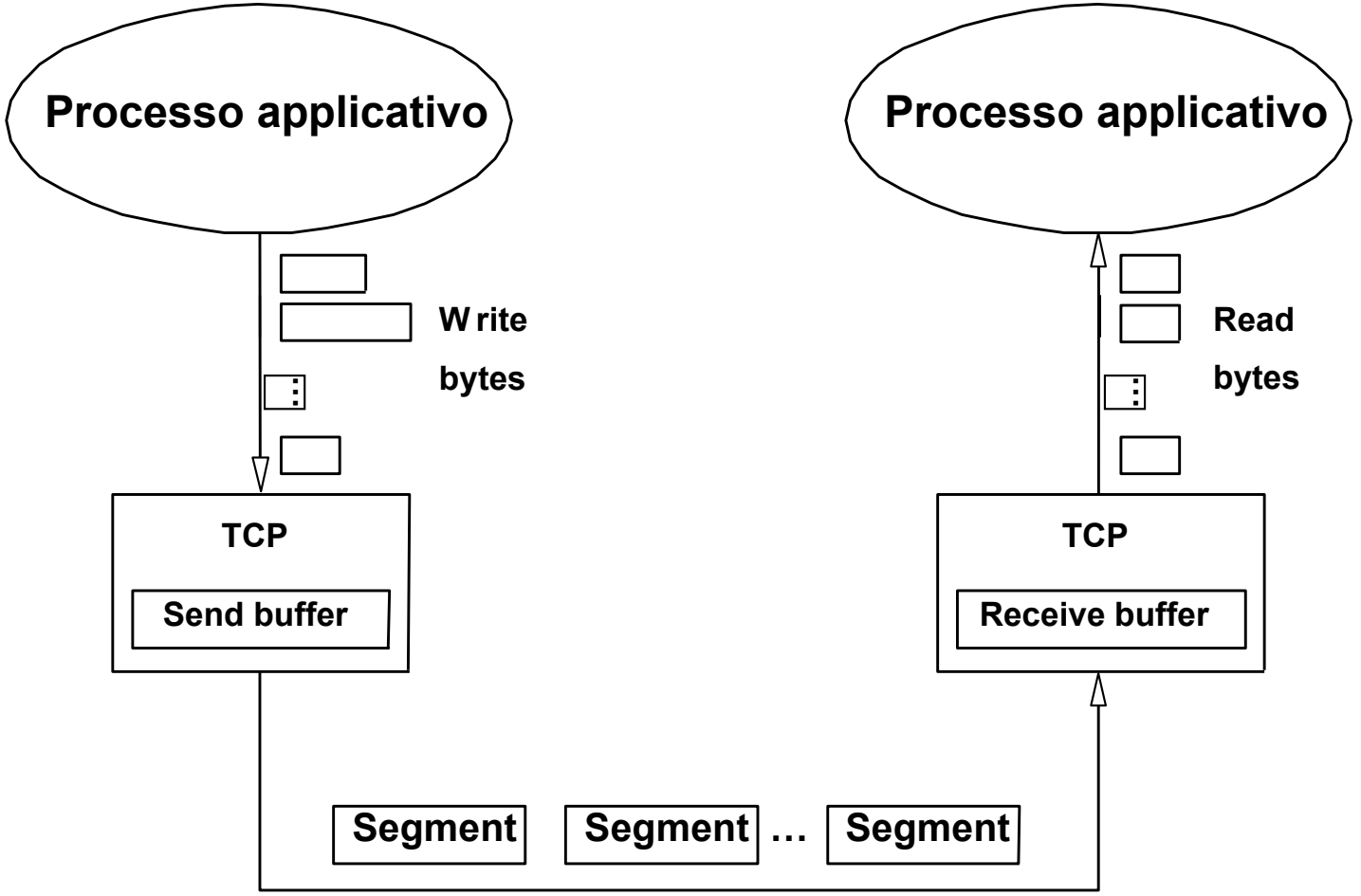
La window size del TCP è calcolata come minimo tra:

- **CongestionWindow (CW):** dimensione della finestra di congestione
- **FlowWindow (AW):** dimensione della finestra di controllo di flusso

Effective window = min(FW, CW)

Calcolo della dimensione della sliding window per il *Controllo di flusso*

Trasmissione dati nel TCP



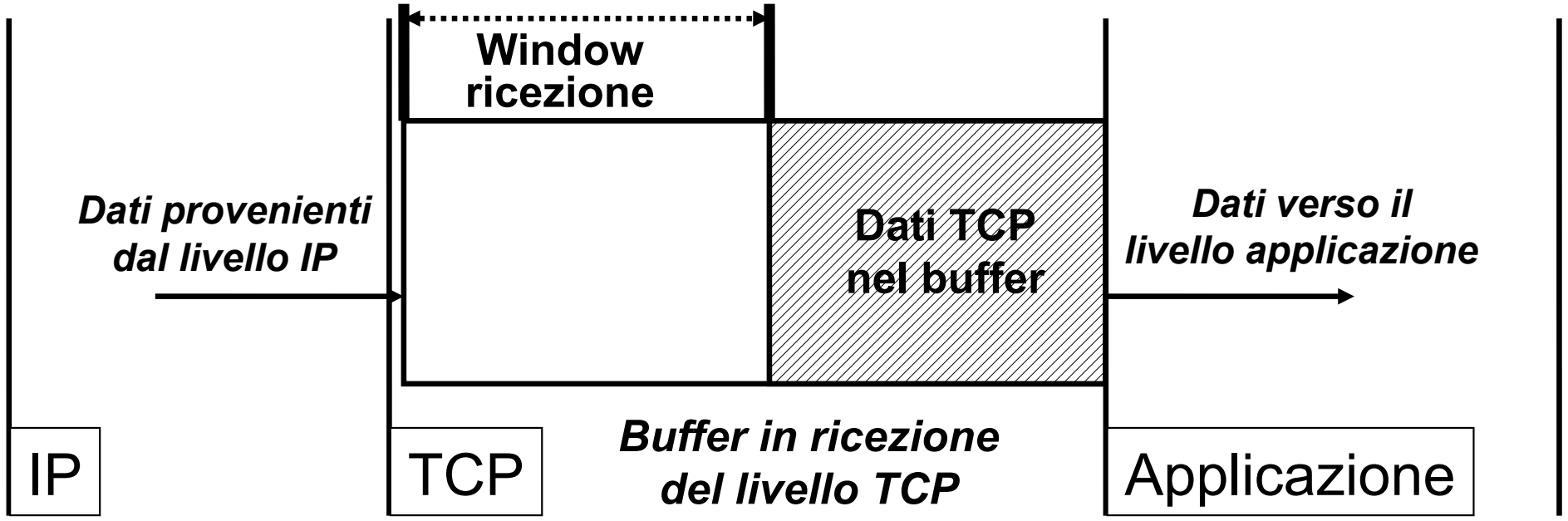
Motivazioni per buffering: *asincronia*

- **Il TCP fa parte del Sistema Operativo, non del livello applicativo che gestisce l'invio e la ricezione dei dati**
- Deve tener conto di tutti gli eventi che si verificano in modo **asincrono**: il livello TCP del mittente non sa quando il processo applicativo deciderà di spedire dati
- ***Il controllo di flusso*** può limitare la decisione di invio in base alle **informazioni ricevute esplicitamente dal destinatario** riguardo la sua capacità di ricevere dati

Controllo di flusso

***Obiettivo*: il mittente non deve riempire il buffer del destinatario inviando una quantità eccessiva di dati a 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)**

- **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)

$$AdvertisedWindow = MaxRcvBuffer - ((NextByteExpected-1)-LastByteRead)$$

FlowWindow: 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):

$$FlowWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)$$

Calcolo della dimensione della sliding window per il *Controllo di congestione*

Controllo della 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 ≠ 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 per cui i due host non conoscono la banda tra essi**
  - **la situazione di congestione è determinata analizzando le perdite di pacchetti ed i ritardi nei nodi terminali**
  - **approccio utilizzato dal TCP**
- **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 del TCP

1. **CONTROLLO DI CONGESTIONE "END-TO-END" (cioè, regolato da mittente e destinatario, non dalla rete)**

2. **"SLOW START"** (non sempre nelle ultime versioni dei SO)

3. **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 della congestione nel TCP

SCELTA: **Congestione end-to-end** → nessuna informazione proveniente dalla rete

Timeout e ritrasmissione contribuiscono ad aumentare la congestione

TCP riduce il tasso di trasmissione in caso di congestione scegliendo il minimo tra:

CongestionWindow (CW): dimensione della finestra di congestione

FlowWindow (AW): dimensione della finestra di controllo di flusso

Effective window = min(FW, CW)

Controllo della congestione nel TCP (2)

- **Misura delle prestazioni di una connessione TCP**:

  - **Throughput** (tasso di trasmissione)

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

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

Due fasi per controllo congestione

- ***Slow-start***
  - all'inizio dell'utilizzo di una nuova connessione la dimensione della finestra di congestione è pari a pochi MSS (e.g., 1 o 2)
  - per ogni segmento acknowledged, incremento progressivo della dimensione della finestra di poche unità in modo costante (e.g., 1 MSS) o esponenziale (e.g., raddoppio ogni volta)
  - ***threshold***: valore della dimensione della finestra, raggiunto il quale lo slow-start termina e si entra in fase di *congestion avoidance*

- ***Congestion avoidance***
  - stato non di congestione: la dimensione della finestra di congestione è incrementata di 1 MSS per ogni segmento acknowledged
  - stato di congestione: nel caso di time-out o ACK duplicati, c'è un ritorno alla fase di *slow start* (diverse versioni del TCP settano i parametri della congestion window in modo differente)

Generico slow start del TCP

Algoritmo di slow-start

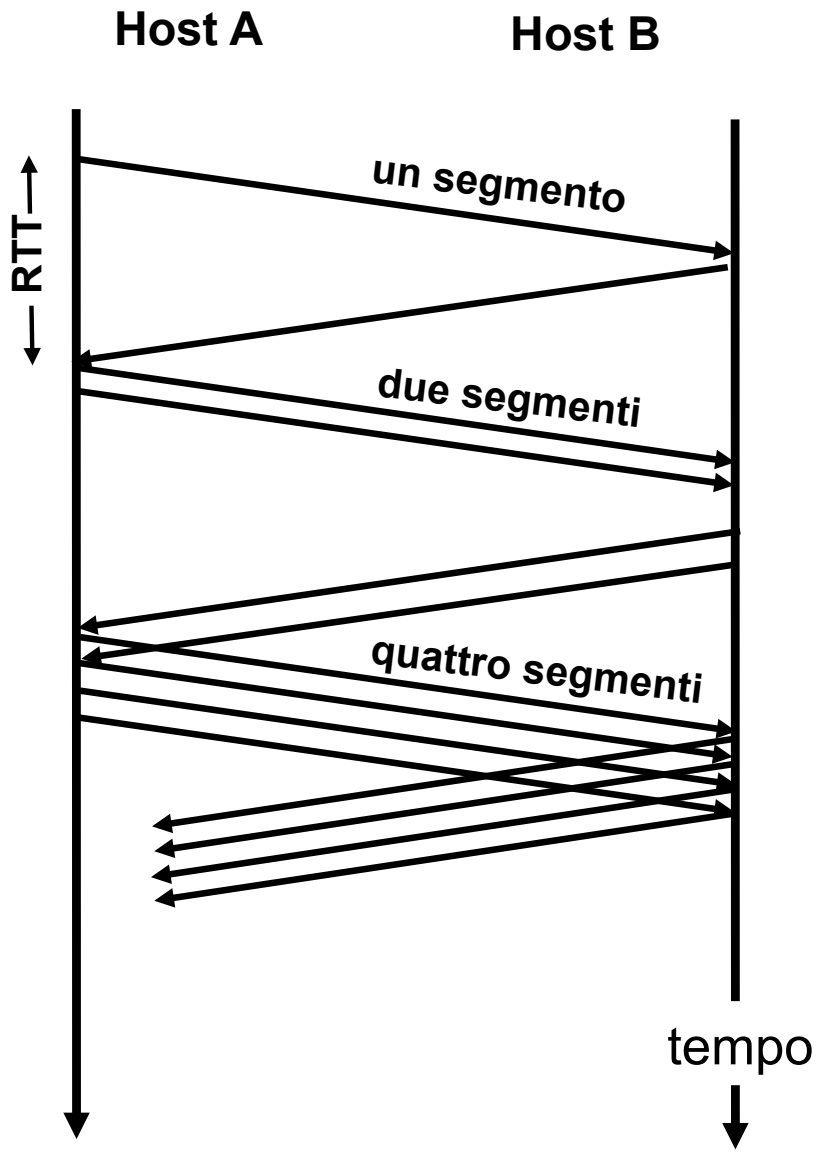
Inizializza: CW = 1

for (ogni segmento ACK)

CW=CW*2

until ((timeout OR

(CW > threshold))



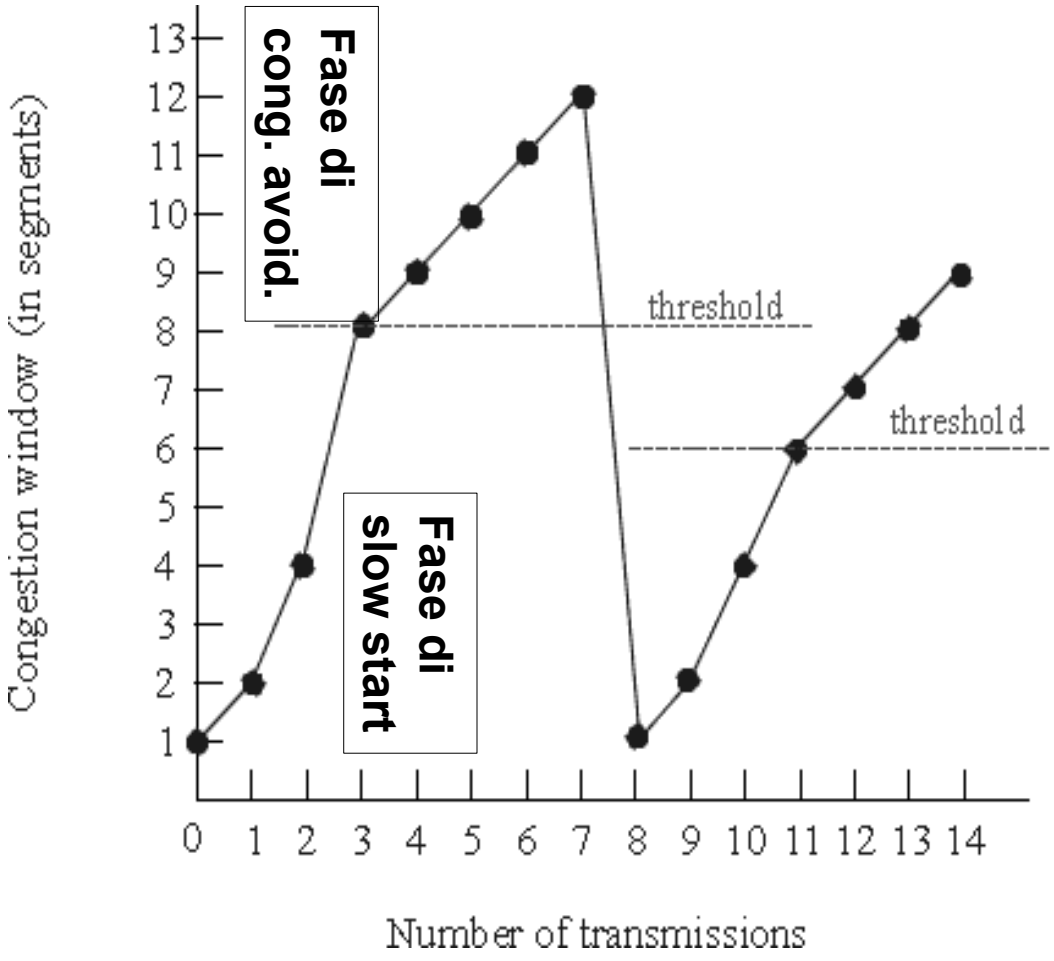
- log N RTT prima di usare finestra di dimensione N → **aumento esponenziale**

Calcolo CW: Algoritmo *Tahoe*

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

Tahoe

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



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:

  ➔ **Meccanismo dell'ACK duplicato**

Ritrasmissione rapida (*cont.*)

- 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 *i* → del tutto simile ad un "not ack" (NACK) sul segmento successivo *i*+1

Calcolo CW: 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

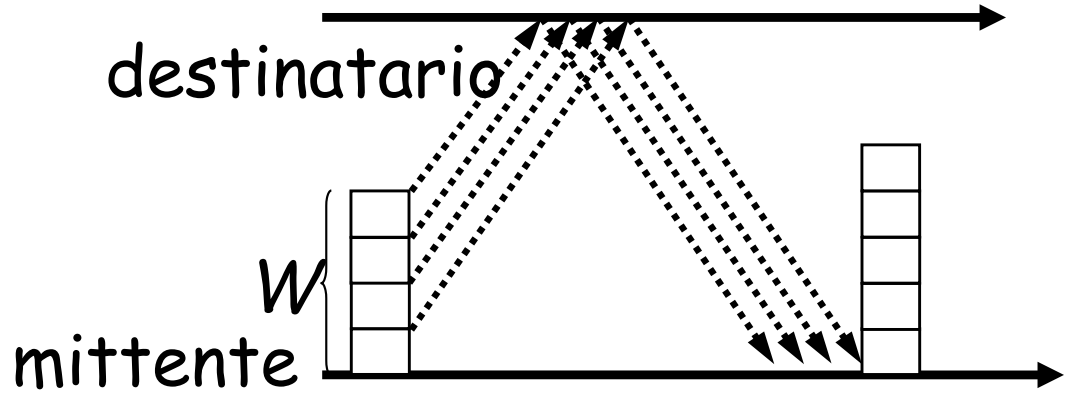
  (<u>MOTIVAZIONE</u>: 1 segmento si è perso, ma la rete funziona perché almeno 3 segmenti sono stati ricevuti dal destinatario)

Reno

```
/* 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
```

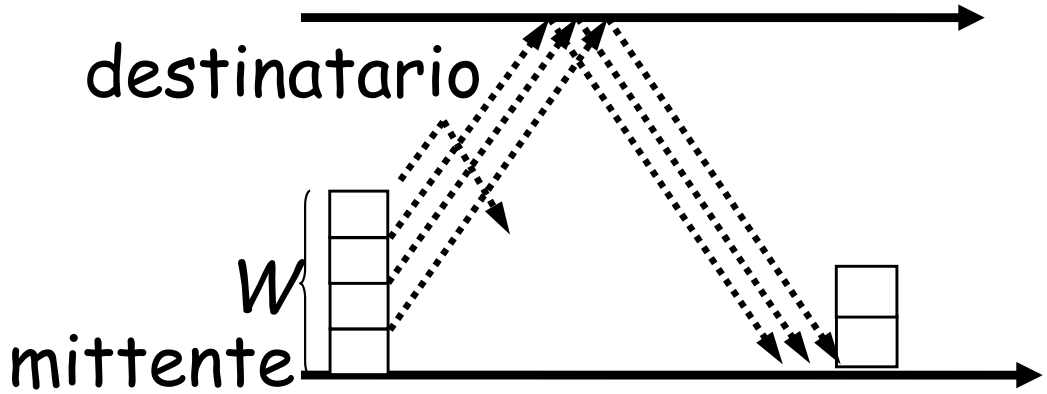
Es., controllo congestione: *Reno*

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



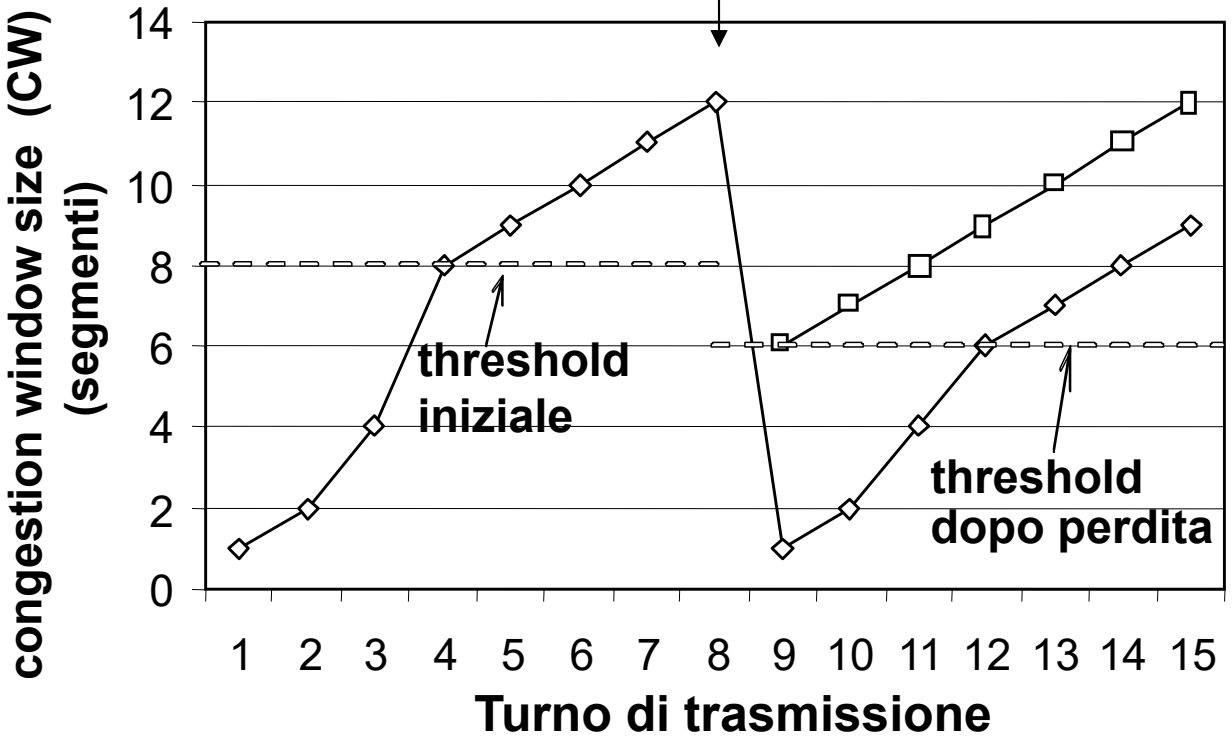
- Riduci la window di metà se viene evidenziata una perdita con un triplo ACK duplicato:

  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

Tante altre versioni

Non c'è una scelta migliore in assoluto: le tecnologie cambiano, i tipi di rete cambiano

- **Vegas**

- **New Reno** (dal kernel Linux 2.6.2)
- **TCP BIC** (dal kernel Linux 2.6.8 a 2.6.18)
- **TCP Cubic** (dal kernel Linux 2.6.19)
- **TCP Proportional Rate Reduction** (da Linux 3.2)

- **Compound TCP** (da Windows Vista)

Es., Controllo di congestione “Vegas”

- Cerca di comprendere problemi di trasmissione, prima che si verifichino
  - Approccio pro-attivo 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 dimensione della CW nel momento in cui si osserva un continuo aumento del RTT**

Congestion detection in TCP Vegas

Throughput(*expected*) = CW / RTT

Throughput(*actual*) =
(n. byte trasmessi precedente RTT) / RTT

X = Throughput(*expected*) - Throughput(*actual*)

- **Si definiscono due soglie: $\alpha < \beta$**

Congestion detection in TCP Vegas

- **Se $X<\alpha$**
  - Incrementa la CW linearmente
- **Se $X>\beta$**
  - Diminuisci la CW linearmente
- **Se $\alpha<X<\beta$**
  - Lascia la CW invariata

TCP Reno *vs.* TCP Vegas

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

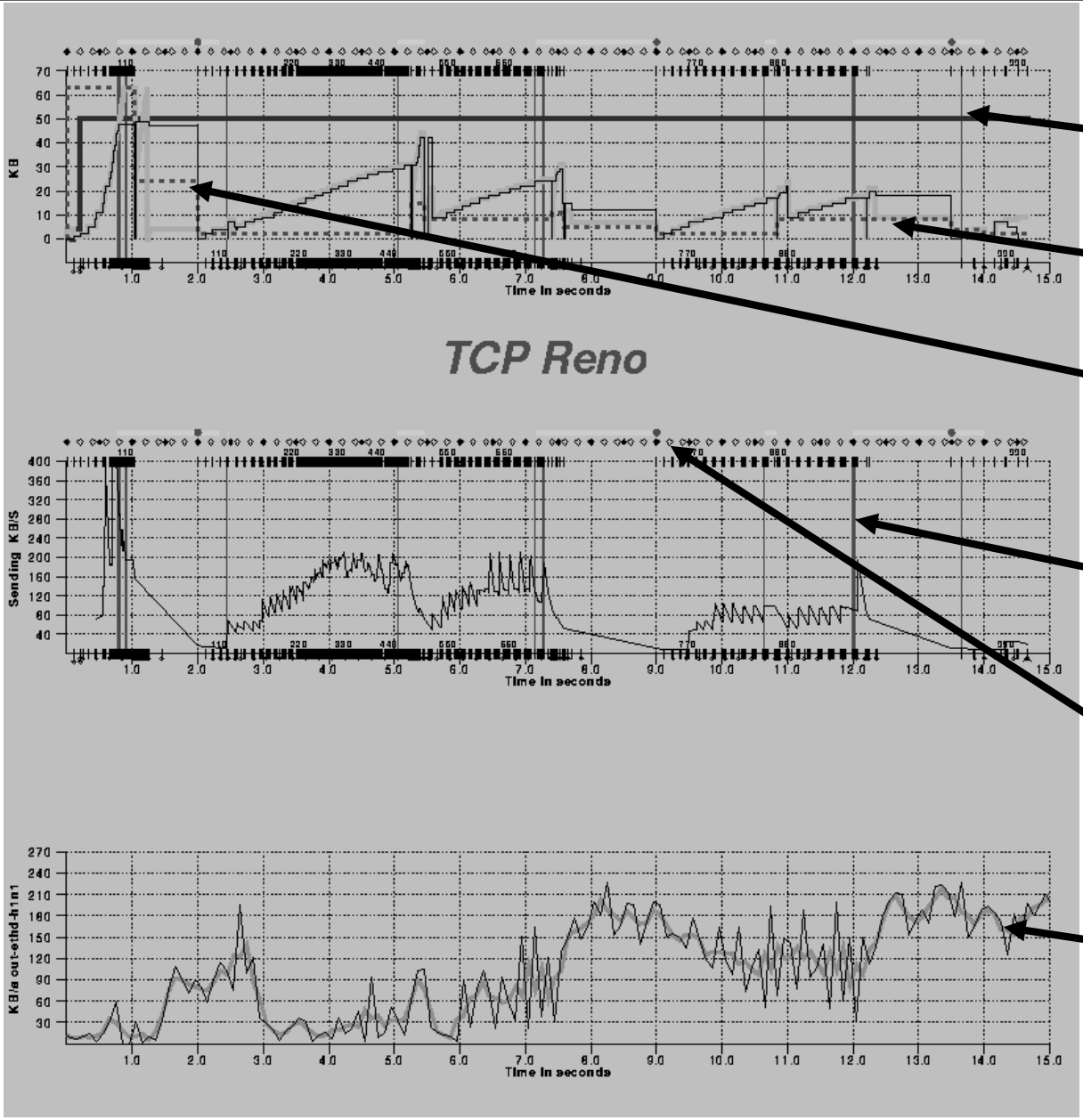
- Individuazione della perdita di pacchetti
  - Timeout
  - RTT
  - Verifiche basate su timer a grana fine per ogni pacchetto

TCP Reno *vs.* TCP Vegas

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

- Congestion detection
  - Aumento di RTT
  - Il throughput diventa insensibile al send rate
- Gestione della CW separata da slow start:
  - Si considera una funzione per il calcolo di X come differenza tra precedente CW e attuale CW, entrambi rapportati a RTT
  - Se X>0 diminuisci CW di 1/8
  - Se X≤0 aumenta MSS

Analisi del protocollo TCP Reno



Send window (buffer size)

Congestion window

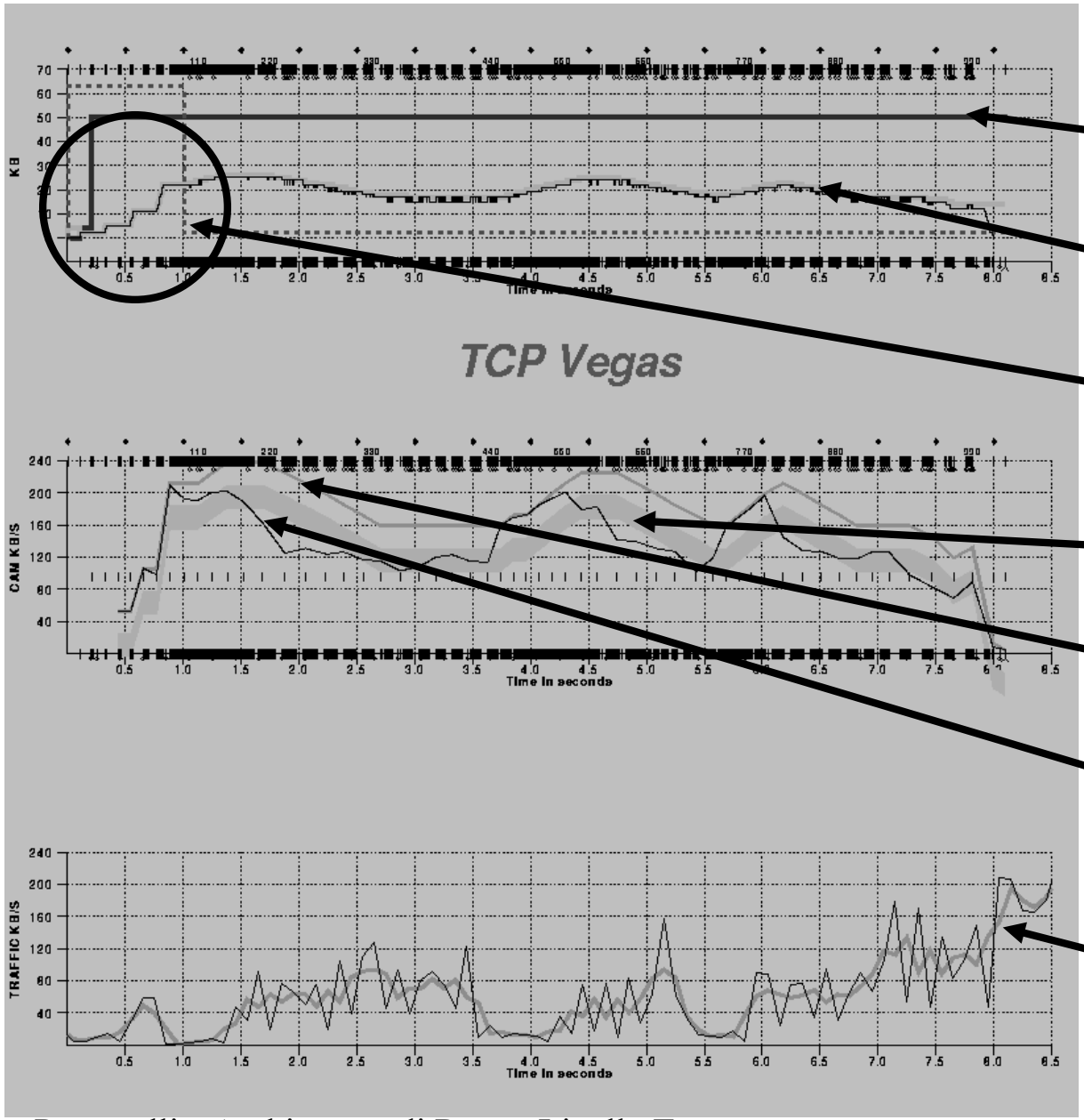
Threshold slow start

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: problemi e soluzioni

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
- Se la situazione persiste, può capitare che il buffer mittente si riempia e il TCP deve bloccare la generazione di nuovi dati da parte del processo

  CONSEGUENZA: Un processo applicativo lento sul destinatario può bloccare un processo mittente veloce!

Come riprendere da una windows=0

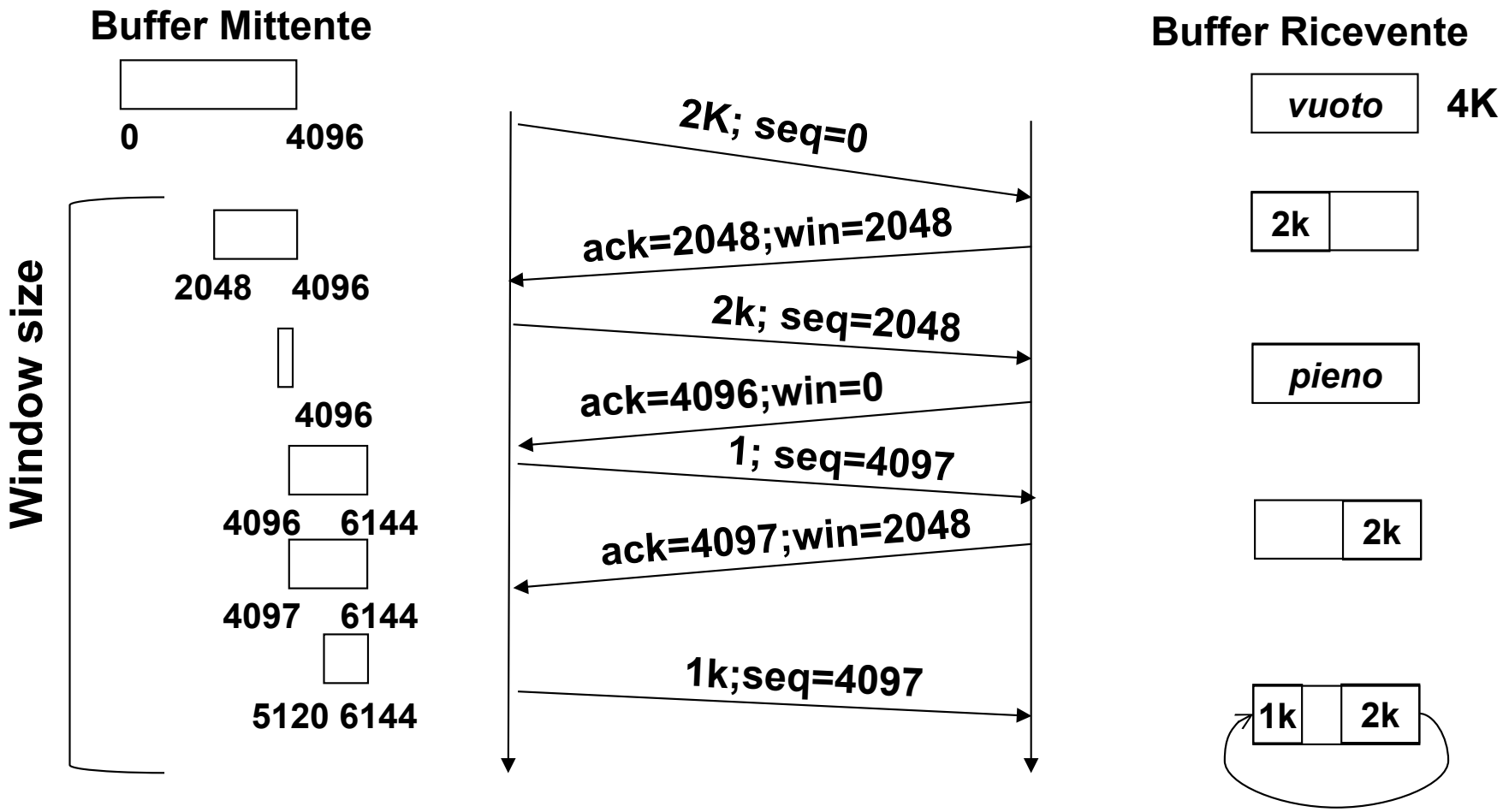
- Una volta che il destinatario ha comunicato una finestra di dimensione 0

  → 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

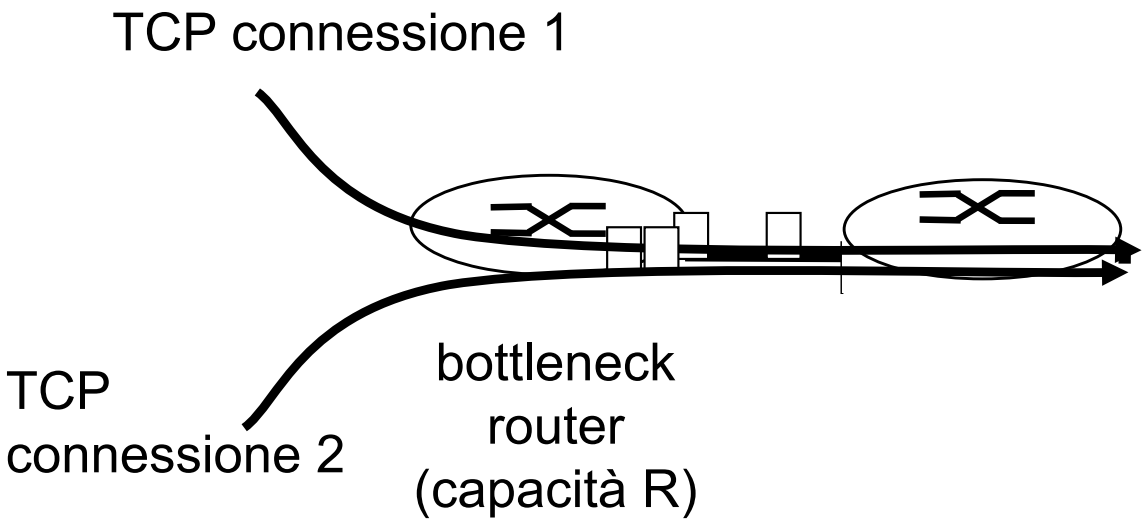
Esempio di funzionamento



Il buffer del ricevente si svuota per motivi indipendenti dalla trasmissione

Fairness del TCP

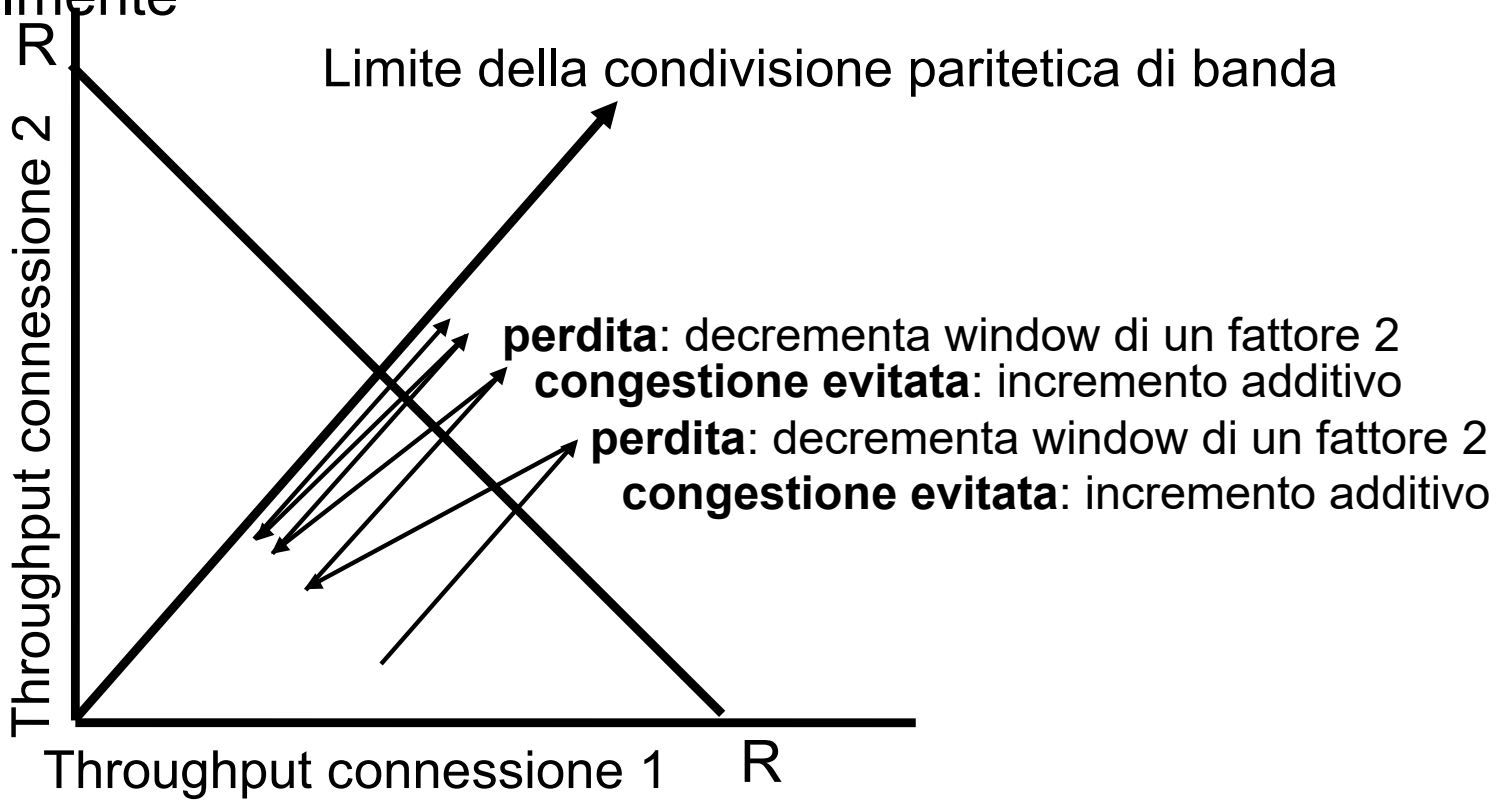
- **Scopo della *fairness*: se vi sono *N* sessioni TCP che condividono lo stesso link, ciascuna deve ottenere 1/*N*-mo della capacità del link**



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



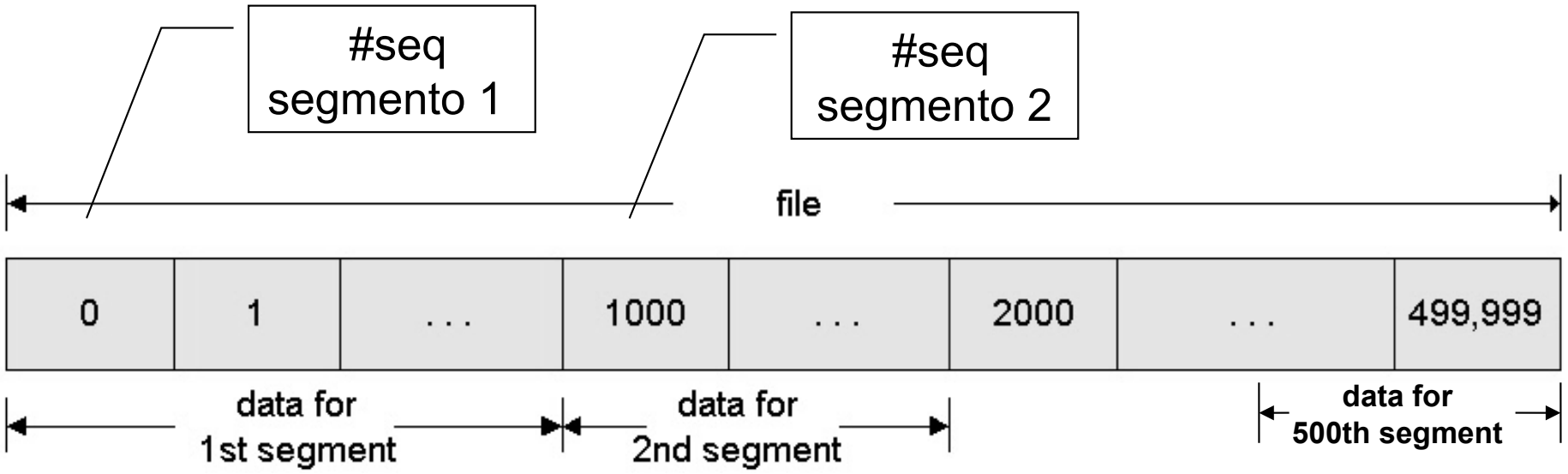
Numeri di sequenza e acknowledgment

Numero di sequenza per un segmento TCP è dato da:

- primo ISN random
- poi offset del primo byte del flusso dati inviati dal mittente

(L'*Initial Sequence Number* è scelto casualmente con l'obiettivo di minimizzare la probabilità che sia presente un segmento identificato con lo stesso numero appartenente ad una connessione precedente con identici numeri di porta)

Esempio: Si supponga di trasferire un file di 500000 byte, con MSS=1000 byte. I numeri sequenza saranno: X, X+1000, X+2000, …



Numeri sequenza e acknowledgment (2)

Il numero di acknowledgment per un segmento TCP:

- TCP è full duplex: l'host A può ricevere dati dall'host B mentre sta inviando dati a B sulla stessa connessione
- Segmento da B a A:

  - **numero di sequenza**: numero sequenziale del byte del flusso dati

  - **numero di acknowledgment**: numero di sequenza del successivo byte che A si aspetta di ricevere da B (tutti i byte precedenti sono stati ricevuti (***acknowledgement incrementale***)

Esempi

- A ha ricevuto da B i segmenti da 0 a 999 byte e da 1000 a 1999 byte, per cui il numero di acknowledgment nel segmento da A a B → 2000
- A ha ricevuto da B i segmenti da 0 a 999 byte e da 2000 a 2999 byte, per cui il numero di acknowledgment nel segmento da A a B → 1000