
La rete Internet e l'architettura di protocolli TCP/IP

Controllo di congestione

- Il controllo della congestione ha lo scopo di evitare (**congestion control**) o risolvere (**congestion avoidance**) eventuali situazioni di sovraccarico nella inter-rete, limitando il traffico offerto alla rete
- Difficoltà:
 - × il protocollo IP (protocollo di rete) non possiede alcun meccanismo per rivelare e controllare la congestione
 - × il TCP è un protocollo end-to-end e può rivelare e controllare la congestione solo in modo indiretto
 - × la conoscenza dello stato della rete da parte delle entità TCP è imperfetta a causa dei ritardi (variabili) di rete
 - × le entità TCP che usano la rete non cooperano tra loro, anzi competono per l'uso delle risorse distribuite

Controllo di congestione

- × Il meccanismo “sliding window” per il controllo di flusso di TCP funziona da estremo ad estremo e quindi, in linea di principio, non può essere usato in modo efficiente per il controllo di congestione
- × Cioè il meccanismo funziona per evitare che un trasmettitore veloce sovraccarichi un ricevitore lento, ma non tiene conto della congestione di rete; ovvero il “collo di bottiglia” può essere la rete e non il ricevitore
- × Tuttavia seppure in modo implicito, e con alcune limitazioni, lo schema sliding window di TCP può proteggere, in caso di congestione, sia il destinatario che la rete

Controllo di congestione

- In caso di congestione infatti il controllo di flusso aiuta perchè:
- Al mittente arriveranno, per una data larghezza di finestra, **meno riscontri** e quindi saranno emessi meno segmenti
- Le **misure di RTT** fatte dal TCP per fissare il timeout permettono (quando la stima è fatta correttamente) di evitare ritrasmissioni inutili che porterebbero ad un aumento della congestione
- Inoltre, il valore stimato di RTT può essere usato dal TCP ricevente come misura della congestione e quindi può aiutarlo a decidere opportunamente la larghezza della finestra da comunicare al TCP mittente
- Infine, il meccanismo di ritrasmissione a intervalli crescenti (**back-off**) coopera per ridurre la congestione

Controllo di congestione

- Tutto questo significa che, calibrando opportunamente i parametri del protocollo, si può effettuare non solo un controllo di flusso ma anche un controllo di congestione
- Le prime implementazioni di TCP utilizzavano per il controllo della congestione anche il protocollo **ICMP**
- ICMP può rallentare il ritmo di trasmissione dell'host mittente, mediante l'invio di messaggi (**Source Quence**), nel momento in cui il destinatario si trovi a dover rifiutare datagrammi a causa della mancanza di risorse di memoria di ricezione
- Questo meccanismo di ICMP, nel caso di rapide variazioni del traffico, sembrò però del tutto insufficiente nel contesto di reti ad alta velocità (LAN)
- Si propose, quindi, fin dalla seconda metà degli anni '80, di implementare un **controllo della congestione basato solo sui time-out** e che prescindere da ICMP

Controllo di congestione

- Il controllo di flusso end-to-end del TCP riesce ad adattare il rate di emissione della sorgente in base al rate di arrivo degli ACK dei segmenti precedenti
- Il rate di arrivo degli ACK è determinato dal collo di bottiglia nella rete, ovvero nel percorso andata e ritorno tra sorgente e destinazione; il collo di bottiglia può essere il ricevitore o la rete
- I bottleneck in rete possono essere:
 - logici, causati dalla congestione nei router (nei buffer)
 - fisici, causati dalla limitazione della banda nei collegamenti fisici (+ facili da gestire)
 - dovuti al ricevitore, per la limitata capacità elaborativa del ricevitore

Controllo di congestione

- Il controllo di flusso di TCP
 - non è in grado di distinguere il tipo di bottleneck di rete
 - non può stabilire il tipo di contromisura più adatta
- TCP utilizza la stima di RTT come misura di congestione, lo scadere del timeout di ritrasmissione è considerato un sintomo di congestione
- Il controllo di flusso TCP ha funzione auto-sincronizzante (self-clocking): il sender usa gli ACK come "clock" per l'invio di nuovi segmenti nella rete
 - il tasso di generazione dei segmenti dipende dal tasso di ricezione degli ACK e questo dipende a sua volta dal link più lento sul path sorgente destinazione (se il bottleneck è nella rete) o dalla velocità del ricevitore (se il bottleneck è nel ricevitore)

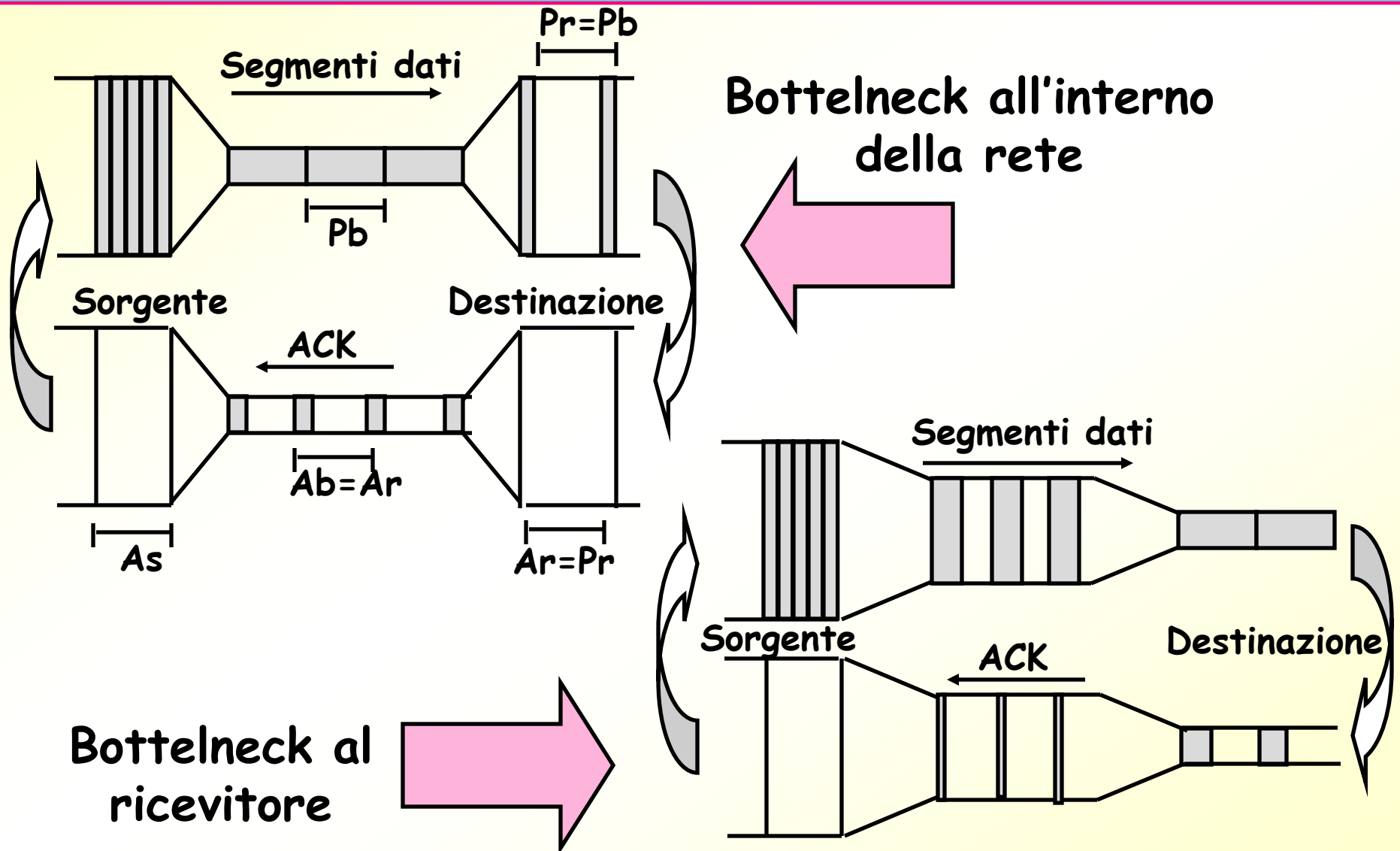
Controllo di congestione

- Bottleneck nella rete (vedi fig. seguente)
 - L'altezza (spessore) del collegamento tra sorgente e destinazione è proporzionale al data rate; sorgente e destinazione sono su reti ad alta capacità collegate da un link a bassa velocità che fa da bottleneck
 - Ogni segmento è rappresentato da un rettangolo la cui area è proporzionale al numero di bit, perciò sui link lenti i segmenti si allungano e si abbassano (dimensione orizzontale=tempo)
 - Il tempo P_b è la minima spaziatura tra i segmenti sul link più lento; all'arrivo dei segmenti a destinazione tale spaziatura viene mantenuta anche se aumenta il data rate perché il tempo di interarrivo non cambia, quindi $P_r = P_b$
 - Se la destinazione riscontra i segmenti appena arrivano (tempo di processamento uguale per tutti), allora la spaziatura degli ACK inviati è determinata dalla spaziatura di arrivo dei segmenti, quindi $A_r = P_r$
 - Dato che un time slot P_b può contenere un segmento dati, potrà a maggior ragione contenere un ACK, quindi $A_b = A_r$

Controllo di congestione

- Bottleneck nel ricevitore (vedi fig. seguente)
 - Il ricevitore può assorbire i segmenti lentamente o per limiti della sua velocità di elaborazione o perché sovraccaricato da segmenti che gli giungono da altre connessioni
 - In fig. assumiamo che il link più lento della rete sia relativamente veloce (circa metà del data rate della sorgente), mentre il pipe a destinazione sia stretto
 - In tal caso, gli ACK saranno generati alla velocità di assorbimento della destinazione, così i segmenti verranno generati alla velocità con cui possono essere gestiti dalla destinazione
- Nota: la sorgente non ha modo di accorgersi se il tasso di arrivo degli ACK rifletta lo stato della rete (controllo di congestione) o della destinazione (controllo di flusso)

Controllo di flusso: self-clocking



Controllo di congestione

- Sono stati definiti dei meccanismi aggiuntivi (oltre a RFC793) per migliorare le prestazioni in caso di congestione

Meccanismo	TCP Berkeley	TCP Tahoe	TCP Reno
Stima varianza RTT	☒	☒	☒
Backoff espon. RTO	☒	☒	☒
Algoritmo di Karn	☒	☒	☒
Slow Start	☒	☒	☒
Congestion Avoidance	☒	☒	☒
Fast Retransmit		☒	☒
Fast Recovery			☒

Controllo di congestione

- Nelle implementazioni attuali, si considera lo scadere di un time-out come un sintomo di congestione delle risorse di interconnessione e si usano nuovi algoritmi per porre rimedio a tali situazioni
- Algoritmo di Jacobson, Algoritmo di Karn + Backoff esponenziale li abbiamo già descritti, ora vedremo gli algoritmi di Slow Start e Congestion Avoidance, Fast Retransmit e Fast Recovery che agiscono sulla dimensione della finestra del sender

Controllo di congestione: CUTE

- Ad esempio, l'algoritmo **CUTE** (**Congestion control Using Time-outs of the End-to-end layer**) fissa il valore della finestra a disposizione del trasmettitore, non pari al valore della Advertised window comunicata dal destinatario, bensì variabile tra un minimo e un massimo
- Tale algoritmo utilizza i seguenti parametri e procedure:
 - **Massimo**: è il valore massimo della finestra in trasmissione; in generale è pari al valore della Advertised window offerta dal destinatario
 - **Minimo**: è il valore minimo della finestra; tipicamente pari al valore di una MSS
 - **Inizializzazione**: rappresenta il valore iniziale della finestra; su reti molto cariche è preferibile partire dal valore minimo

Controllo di congestione: CUTE

- **Incremento**: si incrementa la finestra in trasmissione di un valore pari ad un segmento, ogni N segmenti ricevuti correttamente dal TCP di destinazione (senza mai superare il valore massimo);
- **Decremento**: si decrementa il valore della finestra ogni volta che scade un timeout; il decremento può essere di diversi tipi:
 - **sudden**, la finestra è posta uguale al valore minimo
 - **gradual**, la finestra si riduce di un segmento
 - **binary**, la finestra si divide a metà
- Normalmente i parametri sono scelti in modo da avere un incremento lento e un decremento veloce, per ridurre problemi di instabilità

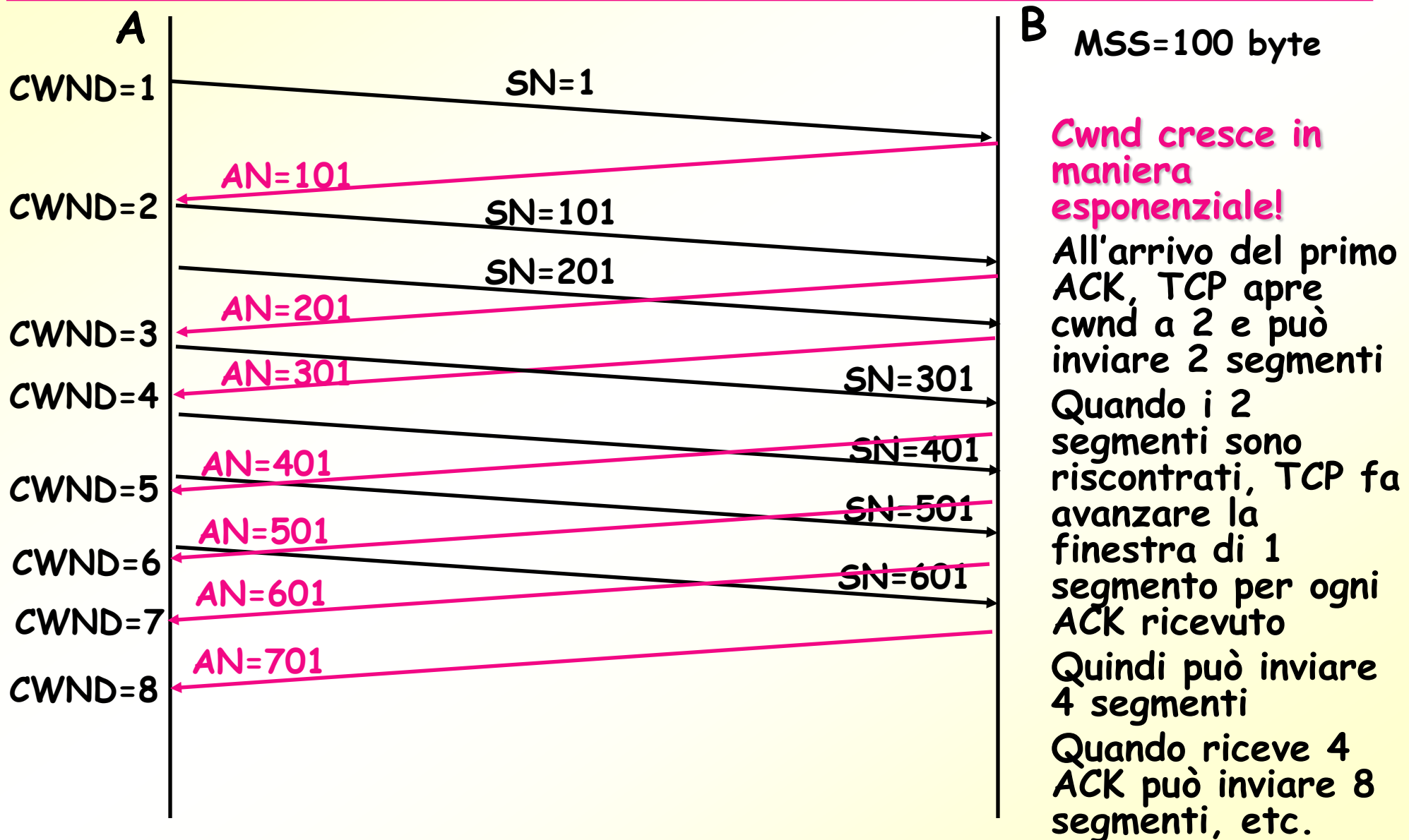
Controllo di congestione: Slow Start

- Slow Start (proposto da Jacobson) tende ad evitare l'insorgere di congestione durante la fase di avvio di una connessione, perciò espande gradualmente la finestra del sender
- Regola l'emissione dei segmenti all'inizio di una connessione e ha lo scopo di raggiungere il ritmo di emissione a regime senza causare congestione
- Si definisce una **Congestion Window** (cwnd) (misurata in segmenti) che tende ad aumentare progressivamente
- La congestion window limita il valore della finestra fino a che questo non sia fissato dalla ricezione degli ACK

Slow Start

- L'ampiezza della finestra *allowed window* (awnd) usata dal sender in segmenti è:
$$awnd = \min [\text{credit}, cwnd]$$
- credit:
 - numero di crediti (in segmenti) concessi nell'ultimo ACK (=RCV.WND/MSS)
- cwnd:
 - congestion window (in segmenti)
 - per il primo segmento (allo start-up della connessione o alla ripartenza dopo una perdita)
 - $cwnd=1$
 - per ogni segmento riscontrato, cwnd è incrementata di 1 segmento fino ad un valore massimo
 - $cwnd = \min [Mwnd, cwnd+1]$
 - Mwnd è il massimo valore della congestion window

Slow Start: esempio



Congestion Avoidance

- Regola l'ampiezza della finestra in caso di congestione durante la connessione (congestion avoidance)
- Il meccanismo è innescato in caso di timeout (ritrasmissione) e consente di controllare il flusso di una sorgente per
 - ✓ consentire l'esaurimento della congestione
 - ✓ evitare un sovraccarico della rete
- Usare la procedura di slow start in caso di congestione potrebbe essere troppo aggressiva perché la crescita della finestra è esponenziale

Congestion Avoidance

Procedura di congestion avoidance allo scadere di un timeout

- Dimezzare il valore della cwnd

$$Cwnd = cwnd / 2$$

- Ad ogni ACK si incrementa linearmente cwnd:

$$cwnd += 1 / cwnd$$

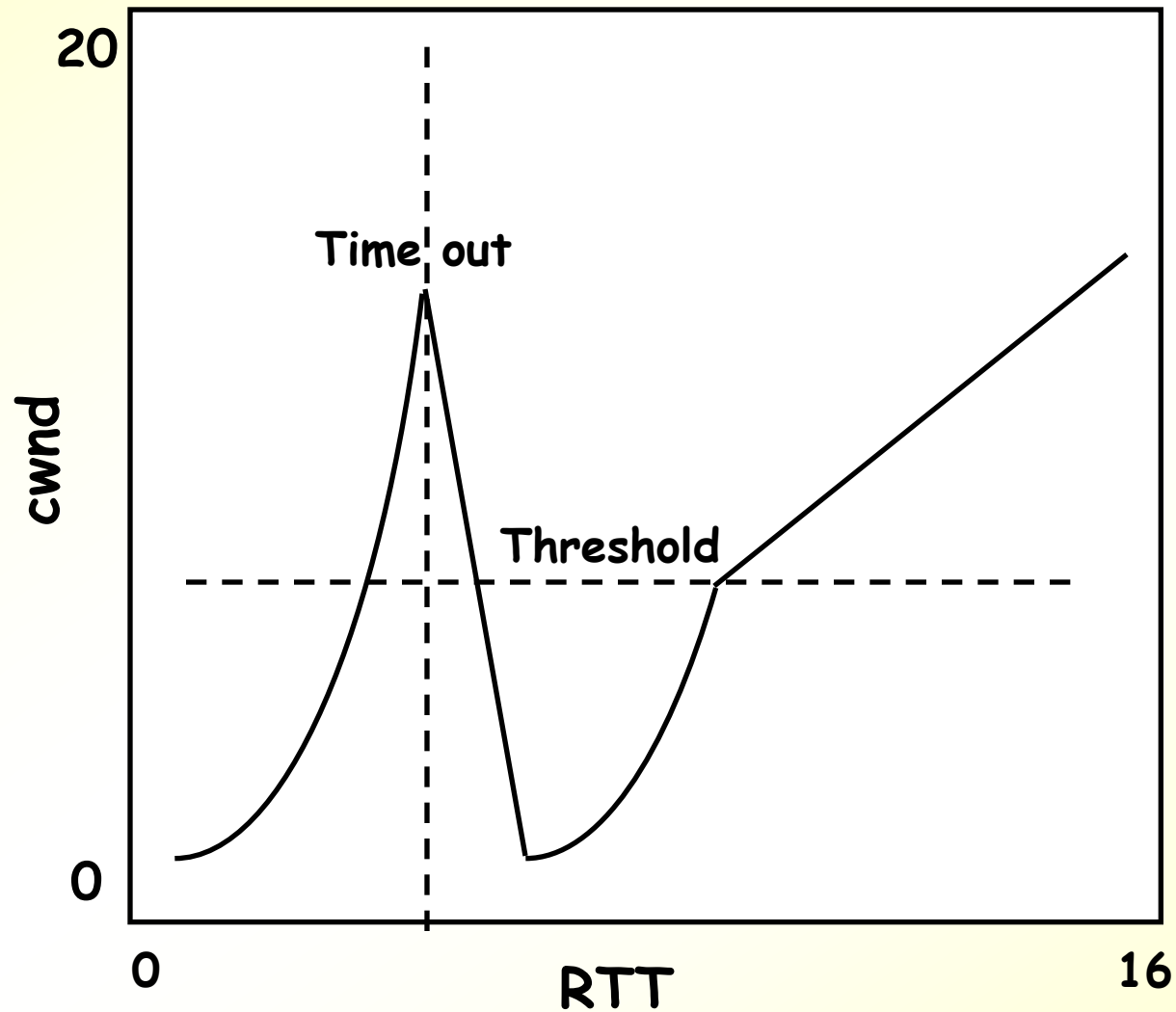
- Inviare $\min(\text{receiver window}, cwnd)$
- Tuttavia, la procedura di congestion avoidance è usata in combinazione con la procedura di slow start nel modo seguente

Slow Start + Congestion Avoidance

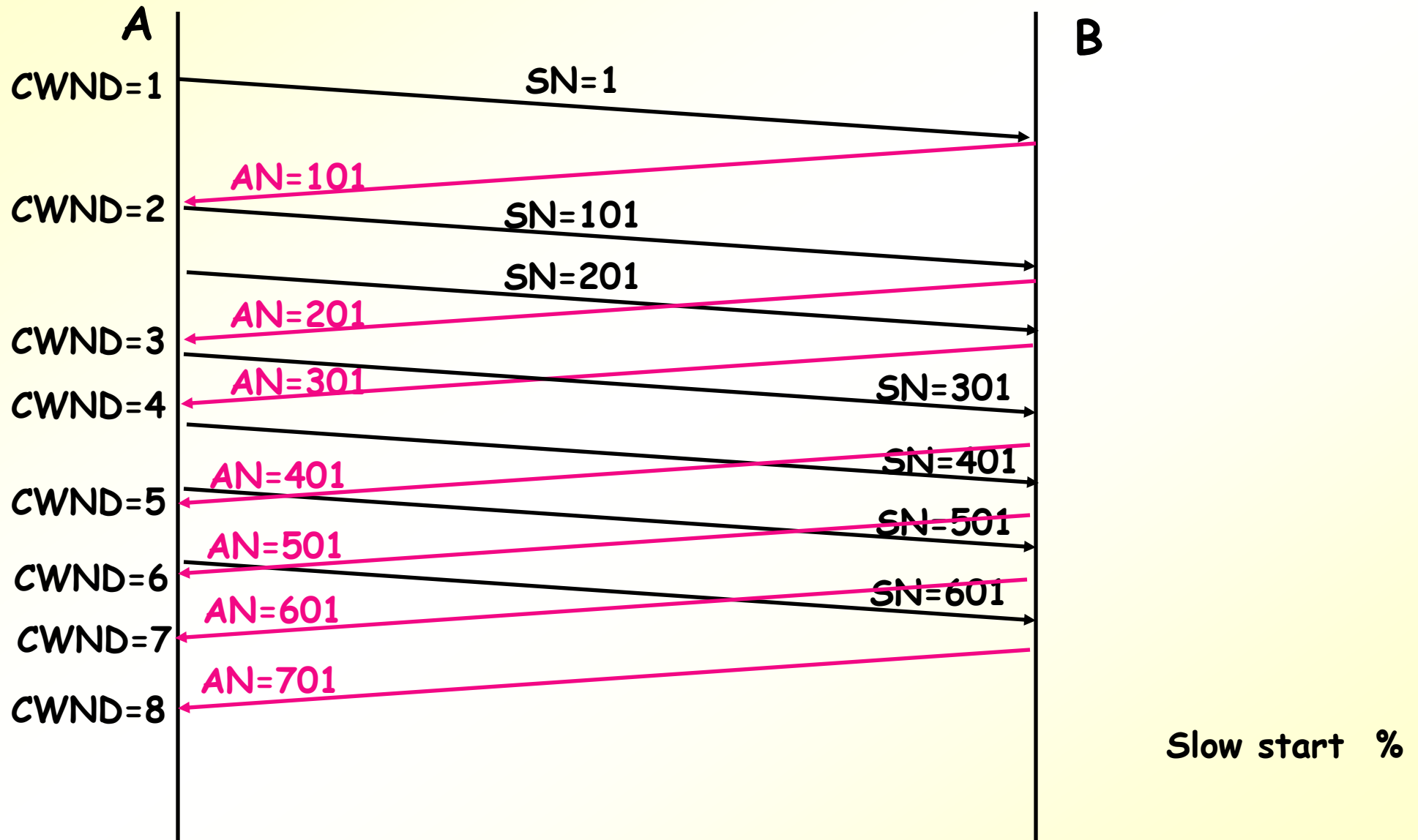
Procedura di slow start+ congestion avoidance allo scadere di un timeout

- fissare una soglia per il passaggio da slow start a congestion avoidance uguale alla metà del valore corrente della congestion window
$$ssthresh = cwnd / 2 \text{ (in effetti, } ssthresh = \min(cwnd/2, \text{credit}) \text{)}$$
- fissare $cwnd=1$ ed eseguire la procedura slow start finchè $cwnd < ssthresh$; in questa fase, $cwnd$ è incrementato di 1 per ogni ACK ricevuto (apertura esponenziale)
- se $cwnd \geq ssthresh$, parte la fase di congestion avoidance, $cwnd$ è incrementato di uno ogni round trip delay ($cwnd += 1/cwnd$) (apertura lineare)

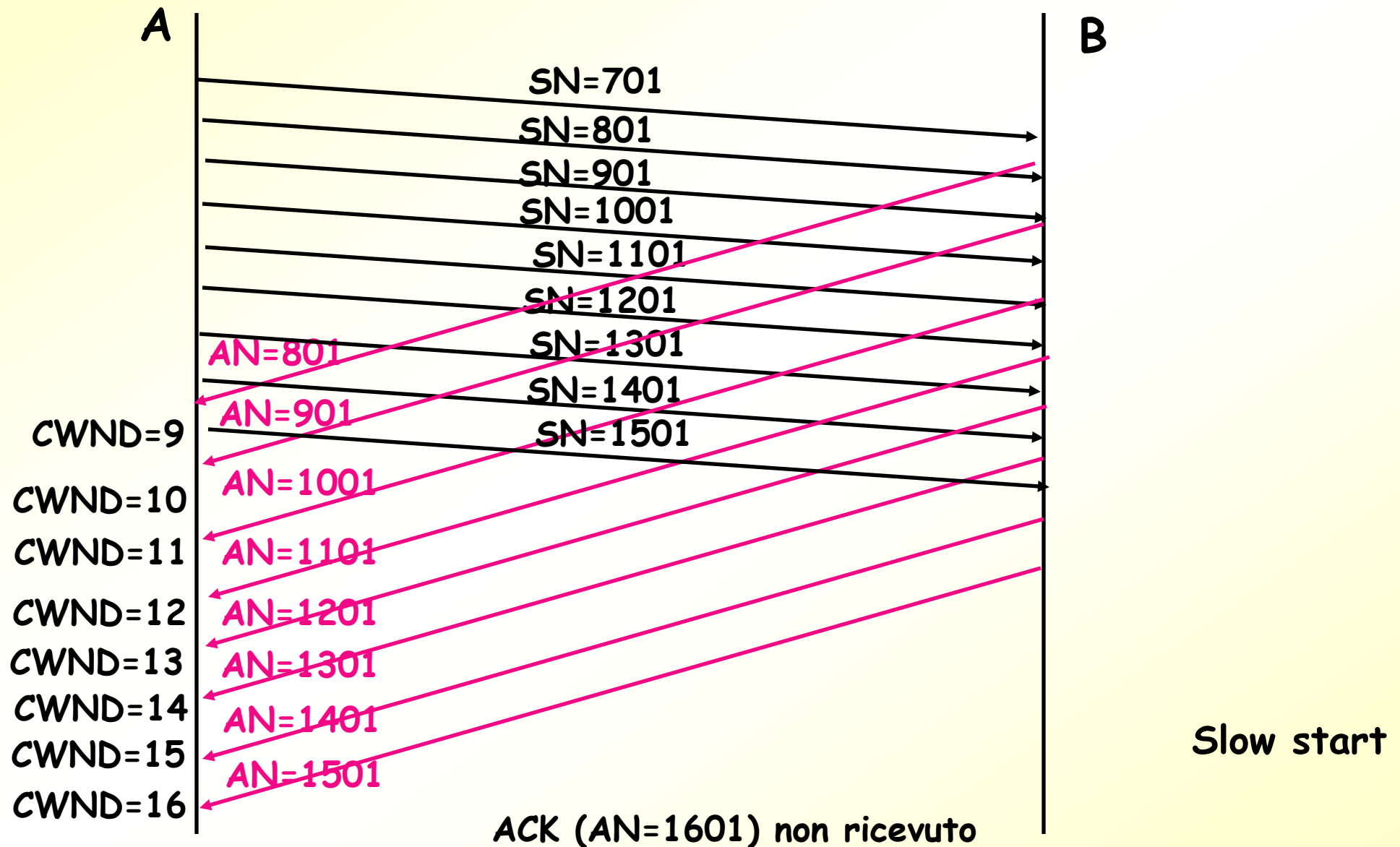
Congestion Avoidance



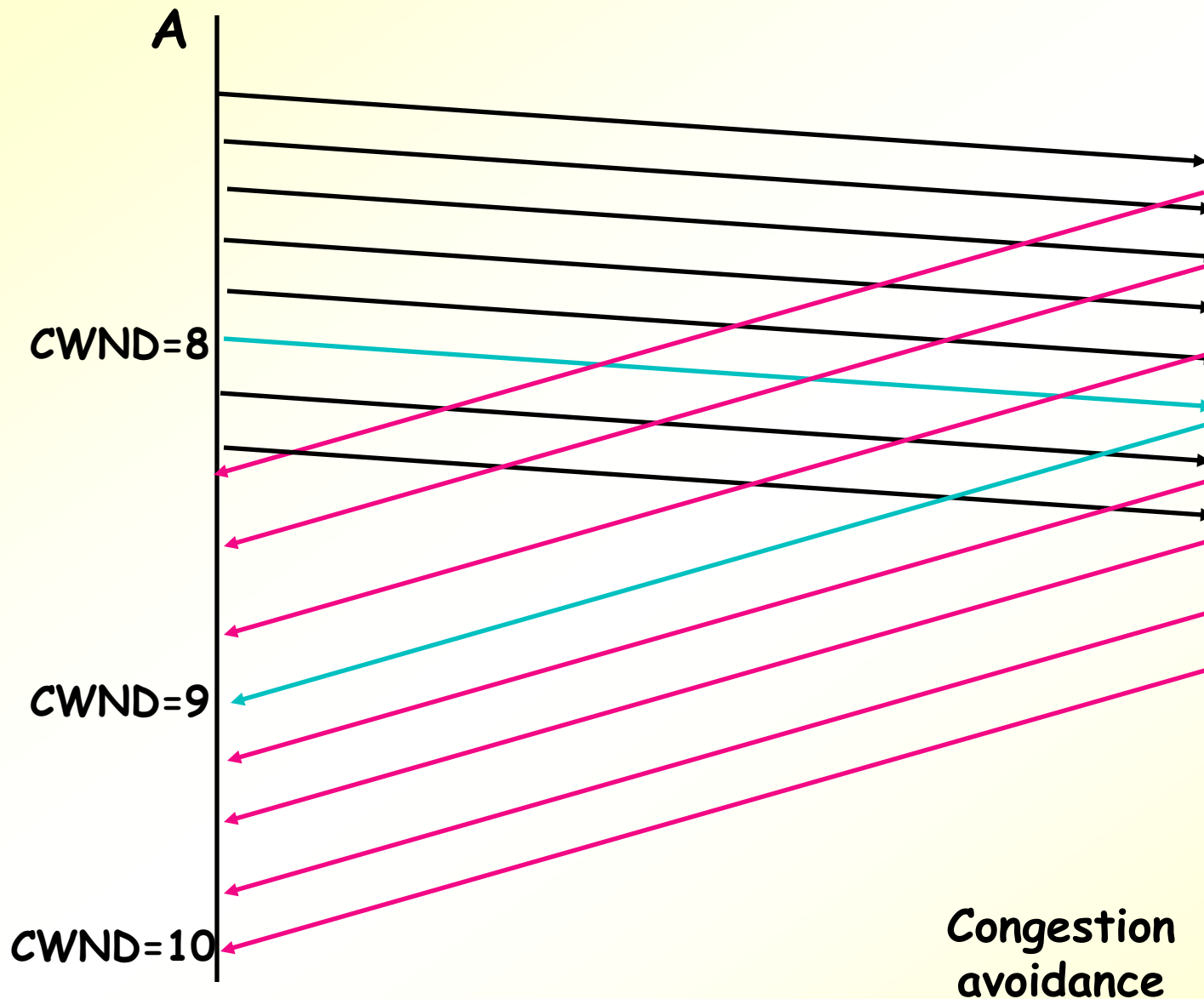
Congestion Avoidance: esempio



Congestion Avoidance: esempio



Congestion Avoidance: esempio



B Dopo il timeout, ssthresh=8

TCP usa slow-start fino a raggiungere il threshold (come in fig. slow start), poi cwnd cresce linearmente con RTT

Ci vogliono 11 RTT per tornare alla dimensione di cwnd che nel caso solo Slow Start si raggiunge in 4 RTT

Fast Retransmit

- Migliora le prestazioni se si perde un singolo segmento
 - × velocizza la ritrasmissione del segmento perso
 - × evita la ritrasmissione dei segmenti successivi già ricevuti con successo
- Assunzioni su cui si basa la procedura (proposta da Jacobson nel 90) sono
 - × il ricevitore che riceve un segmento fuori sequenza emette immediatamente un ACK per l'ultimo segmento ricevuto in ordine; e continua a ripetere quest'ACK per ogni segmento successivo finché non riceve il segmento fuori sequenza
 - × quindi il TCP ricevente genera un ACK cumulativo per tutti i segmenti ricevuti in ordine nel frattempo

Fast Retransmit

- Se una sorgente TCP riceve un ACK duplicato, questo può significare (1) che il segmento che segue quello riscontrato è stato ritardato e arriverà fuori sequenza, o (2) il segmento si è perso. Per essere sicuro che il caso in esame è il (2) e non l'(1) J. suggerì:
 - × la ricezione di tre ACK duplicati per lo stesso segmento è sintomo che il segmento successivo è perso
 - × in questo caso è molto probabile che il segmento sia perso e quindi conviene ritrasmetterlo senza aspettare la scadenza del timeout
- La ritrasmissione del segmento inizia non appena sono ricevuti quattro ACK (3 duplicati e uno normale) del segmento precedente anche se il timeout non è scaduto

Fast Recovery

- Quando il TCP usa il fast retransmit per ritrasmettere un segmento, assume che il segmento sia perso anche se il timeout non è ancora scaduto, quindi deve prendere misure per combattere la congestione usando qualche meccanismo simile a slow start/congestion avoidance
- La tecnica di Fast Recovery proposta da Jacobson è associata alla procedura di fast retransmit
 - × l'arrivo di ACK multipli assicura che i segmenti sono ricevuti abbastanza regolarmente, quindi la procedura normale di slow start/cong. avoidance potrebbe essere troppo conservativa

Fast Recovery

- Rispetto alla procedura normale di slow start/congestion avoidance, il fast retransmit evita la fase iniziale di slow start, cioè:
 - × Si ritrasmette il segmento perso (fast retransmit)
 - × Si dimezza la cwnd
 - × Si procede aumentando linearmente la cwnd ad ogni ACK

Fast Recovery

- La procedura è la seguente
 - × quando sono stati ricevuti tre ACK duplicati
 - si pone: $ssthresh = cwnd/2$
 - viene ritrasmesso il segmento perduto
 - per tener conto dei segmenti già ricevuti (nella cache del ricevitore con numeri di sequenza successivi a quello ritrasmesso) si pone:
$$cwnd = ssthresh + 3$$
- ogni volta che arriva un ulteriore ACK duplicato (per lo stesso segmento), il valore di $cwnd$ viene incrementato di 1 e trasmesso (se possibile) un segmento (tiene conto di eventuali altri Ack duplicati in viaggio nella rete)
- quando viene ricevuto un ACK (riscontro cumulativo del segmento perso più altri)
 - si pone $cwnd = ssthresh$ e si entra nella fase di congestion avoidance

TCP: comandi utente

Open

Format: OPEN (local port, foreign socket, active/passive [, timeout]
[, precedence] [, security/compartment] [, options])
-> local connection name

If the **active/passive** flag is set to passive, then this is a call to LISTEN for an incoming connection. A passive open may have either a fully specified foreign socket to wait for a particular connection or an unspecified foreign socket to wait for any call. On an active OPEN command, the TCP will begin the procedure to synchronize (i.e., establish) the connection at once.

The **timeout**, if present, permits the caller to set up a timeout for all data submitted to TCP. If data is not successfully delivered to the destination within the timeout period, the TCP will abort the connection. The present global default is five minutes.

The TCP or some component of the operating system will verify the users authority to open a connection with the specified **precedence** or **security/compartment**.

A local connection name will be returned to the user by the TCP. It can then be used as a short hand term for the connection defined by the <local socket, foreign socket> pair.

TCP: comandi utente

Send

Format: SEND (local connection name, buffer address, byte count, PUSH flag, URGENT flag [,timeout])

This call causes the data contained in the indicated user buffer to be sent on the indicated connection.

If the **PUSH** flag is set, the data must be transmitted promptly to the receiver, and the PUSH bit will be set in the last TCP segment created from the buffer. If the PUSH flag is not set, the data may be combined with data from subsequent SENDs for transmission efficiency.

If the **URGENT** flag is set, segments sent to the destination TCP will have the urgent pointer set. The receiving TCP will signal the urgent condition to the receiving process if the urgent pointer indicates that data preceding the urgent pointer has not been consumed by the receiving process. The purpose of urgent is to stimulate the receiver to process the urgent data and to indicate to the receiver when all the currently known urgent data has been received.

If a **timeout** is specified, the current user timeout for this connection is changed to the new one.

TCP: comandi utente

Receive

Format: RECEIVE (local connection name, buffer address, byte count) -> byte count, urgent flag, push flag

This command allocates a receiving buffer associated with the specified connection.

The buffer will be filled with as much data as it can hold. If a **PUSH** is seen before the buffer is filled the buffer will be returned partially filled and PUSH indicated.

If there is urgent data the user will have been informed as soon as it arrived via a TCP-to-user signal. The receiving user should thus be in "urgent mode". If the **URGENT** flag is on, additional urgent data remains. If the URGENT flag is off, this call to RECEIVE has returned all the urgent data, and the user may now leave "urgent mode".

To distinguish among several outstanding RECEIVES and to take care of the case that a buffer is not completely filled, the return code is accompanied by both a **buffer pointer** and a **byte count** indicating the actual length of the data received.

TCP: comandi utente

Close

Format: `CLOSE` (local connection name)

This command causes the connection specified to be closed.

Closing connections is intended to be a graceful operation in the sense that outstanding SENDs will be transmitted (and retransmitted), as flow control permits, until all have been serviced. It should also be clear that users should continue to RECEIVE on CLOSING connections, since the other side may be trying to transmit the last of its data. Thus, `CLOSE` means "I have no more to send" but does not mean "I will not receive any more."

It may happen (if the user level protocol is not well thought out) that the closing side is unable to get rid of all its data before timing out. In this event, `CLOSE` turns into `ABORT`, and the closing TCP gives up.

The user may `CLOSE` the connection at any time on his own initiative, or in response to various prompts from the TCP (e.g., remote close executed, transmission timeout exceeded, destination inaccessible). Because closing a connection requires communication with the foreign TCP, connections may remain in the closing state for a short time. Close also implies push function.

TCP: comandi utente

Status

Format: STATUS (local connection name) -> status data

This is an implementation dependent user command. It returns a data block containing the following information:

- local socket,
- foreign socket,
- local connection name,
- receive window,
- send window,
- connection state,
- number of buffers awaiting acknowledgment,
- number of buffers pending receipt,
- urgent state,
- precedence,
- security/compartment,
- and transmission timeout

Abort

Format: ABORT (local connection name)

This command causes all pending SENDs and RECEIVES to be aborted, and a special RESET message to be sent to the TCP on the other side of the connection.

TCP: comandi utente

Opzioni del TCP

- Le opzioni implementative del TCP riguardano:
 - × Send policy: politica di emissione dei segmenti
 - × Deliver policy: politica di consegna dei segmenti
 - × Accept policy: politica di accettazione a destinazione dei segmenti fuori sequenza
 - × Retransmit policy: politica di ritrasmissione dei segmenti allo scadere del timeout
 - × Acknowledge policy: politica di invio dei riscontri dei segmenti correttamente ricevuti
- E' garantita l'interoperabilità tra implementazioni di TCP basate su qualsiasi opzione

Opzioni del TCP

Send policy

Alternativa 1: immediata

- emissione di un segmento al momento in cui si ricevono i dati dall'applicazione
- trasmissioni frequenti di segmenti brevi
- overhead elevato
- silly window syndrome

Alternativa 2: ritardata

- attesa di aver accumulato un determinato ammontare di dati
- overhead minore
- ritardo maggiore

Opzioni del TCP

Delivery policy

Alternativa 1: immediata

- consegna dei segmenti al momento in cui sono ricevuti
- attesa minore
- processing elevato per il sistema operativo se i segmenti sono brevi

Alternativa 2: ritardata

- attesa di aver ricevuto un determinato ammontare di dati (buffer di ricezione)
- attesa maggiore

Opzioni del TCP

Accept policy

Alternativa 1: in ordine

- sono accettati solo i segmenti in sequenza, i segmenti fuori sequenza sono scartati
- semplicità implementativa
- maggiore volume di ritrasmissioni

Alternativa 2: in finestra

- i segmenti fuori sequenza sono accettati se sono compresi nella finestra di ricezione
- sofisticata gestione dei buffer di ricezione

Opzioni del TCP

Acknowledge policy

Alternativa 1: invio immediato

- il riscontro (pacchetto vuoto) viene emesso non appena il segmento è stato accettato
- maggiore volume di traffico
- ritardo minore

Alternativa 2: riscontro cumulativo

- attesa dell'invio di un segmento dati e utilizzazione del piggyback
- eventuale timer che limiti il ritardo di emissione dei riscontri

Opzioni del TCP

Retransmit policy

- Le prestazioni dipendono anche dalle modalità di accettazione e di riscontro utilizzate dal ricevitore

Alternativa 1: ritrasmissione del primo segmento

- è definito un singolo timeout per tutta la coda di segmenti emessi
- all'arrivo di un ACK sono rimossi i segmenti riscontrati e resettato il timer
- se il timeout scade, è riemesso solo il primo segmento ed è resettato il timer
- Prestazioni
 - Ritardi potenzialmente elevati
 - Minimizzazione delle ritrasmissioni

Opzioni del TCP

Retransmit policy

Alternativa 2: ritrasmissione a gruppi

- è definito un singolo timeout per tutta la coda di segmenti emessi
- all'arrivo di un ACK sono rimossi i segmenti riscontrati e resettato il timer
- se il timeout scade, sono riemessi tutti i segmenti della coda ed è resettato il timer

Opzioni del TCP

Retransmit policy

Alternativa 3: individuale

- è definito un timeout per ogni segmento
- all'arrivo di un ACK sono rimossi i segmenti riscontrati e sono disattivati i relativi timer
- se un timeout scade, è ritrasmesso solo il segmento associato al timeout scaduto e resettato esclusivamente il suo timer
- Prestazioni
 - ottimali per il numero di ritrasmissioni
 - implementazione complessa