Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2019-2020

Pietro Frasca

Parte II: Reti di calcolatori

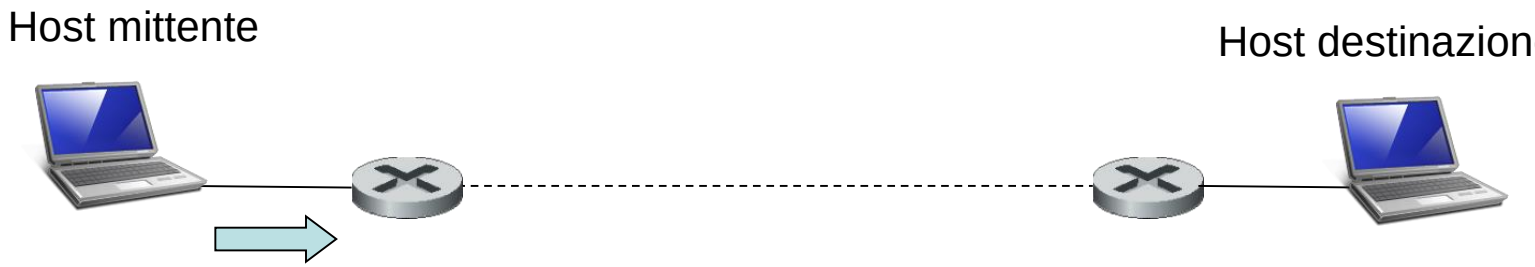
Lezione 15 (39)

Giovedì 30-04-2020

Trasferimento affidabile dei dati

- Il protocollo IP dello strato di rete è inaffidabile, non garantisce né la consegna dei **datagram** e neanche l'integrità dei dati in essi contenuti. I datagram possono arrivare fuori ordine, e i bit nei datagram possono subire modifiche passando da 0 a 1 e viceversa.
- Poiché i segmenti dello strato di trasporto costituiscono il **campo dati** dei datagram IP, anch'essi possono essere soggetti a questi problemi.
- Il TCP realizza un **servizio di trasferimento affidabile dei dati** utilizzando il servizio inaffidabile fornito da IP.
- Il servizio di trasferimento affidabile dati di TCP assicura che tutti i dati inviati dal mittente arrivino integri e nello stesso ordine al destinatario.

- Vediamo come il TCP realizza un trasferimento affidabile di dati, descrivendo un caso semplificato in cui un mittente TCP ritrasmette segmenti solo allo scadere del timeout. In seguito descriveremo anche il caso in cui il client usa riscontri duplicati, oltre ai timeout, per rinviare i segmenti persi.
- Il seguente pseudo codice è relativo a una descrizione molto semplificata di un **mittente TCP**, non considerando la frammentazione del messaggio, il controllo del flusso e della congestione.



```
numSequenza_min = numSequenza_iniziale
numSequenza = numSequenza_iniziale
while (true) {
  switch(evento)
    case evento1: /* dati ricevuti dallo strato applicativo (messaggio) */
      <crea il segmento TCP con il numero di sequenza numSequenza>
      if (<timer non è avviato>)
            <avvia il timer>
      <passa il segmento a IP>
      numSequenza = numSequenza + lunghezza(dati)
      break;
    case evento2: /* timer timeout (il timeout è dato dal valore di
     intervalloTimeout) */
      <ritrasmette il  segmento non ancora riscontrato
        con numero di sequenza sequenza_min (il più piccolo numero di
     sequenza)>
      <avvia il timer>
      break;
    case evento3: /* ACK ricevuto, con valore del campo numero di riscontro
     = numRiscontro */
      if (numRiscontro > numSequenza_min) {
        numSequenza_min = numRiscontro
        if ( <ci sono segmenti non ancora riscontrati>)
          <avvia timer>
      }
}
```

- Vediamo che ci sono tre principali eventi legati alla trasmissione e ritrasmissione dei dati nel mittente TCP:
  1. **ricezione di dati dall'applicazione;**
  2. **timeout del timer;**
  3. **ricezione di un ACK.**

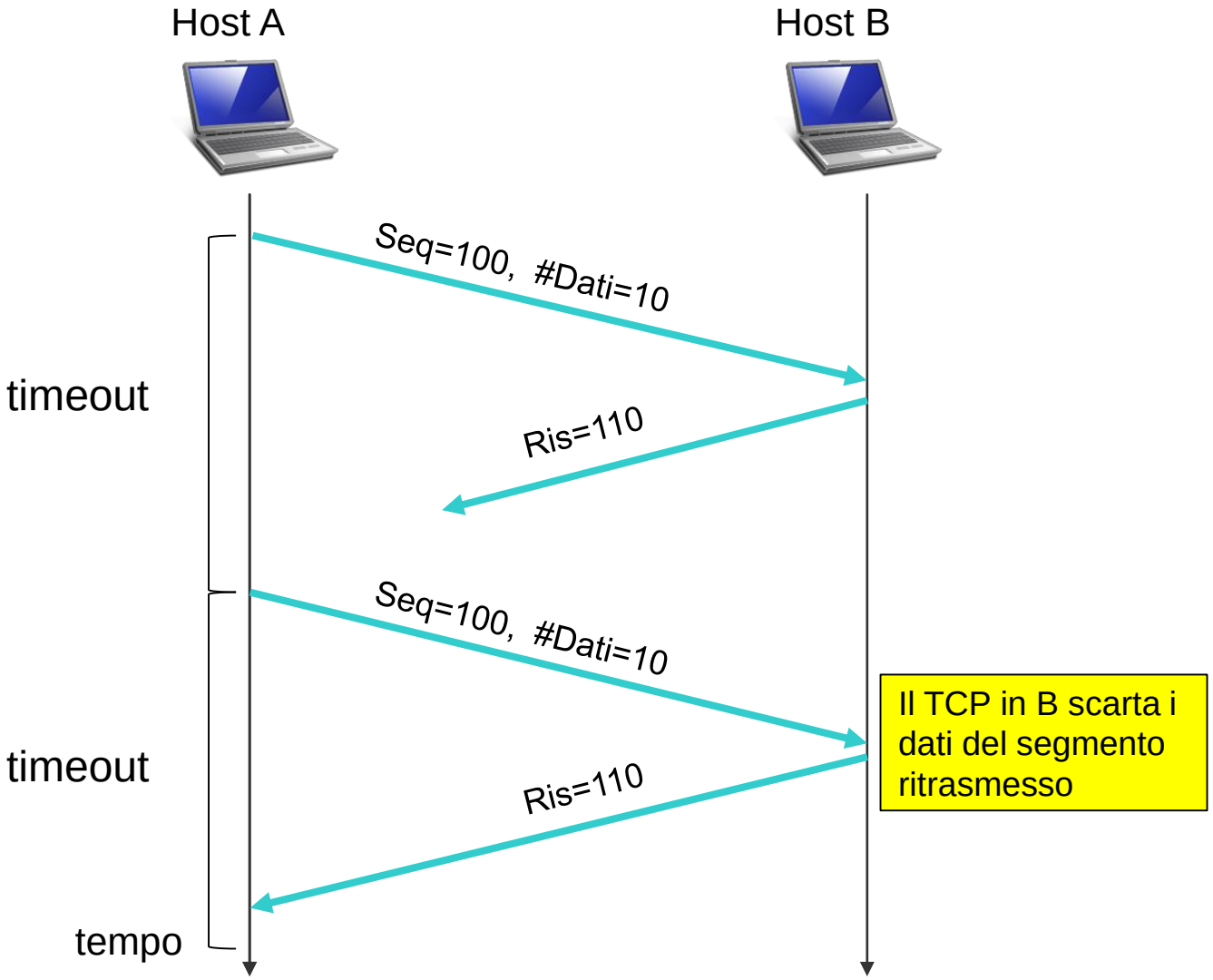
1. Al verificarsi del primo degli eventi principali, il TCP riceve i dati dall'applicazione, li incapsula in un segmento, e passa il segmento a IP. Ogni segmento ha un numero di sequenza. Se il timer non è già stato avviato per qualche altro precedente segmento, il TCP avvia il timer nel momento in cui il segmento viene passato a IP. Il valore di scadenza per questo timer è dato da **intervalloTimeout**, che è calcolato da **RTTstimato** e **DevRTT** come descritto precedentemente.

2. Il secondo evento principale è l'evento **timeout**. Il TCP risponde all'evento timeout ritrasmettendo il segmento che ha causato il timeout stesso. Il TCP quindi riavvia il timer.
3. Il terzo evento è l'arrivo di un segmento di riscontro (ACK) dal ricevente. Al verificarsi di questo evento, il TCP confronta il valore **numRiscontro** contenuto nel campo numero di riscontro con la sua variabile **numSequenza_min**. La variabile di stato **numSequenza_min** è il **numero di sequenza del più vecchio byte non riscontrato**. Come indicato in precedenza il TCP usa riscontri cumulativi, così che **numRiscontro** riscontra la ricezione di tutti i byte prima del byte numero **numRiscontro**. Se **numRiscontro > numSequenza_min**, allora ACK sta riscontrando uno o più segmenti non riscontrati prima. Quindi il TCP mittente aggiorna la sua variabile **numSequenza_min**. Inoltre riavvia il timer se ci sono segmenti attualmente non ancora riscontrati.

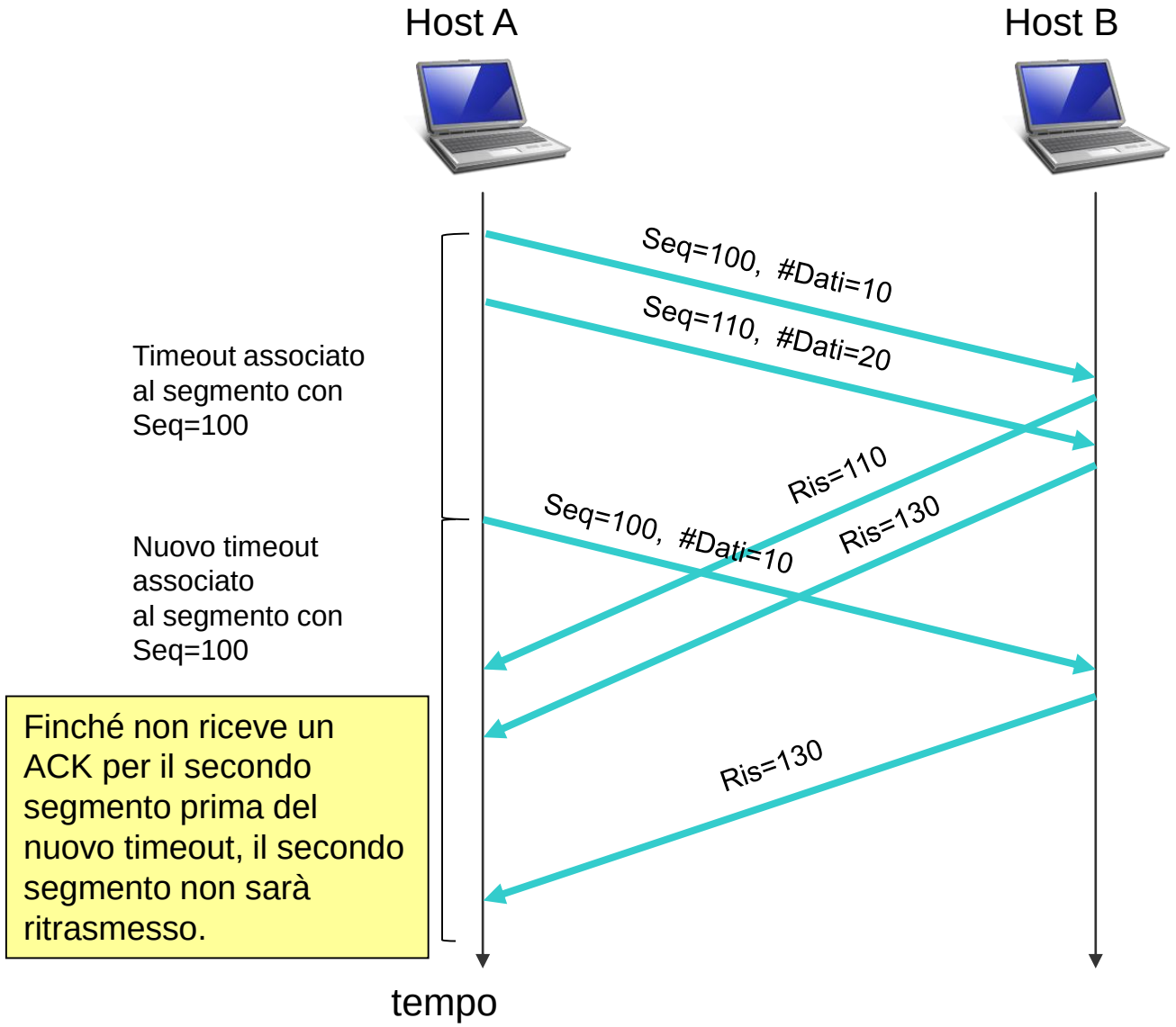
Alcuni tipici scenari

- Per avere un idea più chiara sul funzionamento del TCP esaminiamo ora alcuni tipici scenari.
- Il primo scenario è illustrato nella figura seguente in cui l'host **A** invia un segmento all'host **B**.
- Supponiamo che questo segmento abbia numero di sequenza 100 e che contenga 10 byte di dati.
- Dopo l'invio di questo segmento, l'host **A** aspetta da **B** un segmento con il numero di riscontro uguale a 110. Nonostante il segmento di **A** sia stato ricevuto da **B**, il riscontro da **B** ad **A** si è perso. In questo caso il timer scade, e l'host **A** ritrasmette lo stesso segmento. Ovviamente, quando l'host **B** riceve la ritrasmissione, rileverà dal numero di sequenza che il segmento è già stato ricevuto. Allora, il TCP nell'host **B** scarterà i dati contenuti nel segmento ritrasmesso.

Scenario 1: ritrasmissione a causa di un riscontro perso.

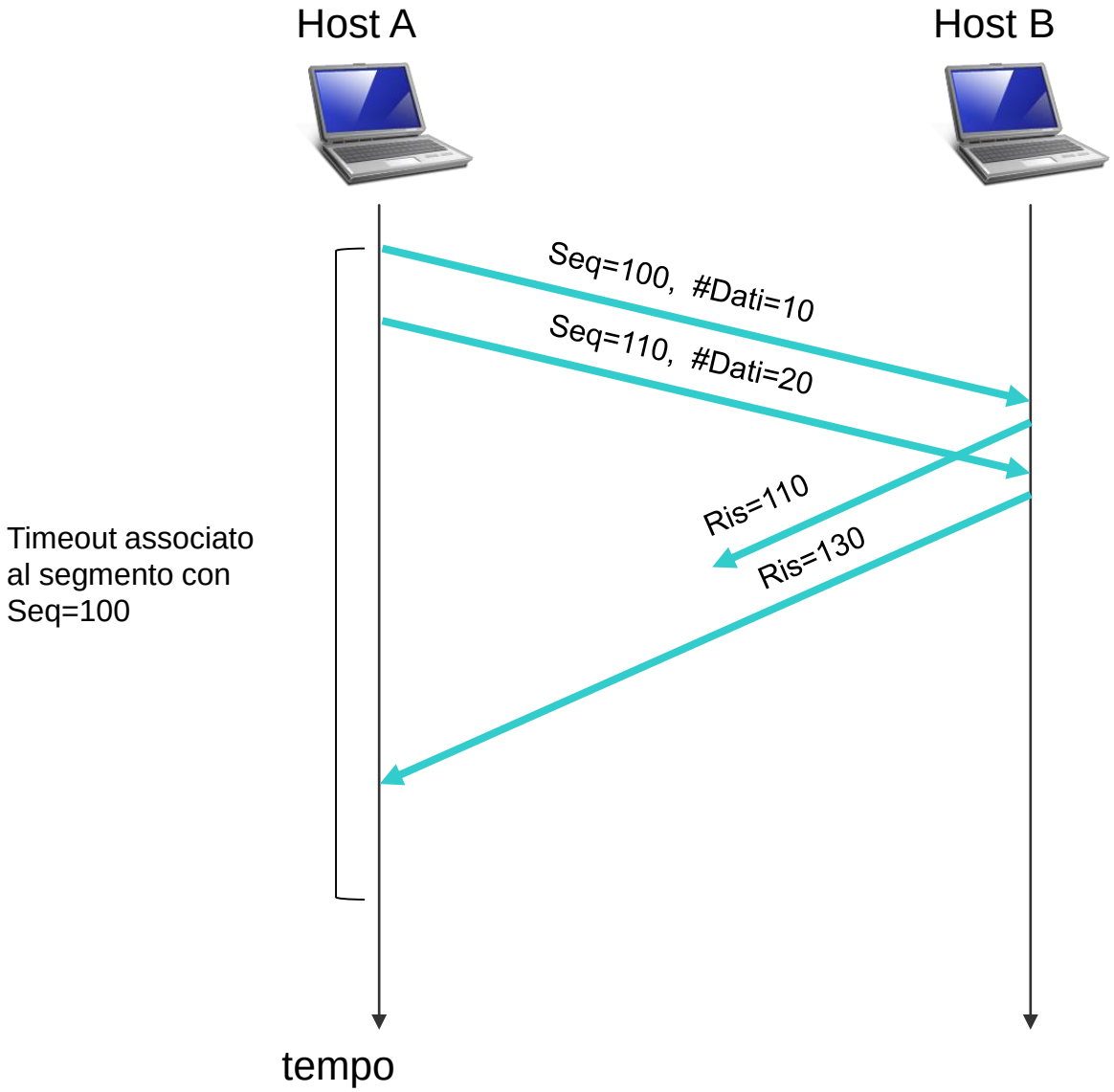


Scenario 2: segmento non ritrasmesso.



- Il primo segmento ha numero di sequenza 100 e 10 byte di dati. Il secondo segmento ha numero di sequenza 110 e 20 byte di dati. Supponiamo che entrambi i segmenti arrivino a B e che B invii due riscontri separati per ciascuno di questi segmenti. Il primo di questi riscontri ha numero di riscontro 110; il secondo ha numero di riscontro 130. Supponiamo ora che nessuno dei riscontri arrivi all'host A prima del timeout del primo segmento. Quando il timer scade, l'host A rispedisce il primo segmento con numero di sequenza 100 e riavvia il timer. **Finché non riceve un ACK per il secondo segmento prima del nuovo timeout, il secondo segmento non sarà ritrasmesso.**

Scenario 3: riscontro cumulativo che evita la ritrasmissione del primo segmento.



- Il riscontro per il primo segmento si perde nella rete, ma appena prima del timeout di questo segmento, l'host A riceve un riscontro con numero di riscontro 130. L'host A perciò sa che l'host B ha ricevuto tutti i dati fino al byte 129 e non rispedisce nessuno dei due segmenti.

Raddoppio dell'intervallo di timeout

- Quando si verifica un evento di timeout, il TCP ritrasmette il segmento non ancora riscontrato che ha il più piccolo numero di sequenza, come descritto prima.
- Ma a ogni ritrasmissione il TCP raddoppia il valore del successivo timeout rispetto al valore precedente, invece di calcolarlo con la relazione:

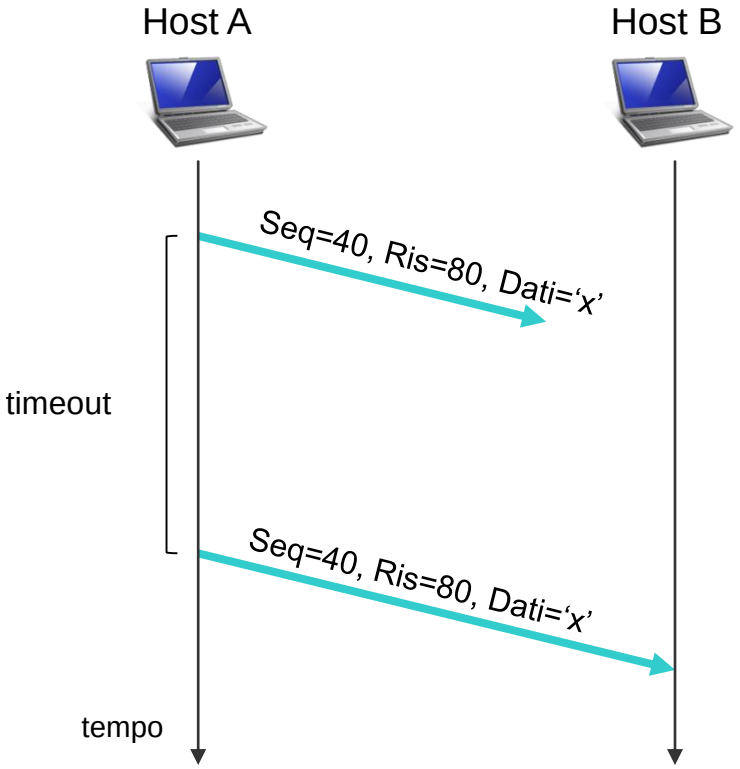
  intervalloTimeout = RTTstimato + 4·DevRTT

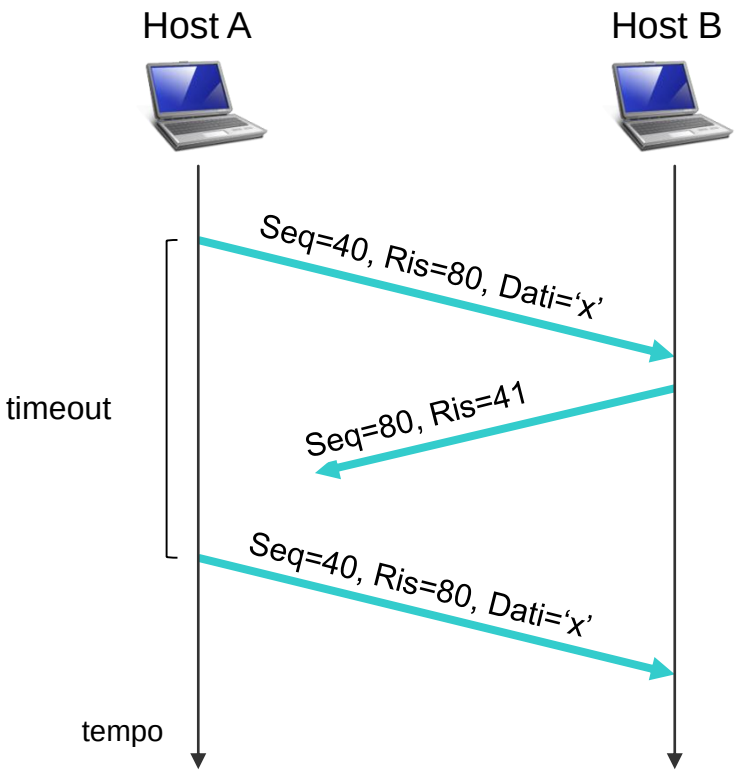
  Ad esempio, supponiamo che all'evento di timeout il valore di **intervalloTimeout** relativo al segmento non riscontrato, con numero di sequenza più piccolo, sia **250 ms**. Il TCP ritrasmetterà questo segmento e assegnerà il nuovo valore di timeout a **500 ms**. Se si verifica un nuovo timeout, dopo 0.5 secondi, il TCP ritrasmette di nuovo il segmento, assegnando ora a intervalloTimeout il valore di **1 s**.

- In tal modo, gli intervalli crescono esponenzialmente dopo ogni ritrasmissione.
- Tuttavia, se il timer viene riavviato per via dell'evento di ricezione di dati dall'applicazione o dall'evento di ricezione di un ACK, **intervalloTimeout** viene calcolato in base alla relazione descritta precedentemente.

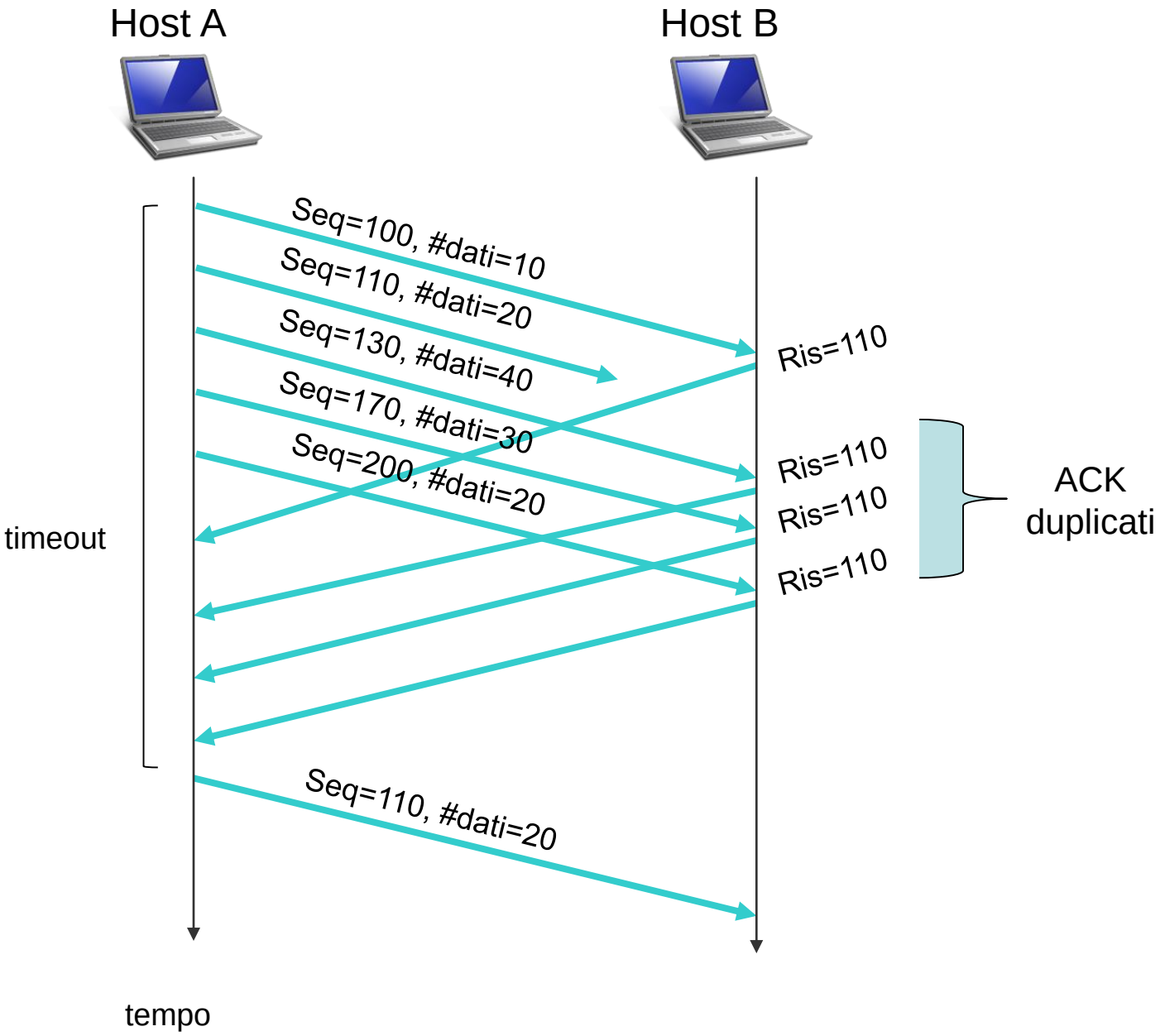
Ritrasmissione rapida

- Un importante problema relativo alle ritrasmissioni riguarda la stima della durata del timeout che potrebbe risultare troppo lungo. Quando si perde un segmento, si genera un lungo periodo di timeout che costringe il mittente a ritardare il rinvio del segmento perso, aumentando di conseguenza il ritardo di trasmissione.





- Spesso però, il mittente TCP può rilevare la perdita dei segmenti **prima che si verifichi l'evento di timeout** grazie agli **ACK duplicati** che riscontrano un segmento il cui ACK è già stato ricevuto dal mittente.
- Quando il destinatario TCP riceve un segmento con numero di sequenza superiore al numero di sequenza atteso, rileva che uno o più segmenti non sono arrivati. Il destinatario allora riscontra di nuovo il segmento che ha ricevuto in ordine corretto, duplicando così un ACK.
- Poiché il mittente spesso invia più segmenti, uno dopo l'altro, se un segmento si perde ci saranno probabilmente vari ACK duplicati.
- Il mittente TCP considera la ricezione di tre ACK duplicati per lo stesso segmento, come prova che il segmento che ha inviato dopo il segmento riscontrato tre volte, è stato perso.


Host A
Host B
Seq=100, #dati=10
Seq=110, #dati=20
Seq=130, #dati=40
Seq=170, #dati=30
Seq=200, #dati=20
Ris=110
Ris=110
Ris=110
Ris=110
ACK
duplicati
timeout
Seq=110, #dati=20
tempo

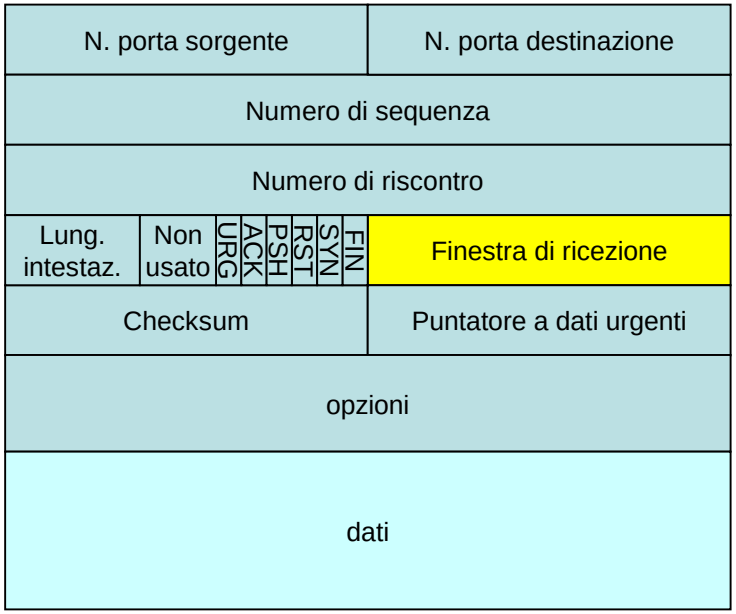
- Nel caso in cui siano stati ricevuti tre ACK duplicati, il mittente TCP effettua una **ritrasmissione rapida**, rinviando il segmento perso prima che scada il timeout.
- Le procedure precedentemente descritte relative ai timer TCP sono complesse e hanno subito molti cambiamenti e sono il risultato di circa cinquanta anni di esperienza.

```
evento3: /* ACK ricevuto, con valore del campo numero di riscontro = numRiscontro */
if (numRiscontro > numSequenza_min) {
   numSequenza_min = numRiscontro
   if ( <ci sono segmenti non ancora riscontrati>)
       <avvia timer>
   }
else { /* un ACK duplicato per un segmento già riscontrato */
   <incrementa il numero di ACK duplicati ricevuti per numRiscontro>
   if ( <numero di ACK duplicati per numRiscontro == 3>) {
      /* ritramissione rapida */
      <rinvia il segmento con numero di sequenza numRiscontro>
    }
 }
```

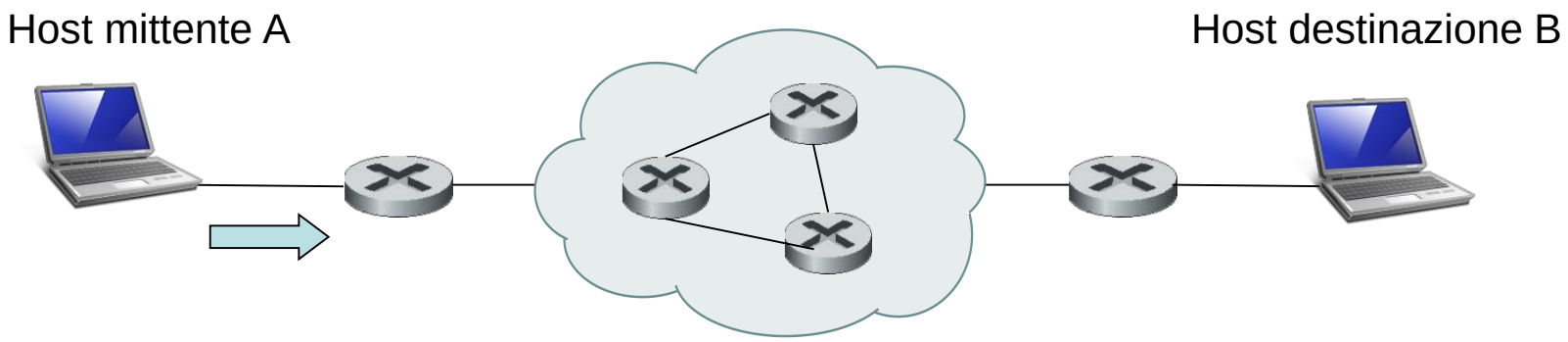
Controllo del flusso

- Il **controllo del flusso** è un servizio che il TCP fornisce per evitare che il mittente possa riempire il buffer di ricezione del destinatario.
- Ricordiamo che il TCP in ciascuna estremità della connessione crea un buffer di trasmissione e uno di ricezione.
- Quando il TCP riceve dati, ne verifica la correttezza e li pone nel buffer di ricezione.
- Il processo applicativo leggerà i dati da questo buffer, ma **non obbligatoriamente nell'istante del loro arrivo**. Infatti, il processo ricevente potrebbe trovarsi ad eseguire altre operazioni e non essere in grado di leggere i dati immediatamente.
- Se il processo è lento a leggere i dati, il mittente può saturare il buffer di ricezione inviando i dati troppo velocemente.

- **Il controllo del flusso è quindi un sistema di regolazione delle velocità di trasmissione dei dati.**
- Il TCP fornisce il controllo del flusso usando il campo di intestazione "***finestra di ricezione"*** *(**receive window**).*
- La "finestra di ricezione" è impostata da un'estremità della connessione per avvisare l'altra di quanto spazio è disponibile nel suo buffer di ricezione.
- La **finestra di ricezione** varia durante il tempo di connessione.



- Come abbiamo già detto, la velocità di trasmissione di un mittente TCP può anche essere ridotta a causa della congestione della rete; questo tipo di controllo del mittente è chiamato **controllo della congestione.**
- Vediamo l'uso della finestra di ricezione in un **esempio di un trasferimento di file**. Supponiamo che un host mittente **A** stia inviando un file all'host **B**.



- Il TCP in B crea un buffer di ricezione la cui dimensione è memorizzata nella variabile **RcvBuffer**. Definisce inoltre le seguenti variabili:

LastByteRead = numero dell'ultimo byte nel flusso di dati letto dal buffer dall'applicazione in B.

LastByteRcvd = numero dell'ultimo byte nel flusso di dati che è stato memorizzato nel buffer di ricezione in B.

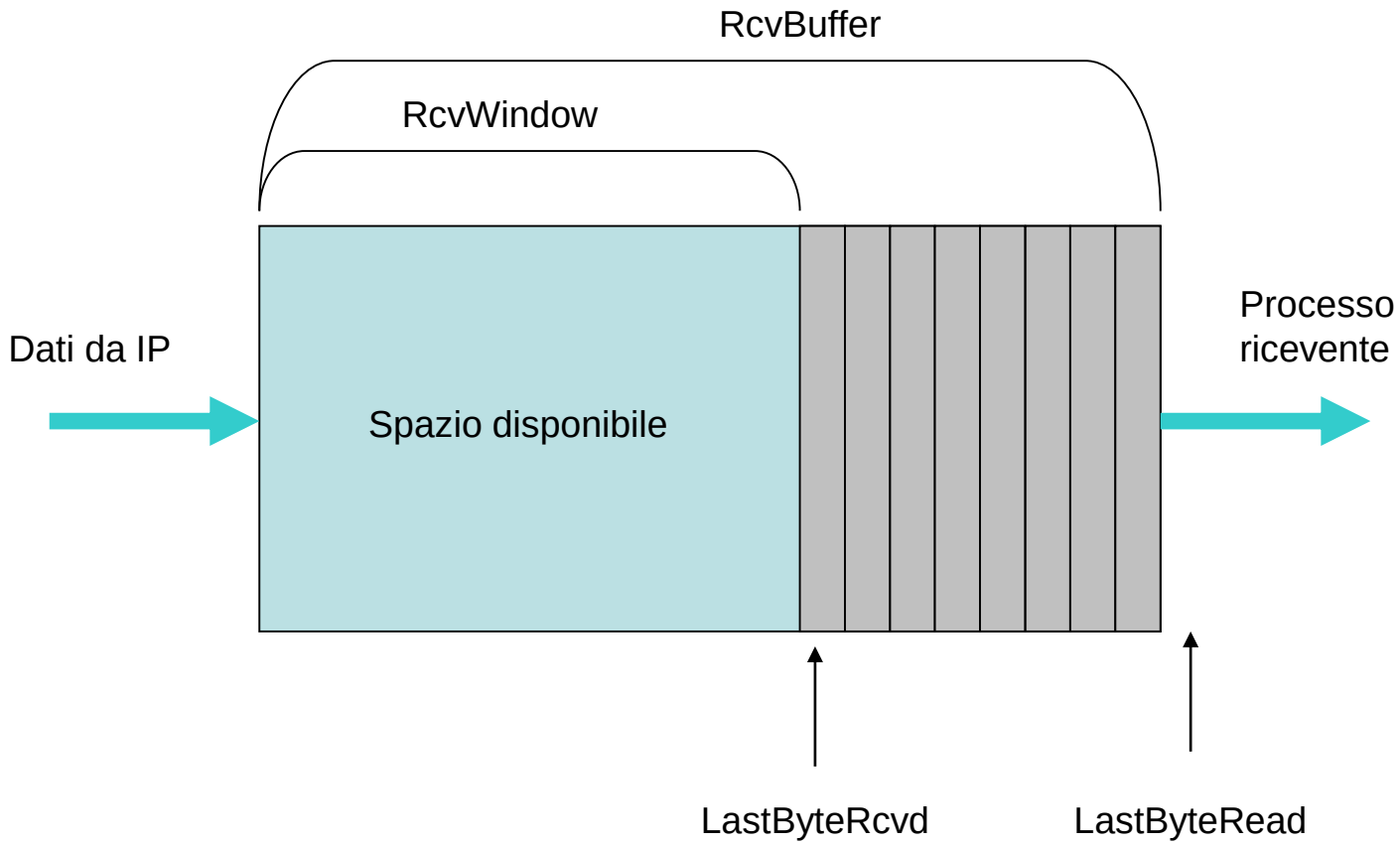
- Poiché il TCP non deve riempire il buffer assegnato, è necessario che sia:

  LastByteRcvd - LastByteRead <= RcvBuffer

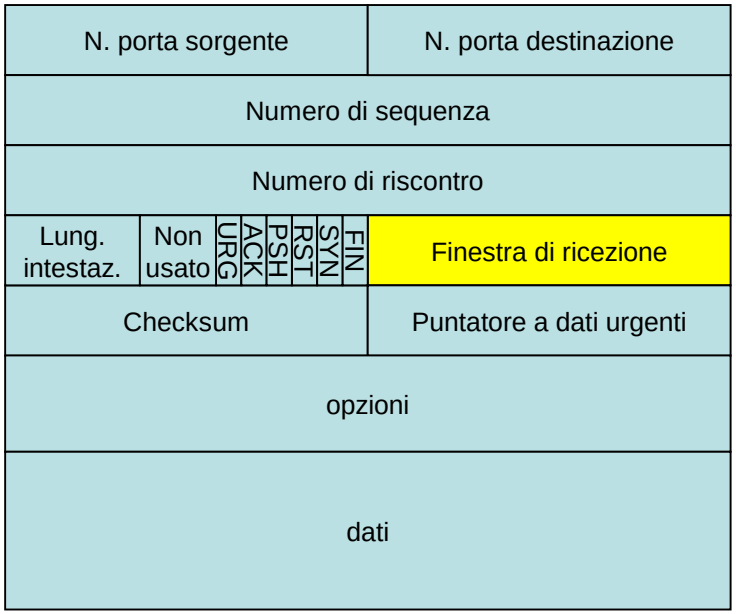
- La finestra di ricezione, indicata da **RcvWindow**, è posta uguale alla quantità di spazio disponibile nel buffer:

  RcvWindow=RcvBuffer–(LastByteRcvd–LastByteRead)

- Poiché lo spazio disponibile cambia con il tempo, **RcvWindow è dinamica**.


RcvBuffer
RcvWindow
Dati da IP
Spazio disponibile
Processo
ricevente
LastByteRcvd
LastByteRead

- Vediamo ora come è usata la variabile **RcvWindow** per fornire il servizio di controllo del flusso.
  L'host **B** comunica all'host **A** quanto spazio ha a disposizione nel buffer di ricezione inserendo il valore corrente di **RcvWindow** nel campo **finestra di ricezione** di ogni segmento che invia ad **A**. Inizialmente, l'host **B** pone **RcvWindow = RcvBuffer**.

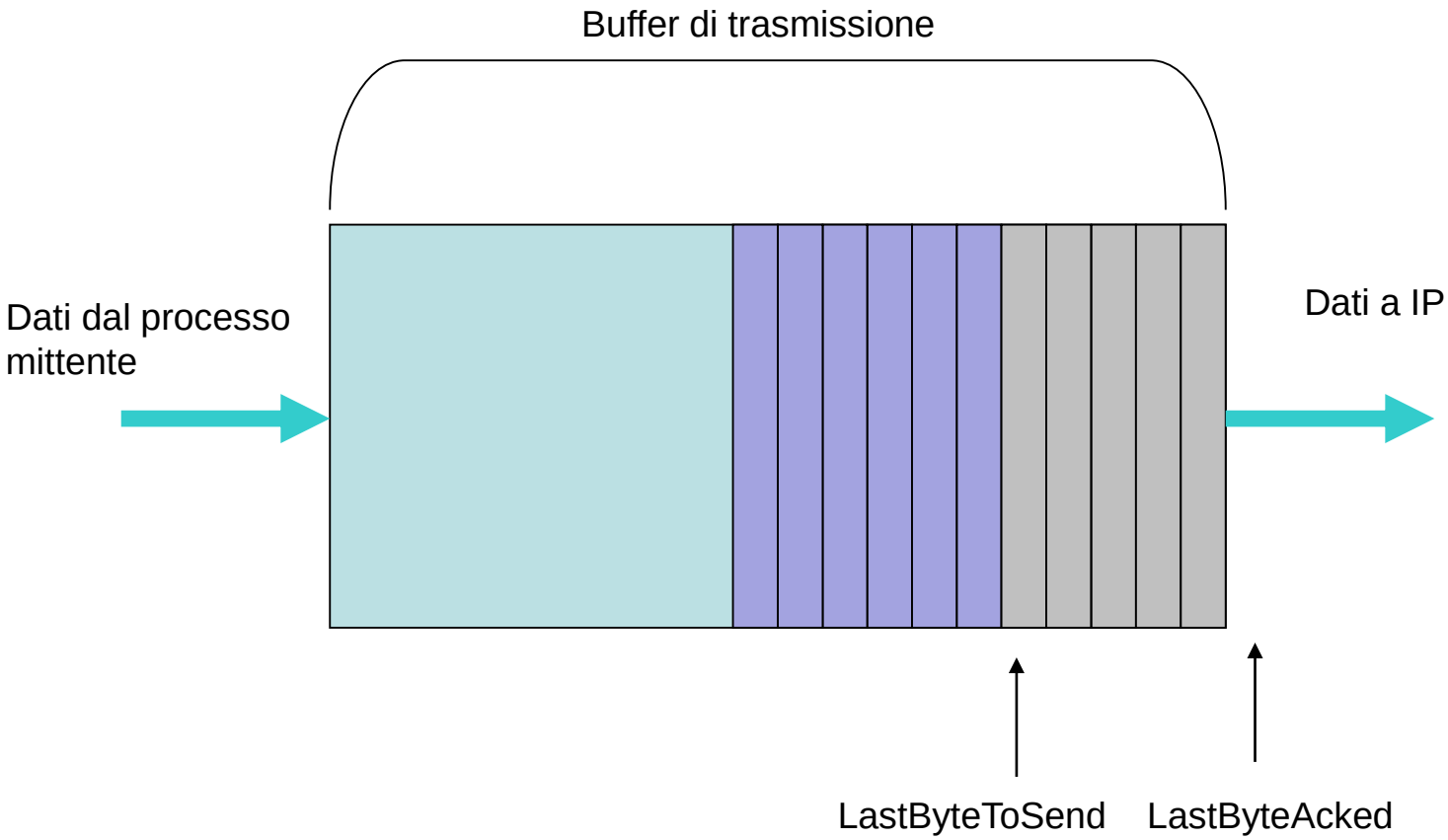


- L’host mittente **A** a sua volta utilizza due variabili:
  - **LastByteToSend** (ultimo byte da inviare) e
  - **LastByteAcked** (ultimo byte riscontrato).
- La differenza tra queste due variabili,

  LastByteToSend – LastByteAcked

  è pari al **numero di byte che A invierà** a **B nel prossimo segmento**.
- Mantenendo tale quantità di byte inferiore al valore di **RcvWindow**, l’host **A** non potrà saturare il buffer di ricezione dell’host **B**. Quindi, l’host **A** invia per la durata della connessione un numero di byte pari a:

  LastByteToSend - LastByteAcked <= RcvWindow


Buffer di trasmissione
Dati dal processo mittente
Dati a IP
LastByteToSend
LastByteAcked

- in questo schema ora descritto potrebbe verificarsi un problema tecnico. Per vederlo, supponiamo che il buffer di ricezione dell'host **B** si saturi, così che **RcvWindow = 0**. Oltre all'avviso all'host **A** che **RcvWindow = 0**, supponiamo anche che **B non abbia dati da inviare ad A**. Vediamo cosa accade. Quando l'applicazione di **B** svuota il buffer, il TCP non invia nuovi segmenti all'host **A** con il nuovo valore RcvWindow: in effetti, il TCP manda un segmento all'host A solo se ha dei dati o un riscontro da mandare. Quindi, l'host A non viene mai informato che si è liberato dello spazio nel buffer di ricezione del host B. Per risolvere questo problema, il TCP dell'host A **continua a inviare segmenti con un byte di dati** quando la finestra di ricezione di B è a zero. Questi segmenti saranno riscontrati dal ricevitore. A un certo punto il buffer comincerà a svuotarsi e i riscontri conterranno un valore di RcvWindow diverso da zero.

Controllo della congestione del TCP

- Un altro servizio molto importante del TCP è il **controllo della congestione (Rfc 5681).**
- Il TCP regola la velocità di trasmissione del mittente in funzione del livello di congestione presente nel percorso relativo alla sua connessione.
- Se un mittente TCP rileva che c'è poco traffico, allora aumenta la sua velocità di trasmissione, se invece percepisce che c'è congestione lungo il percorso, allora riduce la sua velocità di trasmissione.
- Ci sono quindi due problemi principali da risolvere:
  - **rilevazione della quantità di congestione;**
  - **regolazione della velocità di trasmissione del mittente;**

  sono stati proposti e realizzati vari algoritmi per la soluzione dei problemi di cui sopra.

- Esamineremo l'algoritmo di controllo della congestione di **TCP Reno**, che viene implementato in molti sistemi operativi. Per descrivere l'algoritmo, supporremo che il mittente TCP stia inviando un file di grande dimensione.
- Esaminiamo prima in che modo un mittente limita la velocità di trasmissione.
- Per il controllo della congestione il TCP, in entrambi i lati della connessione, utilizza una variabile detta **finestra di congestione** *(**congestion window**).*
- La finestra di congestione, che indichiamo con **CongWindow**, stabilisce la quantità di byte che un host può inviare all'altra estremità della connessione.
- Poiché la velocità di trasmissione è anche regolata dalla finestra di ricezione (RcvWindow), la quantità dei byte non riscontrati che un host può inviare non deve superare il minimo tra CongWindow e RcvWindow:

LastByteToSend-LastByteAcked<= min(CongWindow, RcvWindow)

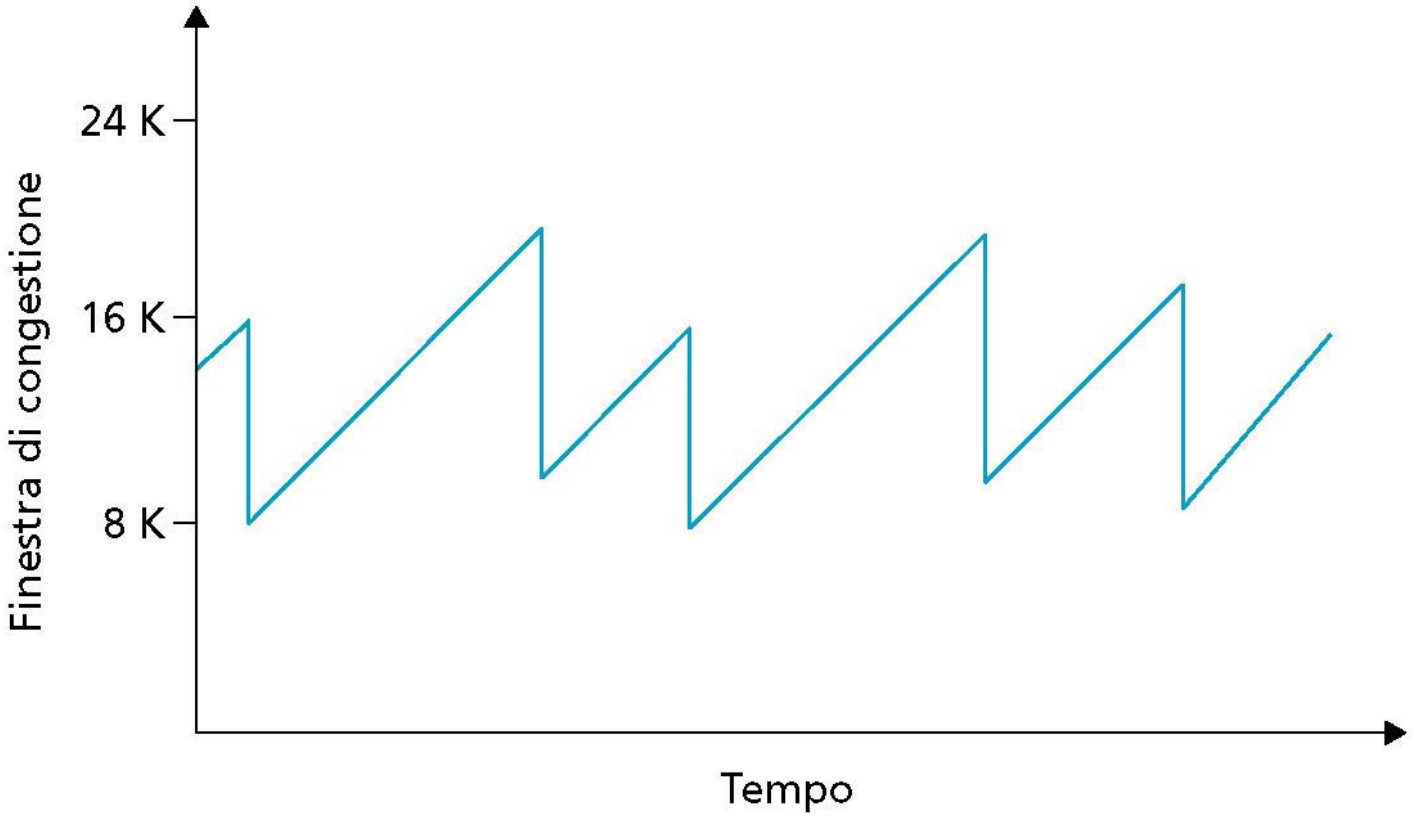
- Per analizzare il controllo della congestione, assumiamo che il buffer di ricezione del TCP sia abbastanza grande in modo da poter trascurare il limite imposto dalla finestra di ricezione **RcvWindow**. In questo caso, la quantità di dati non riscontrati che un host può inviare è limitata unicamente dalla **CongWindow**.
- La relazione di vincolo di cui sopra asserisce che variando il valore di CongWindow, il mittente TCP può variare la velocità con cui invia i dati nella sua connessione**.**
- Quando c'è elevata congestione nella rete, i buffer dei router lungo il percorso si riempiono, causando la perdita di pacchetti. Un pacchetto perso, a sua volta, genera nel TCP mittente un **evento di perdita**, cioè, o un timeout o la ricezione di tre ACK duplicati, che è considerato dal mittente TCP come **una rilevazione di congestione nel percorso dal mittente al ricevente**.

- L’algoritmo è basato su tre componenti principali:
  1. **incremento additivo, decremento moltiplicativo**
  2. **partenza lenta *(slow start)***
  3. **reazione a eventi di timeout.**

Incremento additivo, decremento moltiplicativo

- L’idea su cui si basa il controllo di congestione del TCP è di **diminuire la dimensione della finestra di congestione CongWindow** del mittente, quando si verifica un evento di perdita, in modo da ridurre la sua velocità di trasmissione.
- Nel caso di congestione, tutte le connessioni TCP che passano attraverso gli stessi router congestionati probabilmente subiranno eventi di perdita e quindi tutti i mittenti ridurranno le loro velocità di trasmissione diminuendo i valori delle loro **CongWindow.**

- L’effetto totale, quindi, è che **si avrà una riduzione della congestione nei router congestionati**.
- Il TCP usa un criterio detto a **"decremento moltiplicativo"**, che dimezza il valore corrente di **CongWindow** dopo un evento di perdita. Tuttavia, il valore minimo di CongWindow non scende sotto il valore di un ***MSS***.
- Il motivo per aumentare la velocità è che se non si rileva congestione, allora è probabile che ci sia della banda disponibile che potrebbe essere utilizzata dalla connessione TCP.
- Il mittente TCP interpreta l’esistenza di disponibilità di banda ogni volta che riceve un **ACK** e di conseguenza aumenta **CongWindow**.
- Il valore di CongWindow segue continuamente dei cicli durante i quali esso cresce e poi improvvisamente diminuisce il suo valore corrente quando si verifica un evento di perdita.

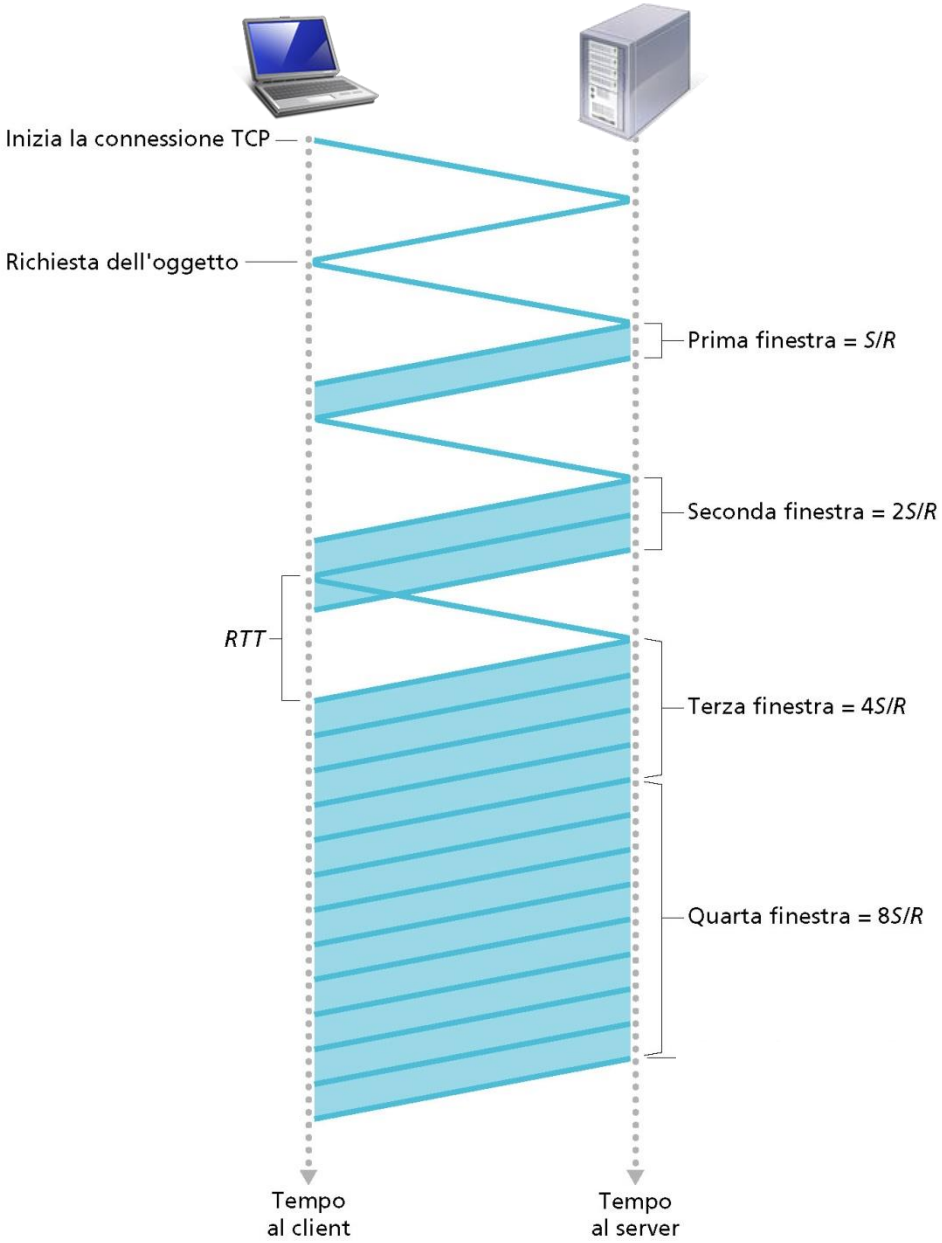


Controllo della congestione a incremento additivo-decremento moltiplicativo

- In prima approssimazione, possiamo dire che l'andamento della finestra di congestione sia a ***"dente di sega"***, supponendo che la crescita sia lineare e la riduzione istantanea, a "decremento moltiplicativo". Questo andamento è detto "**incremento additivo e decremento moltiplicativo**" (**AIMD**, *additive-increase moltiplicative-decrise*).

Partenza lenta (slow start)

- Al momento dell'instaurazione di una connessione TCP, **CongWindow** è inizializzata al valore di un ***MSS,*** fornendo una velocità di trasmissione di circa ***MSS/RTT byte/sec.***
- Per esempio se MSS = 500 byte e RTT = 250 ms, allora la velocità di trasmissione iniziale è solo di circa 16 kbit/sec (500*8/0,25).
- Dato che la banda disponibile per la connessione potrebbe essere molto maggiore di ***MSS/RTT,*** il mittente TCP aumenta la sua velocità in **modo esponenziale**,


Inizia la connessione TCP
Richiesta dell'oggetto
Prima finestra = S/R
Seconda finestra = 2S/R
RTT
Terza finestra = 4S/R
Quarta finestra = 8S/R
Tempo al client
Tempo al server

raddoppiando il proprio valore di **CongWindow ogni *RTT*** fino a raggiungere il valore della variabile **Threshold (soglia)**.

- La variabile **Threshold** è inizializzata a un valore grande, in genere **64 kbyte,** in modo che non abbia alcun effetto iniziale.
- Quando si verifica un evento di perdita, il valore di **Threshold** è posto pari alla metà del valore corrente di **CongWindow.** Questa prima fase, in cui CongWindow cresce in modo esponenziale, è detta "**partenza lenta"**.
- Il TCP nel mittente aumenta la velocità di trasmissione in modo esponenziale aumentando il valore di CongWindow di un *MSS* ogni volta che viene riscontrato un segmento trasmesso. In particolare, il TCP invia il primo segmento e aspetta il riscontro.
- Se questo segmento viene riscontrato prima di un evento di perdita, il mittente aumenta la finestra di congestione di un *MSS* e invia due segmenti della massima dimensione.

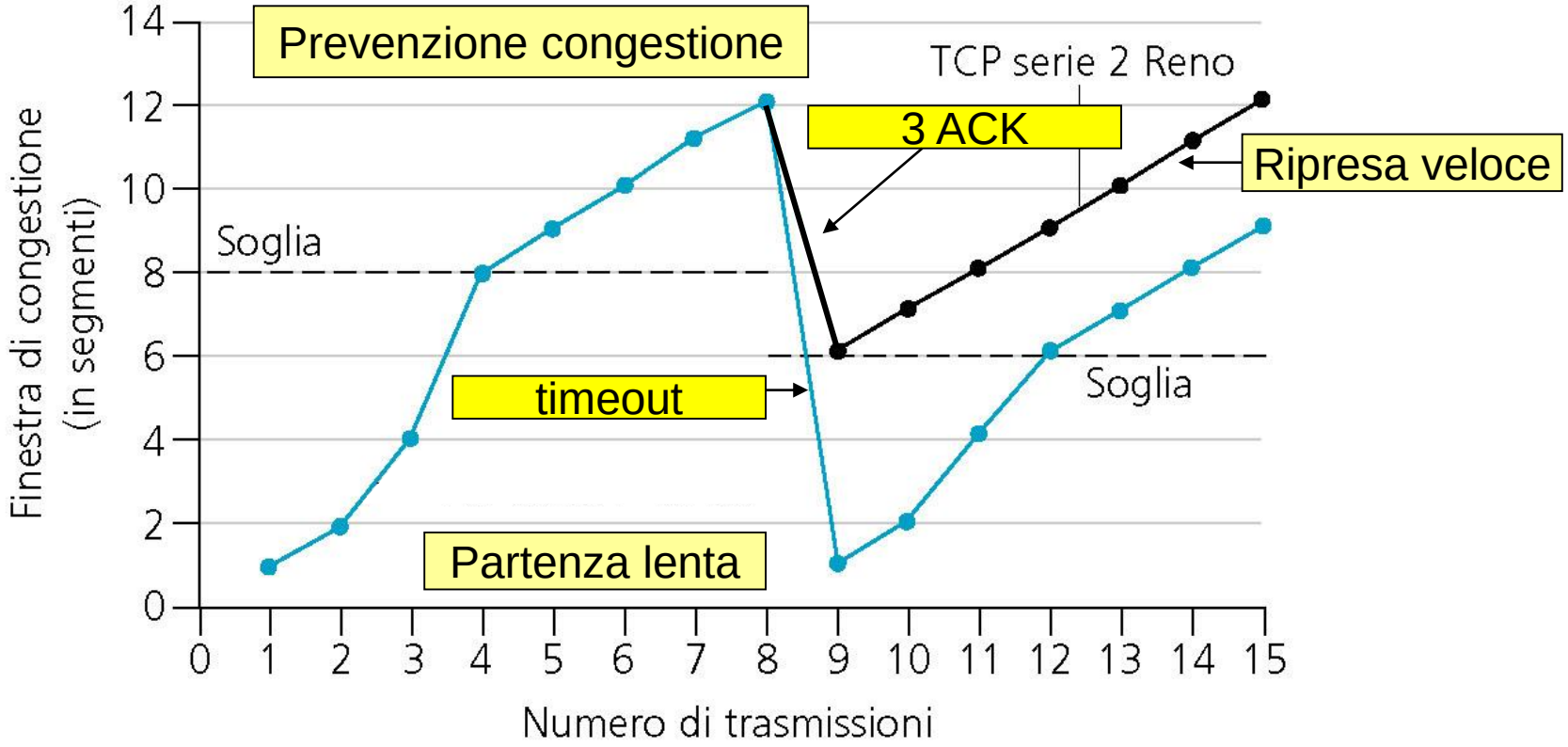
- Se questi segmenti vengono riscontrati prima di un evento di perdita, il mittente TCP aumenta la finestra di congestione di un *MSS* per ognuno dei segmenti riscontrati, portando la dimensione della finestra di congestione a quattro *MSS,* e invia quattro segmenti della massima dimensione. Questa procedura continua fino a che i riscontri arrivano prima di eventi di perdita. **Quindi, durante la fase di partenza lenta il valore di CongWindow raddoppia effettivamente a ogni *RTT*.**

Prevenzione della congestione

- Raggiunto il valore di **soglia (Threshold)**, il mittente continua a aumentare la sua velocità, ma in **modo lineare**, fino a quando si verifica un evento di perdita. A questo punto, il valore di **CongWindow** viene dimezzato e il valore di **soglia** è posto alla metà del valore che aveva **CongWindow** prima dell'evento di perdita. Questa fase di crescita lineare di CongWindow è detta "**prevenzione della congestione" *(congestion avoidance)***.

Reazione agli eventi di timeout

- Il controllo della congestione del TCP Reno si comporta in modo diverso a seconda che l'evento di perdita sia dovuto a un **evento di timeout** o che sia dovuto a un **ACK ripetuto tre volte**.
- Dopo un ACK replicato tre volte, la finestra di congestione è dimezzata e poi aumenta linearmente.
- Invece, dopo un evento di timeout, il mittente TCP entra in una **fase di partenza lenta**, cioè, la finestra di congestione è ripristinata al valore di un *MSS* e quindi è incrementa esponenzialmente. La finestra continua a crescere esponenzialmente finché **CongWindow raggiunge la metà del valore che aveva prima dell'evento di timeout**.
- A quel punto, CongWindow cresce linearmente, come avrebbe fatto dopo un ACK duplicato tre volte.
- Come detto Il TCP gestisce questi andamenti utilizzando una variabile chiamata **threshold** (**soglia**), che determina la dimensione della finestra alla quale deve terminare la fase



Andamento della finestra di congestione TCP

Di partenza lenta, e deve cominciare la fase di prevenzione della congestione.

- Il controllo di congestione del TCP si comporta diversamente dopo un evento di timeout e dopo la ricezione di un ACK duplicato tre volte.
- In particolare, il mittente TCP, dopo un evento di timeout, riduce la sua finestra di congestione a 1 MSS, mentre dopo aver ricevuto un ACK duplicato tre volte dimezza soltanto la sua finestra di congestione
- Il motivo che ha portato a gestire in modo diverso i due eventi di perdita è che, sebbene sia stato perso un pacchetto, l'arrivo di tre ACK duplicati indica che alcuni segmenti sono stati ricevuti dal mittente.
- Quindi, a differenza del caso di timeout, la rete mostra di essere in grado di consegnare almeno alcuni segmenti, anche se altri si sono persi a causa della congestione.

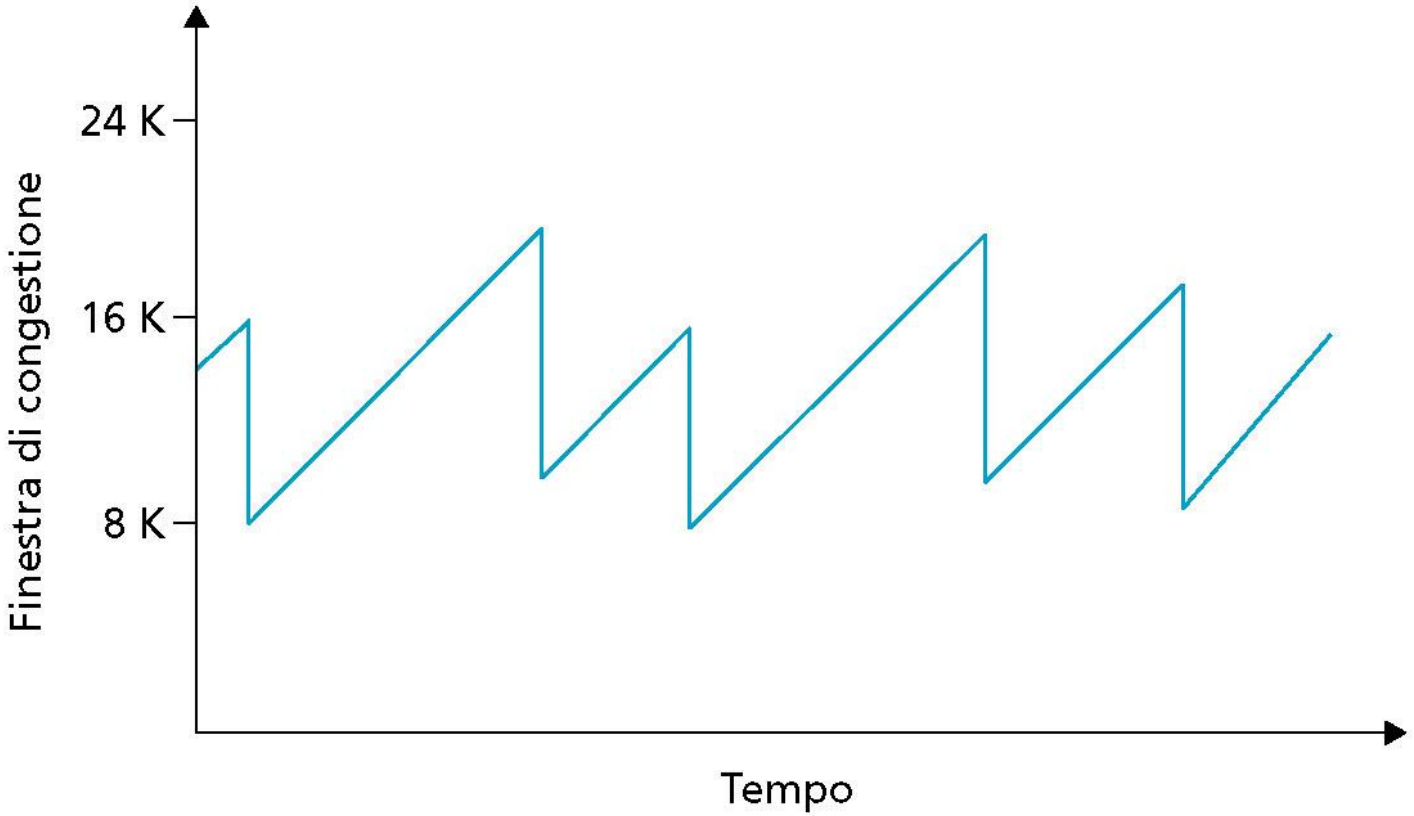
- Questa eliminazione della fase di partenza lenta dopo un ACK duplicato tre volte è chiamata ***ripresa veloce (fast recovery).***
- Sono state proposte molte varianti all'algoritmo Reno.
- L'algoritmo TCP Vegas, sviluppato nell'Università di Arizona, tenta di evitare la congestione mantenendo comunque un buon throughput. L'idea alla base di Vegas è di
  - (1) rilevare la congestione nei router tra la sorgente e la destinazione *prima* che si verifichi la perdita di pacchetti osservando i tempi di RTT. Maggiori sono i tempi di RTT dei pacchetti, maggiore è la congestione nei router.
  - (2) abbassare linearmente la velocità quando viene rilevata questa imminente perdita di pacchetti.

  Il TCP Vegas è stato implementato nel kernel Linux.

  Il controllo della congestione di TCP si è evoluto nel corso degli anni e continua ancora ad evolvere.

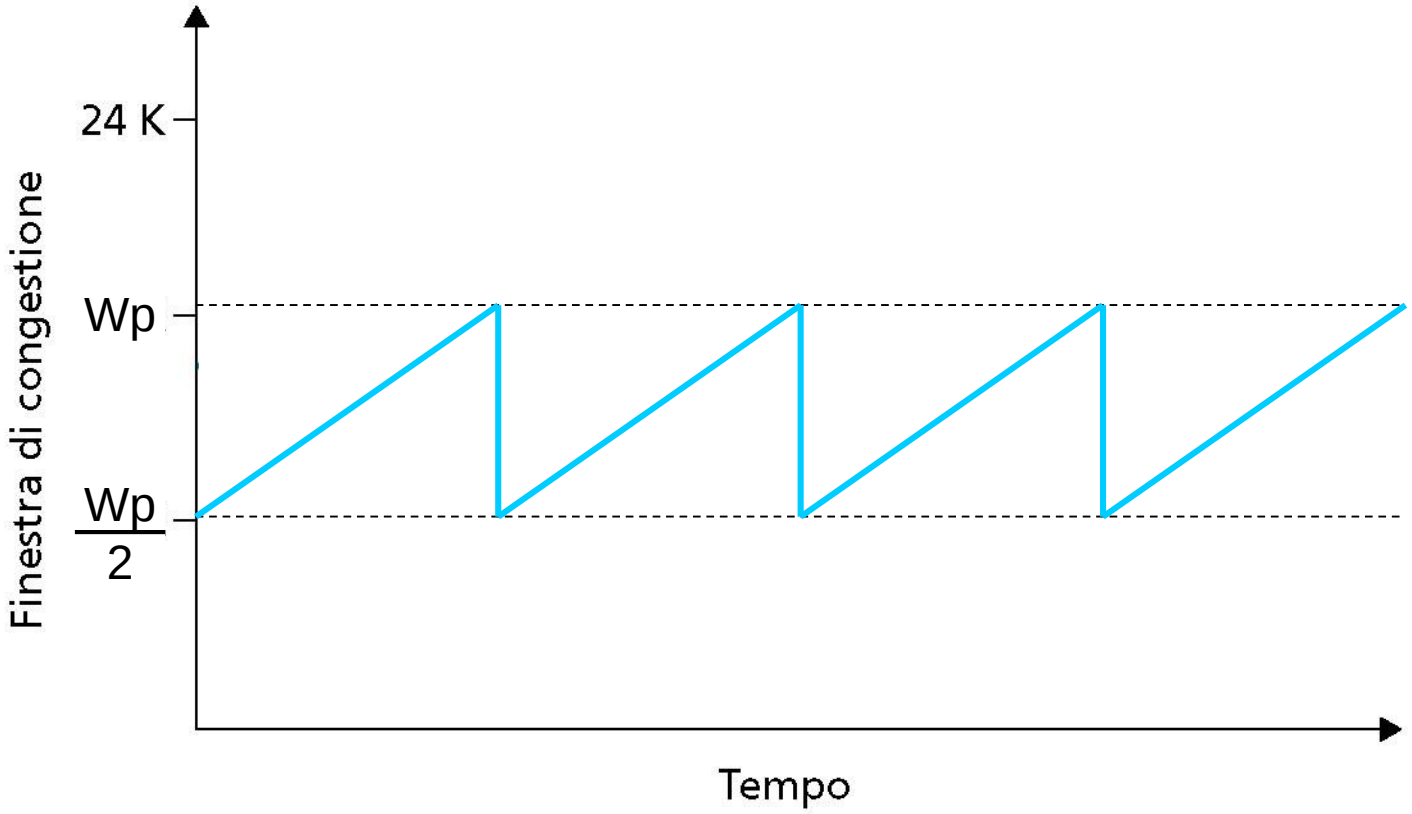
Calcolo semplificato del throughput di TCP

- Dato l'andamento a dente di sega di TCP, calcoliamo **approssimativamente** il valore del **throughtput medio** (velocità media) di una connessione di lunga durata.
- In questa analisi trascureremo le fasi di partenza lenta che si verificano dopo gli eventi di timeout in quanto sono generalmente molto brevi, dato che la velocità del mittente cresce esponenzialmente.
- Quando la dimensione della finestra è di ***W*** byte e il tempo di round-trip corrente è di ***RTT*** secondi, la velocità di trasmissione di TCP è circa ***W/RTT***.
- Il TCP aumenta la dimensione della finestra ***W*** di un *MSS* ogni *RTT* finché si verifica l'evento di perdita. Indichiamo con ***Wp*** il valore di ***W*** quando si verifica un evento di perdita.



Controllo della congestione a incremento additivo-decremento moltiplicativo

- Assumendo che ***Wp*** e ***RTT*** siano approssimativamente costanti per la durata della connessione, la velocità di trasmissione di TCP varia tra ***Wp/(2·RTT)*** e ***Wp/RTT***.
- Queste assunzioni portano a un modello molto semplificato del comportamento del TCP in regime stazionario.



- La comunicazione perde un pacchetto quando la velocità di trasmissione arriva a *Wp/RTT:* la velocità è allora dimezzata e poi aumentata di *MSS/RTT* a ogni *RTT* finché raggiunge nuovamente *Wp/RTT.* Questo processo si ripete continuamente. Poiché il throughput del TCP aumenta linearmente fra due valori estremi, abbiamo:

Throughput = (*Wp/(2·RTT)+ Wp/RTT)/2 =*
= 3·Wp/4·RTT = 0.75·Wp/RTT