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

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
- 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:

$$SRTT[i] = (1 - \alpha) SRTT[i-1] + \alpha RTT[i]$$

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

$$RTO = \beta SRTT = 2 SRTT$$

- Scegliere $\beta=2$, un valore costante, non risponde ai cambiamenti della varianza
- In realtà, SRTT oscilla in maniera random attorno al valor medio con una deviazione standard $\alpha SDEV(RTT)$
- Primo problema: 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: 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 $(1-\rho)$). 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 β 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:

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

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

$$MDEV(x) = E [|X - E[X]|]$$

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

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

Algoritmo di Jacobson

- J. propose di calcolare RTO come:

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

- 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 secondi 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, $SRTT(K+1)$:

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

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

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

- quindi:

$$SRTT(k+1) = SRTT(k) + \alpha SERR(k+1)$$

- 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 $RTO(K+1)$

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

- $\alpha=0.125$ (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

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 $RTT=0$, $RTO=3$ secondi (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

$$\begin{aligned} SRTT &\leftarrow R \\ RTTVAR &\leftarrow R/2 \\ RTO &\leftarrow SRTT + \max(G, K \cdot RTTVAR) \end{aligned}$$

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 , finchè 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 ritrasmissione
- 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 flusso

- Il controllo di flusso è una procedura attuata in modo coordinato tra due entità TCP, sorgente e destinatario, intesa a limitare il flusso dei dati trasmessi, in funzione delle risorse a disposizione “nei sistemi terminali” e prescindendo dal traffico presente in rete
 - × tale meccanismo è indispensabile in Internet dove calcolatori di dimensione e capacità di calcolo diverse comunicano tra loro; il più lento dei due deve poter rallentare l'emissione di informazione dell'altro
- Lo scopo del controllo di flusso è di evitare che un mittente invii dei segmenti ad un destinatario che, “in quel momento”, non è in grado di riceverli, a causa di indisponibilità di risorse sia di elaborazione che di memorizzazione

Controllo di flusso

- Il controllo di flusso nel TCP è implementato mediante lo stesso meccanismo usato per il controllo di errore, “a finestra scorrevole” (**sliding window**) di tipo “credit allocation”
 - l'ampiezza della finestra è variabile
 - il meccanismo è “orientato al byte”: la finestra rappresenta, istante per istante, il numero massimo di byte che possono essere trasmessi verso il destinatario
- Lo schema è basato sul campo **Window** (16 bit) in cui il destinatario scrive la larghezza (in byte) della finestra di trasmissione W che il mittente dovrà usare dal quel momento in poi
- Il mittente userà tale valore W finché non riceve dal destinatario un successivo segmento con una dimensione di finestra diversa

Controllo di flusso

- Un riscontro ($\text{ACK Number}=x$ e $\text{Window}=W$) significa che:
 - × sono riscontrati tutti gli ottetti ricevuti fino a quello numerato con $X-1$
 - × il trasmittente è autorizzato a trasmettere, senza attendere altri riscontri, fino a ulteriori W ottetti, ovvero fino all'ottetto numerato con $x+W-1$
- Solitamente la RW ha un'ampiezza pari a $n \cdot \text{MSS}$, dove n è un numero intero per evitare che il TCP trasmittente esegua una segmentazione poco efficiente
 - se la finestra di ricezione è pari a 301 byte e la MSS 100 byte, il TCP trasmittente sarebbe indotto a formare segmenti da 1 byte, a cui aggiungere l'header del TCP e di IP (tipicamente di 20 byte ciascuno). Se a ciò si aggiunge l'header di livello 2 (8 byte nel caso di PPP), per inviare un'informazione di 1 byte è necessario inviarne 49

Controllo di flusso

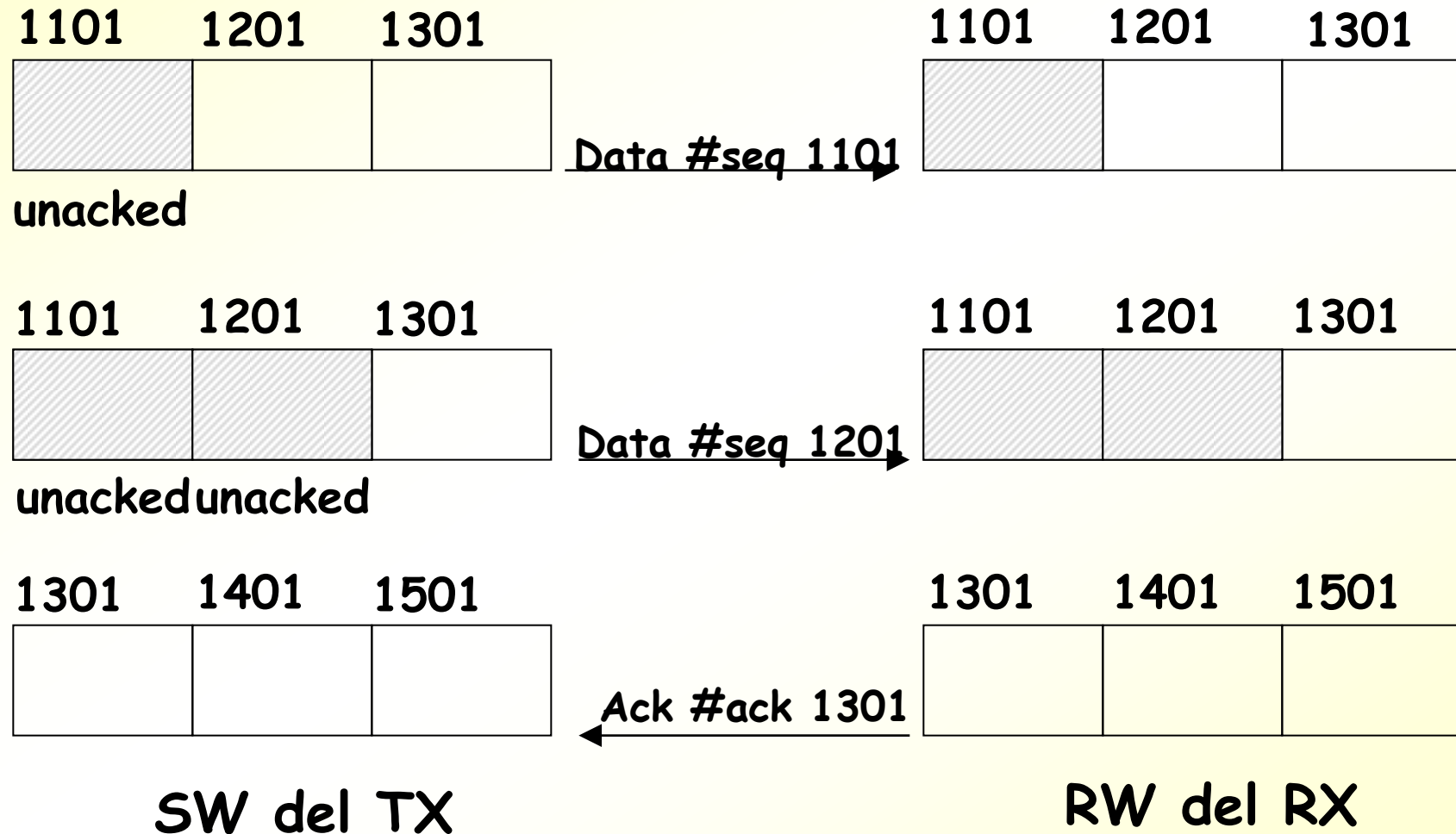
- La finestra comunicata dal ricevitore, **Advertised Window**, rappresenta dunque anche la quantità di dati che l'entità destinataria è disposta a ricevere (funzione della capacità di calcolo e della memoria in ricezione)
- Il ricevitore può variare "dinamicamente" la dimensione della finestra in funzione delle sue esigenze e limitare il ritmo di trasmissione qualora non sia in grado di gestirlo
- Quindi il ricevitore informa il trasmettitore, avendo presente però che quest'ultimo la modificherà solo dopo aver ricevuto dati, corretti ed in sequenza, che abbiano "riempito" le finestre precedentemente offerte

Controllo di flusso: esempio 1

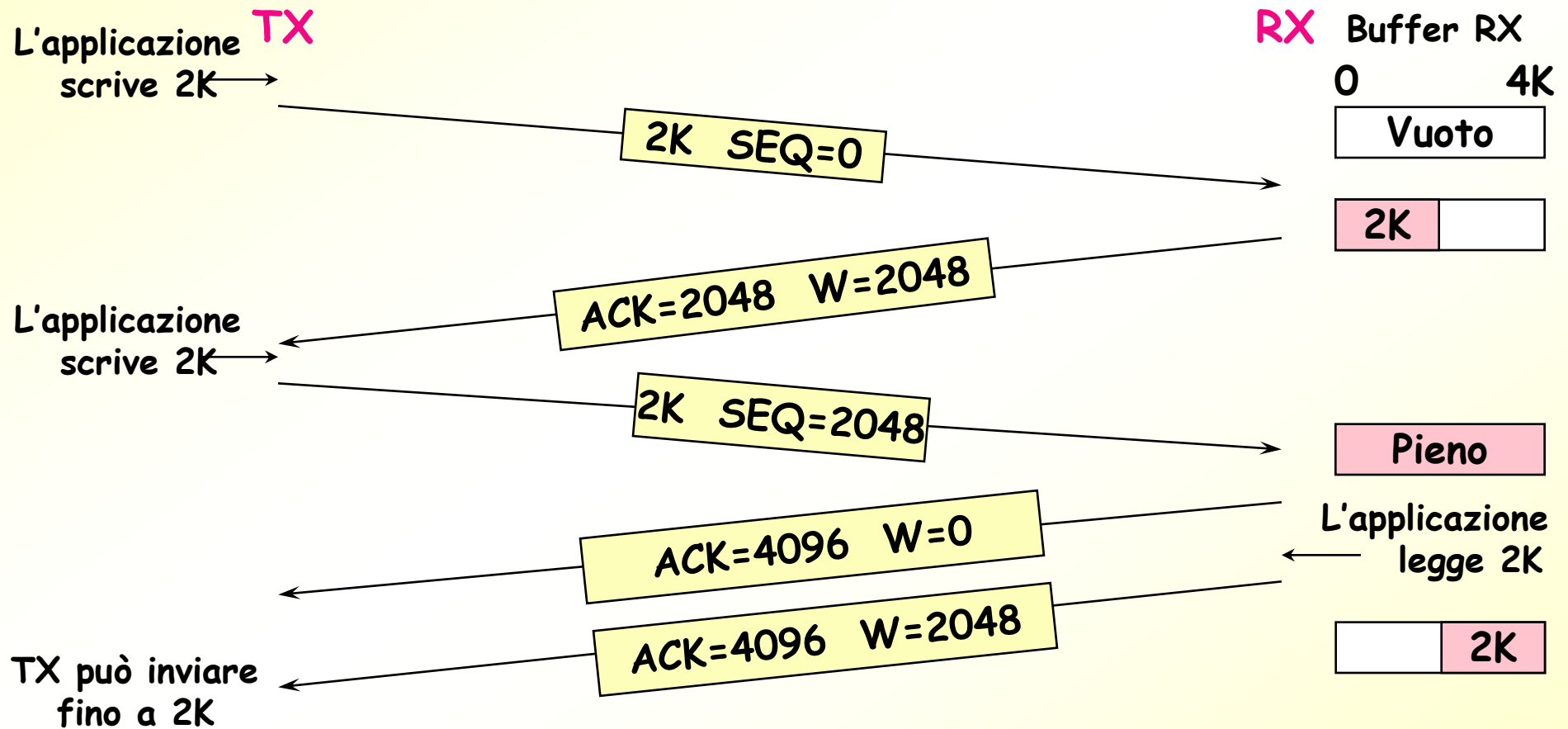
Esempio

- Si abbia una MSS di 100 byte e una finestra $RW=3MSS=300$ byte. In un certo istante, il ricevitore accetta i byte da 1101 a 1400.
- Il trasmettitore invia i byte dal 1101 al 1200 che non vengono mandati subito all'applicazione (magari perché il TCP è uno dei tanti processi concorrenti che gira su un PC o su una workstation, tra i quali il S.O. cicla); allora la RW effettiva si è ridotta a 200 byte.
- Il trasmettitore invia un altro segmento che contiene i byte dal 1201 al 1300; il TCP ricevente invia tutti i byte al livello superiore e svuota la sua finestra, che torna alla sua dimensione originale di 300 byte, accettando i byte da 1301 a 1600.

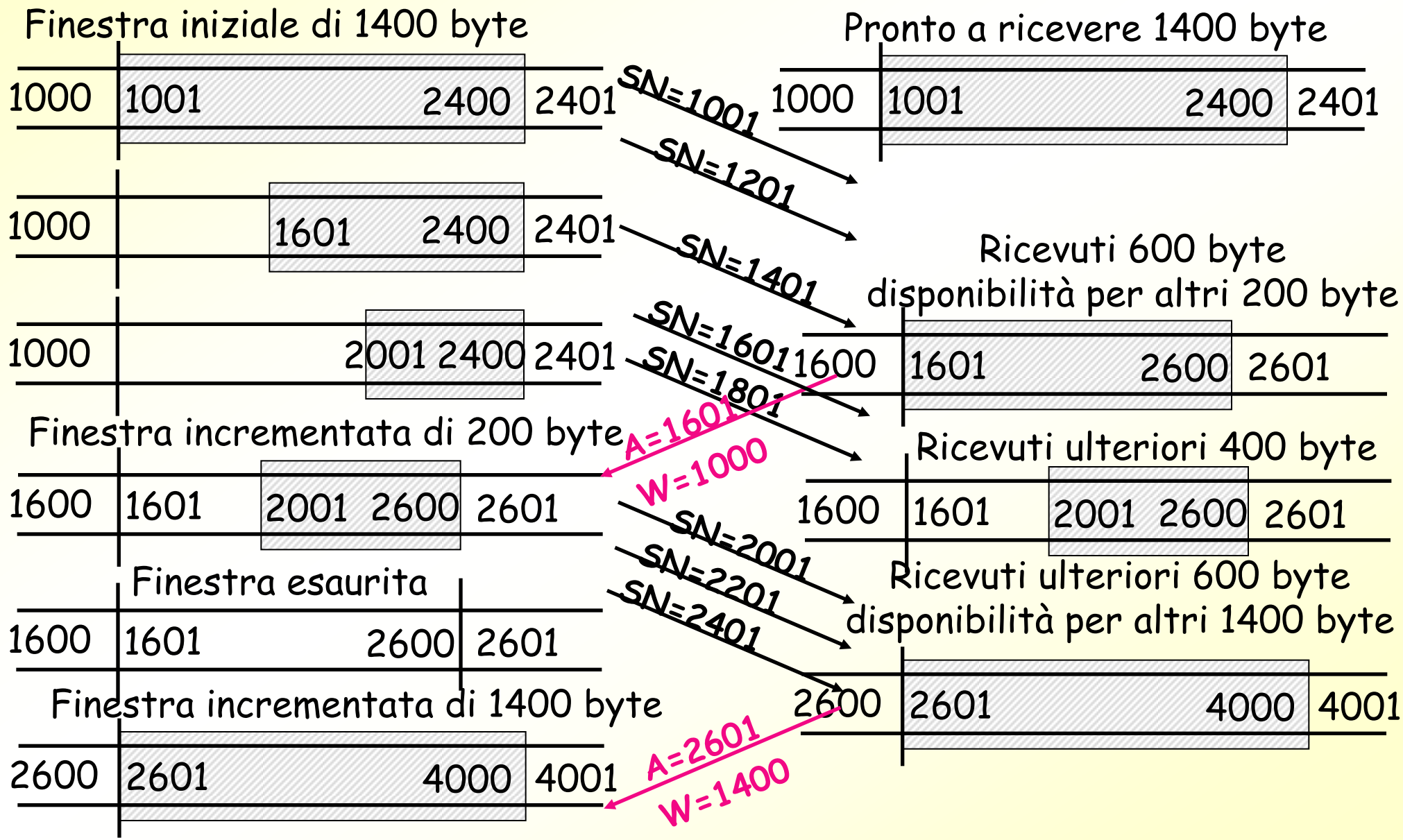
Controllo di flusso: esempio 1



Controllo di flusso: esempio 2

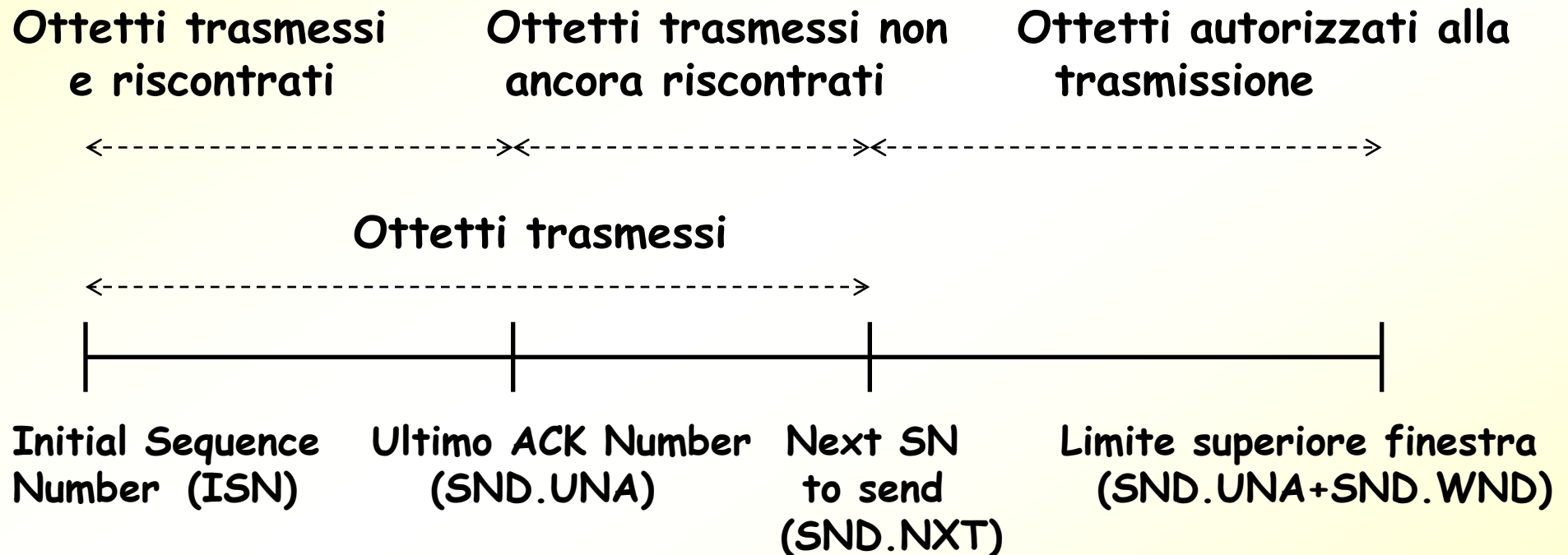


Controllo di flusso: esempio 3



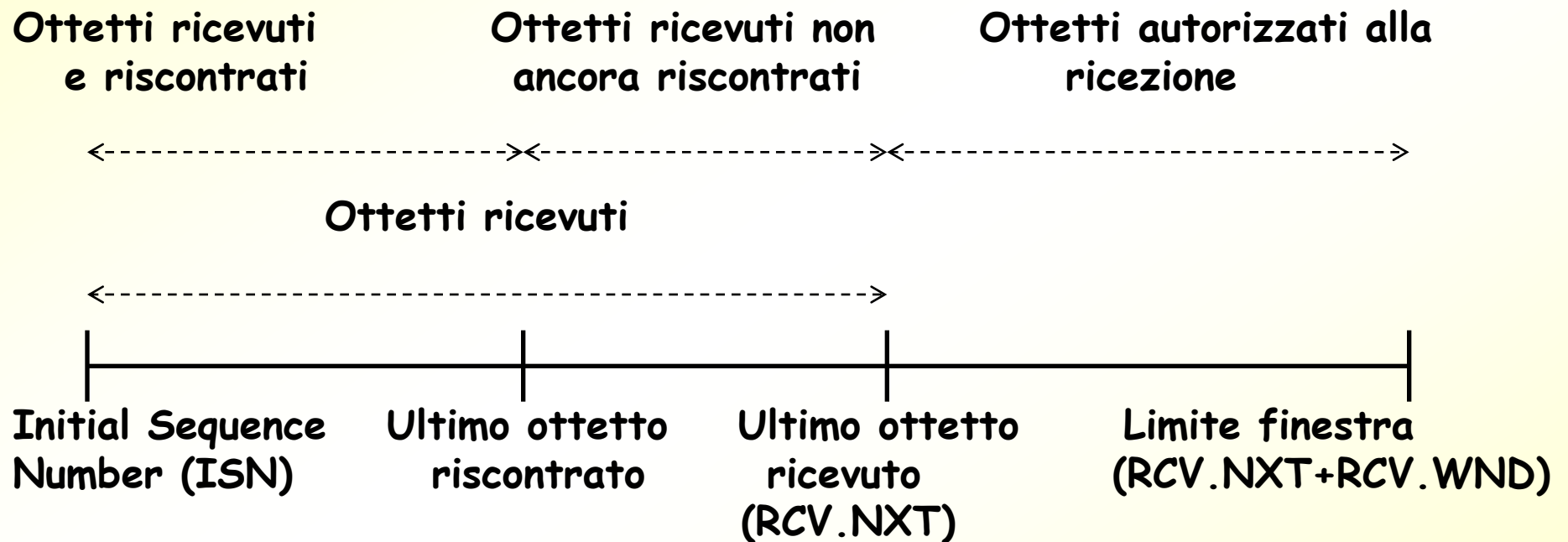
Controllo di flusso

- Puntatori del controllo a finestra lato emittente



Controllo di flusso

- Puntatori del controllo a finestra lato ricevente

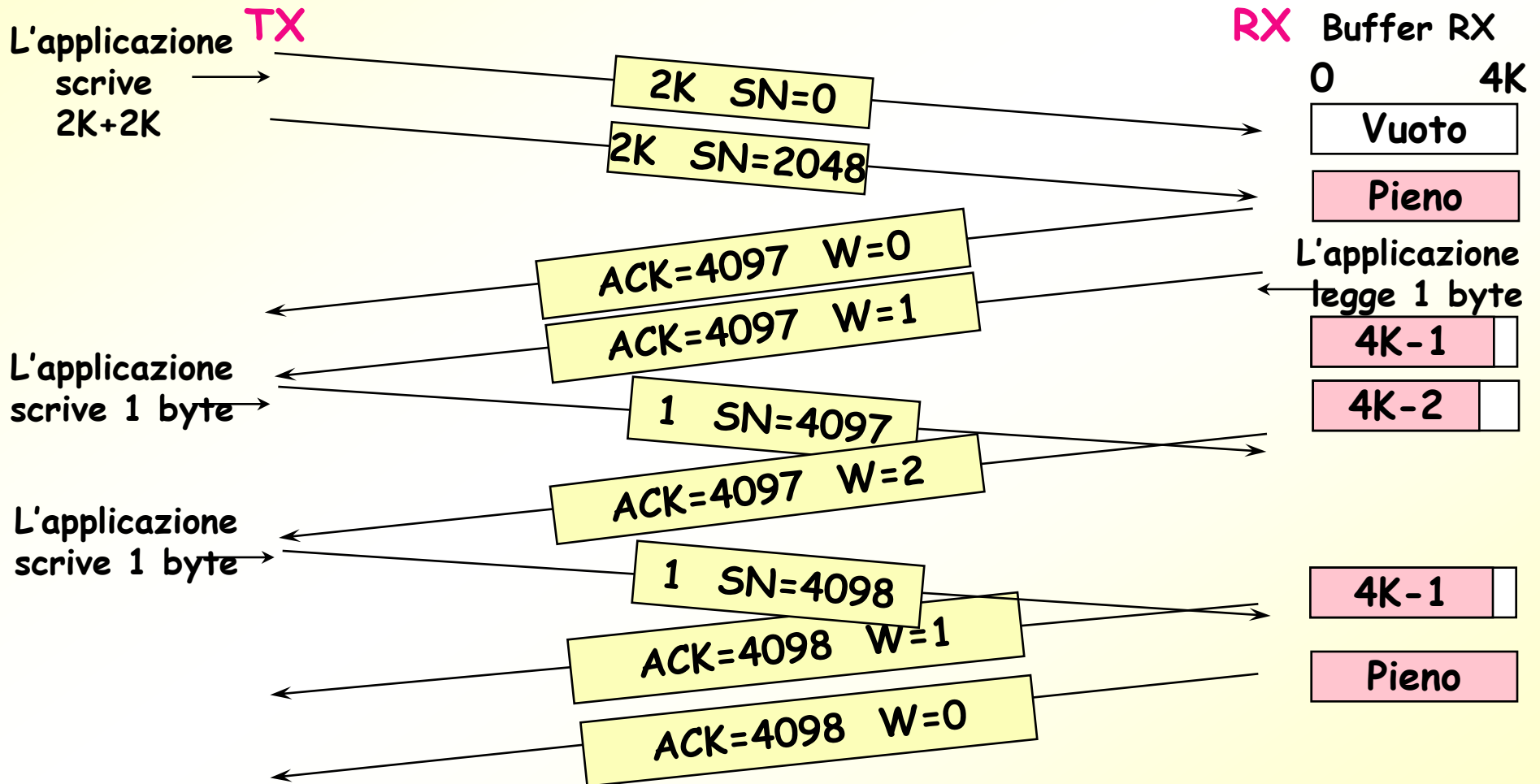


Sindrome della silly window

- Nelle prime implementazioni del TCP si era riscontrato un problema, noto come "Silly window syndrome", che si riscontrava quando l'applicazione sorgente passa i dati al suo TCP a grossi blocchi mentre l'applicazione ricevente assorbe i byte lentamente 1 alla volta
- In questo caso il ricevitore vedeva ogni volta un byte libero nel buffer e con l'invio dell'ACK e del campo window sollecitava la trasmissione di un segmento con un solo byte. Naturalmente una quantità di segmenti con un solo byte porta ad uno spreco di risorse a causa dell'elevatissimo overhead

Sindrome della silly window

Esempio:



Sindrome della silly window

- Per superare il problema della "silly window syndrome" (Clark 1982) si sono introdotti i seguenti meccanismi:
- **lato ricevitore**: il ricevitore non deve inviare "window update" per piccole variazioni della finestra di ricezione, ovvero: il ricevitore dovrebbe aspettare di avere a disposizione almeno 1 MSS o la metà della finestra di ricezione prima di inviare un window update
 - × quindi il ricevitore mente indicando un buffer pieno fino a che il buffer non si è svuotato per metà o per una quantità almeno pari a 1 MSS
- **lato trasmettitore**: il sender deve evitare di emettere segmenti troppo piccoli, ovvero tenta di creare segmenti non più piccoli di $1/2$ MSS (se non sollecitato con primitive a fare il PUSH dei dati)

Controllo di flusso: throughput

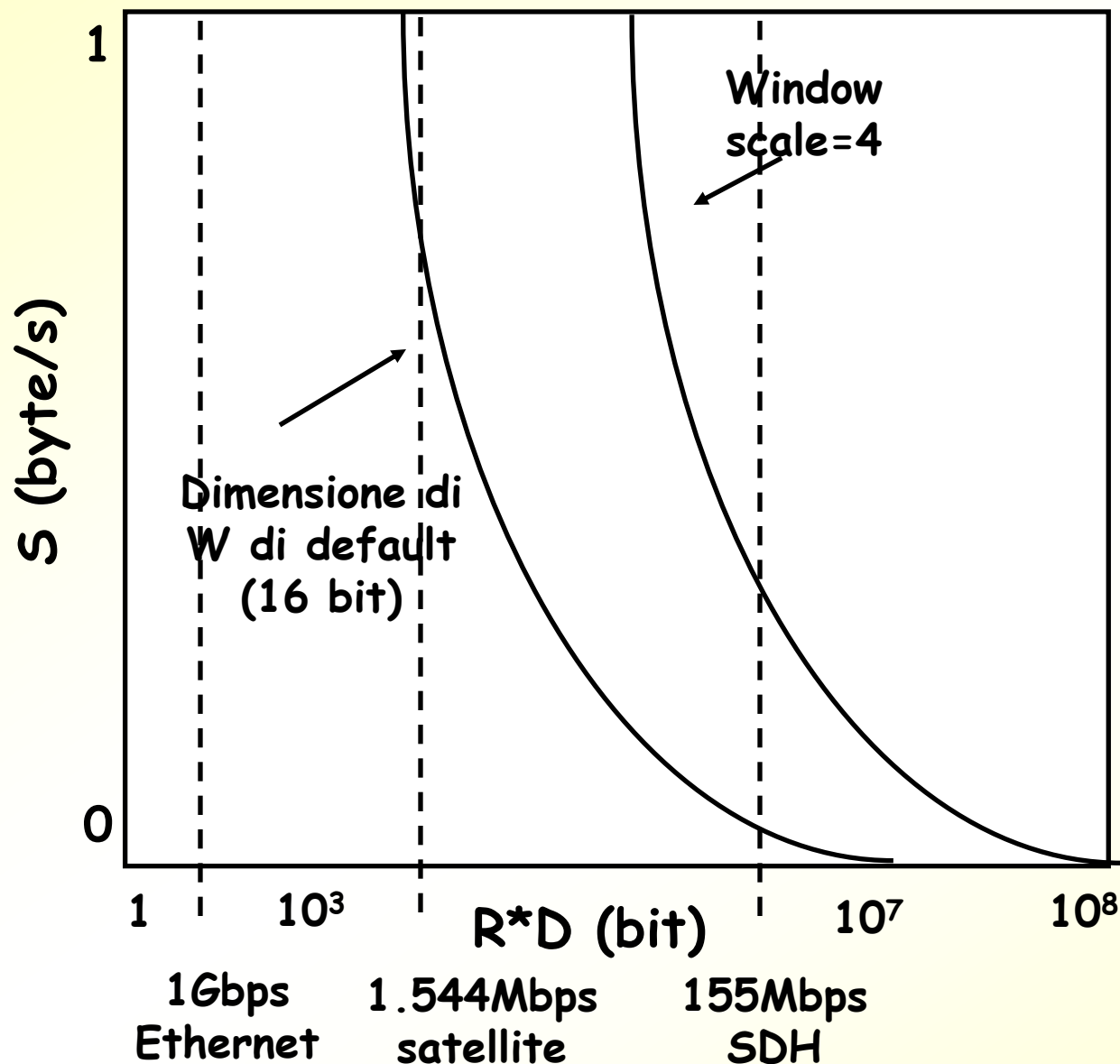
- Il throughput (S) (byte/s) di una connessione TCP dipende da:
 - dimensione della finestra TCP in byte (W)
 - ritardo di propagazione (D) in sec tra sorgente e destinazione TCP
 - bit rate (R) in bps della sorgente TCP

$$S = \begin{cases} 1 & \text{se } W > RD/4 \\ 4W/RD & \text{se } W < RD/4 \end{cases}$$

Controllo di flusso: throughput

- Nel calcolo del throughput (S) ignoriamo l'overhead del segmento TCP
 - una sorgente TCP emetta una sequenza di byte verso una destinazione
 - il primo byte impiegherà un tempo D per arrivare a destinazione, più un tempo addizionale D perché il riscontro torni alla sorgente
 - durante questo tempo ($2D$) la sorgente potrebbe trasmettere $2D \cdot R$ bit, ovvero $RD/4$ byte
 - in realtà, però, la sorgente è limitata dalla dimensione W della finestra TCP finché non riceve almeno un riscontro
 - quindi se $W > RD/4$ (il riscontro arriva prima che la finestra si chiuda) si ottiene il throughput max sulla connessione, se invece $W < RD/4$ (il sender si ferma dopo W in attesa del riscontro per riaprire la finestra) il max throughput normalizzato è dato dal rapporto W su $RD/4$

Controllo di flusso: throughput



- La max dimensione di W è $2^{16}-1=65535$ byte
- RD nel caso Ethernet da 1Gbps con estensione 100m è $<10^3$ bit; nel caso di link satellitare a 1.544Mbps è $<10^6$
- In entrambi i casi la dimensione di finestra di default va bene
- Nel caso link SDH a 155Mbps tra 2 punti distanti, RD è $<10^7$ e la dimensione classica di W non va bene, allora si usa un fattore di scala che permette di allargare la finestra a $2^{20}-1=10^6$ byte

Controllo di flusso: Window size

- La dimensione della finestra di 64KB può essere un problema per le linee con High Bandwidth o High Delay o High Bandwidth Delay Product (DR)
 - Es. connessioni satellitari
- Con alti ritardi di propagazione e elevate velocità di trasmissione, la finestra si svuota velocemente, ma il sender non può inviare altri dati finché non riceve un riscontro

Soluzioni:

- RFC 1323 propone un "Window scale option" che permette di negoziare un fattore di scala che estende la dimensione di Window di altri 16 bit (fino a 2^{32} byte)
- RFC 1106 propone l'uso del Selective Repeat al posto del Go-back N; introducendo i NACK per chiedere la ritrasmissione di un segmento specifico

Controllo di flusso: Window size
