



Corso di
SISTEMI TELEMATICI
a.a. 2011-2012

**Lo strato di Trasporto: controllo della
congestione**

Calcolo del RTT



• L'SRTT privilegia i valori dei campioni di RTT più recenti:

$$\text{SRTT}(k+1) = a \text{SRTT}(k) + (1-a) \text{RTT}(k+1)$$

$0 \leq a \leq 1$; tanto più il valore del peso (smoothing factor) a è vicino a 0, tanto maggiore sarà il peso dato all'ultima osservazione di RTT (normalmente $0.8 \leq a \leq 0.9$)

- Valori minori del peso corrispondono ad un aggiornamento veloce del SRTT; valori maggiori rendono il SRTT insensibile a brevi variazioni del ritardo di trasferimento

Calcolo del RTT



- Usando un valore costante di a ($0 < a < 1$), indipendentemente dal numero di osservazioni passate, si considerano comunque tutte le osservazioni ma si dà minor peso a quelle più distanti
- Infatti:

$$\text{SRTT}(k+1) = (1-a) \text{RTT}(k+1) + a (1-a) \text{RTT}(k) + a^2(1-a) \text{RTT}(k-1) + \dots + a^k(1-a) \text{RTT}(1) + a^{k+1} \text{RTT}(0)$$

dato che a e $(1-a)$ sono ≤ 1 ogni termine successivo è più piccolo; per es. per $a = 0.8$ ($= 7/8$), si ha:

$$\text{SRTT}(k+1) = 0.2 \text{RTT}(k+1) + 0.16 \text{RTT}(k) + 0.128 \text{RTT}(k-1) + \dots$$

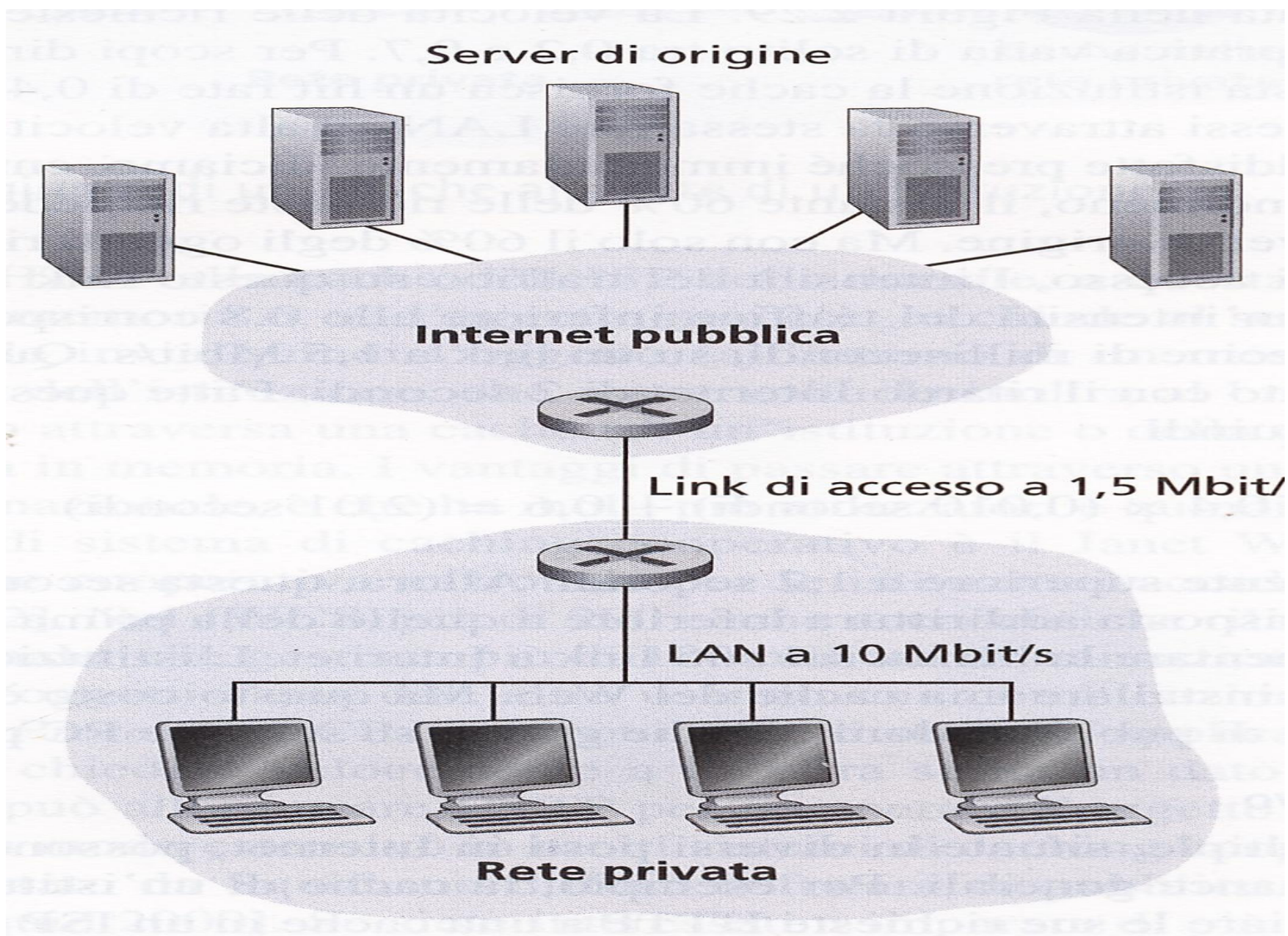
Calcolo del RTO



La specifica TCP (RFC 793) prevede che il valore di RTO sia proporzionale a SRTT secondo la seguente espressione:

$$RTO(k+1) = \min[UBOUND, \max [LBOUND, b \text{ SRTT}(k+1)]]$$

- ✕ **LBOUND** (valore tipico 1 s) e **UBOUND** (valore tipico 1 m) sono i prefissati limiti inferiore e superiore di RTO
- ✕ **b** è una costante il cui valore deve essere scelto $1.3 < b < 2.0$ (delay variance factor)
- ✕ Il lower bound **DOVREBBE** essere misurato in frazioni di secondo (per accomodare LAN ad alta velocità) e l'upper bound dovrebbe essere $2 \cdot MSL$, cioè 240 secondi



Esempio di bottleneck



Algoritmi di Karn e Jacobson

- Ci sono due problemi legati al calcolo del RTO specificato nel RFC-793:
 - la misura accurata del RTT è difficile quando ci sono ritrasmissioni
 - l'algoritmo per calcolare SRTT è inadeguato, perché assume erroneamente che la varianza nei valori di RTT è piccola e costante



Algoritmi di Karn e Jacobson

• Questi problemi sono rispettivamente risolti dagli algoritmi di Karn e Jacobson

- Nel calcolo di SRTT, l'algoritmo di Karn ignora i riscontri di segmenti ritrasmessi
- L'algoritmo di Jacobson incorpora la misura della varianza del RTT ed è importante soprattutto su link a bassa velocità, dove la variazione delle dimensioni dei pacchetti causa un'ampia variazione di RTT. L'algoritmo migliora l'utilizzazione di un link da 9.6 kbps dal 10% al 90%



Stima della varianza di RTT

- Il metodo di misura standard di SRTT non è adatto a situazioni in cui la varianza del ritardo di rete è elevata

- ✗ se il bit rate della connessione TCP basso, allora il ritardo di trasmissione può essere elevato rispetto al ritardo di propagazione e perciò la varianza del ritardo dipende dalla dimensione dei datagrammi IP (dai dati e non dalla rete)
- ✗ variazioni repentine di traffico e di carico in rete possono provocare brusche variazioni di RTT
- ✗ l'entità TCP ricevente può inviare i riscontri in maniera cumulativa o comunque dopo un certo tempo di processamento



Stima della varianza di RTT

- L'RFC 793 specifica il timeout come il doppio del valore di RTT stimato:

$$SRTT[k+1] = (1-a) SRTT[k] + a RTT[k+1]$$

$$0 < a < 1; \text{tipicamente } a = 1/8 = 0.125$$

$$RTO = b SRTT = 2 SRTT$$

- Scegliere $b=2$, un valore costante, non risponde ai cambiamenti della varianza



Stima della varianza di RTT

- In realtà, SRTT oscilla in maniera random attorno al valor medio con una deviazione standard $SDEV(RTT)$
- Primo problema (sovrastima di RTO!):
supponiamo una rete non-congestionata tale che l'RTT resti quasi costante per un lungo intervallo. Improvvisamente un pacchetto si perde. Dopo RTT secondi, l'ACK non è ancora stato ricevuto. E' quasi certo che il pacchetto è stato scartato, tuttavia bisogna aspettare RTT secondi prima di ritrasmetterlo!



Stima della varianza di RTT

• Secondo problema (sottostima di RTO):

dalla teoria delle code, se il carico di rete aumenta, il valore medio di RTT aumenta e (cosa peggiore) la sua varianza cresce anche di più (inversamente proporzionali a $(1-r)$). In queste circostanze, fissare RTO pari solo al doppio del valore misurato di RTT potrebbe essere troppo piccolo (anche minore del RTT reale). Cioè, i pacchetti che impiegano tanto tempo ad arrivare per via della congestione di rete (ma che arriveranno) sono ritrasmessi aumentando ancora la congestione

- Jacobson afferma che questo succede con carichi di rete superiori al 30%
- J. propone di usare un valore di b grossolanamente proporzionale alla deviazione standard della pdf del tempo di arrivo degli ack



Algoritmo di Jacobson

• Per misurare la variazione di RTT ci sono varie alternative: la scelta convenzionale è la varianza e quindi la deviazione standard:

$$s^2 = 1/n \sum |RTT - SRTT|^2$$

• La stima della deviazione standard di RTT è troppo complessa (calcolo di quadrati e radici quadrate)



Algoritmo di Jacobson

- J. suggerì di usare la DEVIAZIONE MEDIA MDEV(RTT) dei campioni del RTT (o errore di predizione medio) come stimatore della deviazione standard di RTT:

$$\text{MDEV}(x) = E [|X - E[X]|]$$

$$\text{MDEV}^2(\text{RTT}) = 1/n \sum (|\text{RTT} - \text{SRTT}|)^2$$

- Se gli errori di predizione sono distribuiti normalmente, $\text{MDEV} = (\pi/2)^{1/2} \text{SDEV} \approx 1.25$, cioè MDEV è una buona approssimazione di SDEV ed è molto più facile da calcolare



Algoritmo di Jacobson

- J. propose di calcolare RTO come:

$$RTO(k+1) = SRTT(k+1) + u * MDEV(RTT)$$

dove u è di solito fissato a $u=4$

Giustificazione

- Se c'è poca differenza tra i valori campionati di RTT e la stima del SRTT per lungo tempo ($MDEV(RTT) \rightarrow 0$), si può pensare che SRTT è una buona stima di RTT. Quindi, se un riscontro impiega più di SRTT sec ad arrivare, si può pensare, a ragione, di ritrasmettere il segmento
- D'altra parte, se RTT ha un'alta varianza (grande $MDEV(RTT)$), allora la stima non è molto affidabile ed è opportuno avere un margine di sicurezza proporzionale all'incertezza



Algoritmo di Jacobson

La procedura di calcolo dinamica di RTO è

- calcolo della stima di RTT, $SRTT(K+1)$:

$$SRTT(k+1) = (1-a) SRTT(k) + a RTT(k+1) = \\ SRTT(k) + a (RTT(k+1) - SRTT(k))$$

- valori tipici di a sono 0.1-0.2
- calcolo dell'errore nella predizione di RTT, $SERR(K+1)$:

$$SERR(K+1) = RTT(K+1) - SRTT(K)$$

- quindi:

$$SRTT(k+1) = SRTT(k) + a SERR(k+1)$$

- la nuova predizione è basata sulla vecchia più una frazione dell'errore nella predizione



Algoritmo di Jacobson

- Calcolo della deviazione media $MDEV(K+1)$
- J. propose di usare la stessa tecnica di media esponenziale (exponential smoothing) usata per la stima di RTT anche per la stima di $MDEV$, indicata con $SDEV$, quindi:

$$SDEV(k+1) = (1-h) SDEV(k) + h |SERR(k+1)|$$

- calcolo del valore del timeout $RTO(K+1)$

$$RTO(k+1) = SRTT(k+1) + u SDEV (k+1)$$

- $a=0.125$ ($1/8$), $h=0.25$ ($1/4$), $u=4$ (all'inizio J. disse $u=2$, poi si accorse che $u=4$ aveva maggiori vantaggi: (1) la moltiplicazione per 4 può essere fatta con un solo shift; (2) minimizza timeout e ritrasmissioni non necessarie perché meno dell'1% di tutti i pacchetti arrivano con più di 4 deviazioni standard di ritardo)



Algoritmo di Jacobson

Con l'algoritmo di J. i valori di RTO calcolati sono piuttosto conservativi rispetto ai valori misurati di RTT finché i campioni di RTT variano, poi cominciano a convergere verso RTT quando i valori dei campioni si stabilizzano, cioè la stima della variazione SDEV si riduce (vedi fig. libro Stallings pg. 260)



Algoritmi di Jacobson

- L'algoritmo di base

Per calcolare il valore corrente di RTO, un sender TCP mantiene 2 variabili di stato, SRTT e RTTVAR (round-trip time variation). Inoltre, assumiamo una granularità del clock di G secondi

Le regole per il calcolo di SRTT, RTTVAR, e RTO sono:

- (1) Finché non viene misurato un RTT per un segmento, il sender DOVREBBE porre $RTT=0$, $RTO=3$ sec (RFC 1122), anche se si applica il "back off" sulle ritrasmissioni ripetute
- (2) Quando si effettua la prima misura di RTT, R, l'host DEVE porre

$SRTT \leftarrow R$

$RTTVAR \leftarrow R/2$

$RTO \leftarrow SRTT + \max(G, K \cdot RTTVAR)$

dove $K = 4$



Algoritmi di Jacobson

- (3) Quando si effettua una misura successiva R' di RTT, l'host DEVE porre

$$\begin{aligned} \text{RTTVAR} &\leftarrow (1-\text{beta}) * \text{RTTVAR} + \text{beta} * |\text{SRTT}-R'| \\ \text{SRTT} &\leftarrow (1-\text{alpha}) * \text{SRTT} + \text{alpha} * R' \end{aligned}$$

Il valore di SRTT usato nell'aggiornamento di RTTVAR è il suo valore prima dell'aggiornamento di SRTT stesso usando il secondo assegnamento. Il calcolo DOVREBBE essere eseguito usando $\text{alpha}=1/8$ e $\text{beta}=1/4$

Dopo il calcolo, un host DEVE aggiornare

$$\text{RTO} \leftarrow \text{SRTT} + \max(G, K * \text{RTTVAR})$$

- (4) Una volta che RTO è calcolato, se è minore di 1 sec allora RTO DOVREBBE essere arrotondato a 1 sec. Un valore minimo di RTO più grande è necessario per mantenere il TCP conservativo e per evitare ritrasmissioni spurie
- (5) Un valore massimo PUO' essere fissato per RTO purché sia almeno 60 sec



Algoritmo di Karn / Backoff exp

Altri due fattori devono essere considerati per migliorare le prestazioni del TCP:

1. **Quale valore di RTO usare su un segmento ritrasmesso?**
Si usa l'algoritmo di backoff esponenziale
2. **Quali campioni usare in input all'algoritmo di Jacobson.**
L'algoritmo di Karn determina quali campioni di RTT usare per non inficiare la stima



Backoff esponenziale di RTO

Quando scade il timeout relativo a un segmento, il TCP sender ritrasmette il segmento stesso; secondo la specifica originale RFC793 il sender usa sempre lo stesso valore di RTO per tutte le successive ritrasmissioni del pacchetto

- Questa tecnica non è consigliabile se la scadenza del timeout è legata a uno stato di congestione di rete perché lo aggraverebbe
 - ✗ è consigliabile variare il valore di RTO delle sorgenti che sono coinvolte nella congestione per evitare ritrasmissioni contemporanee
 - ✗ è consigliabile aumentare RTO ogni volta che il TCP sender ritrasmette lo stesso segmento (backoff)



Backoff esponenziale di RTO

Una tecnica semplice per implementare il backoff è il backoff esponenziale

- Ogni volta che deve ritrasmettere un segmento, TCP moltiplica il valore di RTO precedentemente calcolato per un opportuno valore q , finché il segmento non vada a buon fine
- La sorgente TCP aumenta il valore di RTO per ogni ritrasmissione (backoff process)

$$RTO_{i+1} = q RTO_i$$

- normalmente $q=2$ (binary exponential backoff)



Backoff esponenziale di RTO

- Normalmente, il valore di RTO viene raddoppiato a ogni ritrasmissione fino al raggiungimento di un fattore moltiplicativo pari a 64, ottenuto alla settima trasmissione
- In base a quest'equazione, RTO cresce esponenzialmente ad ogni ritrasmissione
- Oltre questo valore, la connessione viene reinizializzata (con una procedura di reset): il valore di RTO torna al valore precedente solo dopo la ricezione di un riscontro relativo ad un segmento che è stato trasmesso una sola volta

Algoritmo di Karn



TCP usa l'algoritmo di Karn per scegliere i campioni di RTT da usare per la stima di SRTT

- In caso di ritrasmissione TCP non distingue se il riscontro si riferisce
 - × alla prima trasmissione del segmento
 - × alla ritrasmissione del segmento
- Un errore di attribuzione può causare
 - × timeout troppo elevato (perdita di efficienza e inutili ritardi)
 - × timeout troppo breve (ritrasmissioni eccessive nuovi errori di misura)

Algoritmo di Karn



- Nel calcolo di SRTT e SDEV, secondo l'algoritmo di Karn, **NON DEVONO** essere inseriti i campioni di RTT relativi a segmenti ritrasmessi (per i quali è ambiguo se la reply si riferisce alla prima istanza del pacchetto o all'ultima)
- Quindi TCP aggiorna SRTT e SDEV solo con riferimento ai segmenti trasmessi una sola volta
- L'unico caso in cui il TCP può prendere campioni RTT dai segmenti ritrasmessi è quando si usa l'opzione "timestamp" che rimuove l'ambiguità

Algoritmo di Karn



- Inoltre l'algoritmo di Karn stabilisce di calcolare il valore di RTO dei segmenti ritrasmessi con la procedura di exponential backoff
- Si usa il backoff esponenziale nel calcolo di RTO finché arriva un riscontro relativo a un segmento che non è stato ritrasmesso
- A questo punto si riattiva l'algoritmo di Jacobson per il calcolo di RTO



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



Proprietà autosincronizzante del TCP

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



Proprietà autosincronizzante del TCP

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)
- **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

(continua...)



Controllo di congestione

- 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 $A_b = A_r$

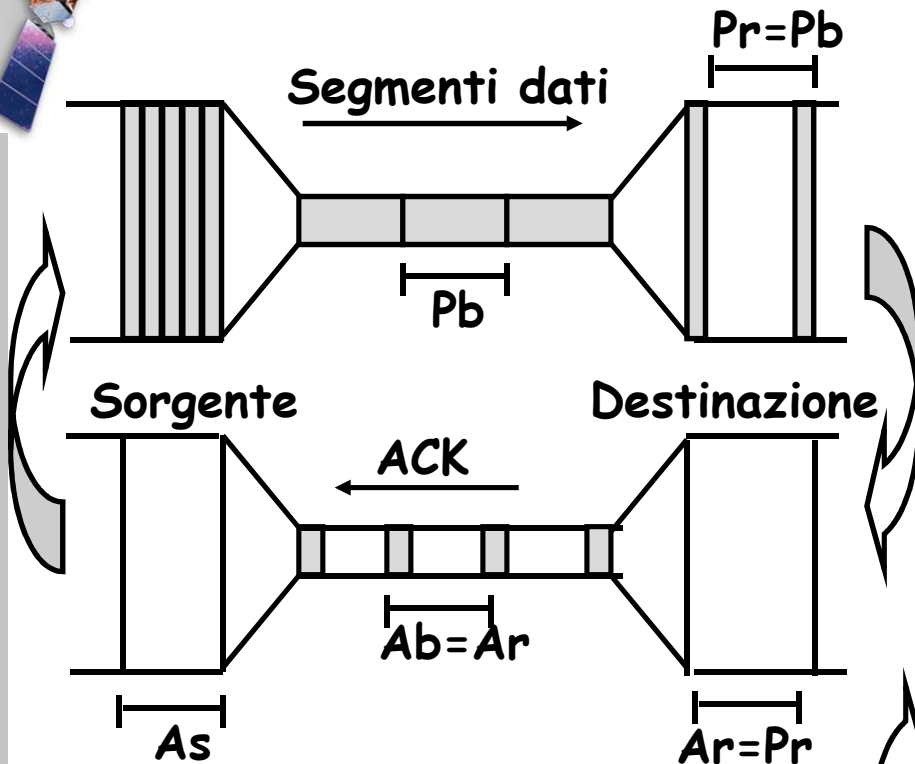


Controllo di congestione

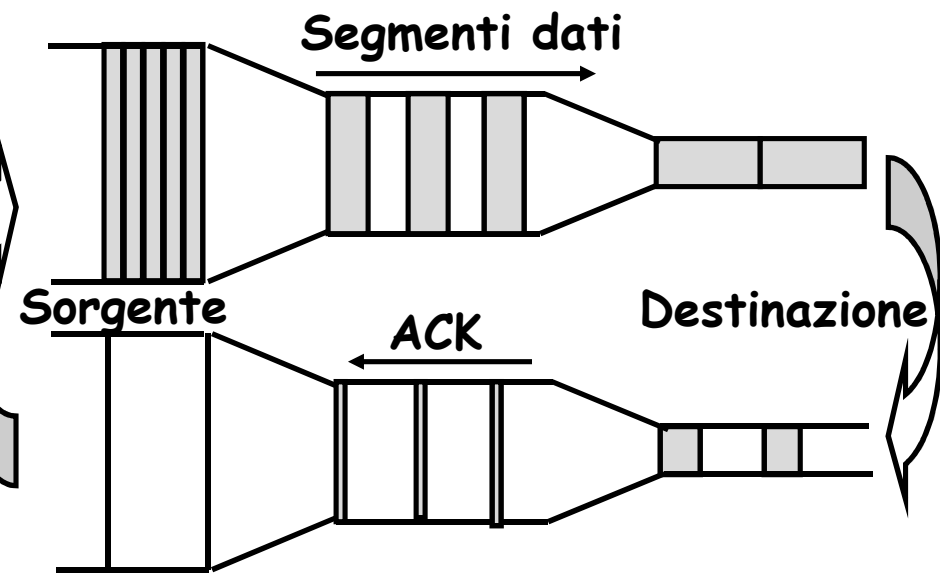
• Bottleneck nel ricevitore (vedi fig. seguente)

- 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

Controllo di flusso: self-clocking



Bottleneck all'interno della rete



Bottleneck al ricevitore



Controllo di congestione

- **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)
- **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**



Controllo di congestione

Sono stati definiti dei meccanismi aggiuntivi 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: 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



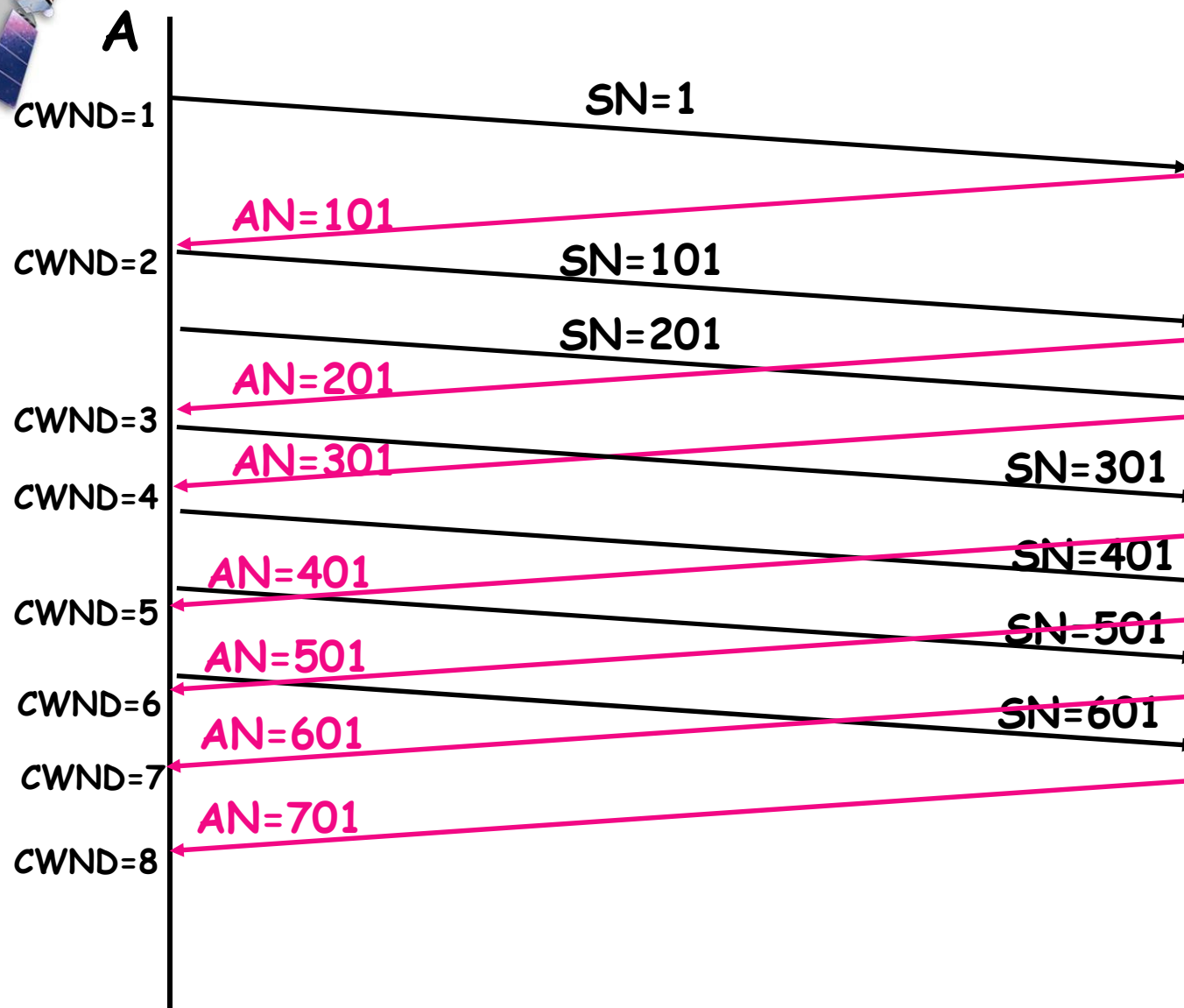
Slow Start

- L'ampiezza della finestra allowed window (awnd) usata dal sender in segmenti è:

$$\text{awnd} = \min [\text{RW}, \text{cwnd}]$$

- RW: 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 un timeout)
 - cwnd=1 (si può partire anche da 2)
 - per ogni segmento riscontrato, cwnd è incrementata di 1 segmento fino ad un valore massimo Mwnd
 - cwnd=Min [Mwnd, cwnd+1]

Slow Start: esempio



B MSS=100 byte

Cwnd cresce in maniera esponenziale!

All'arrivo del primo ACK, TCP apre cwnd a 2 e può inviare 2 segmenti

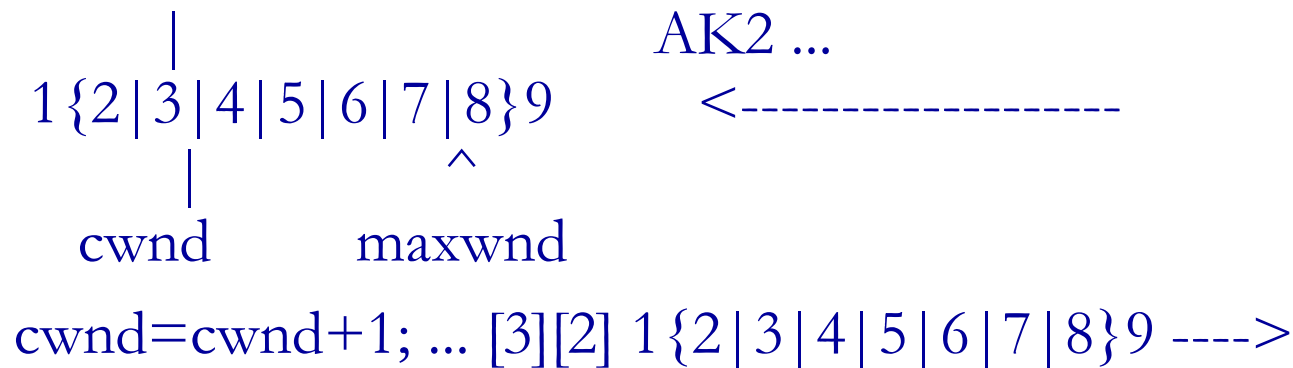
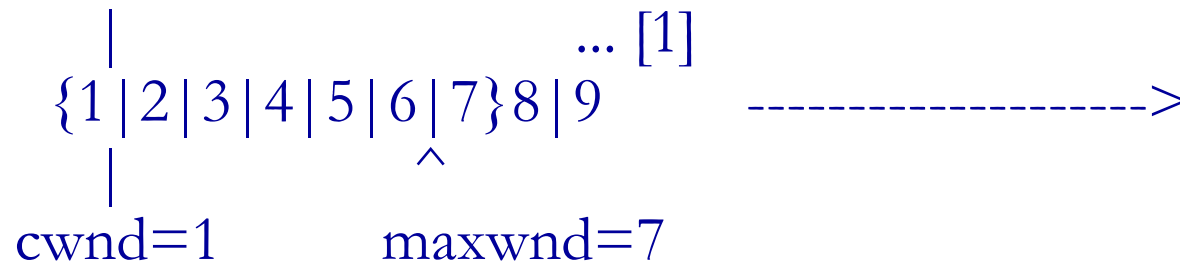
Quando i 2 segmenti sono riscontrati, TCP fa avanzare la finestra di 1 segmento per ogni ACK ricevuto

Quindi può inviare 4 segmenti

Quando riceve 4 ACK può inviare 8 segmenti, etc.

Slow Start

- Slow-start impiega $RTT * \log_2(\text{MaxWin})$ a raggiungere la massima dimensione della finestra (MaxWin)
- Se la rete è congestionata, gli ACK impiegano più tempo ad arrivare e slow-start diventa più lento



Slow Start

- In regime di slow start: al tempo 0 viene trasmesso 1 segmento di lunghezza MSS, al tempo RTT 2 segmenti MSS, al tempo 2RTT 4 segmenti MSS e così via; al tempo $k \cdot \text{RTT}$ sono spediti 2^k segmenti che corrispondono a $2^k \cdot \text{MSS}$ byte
- In un tempo pari a $N \cdot \text{RTT}$, assumendo che si mantenga valida la condizione $\text{CW} < \text{RW}$ con $\text{CW} = \text{MSS} \cdot 2^k$, il numero totale di byte trasmessi è pari a:

$$\begin{aligned}
 \sum_{k=1..N} (2^k \cdot \text{MSS}) &= \text{MSS} * \sum_{k=1..N} 2^k = \\
 &= \text{MSS} * \frac{1-2^{N+1}}{1-2} = \text{MSS} * (2^{N+1}-1) \text{ [byte]}
 \end{aligned}$$

Slow Start

- Supponiamo che $RW = m * MSS$ e che la rete non si congestioni, il regime di slow start finisce quando $CW > RW$, ovvero:

$$2^n * MSS > m * MSS$$

$$2^n > m$$

$$n > \log_2 m$$

- dove n rappresenta il numero di RTT dopo il quale il numero di byte trasmessi è costante, cioè il numero di RTT necessari affinché CW raggiunga il valore di regime; definiamo n_{inf} è il numero intero immediatamente inferiore a $\log_2 m$:

$$n_{inf} = \lfloor \log_2 m \rfloor$$

Slow Start

- Quindi, in regime di slow start il TCP trasmette un numero di byte pari a:

$$MSS * (2^{n_{inf}+1}-1) = MSS * (2^{\lfloor \log_2 m \rfloor +1}-1)$$

- Per $n > n_{inf}$ ad ogni RTT saranno trasmessi $m * MSS$ byte, per cui in un tempo pari a $h * RTT$ verranno trasmessi $h * m * MSS$ byte
- In generale il numero di byte trasmessi dal TCP in $L * RTT$ secondi, nel caso in cui $RW = m * MSS$, è dato da:

$$\text{se } L \leq n_{inf} \Rightarrow MSS * (2^{L+1}-1)$$

$$\text{se } L > n_{inf} \Rightarrow MSS * (2^{\lfloor \log_2 m \rfloor +1} -1) + (L - n_{inf}) * m * MSS$$

Slow Start

- Nelle condizioni in cui il time-out non scatti mai e il ricevitore dia sempre la massima RW disponibile, il tempo impiegato per trasmettere un file di B byte è $L \cdot RTT$, con L intero:
- se $B \leq MSS \cdot (2^{n_{inf}+1} - 1)$, cioè B è più piccolo del numero di byte che si possono trasmettere in regime di slow start:

$$B = MSS \cdot (2^{L+1} - 1) \quad \Rightarrow \quad L = \lceil \log_2(1 + B/MSS) - 1 \rceil$$

- se $B > MSS \cdot (2^{n_{inf}+1} - 1)$, cioè la trasmissione avverrà parte in slow start parte in regime, in regime il numero di byte da trasmettere è:

$$B - MSS \cdot (2^{\lfloor \log_2 m \rfloor + 1} - 1) = (L - n_{inf}) \cdot m \cdot MSS$$

$$L = \lfloor \log_2 m \rfloor + \left\lceil \frac{B - MSS \cdot (2^{\lfloor \log_2 m \rfloor + 1} - 1)}{m \cdot MSS} \right\rceil$$



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 $\min(RW, cwnd)$
- 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

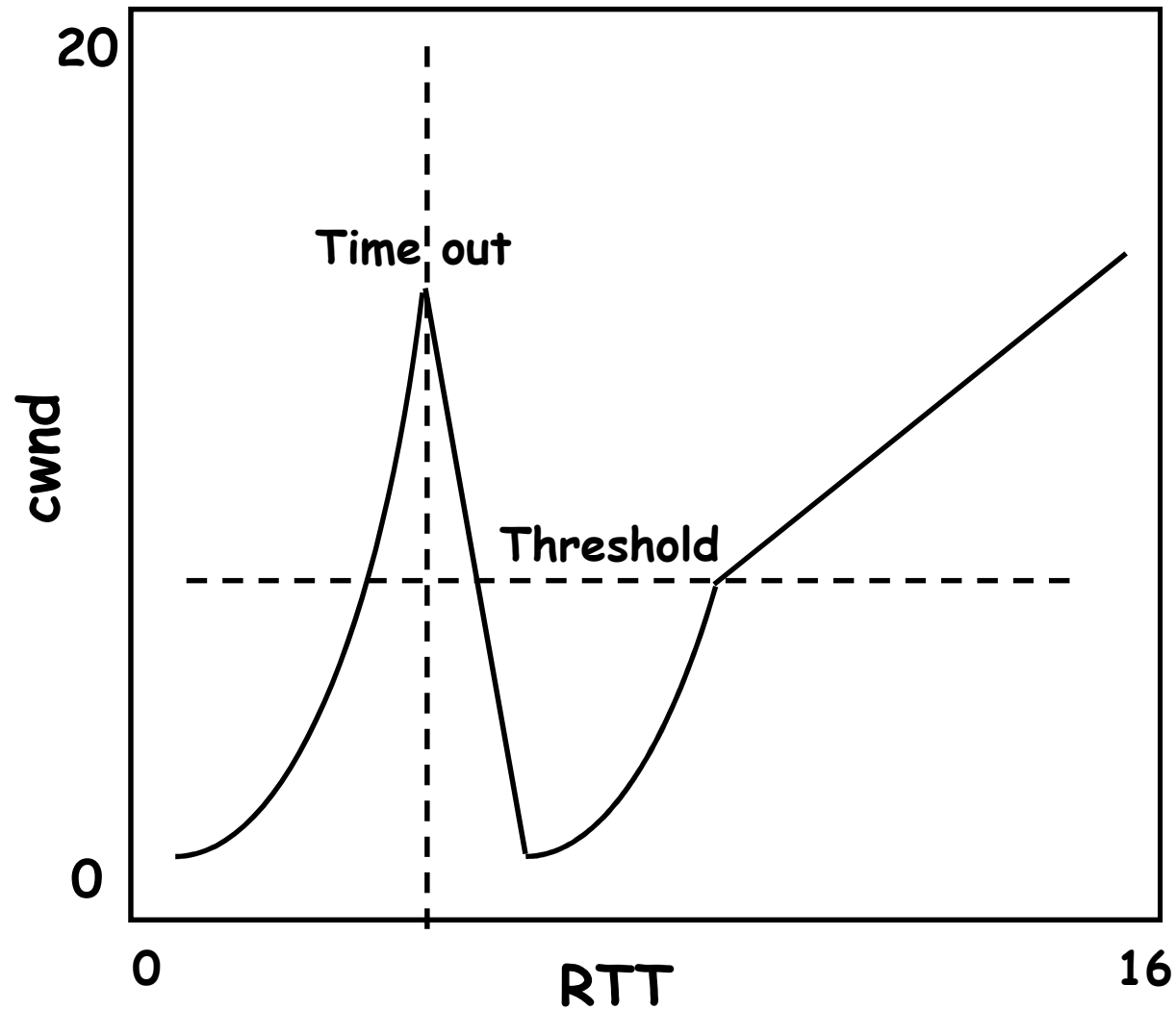
$$ssthresh = cwnd / 2$$

(in effetti, $ssthresh = \min(cwnd/2, RW)$)

- 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)

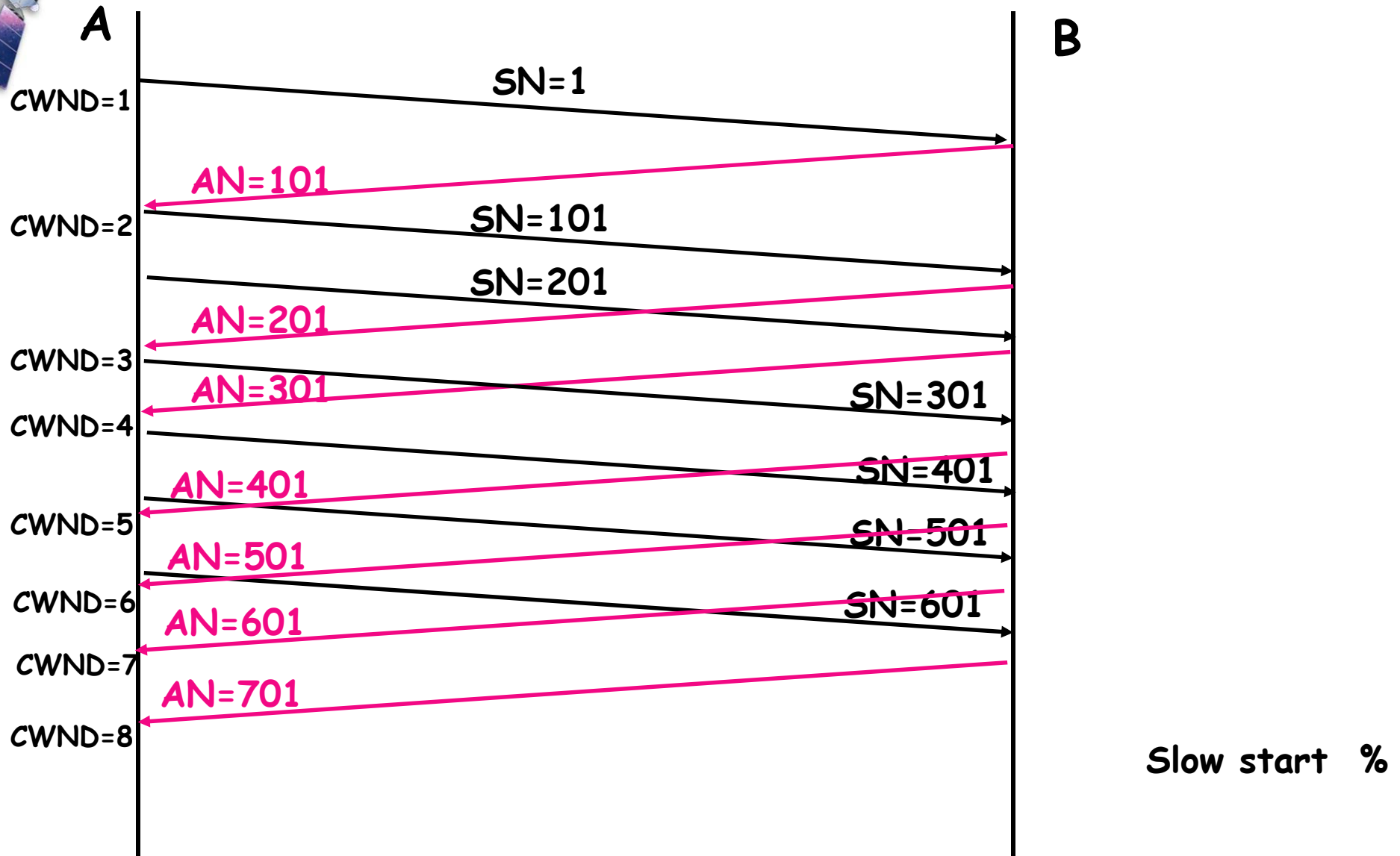


Congestion Avoidance



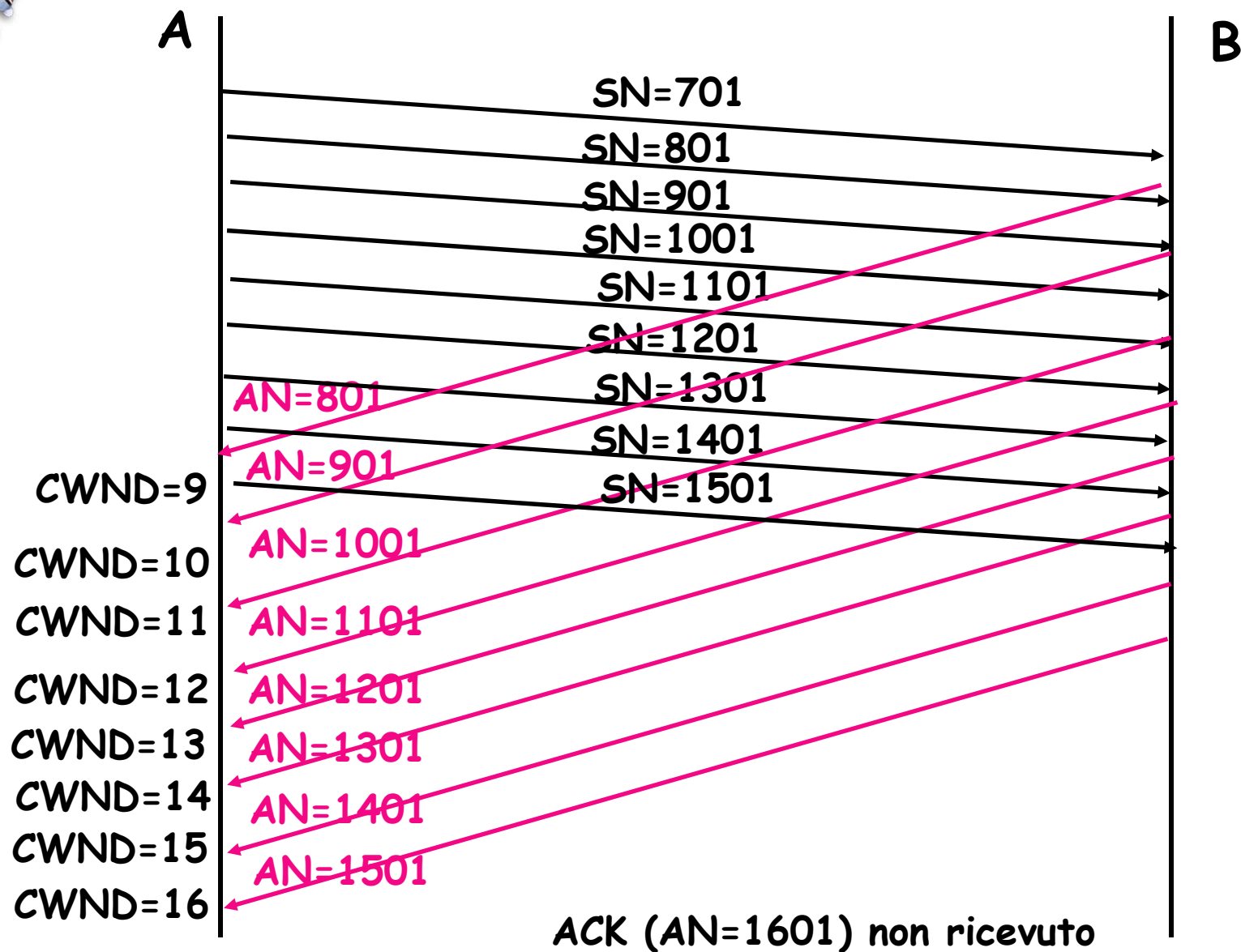


Congestion Avoidance: esempio



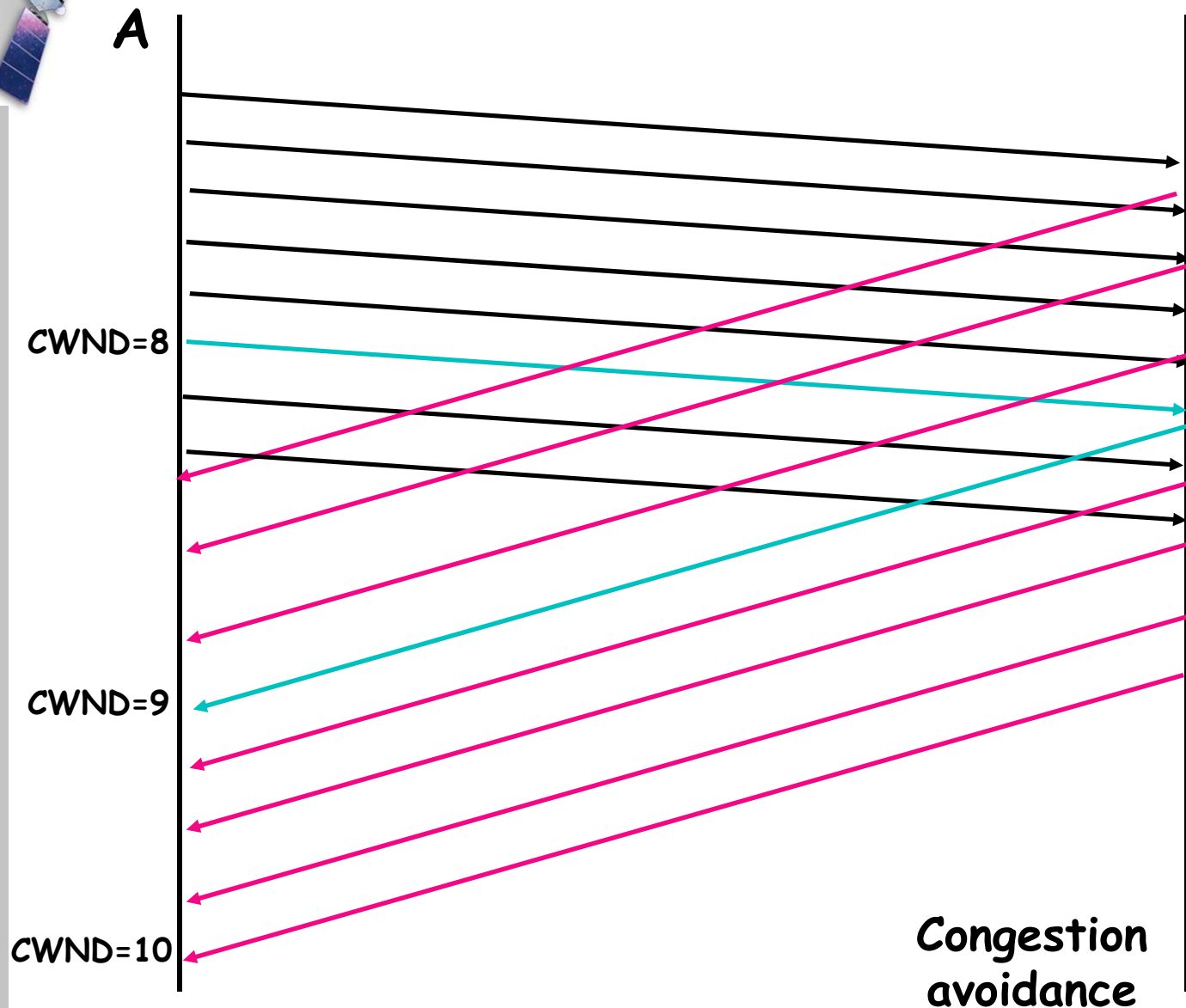


Congestion Avoidance: esempio





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:

$$cwnd = ssthresh + 3$$

Fast Recovery



- 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