Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2016-17

Pietro Frasca

Parte II: Reti di calcolatori

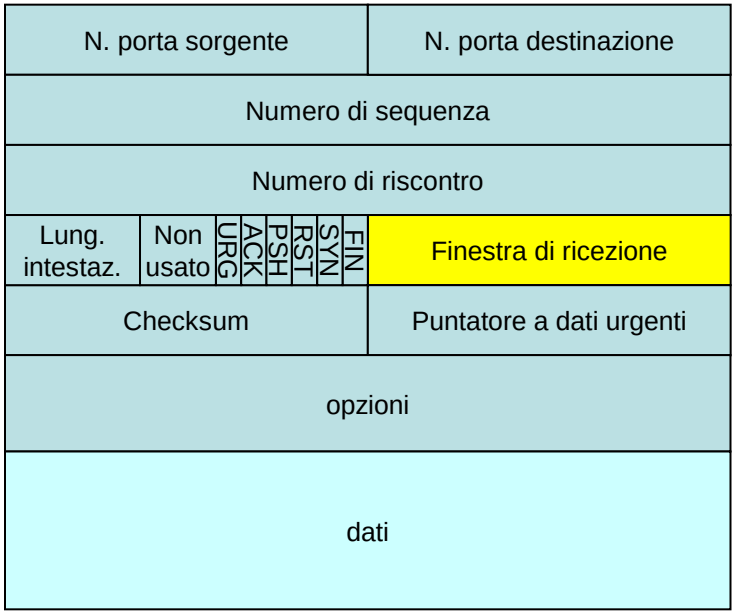
Lezione 14 (38)

Venerdì 28-04-2017

Controllo del flusso

- Il **controllo del flusso** è un servizio che il TCP fornisce per evitare che il mittente possa saturare il buffer di ricezione del destinatario.
- Ricordiamo che il TCP in ciascuna estremità della connessione crea un buffer di ricezione e uno di spedizione.
- Quando il TCP riceve dati, ne verifica la correttezza e li mette 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 riempire 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.

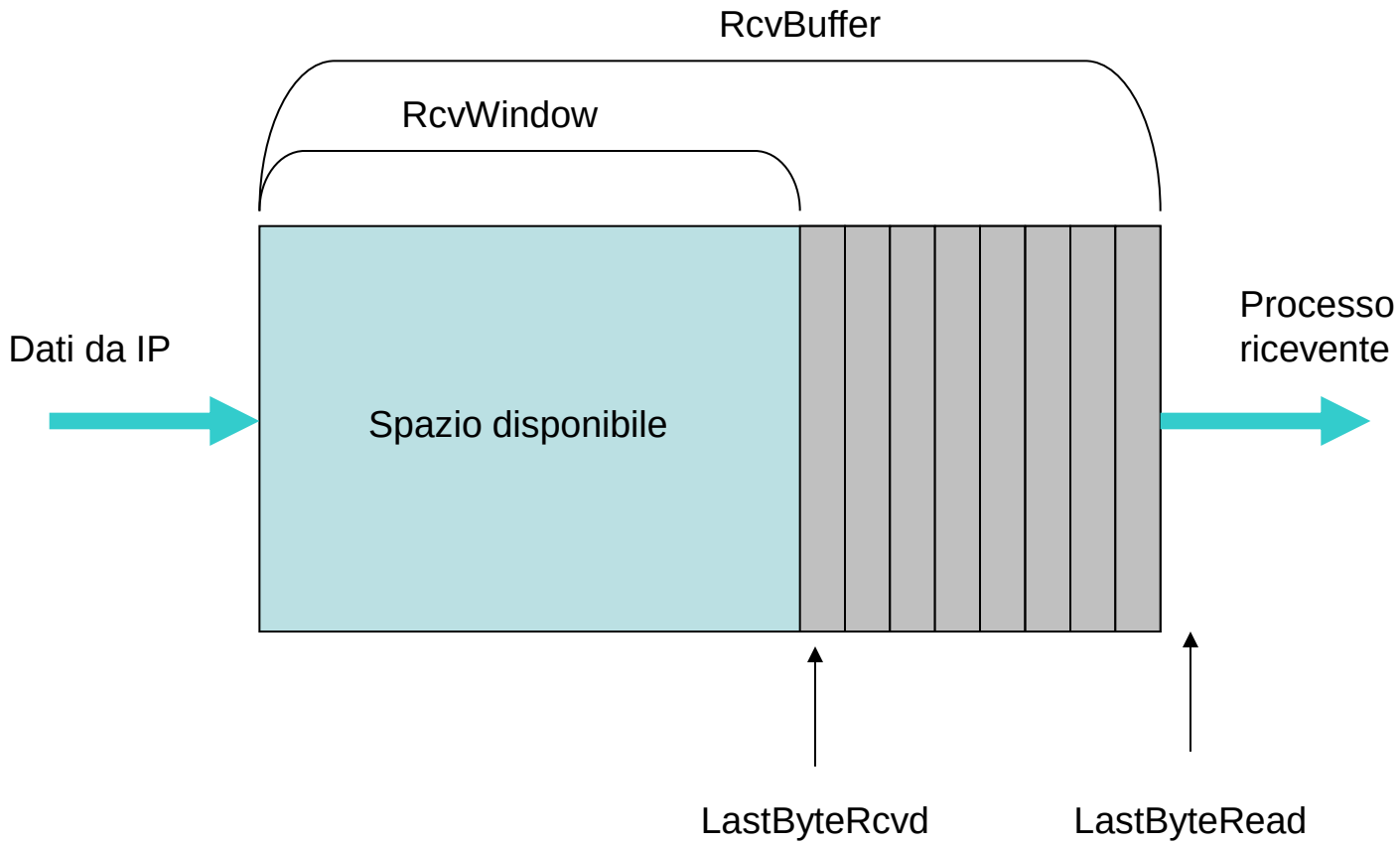
- Poiché il TCP non deve saturare il buffer assegnato, è necessario che sia:

  LastByteRcvd - LastByteRead <= RcvBuffer

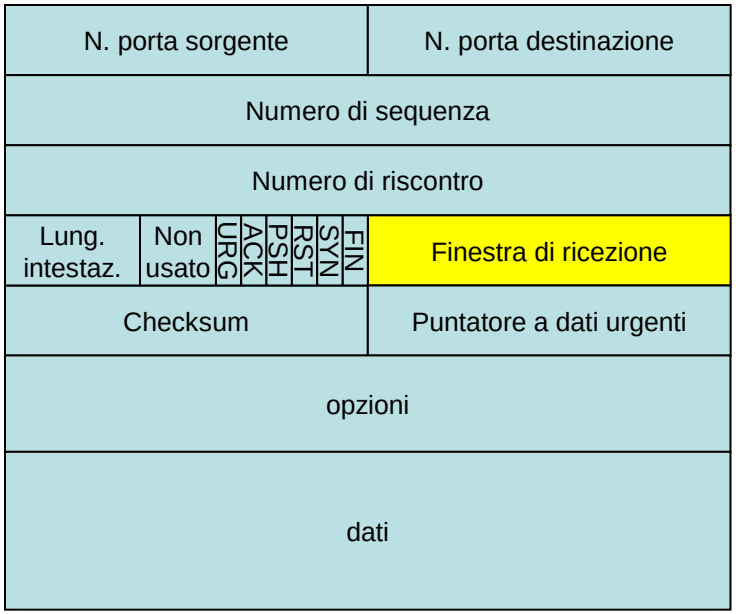
- La finestra di ricezione, indicata da **RcvWindow**, è posta uguale alla quantità di spazio disponibile nel buffer:

  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**.

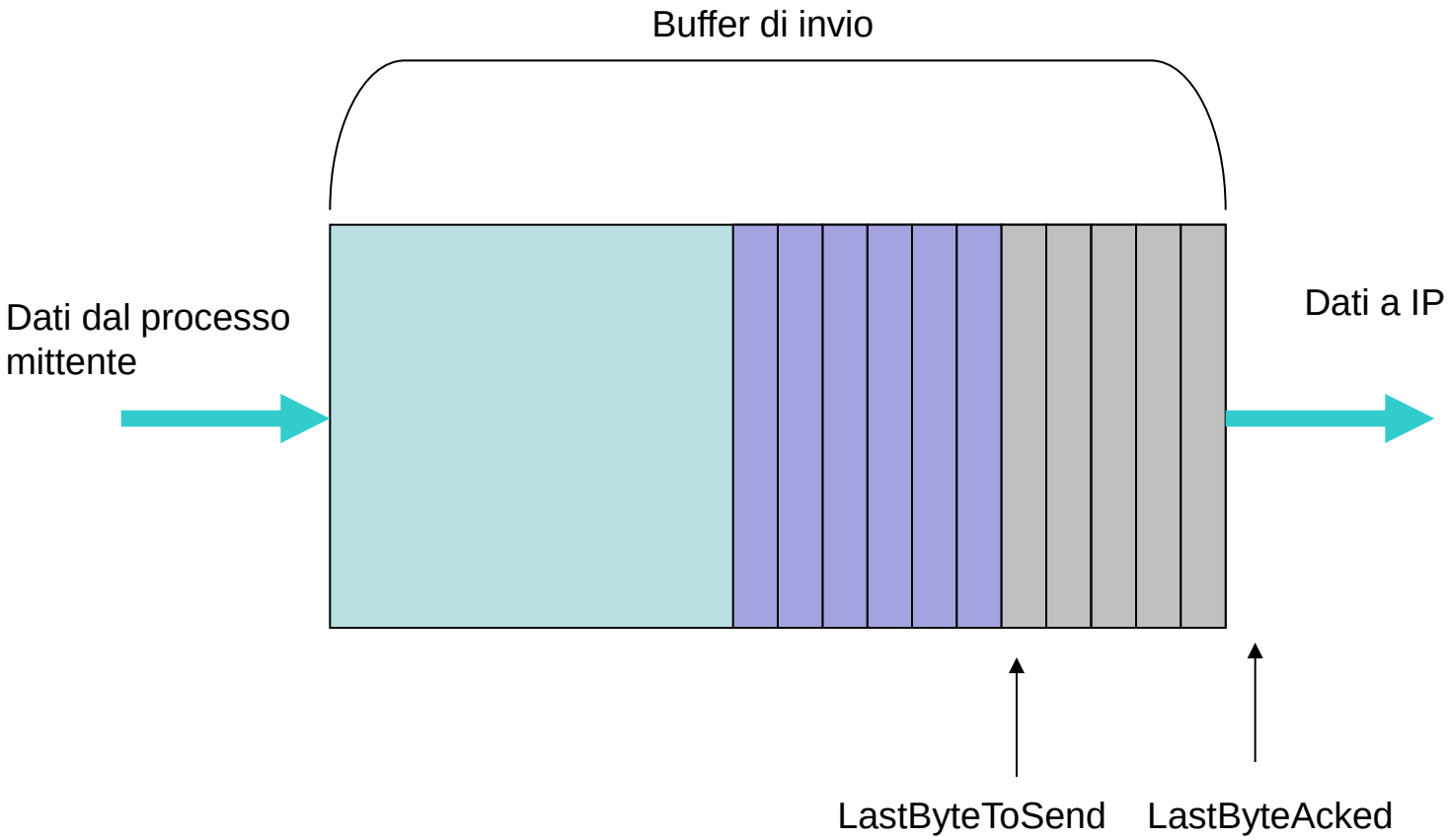


- L’host mittente **A** a sua volta utilizza due variabili:
  - **LastByteToSend** (ultimo byte da inviare) e
  - **LastByteAcked** (ultimo byte riscontrato).
- La differenza tra queste due variabili,

  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 invio
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.**
- 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 nella maggior parte dei 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**, limita la quantità di byte che un host può inviare in una connessione TCP.
- In particolare, la quantità dei dati non riscontrati che un host può avere all'interno di una connessione TCP **non deve superare il minimo tra CongWindow e RcvWindow**:

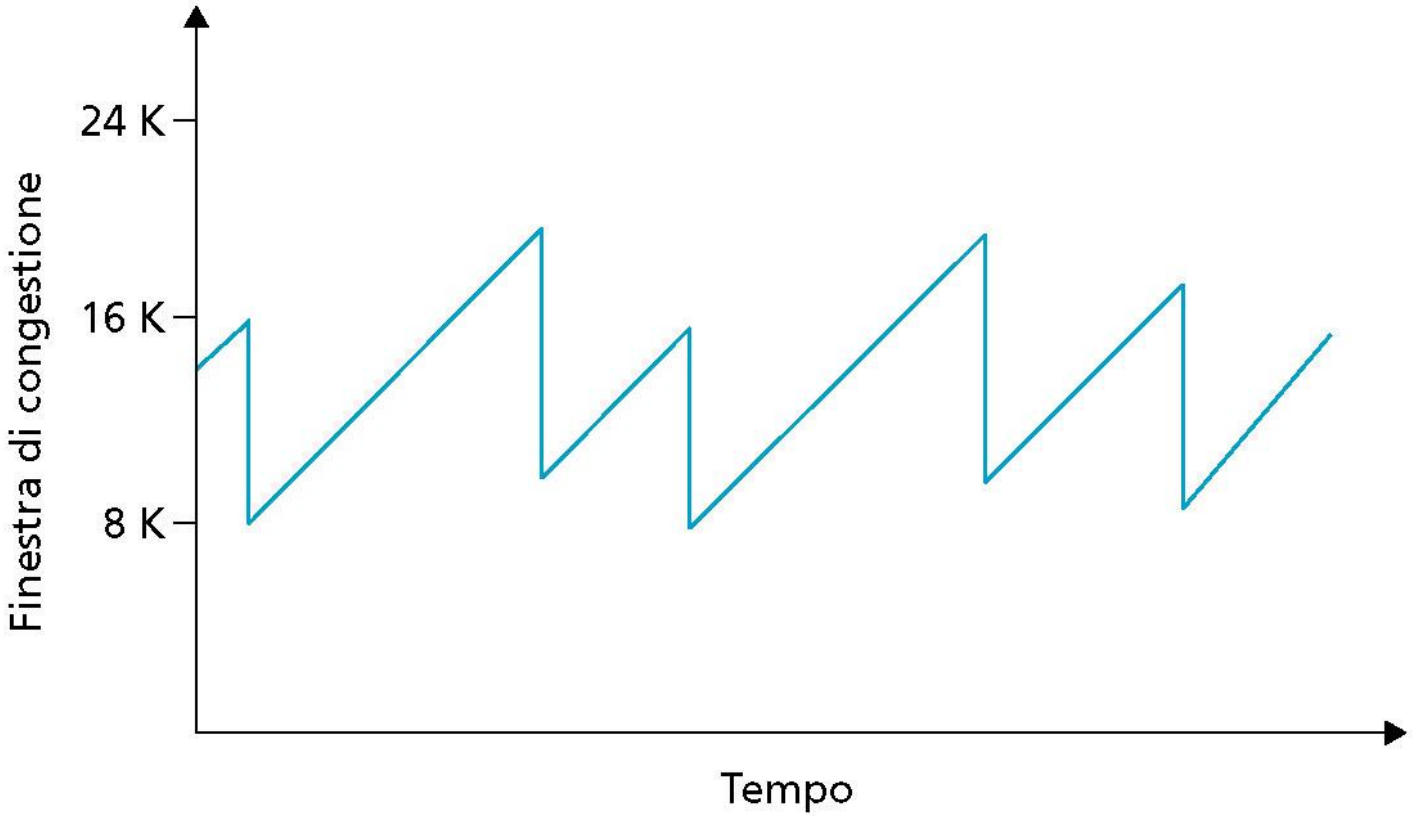
LastByteToSend-LastByteAcked<= min(CongWindow, RcvWindow)

- Per analizzare il controllo della congestione, assumiamo che il buffer di ricezione del TCP sia abbastanza grande in modo da poter trascurare il vincolo imposto dalla finestra di ricezione. In questo caso, la quantità di dati non riscontrati che un host può avere all'interno di una connessione TCP è limitata unicamente dalla **CongWindow**.
- La relazione di vincolo di cui sopra limita la quantità di dati non riscontrati dal mittente e quindi indirettamente limita la quantità di dati inviati dal mittente. **Variando il valore di CongWindow, il mittente può quindi variare la velocità con cui invia i dati nella sua connessione TCP.**
- 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 un **evento di perdita** al mittente: o un timeout o la ricezione di tre ACK duplicati, che è considerato dal mittente 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 globale, 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 è fissato al 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.

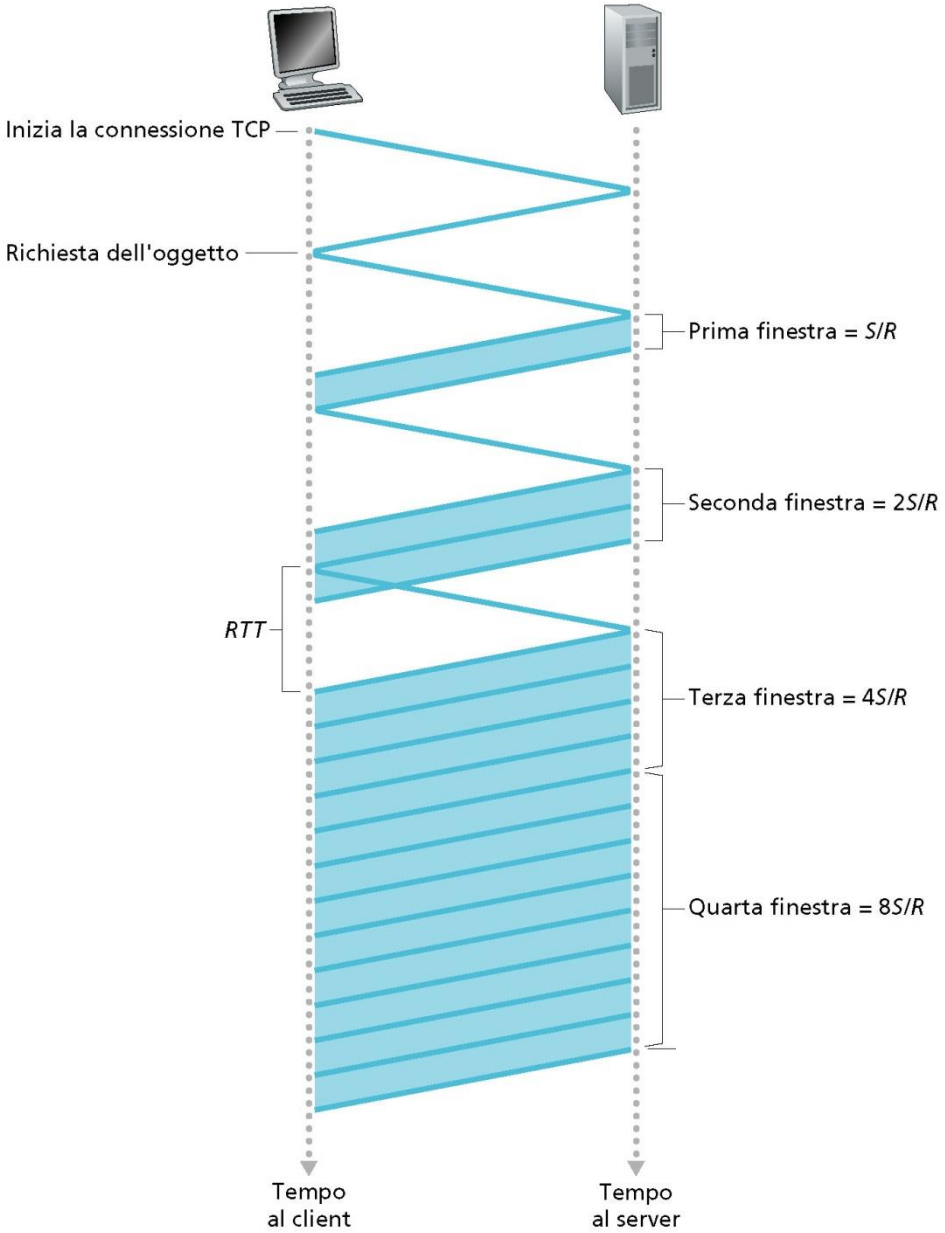
- Il mittente TCP interpreta l'esistenza di disponibilità di banda ogni volta che riceve un **ACK** e di conseguenza aumenta **CongWindow**.
- Il valore di CongWindow segue continuamente dei cicli durante i quali esso cresce e poi improvvisamente diminuisce il suo valore corrente quando si verifica un evento di perdita.
- 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 a "decremento moltiplicativo". Questo andamento è detto "**incremento additivo e decremento moltiplicativo**" (**AIMD**, *additive-increase moltiplicative-decrise*).

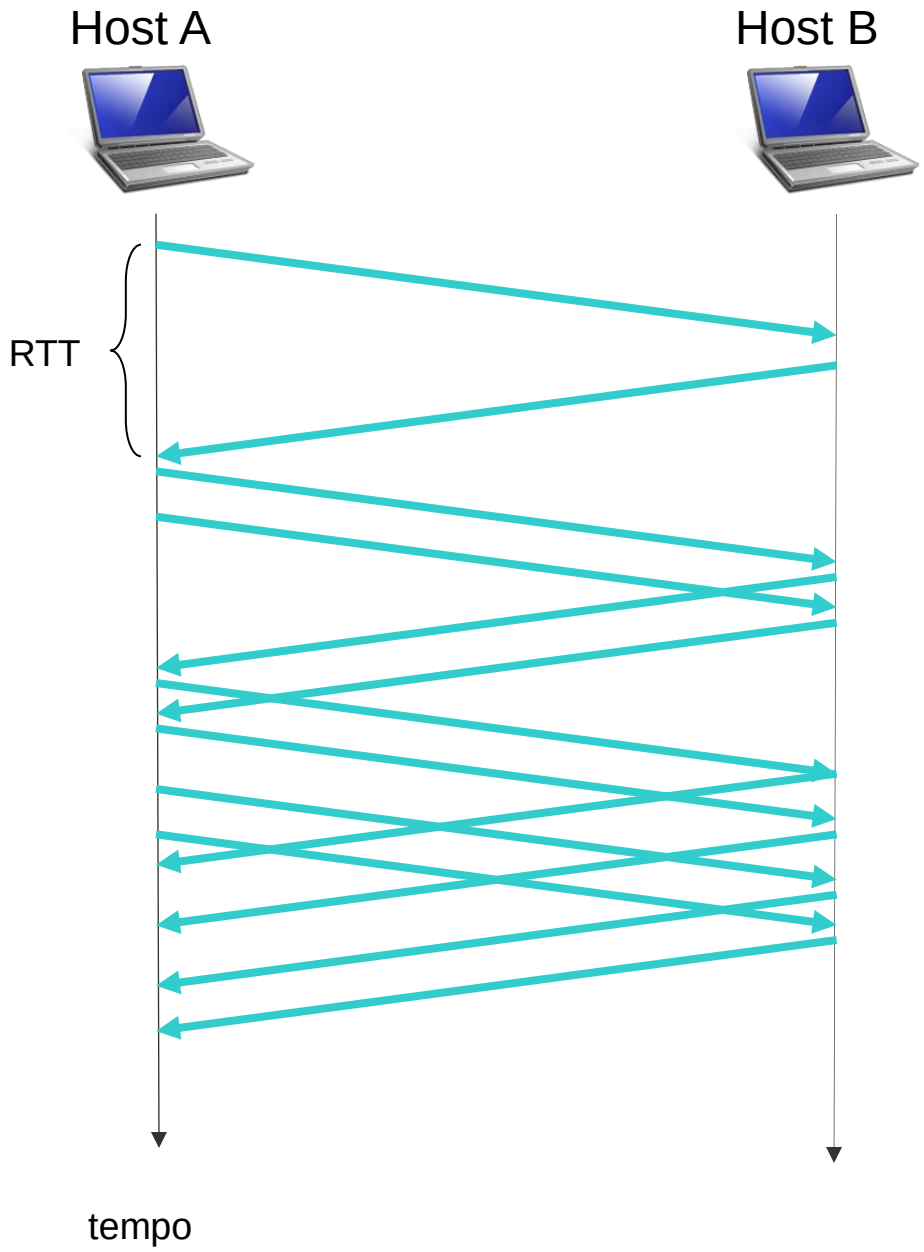


Controllo della congestione a incremento additivo-decremento moltiplicativo

Partenza lenta (slow start)

- Quando si instaura 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**, raddoppiando il proprio valore di **CongWindow ogni *RTT*** fino a raggiungere il valore di **Threshold (soglia)** una variabile TCP. La variabile **Threshold** è inizialmente posta a un valore grande, in genere **65 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 attuale di **CongWindow.** Questa prima fase, in cui CongWindow cresce in modo esponenziale, è chiamata "**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.


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


Host A
Host B
RTT
tempo

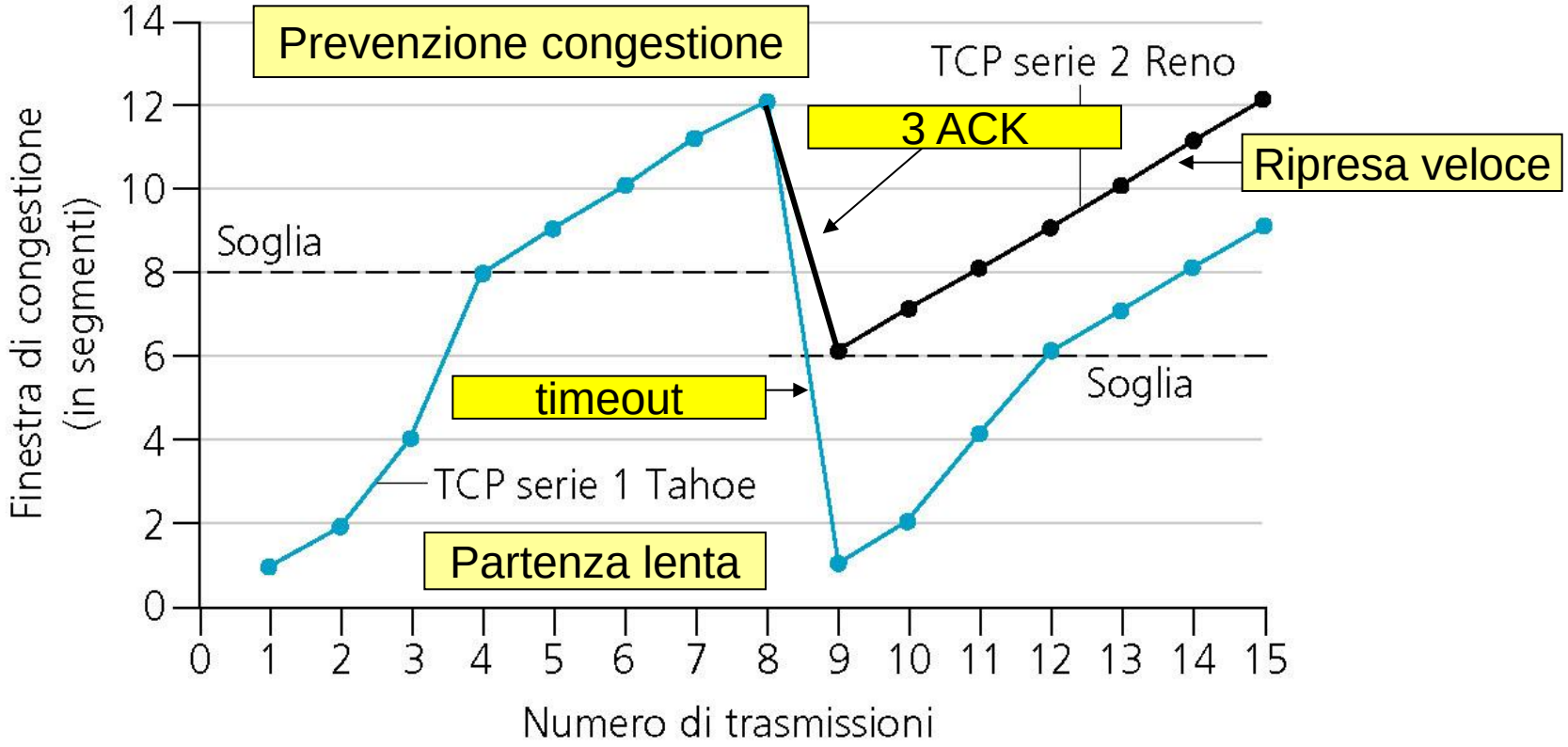
- Se questo segmento viene riscontrato prima di un evento di perdita, il mittente aumenta la finestra di congestione di un *MSS* e invia due segmenti della massima dimensione. Se questi segmenti vengono riscontrati prima di un evento di perdita, il mittente TCP aumenta la finestra di congestione di un *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, al che il valore di **CongWindow** viene dimezzato e il valore di **soglia** viene 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 reagisce in modo diverso a seconda che l'evento di perdita sia dovuto a un **evento di timeout** a 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.
- Ma dopo un evento di timeout, il mittente TCP entra in una **fase di partenza lenta**: cioè, riassegna alla finestra di congestione il valore di un *MSS* e quindi incrementa la finestra 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 queste dinamiche utilizzando una variabile chiamata **threshold** (**soglia**), che determina la dimensione della finestra alla quale deve terminare la partenza lenta, e deve cominciare la prevenzione della congestione.



Andamento della finestra di congestione TCP

- Vediamo ora perché 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, perché 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 esso dimezza soltanto la sua finestra di congestione?
- Una vecchia versione di TCP, nota come TCP **Tahoe**, portava incondizionatamente la finestra di congestione a 1 MSS ed entrava nella fase di partenza lenta dopo qualunque tipo di evento di perdita.
- Come abbiamo visto, la versione più recente di TCP, TCP **Reno**, elimina la fase di partenza lenta dopo un ACK duplicato tre volte. Il motivo che ha portato a cancellare la fase di partenza lenta in questo caso 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).***

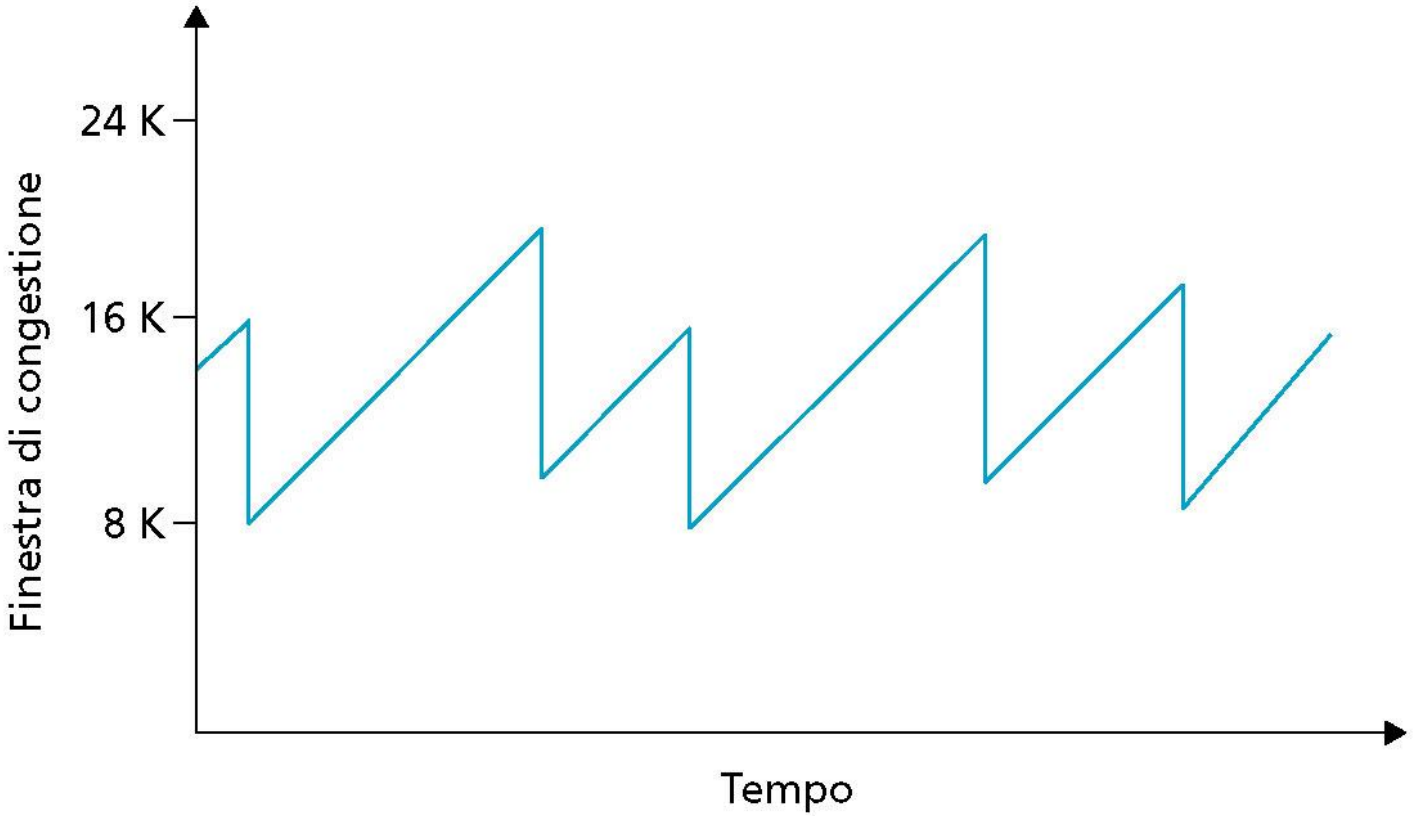
- Sono state proposte molte varianti all'algoritmo Reno.
- L'algoritmo TCP Vegas, sviluppato nell'Università di Arizona, tenta di evitare la congestione mantenendo comunque un buon throughput. L'idea alla base di Vegas è di
  - (1) rilevare la congestione nei router tra la sorgente e la destinazione *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.

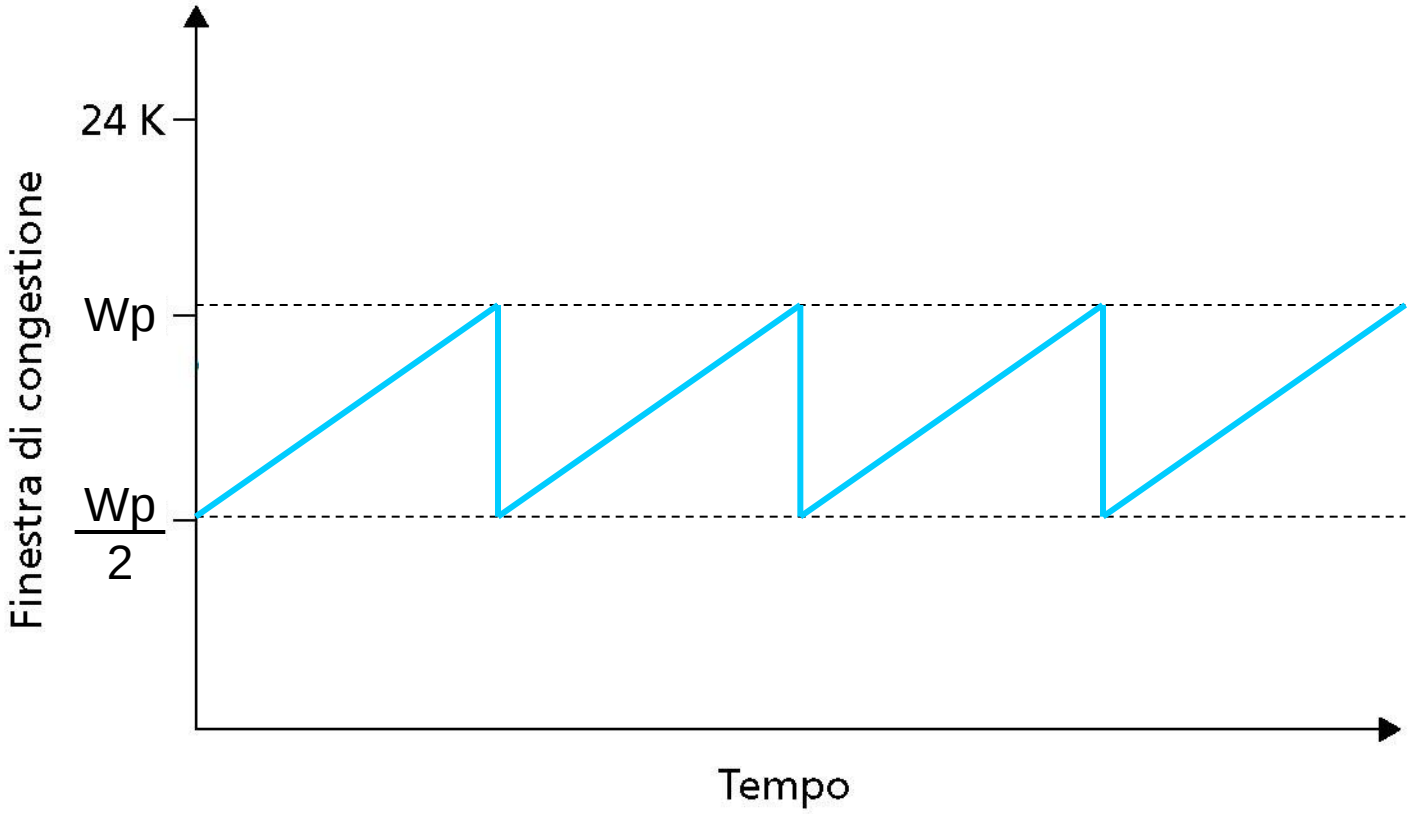
Descrizione macroscopica del throughput di TCP

- 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 stato stazionario.



- La comunicazione perde un pacchetto quando la velocità di trasmissione aumenta fino 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