



3 – Il Livello di Trasporto

Antonio Capone, Matteo Cesana,
Ilario Filippini, Guido Maier

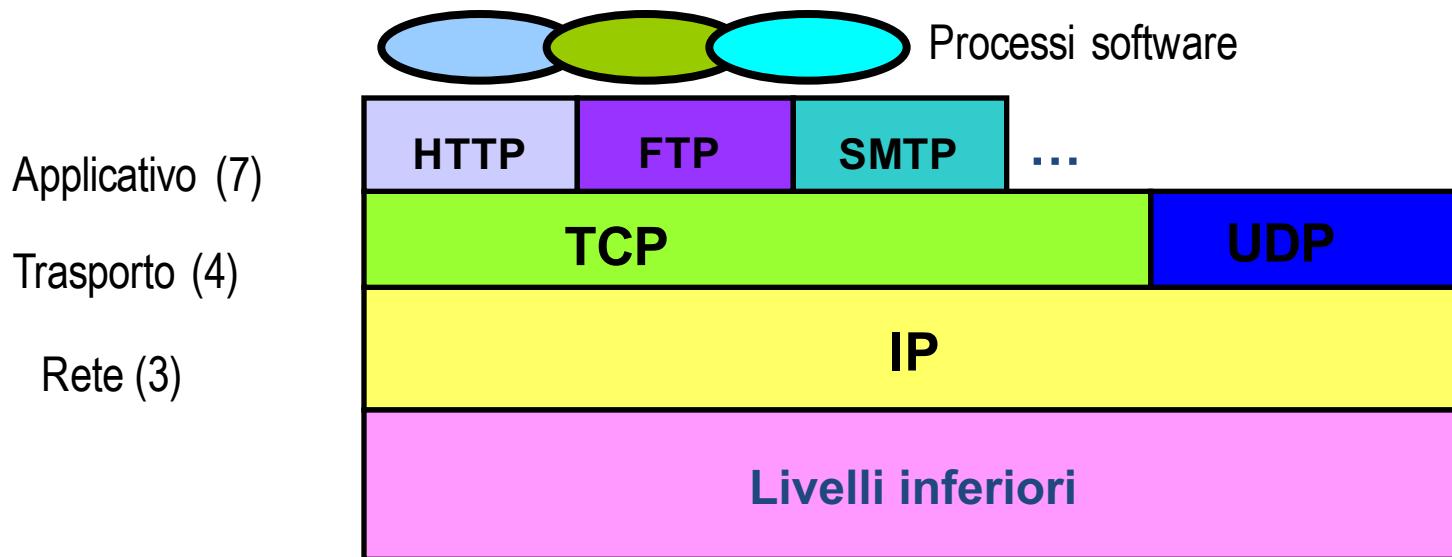
Livello di Trasporto

- **Introduzione**
- **Protocollo UDP**
- **Trasporto affidabile**
 - Protocolli di ritrasmissione
 - Controllo di flusso a finestra mobile
- **Protocollo TCP**
 - Generalità
 - Formato e connessioni
 - Controllo di flusso
 - Controllo d'errore
 - Controllo di congestione



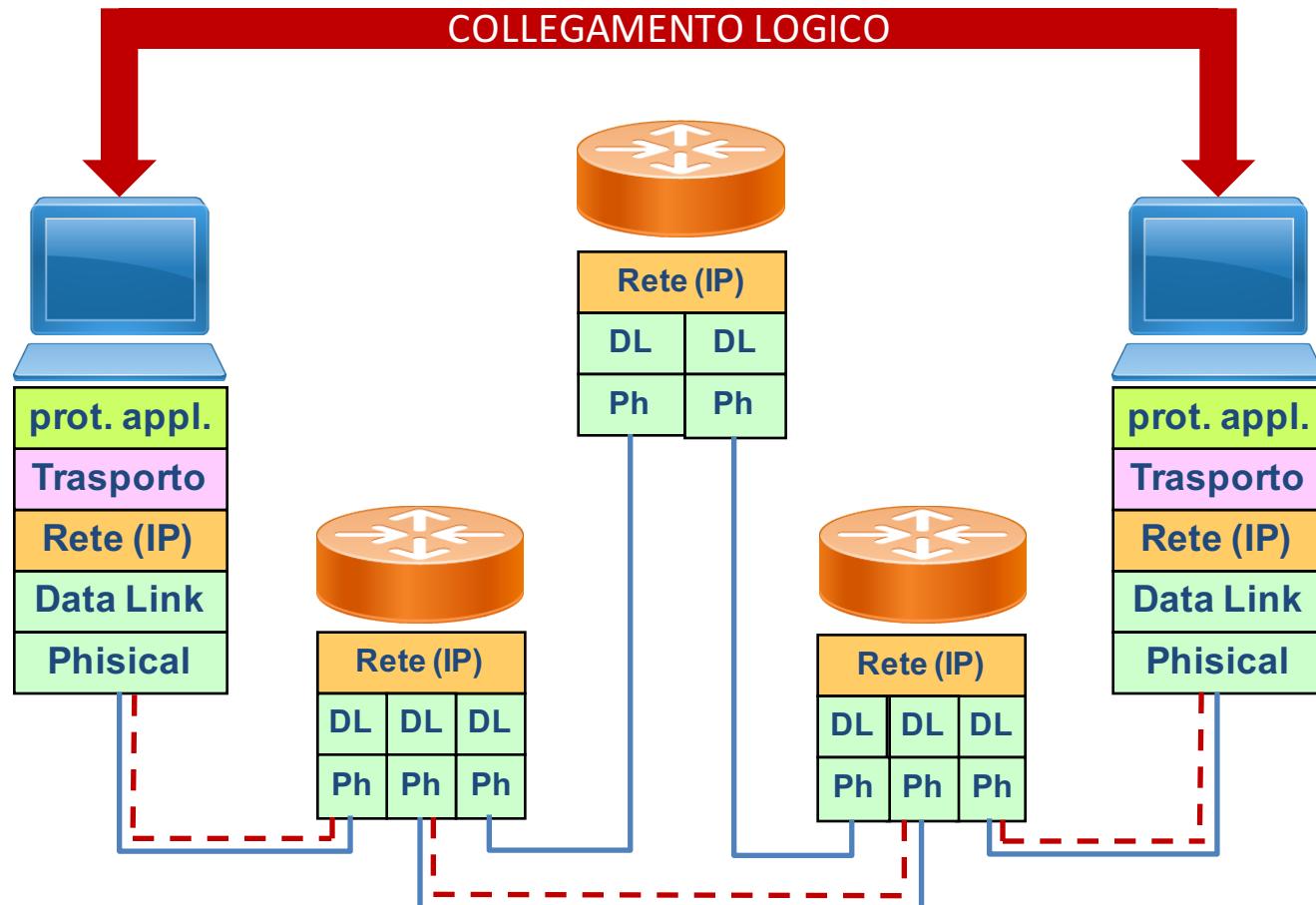
Servizio di trasporto

- Il livello di trasporto ha il compito di instaurare un collegamento logico tra le applicazioni residenti su *host* remoti
- Il livello di trasporto rende trasparente il trasporto fisico (attraverso la rete) dei messaggi alle applicazioni



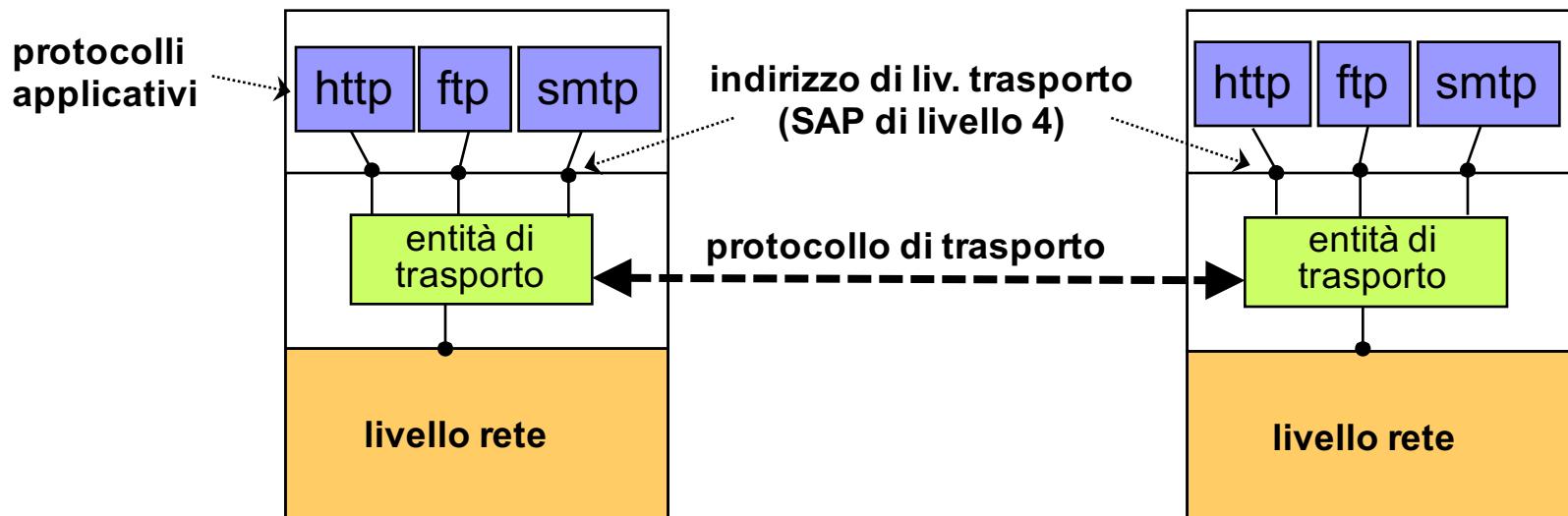
Servizio di trasporto

- Il livello di trasporto è presente solo negli *end systems (host)*
- Esso consente il collegamento logico tra processi applicativi



Servizio di trasporto

- Più applicazioni possono essere attive su un *end system*
 - Il livello di trasporto svolge funzioni di *multiplexing/demultiplexing* per applicazioni
 - Ciascun collegamento logico tra applicazioni è indirizzato dal livello di trasporto



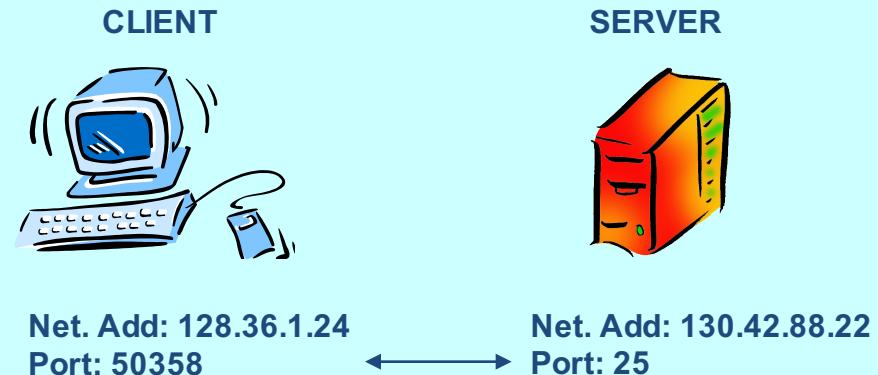
Indirizzamento: le *porte*

- In Internet le funzioni di *multiplexing/demultiplexing* vengono gestite mediante indirizzi contenuti nelle PDU di livello di trasporto
- Tali indirizzi sono lunghi 16 bit e prendono il nome di *porte*
- I numeri di porta possono assumere valori compresi tra 0 e 65535
- I **numeri noti** sono assegnati ad importanti applicativi dal lato *server* (HTTP, FTP, SMTP, DNS, ecc.)
- I **numeri dinamici** sono assegnati dinamicamente ai processi applicativi lato *client*
- I **numeri registrati** sono assegnati a specifiche applicazioni da chi ne faccia richiesta (tipicamente protocolli proprietari)

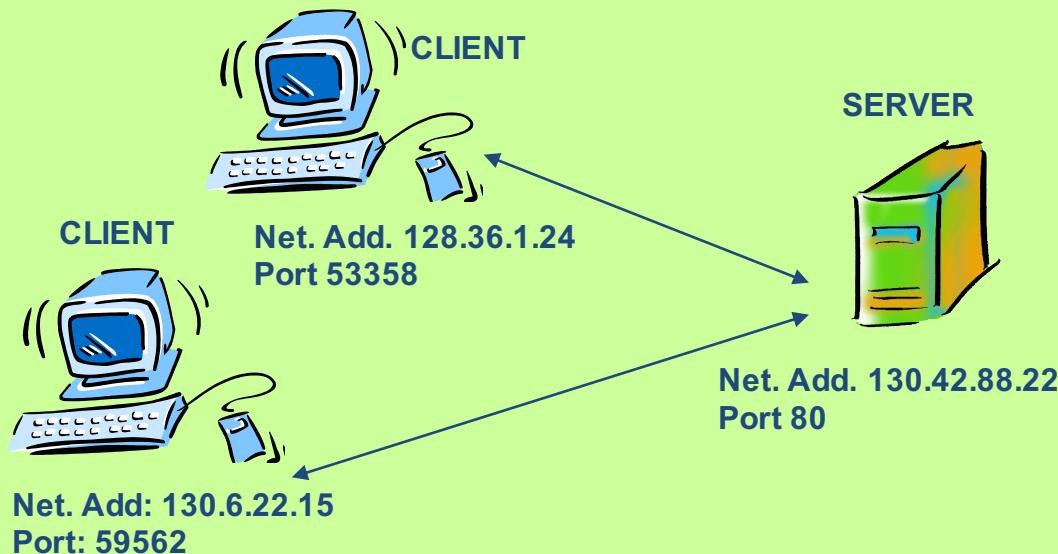


Socket e multiplazione

Un *client* trasmette segmenti verso la porta di un server SMTP remoto

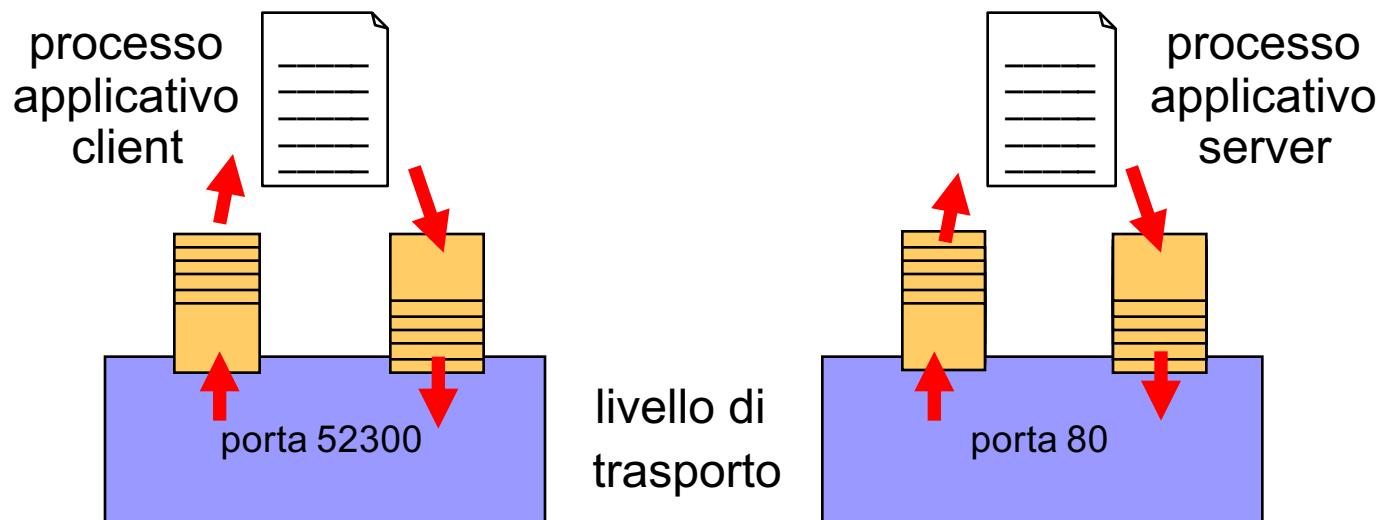


Due client accedono alla stessa porta di un server HTTP.
Non c'è comunque ambiguità,
perché la coppia di *socket* è
diversa



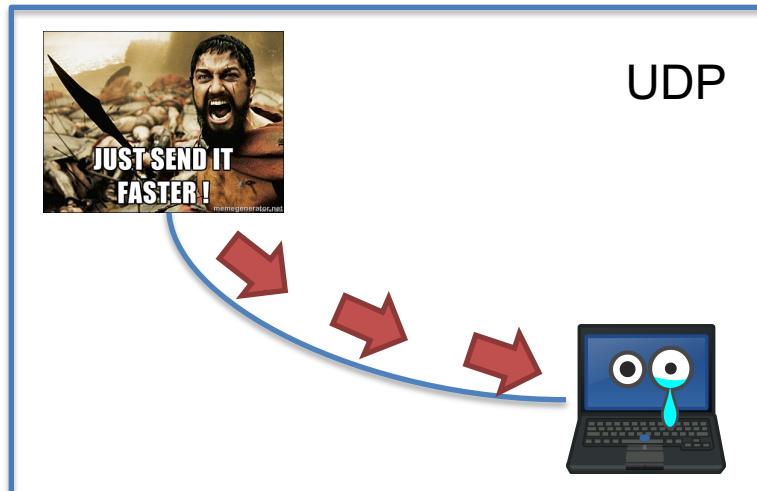
Servizio di Buffering

- I protocolli di trasporto sono implementati nei più diffusi **sistemi operativi**
- Quando un processo viene associato ad una porta (lato *client* o lato *server*) viene associato dal sistema operativo a due code, una d'ingresso e una d'uscita
- Funzionalità di *buffering* dei dati



Servizio di trasporto

- Il servizio di rete è non affidabile
 - Fa del proprio meglio per consegnare i singoli messaggi indipendentemente a destinazione
- Il servizio di trasporto fornito può essere di vari tipi
 - Trasporto affidabile (garanzia di consegna dei messaggi nel corretto ordine)
 - Trasporto non affidabile (solo funzionalità di multiplexing)
 - Trasporto orientato alla connessione
 - Trasporto senza connessione
- Nella suite IP sono definiti due tipi di trasporto
 - **TCP (Transmission Control Protocol)**, orientato alla connessione e affidabile
 - **UDP (User Datagram Protocol)**, senza connessione e non affidabile



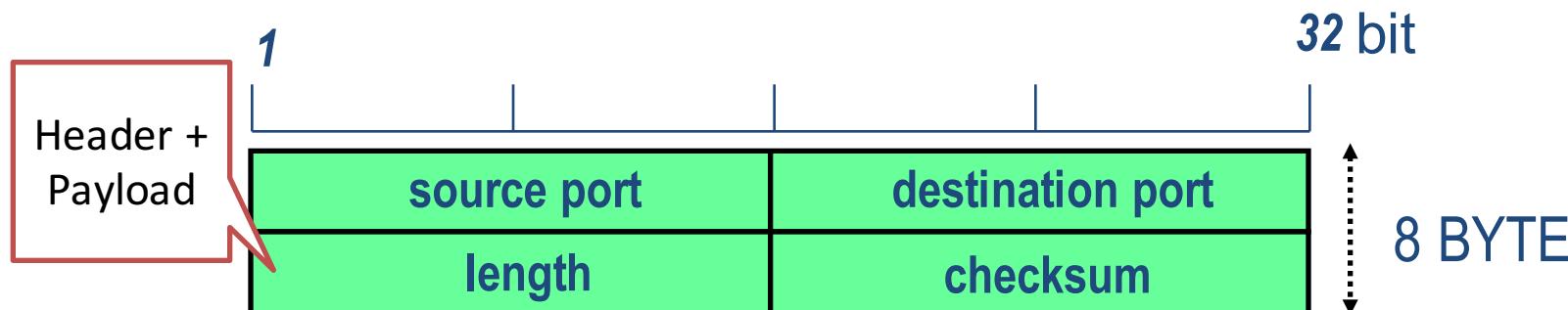
Livello di Trasporto

- Introduzione
- **Protocollo UDP**
- **Trasporto affidabile**
 - Protocolli di ritrasmissione
 - Meccanismo a finestra mobile
- **Protocollo TCP**
 - Generalità
 - Formato e connessioni
 - Controllo di flusso
 - Controllo d'errore
 - Controllo di congestione



User Datagram Protocol (UDP) – RFC 768

- E' il modo più semplice di usare le funzionalità di IP
- Non aggiunge nulla a IP se non:
 - Indirizzamento delle applicazioni (*mux/demux*)
 - Blando controllo d'errore sull'*header* (senza correzione)
- ... e quindi
 - E' un protocollo *datagram*
 - Non garantisce la consegna
 - Non esercita nessun controllo (né di flusso, né di errore)



Perchè usare UDP e non TCP?

- **Minore latenza**
 - Non occorre stabilire una connessione
- **Maggiore semplicità**
 - Non occorre tenere traccia dello stato della connessione
 - Poche regole da implementare
- **Minore overhead**
 - *Header UDP è minore dell'header TCP*

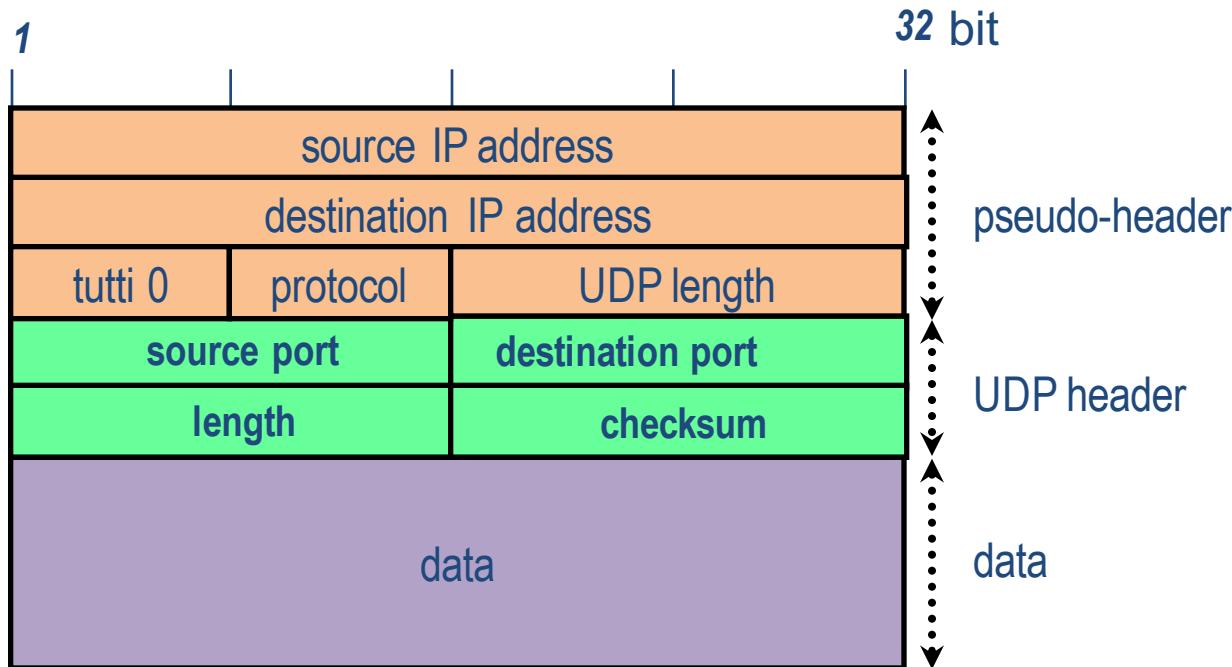


Non è possibile visualizzare questa immagine.



Il campo *Checksum*: controllo di integrità

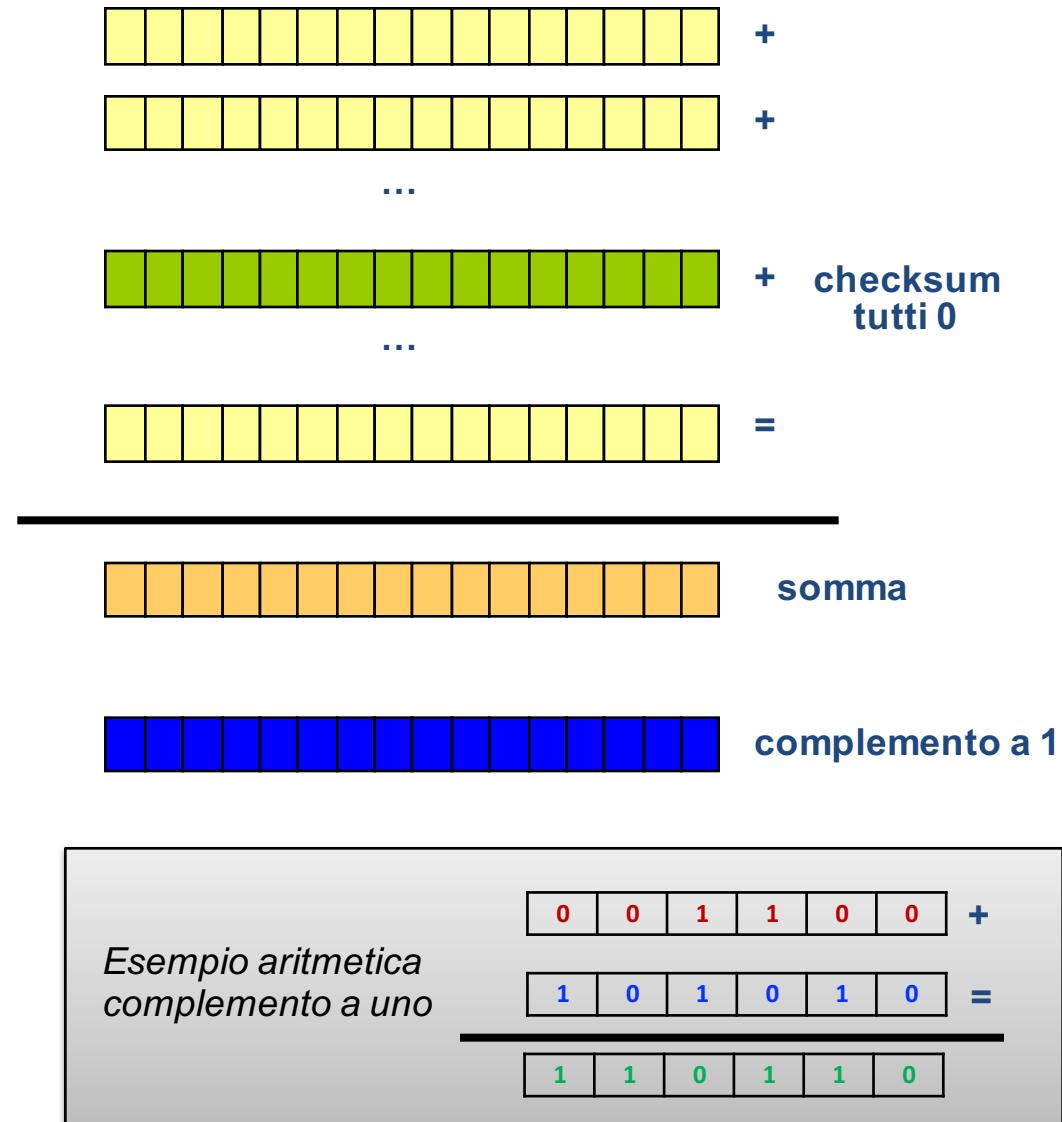
- Informazione ridondante inserita nell'*header* del segmento UDP per controllo d'errore
- Il campo di *checksum* (16 bit) è calcolato dal trasmettitore ed inserito nell'*header*
- Il ricevitore ripete lo stesso calcolo sul segmento ricevuto (comprensivo di *checksum*)
- Se il risultato è corretto accetta il segmento, altrimenti lo scarta
- Viene calcolato considerando l'*header* UDP, uno *pseudo-header* IP ed i *dati*



Calcolo del Checksum

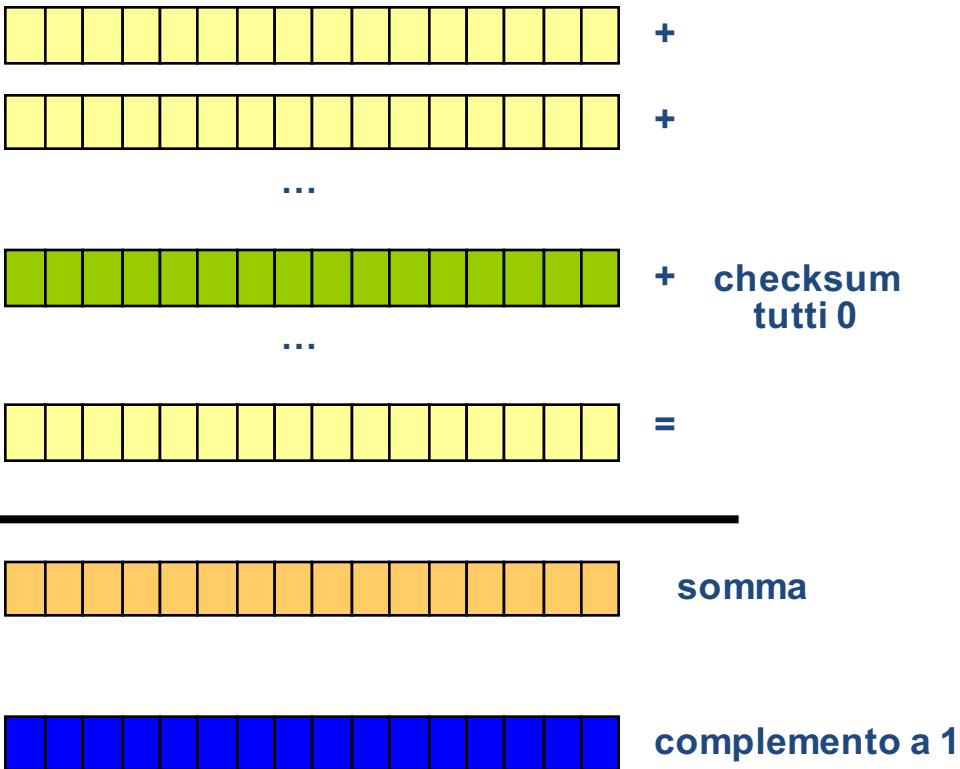
lato trasmettitore

- L'insieme di bit è diviso in blocchi da 16 bit
- Il campo *Checksum* è inizializzato a 0
- Tutti i blocchi vengono sommati in aritmetica complemento a uno
- Il risultato è complementato ed inserito nel campo di *checksum* del segmento inviato



Calcolo del Checksum *lato ricevitore*

- L'insieme di bit è diviso ancora in blocchi da 16 bit
- Tutti i blocchi vengono sommati in aritmetica complemento a uno
- Il risultato è complementato
 - Se sono tutti 0 il pacchetto è accettato
 - Altrimenti è scartato



Wireshark: Protocollo UDP

- File cattura : http-ethereal-trace-5
- Attività:
 - Scegliere un pacchetto UDP ed esaminare i campi dell'*header* UDP
 - Quanto sono lunghi?
 - Quale è il numero di porta massimo?
 - Quale è la lunghezza massima possibile per il payload UDP?
 - Guardando nell'*header* IP, verificare il contenuto del campo *protocol*
 - Scegliere una coppia di pacchetti “Domanda-Risposta”, verificare la relazione tra i numeri di porta



Livello di Trasporto

- Introduzione
- Protocollo UDP
- **Trasporto affidabile**
 - Protocolli di ritrasmissione
 - Controllo di flusso a finestra mobile
- **Protocollo TCP**
 - Generalità
 - Formato e connessioni
 - Controllo di flusso
 - Controllo d'errore
 - Controllo di congestione



Collegamento ideale

- Collegamento ideale
 - Tutto ciò che viene trasmesso arriva nello stesso ordine e viene correttamente interpretato a destinazione
- Es., come essere sicuri che una sequenza di ordini impartiti sia stata compresa con certezza?
- E' possibile oppure è uno sforzo velleitario?
- E' necessario esserne assolutamente certi?



Recupero d'errore

- Ingredienti di un **Protocollo di Ritrasmissione**
 - Ciascuna trama ricevuta correttamente viene riscontrata positivamente con un messaggio di (*Acknowledgment* o ACK)
 - A volte l'errore può essere segnalato da un messaggio detto di NACK (Not ACK)
 - La mancanza di ACK o la presenza di NACK segnala la necessità di ritrasmettere
 - La procedura si ripete finché la trama viene ricevuta corretta
- Necessita di canale di ritorno e di messaggi di servizio (ACK, NACK)
- Nota: anche i messaggi di servizio possono essere corrotti da errori!



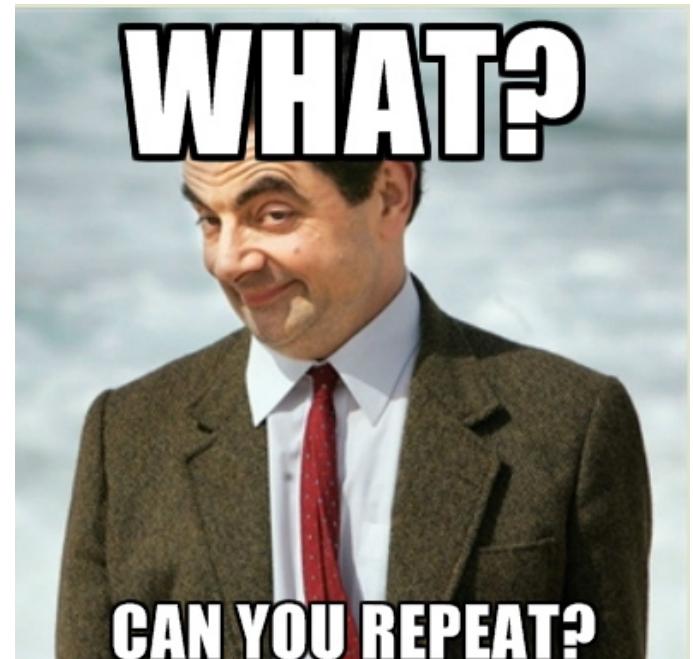
Controllo d'integrità e recupero d'errore

- Queste procedure possono essere attivate a qualunque livello
- Storicamente sono state sempre presenti a livello di linea sui collegamenti fisici a causa delle cattive linee fisiche del passato
- Presente a livello di trasporto per recupero *end-to-end*
 - Proteggere i collegamenti fisici non basta, i pacchetti possono andare persi nei buffer dei router
- Nei sistemi moderni il recupero d'errore a livello di linea può essere assente



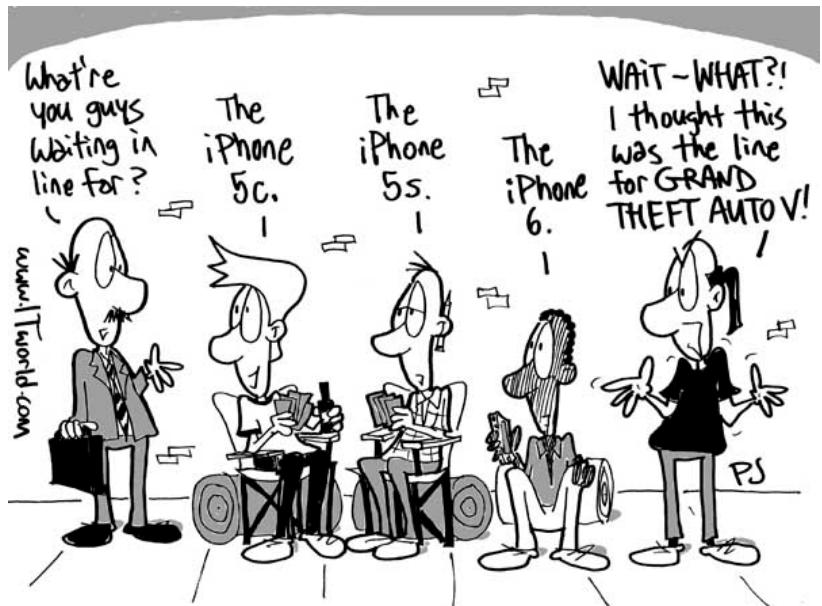
Protocolli di ritrasmissione

- Obiettivo:
 - integrità del messaggio/pacchetto
 - ordine della sequenza di pacchetti
 - no duplicazione
- Usando i messaggi:
 - ACK: riscontro positivo
 - NACK: riscontro negativo
- e meccanismi come:
 - *Timeout*
 - Finestra di trasmissione



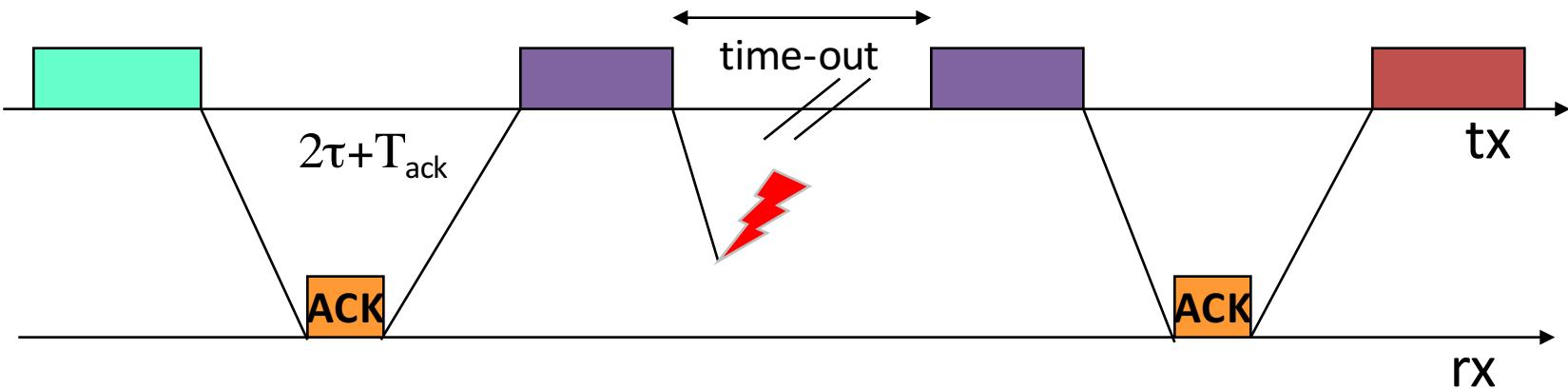
Protocollo Stop and Wait

- Utilizza il solo ACK
- e un contatore di *time out*
- Ogni messaggio ricevuto correttamente è riscontrato dal ricevitore (ACK)



Protocollo Stop and Wait

- Il trasmettitore trasmette un pacchetto e inizializza un contatore (*time-out*)
- Se il primo evento successivo è
 - l'ACK, trasmette il pacchetto successivo
 - il time-out, ritrasmette il pacchetto corrente



Funzionamento corretto se **time-out $\geq 2\tau + T_{ack}$**



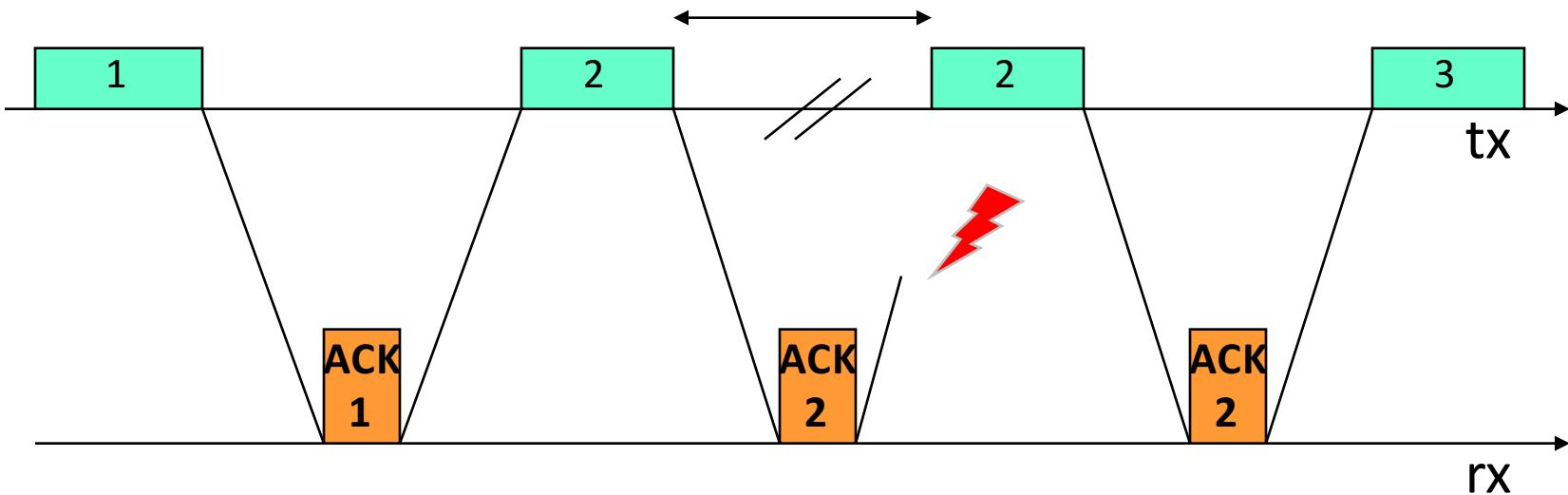
Protocollo Stop and Wait

- Numerazione dei pacchetti (SN) e degli ACK (RN)
 - Se un pacchetto viene erroneamente trasmesso più volte
 - Il ricevitore può riconoscere la duplicazione perché i pacchetti hanno lo stesso SN
 - Quando il trasmettitore riceve un ACK deve poterlo assegnare al pacchetto corretto
 - Confrontando RN può assegnarlo al pacchetto con il rispettivo SN



Protocollo Stop and Wait

- Ma se devo aspettare ACK prima di trasmettere il pacchetto successivo, come posso avere duplicati?
- Ecco cosa succede se ACK va perso e scade il *timeout*:



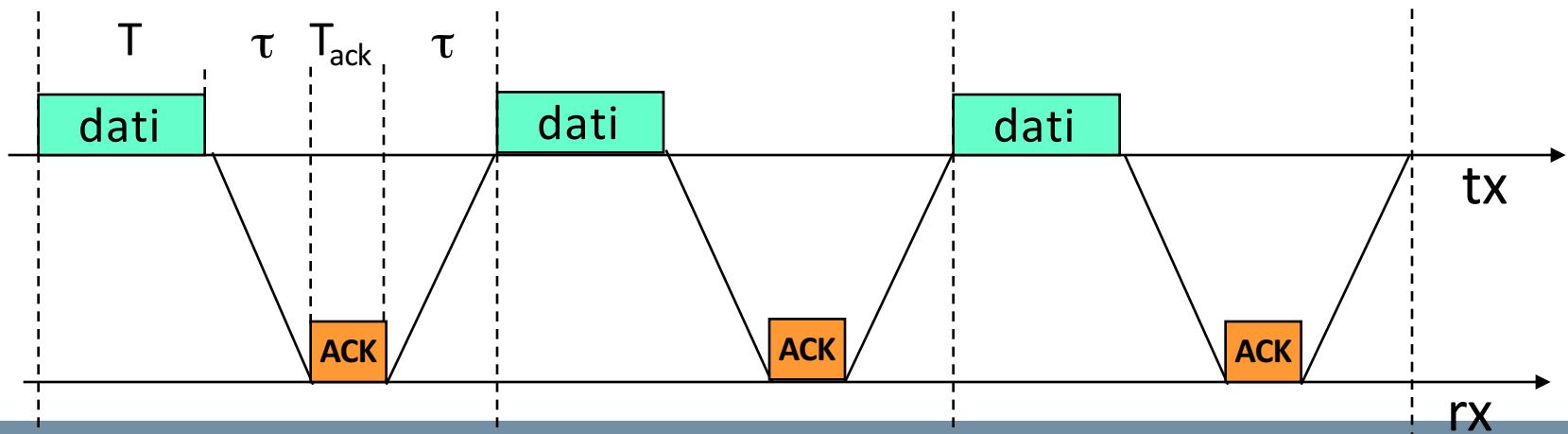
Protocollo Stop and Wait

- Efficienza del protocollo:
 - frazione di tempo in cui il canale è usato per trasmettere informazione utile in assenza di errori

$$\eta = \frac{T}{T + T_{ack} + 2\tau}$$

T = tempo di trasmissione

τ = tempo di propagazione



Protocollo Stop and Wait

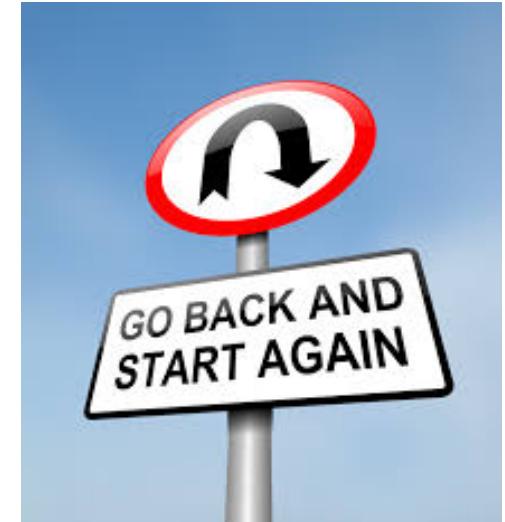
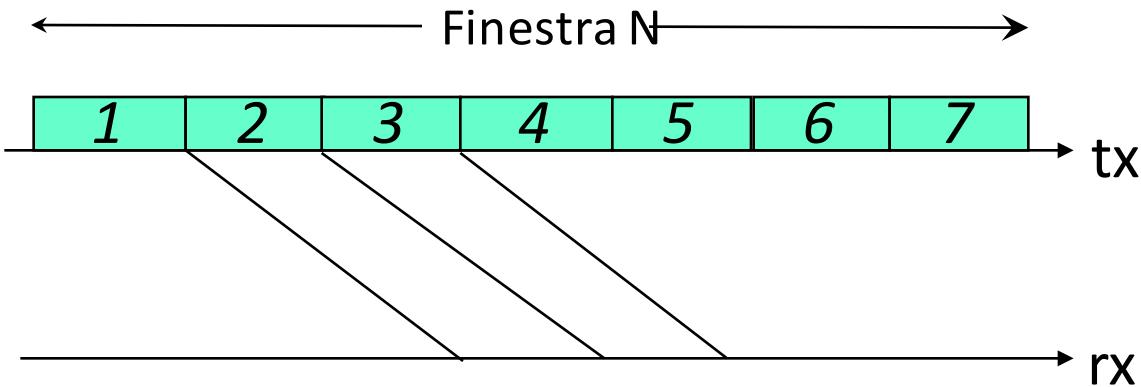
- Efficienza bassa se $T \ll \tau$
- Protocollo non adatto a situazioni con elevato ritardo di propagazione e/o elevato ritmo di trasmissione
- Utilizzato spesso in modalità *half-duplex*

$$\eta = \frac{T}{T + T_{ack} + 2\tau} = \frac{1}{1 + \frac{T_{ack}}{T} + \frac{2\tau}{T}}$$



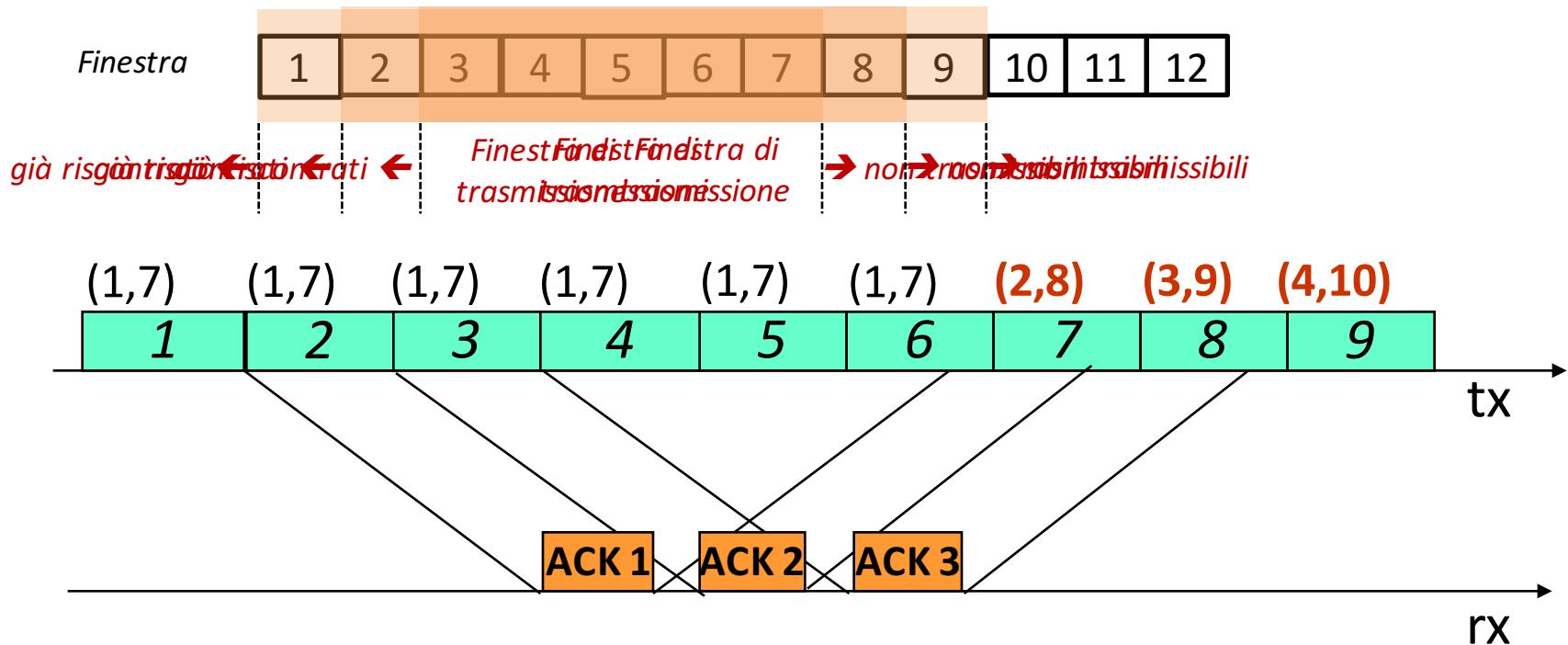
Protocollo Go-back-N

- Variante rispetto allo *Stop and Wait*:
 - Si possono trasmettere fino a N pacchetti (finestra) senza aver avuto il riscontro



Finestra Go-back-N

- Se il riscontro del primo pacchetto arriva prima della fine della finestra, la finestra viene fatta scorrere di una posizione (**sliding window**)

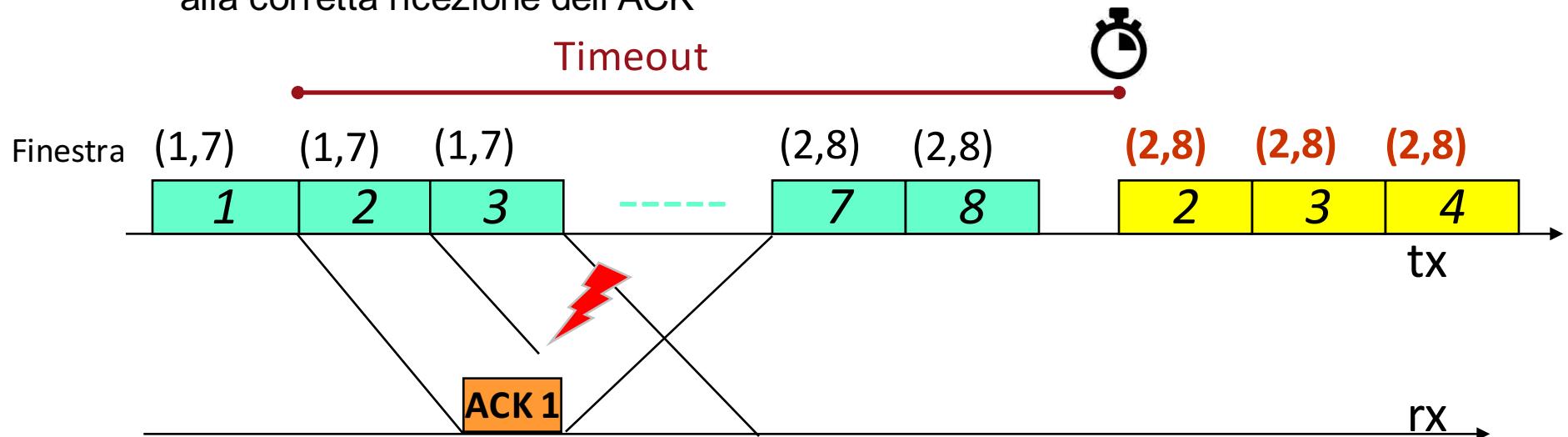


Se non ci sono errori la trasmissione non si ferma mai (efficienza 100%)



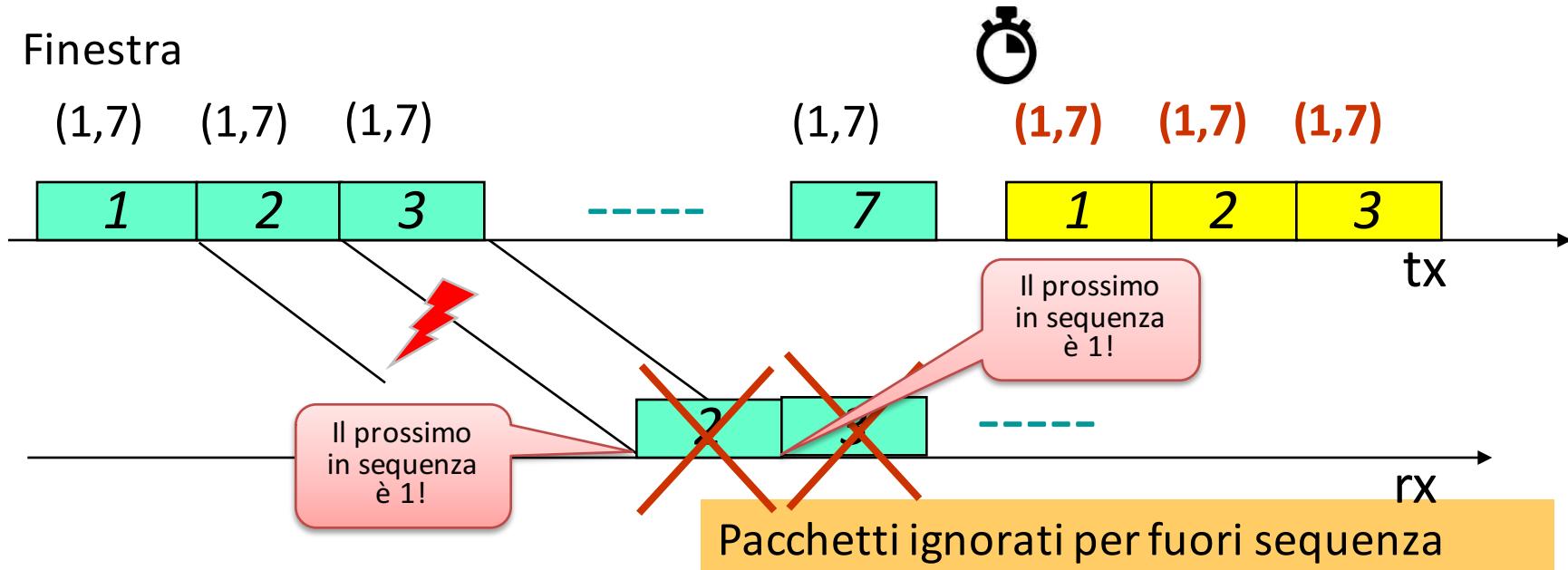
Finestra Go-back-N

- ... altrimenti (quando si verifica un errore): **si ricomincia a trasmettere la finestra dal primo pacchetto non riscontrato**
 - “Torna indietro di N pacchetti”
- il *time out* ha lo stesso significato dello *Stop&Wait*
 - Raggiunto l’ultimo pacchetto della finestra, la trasmissione si blocca in attesa di un nuovo ACK o della scadenza del *timeout*
 - La ritrasmissione del primo pacchetto non riscontrato inizia allo scadere del *timeout*
 - All’inizio di ogni pacchetto viene fatto partire un *timeout*, che viene cancellato alla corretta ricezione dell’ACK



Finestra Go-back-N

- Ciò può causare la ritrasmissione di pacchetti corretti, ma semplifica il funzionamento perché
- .. permette al ricevitore di ignorare le ricezioni fuori sequenza (l'ordine è mantenuto automaticamente)
 - Per i pacchetti ignorati, non viene trasmesso alcun ACK

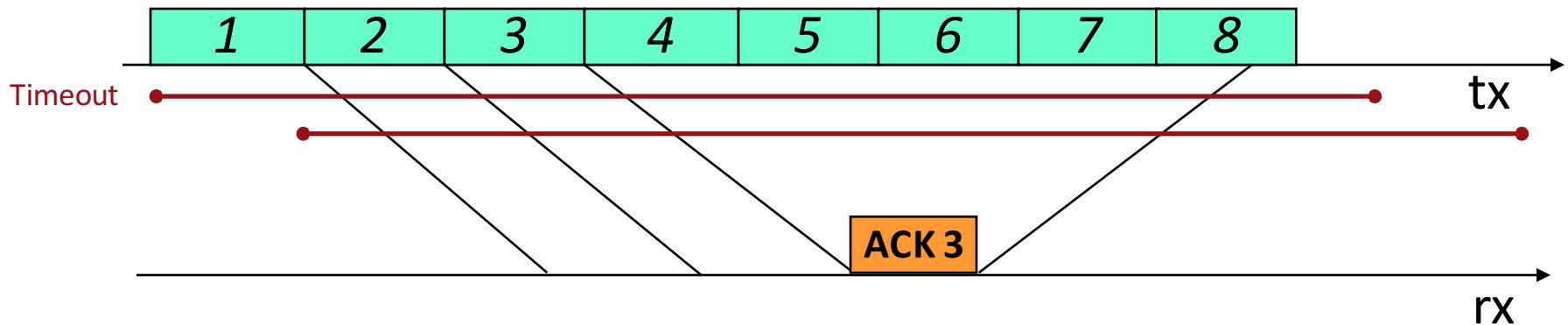


Go-back-N: ACK collettivo

- Escludendo il fuori sequenza, il riscontro (ACK) può essere collettivo
- Questo, se non scade alcun *timeout*, rimedia alla perdita di ACK

Finestra

(1,7) (1,7) (1,7) (1,7) (1,7) (1,7) (1,7) (1,7) (4,10)



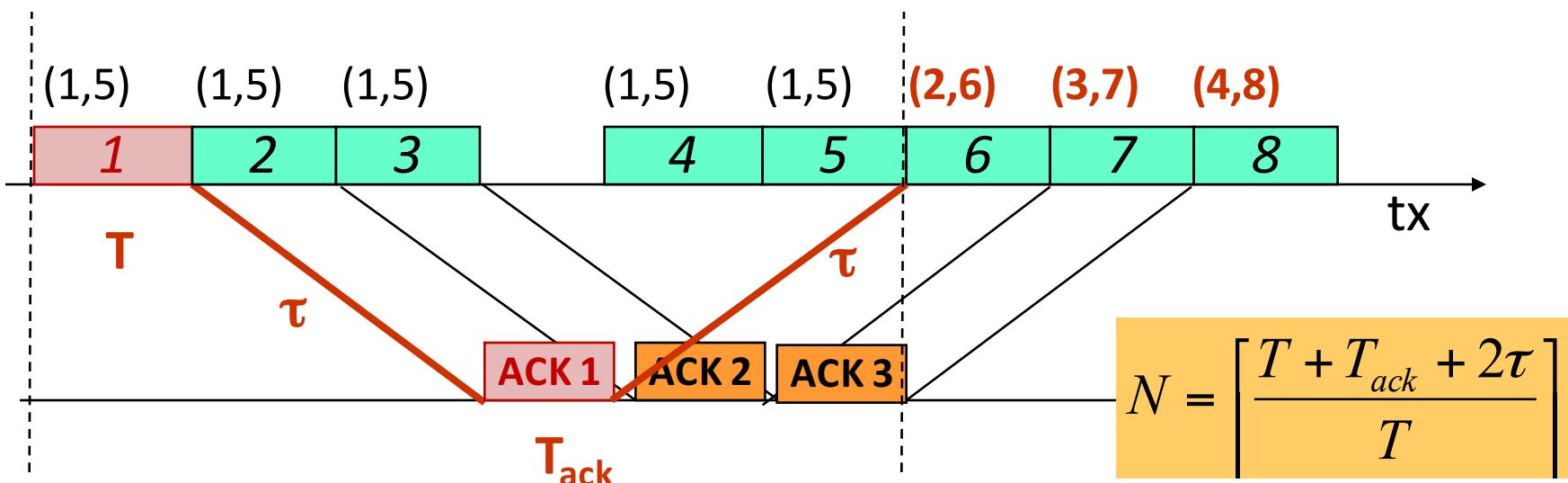
Applet Go-Back-N

http://media.pearsoncmg.com/aw/ecs_kuros_e_compnetwork_6/video_applets/GBNindex.html



Dimensionamento finestra

- La finestra ottima coincide con il *Round Trip Time*:
 - Trasmissione pacchetto, propagazione tx-rx,
Trasmissione ACK e propagazione rx-tx
- La finestra aumenta di un pacchetto alla volta e la trasmissione non si interrompe mai



Dimensionamento finestra

- La finestra può anche essere dimensionata in tempo, in byte,
- Il dimensionamento si complica se i tempi di propagazione non sono noti
 - es. con Go-back-N a livello di trasporto i tempi di attraversamento della rete (τ) possono variare col tempo.
 - e/o i pacchetti sono di lunghezza variabile
- Rimedi
 - Fare la finestra grande
 - Non pregiudica il funzionamento in assenza d'errore
 - Ma in caso d'errore, aumenta il ritardo con cui si scopre ed il numero di ritrasmissioni inutili
 - Uso del NACK
 - Stimare il tempo di RTT e adattare la finestra o il *timeout*



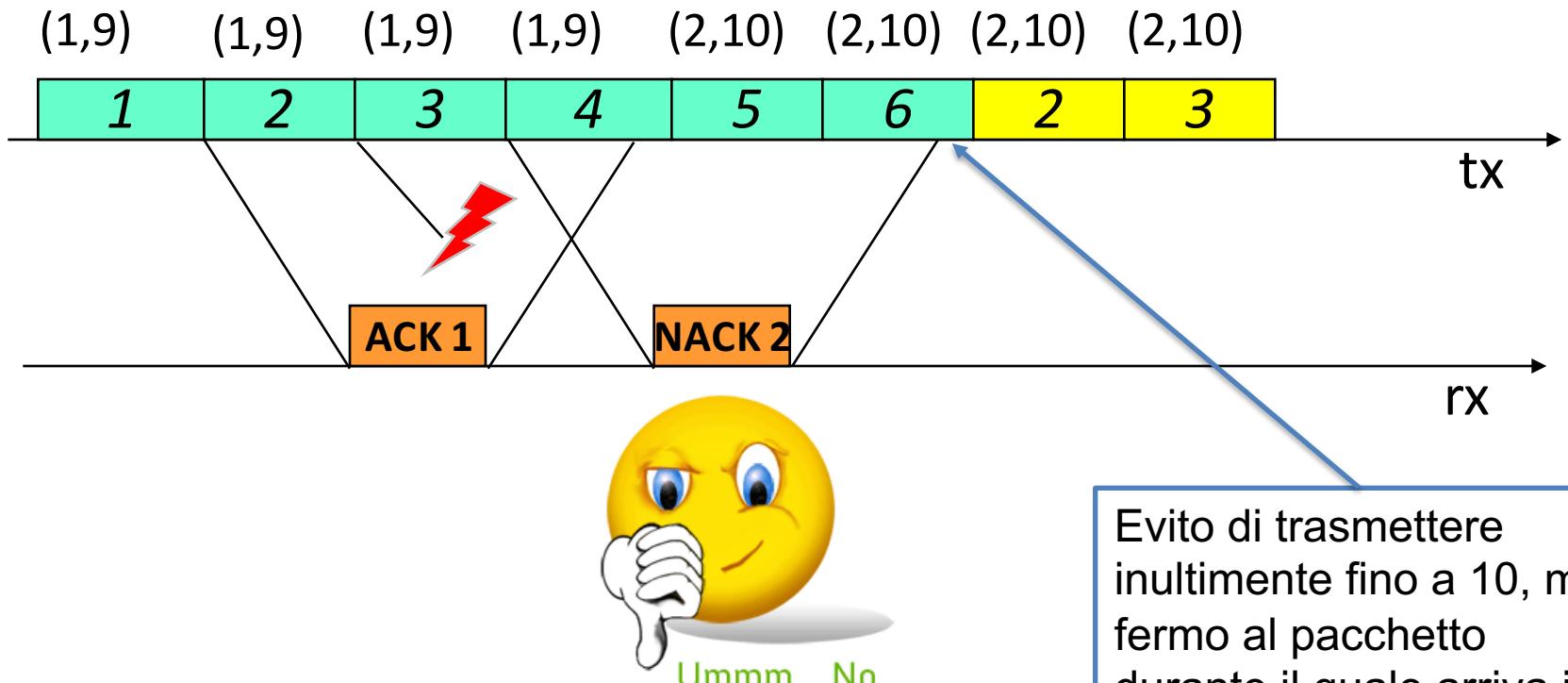
Uso del NACK

- L'uso del NACK può abbreviare i tempi di ritrasmissione in caso d'errore, evitando di aspettare la fine della finestra
- Non è possibile inviare immediatamente il NACK perché il ricevitore dovrebbe conoscere il SN del pacchetto errato
 - Ma se è andato perso o è errato, il suo contenuto non può essere letto
- Se arriva un pacchetto fuori sequenza, posso ipotizzare che sia andata perso il pacchetto precedente
- Non è possibile applicarlo nei meccanismi in cui non è garantita la consegna in ordine (livelli superiori al livello di *link*)



Uso del NACK

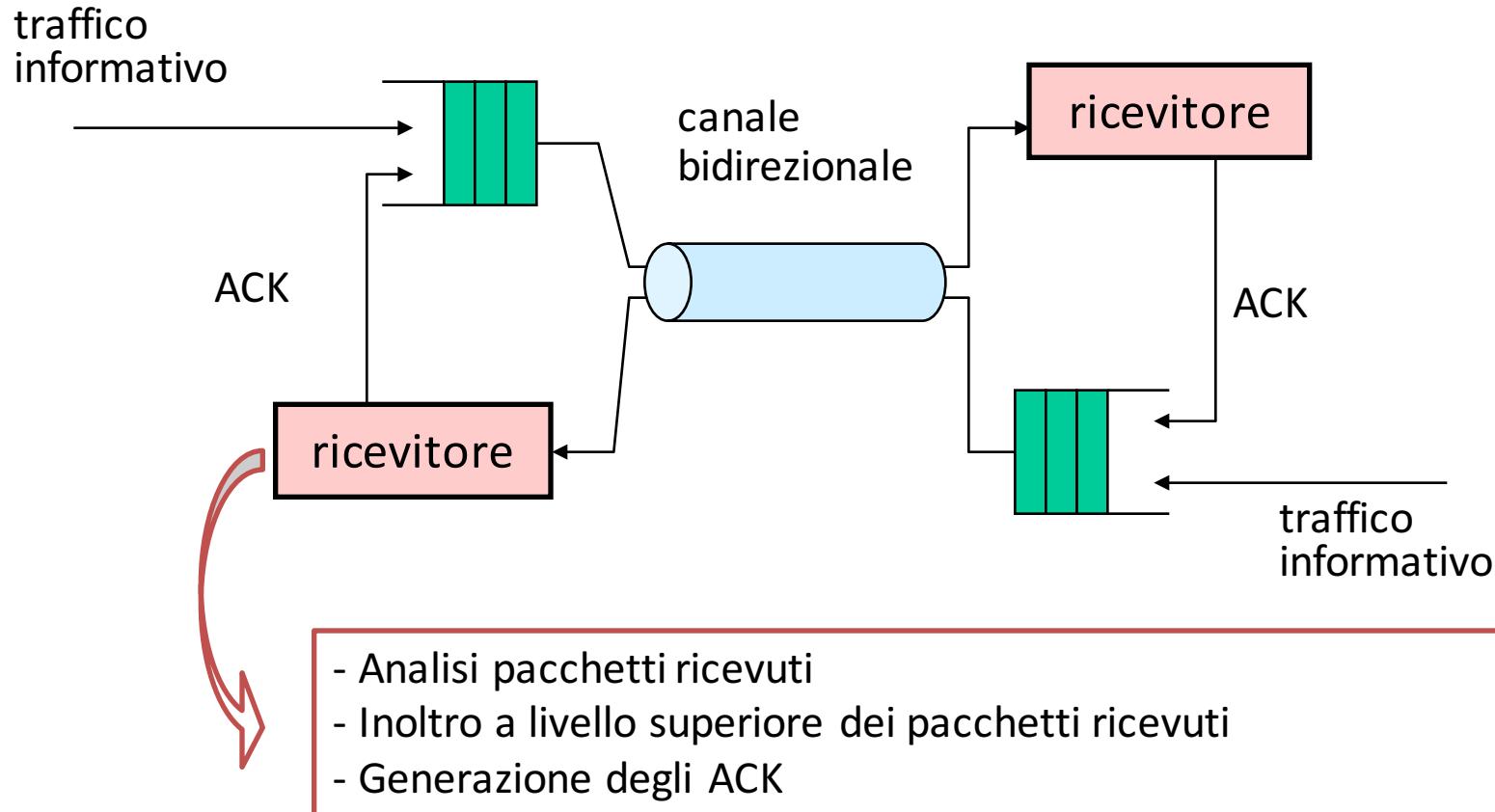
Finestra



Evito di trasmettere inultimamente fino a 10, mi fermo al pacchetto durante il quale arriva il NACK

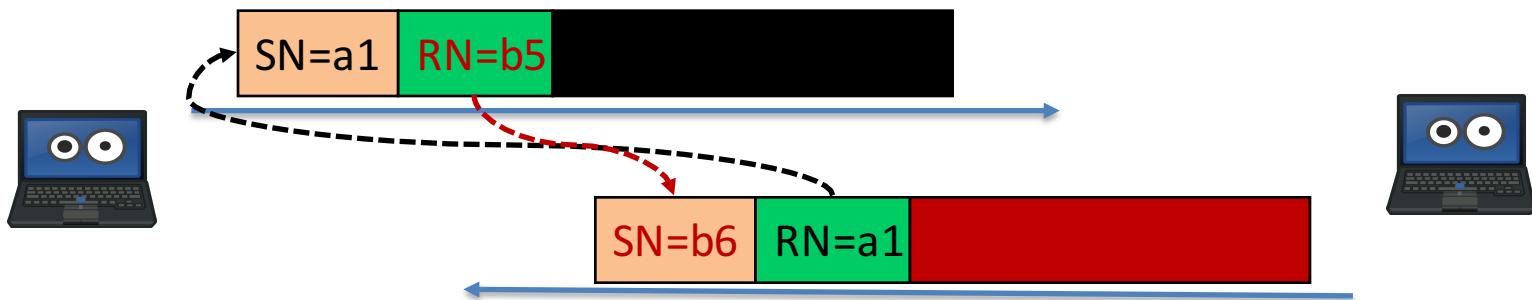


Go-back-N full-duplex



Go-back-N e piggy backing

- Gli ACK possono anche essere inseriti negli *header* dei pacchetti che viaggiano in direzione opposta (*Piggy-backing*)



- SN: numero di sequenza del pacchetto trasmesso (canale diretto)
- RN: numero di sequenza delle pacchette atteso in direzione opposta, vale come riscontro cumulativo dei pacchetti fino a $RN-1$



Regole Go-back-N

- Trasmettitore:
 - N : dimensione finestra
 - N_{last} : ultimo riscontro ricevuto
 - N_C : numero corrente disponibile per pacchetto in trasmissione
 - **Regole:**
 - Ogni nuovo pacchetto viene messo in attesa se $N_C \geq N_{last} + N$, altrimenti, se $N_C < N_{last} + N$, il pacchetto viene trasmesso con SN pari a N_C , viene inizializzato il timer di timeout e il valore di N_C viene incrementato di uno ($N_C := N_C + 1$);
 - Ad ogni riscontro RN ricevuto, si pone $N_{last} = RN$;
 - I pacchetti nella finestra vengono trasmessi senza vincoli di temporizzazione;
 - In caso di scadenza di timeout la ritrasmissione deve ripartire dal pacchetto N_{last}
 - Intervallo N_{last} e $(N_{last} + N - 1)$: finestra di trasmissione

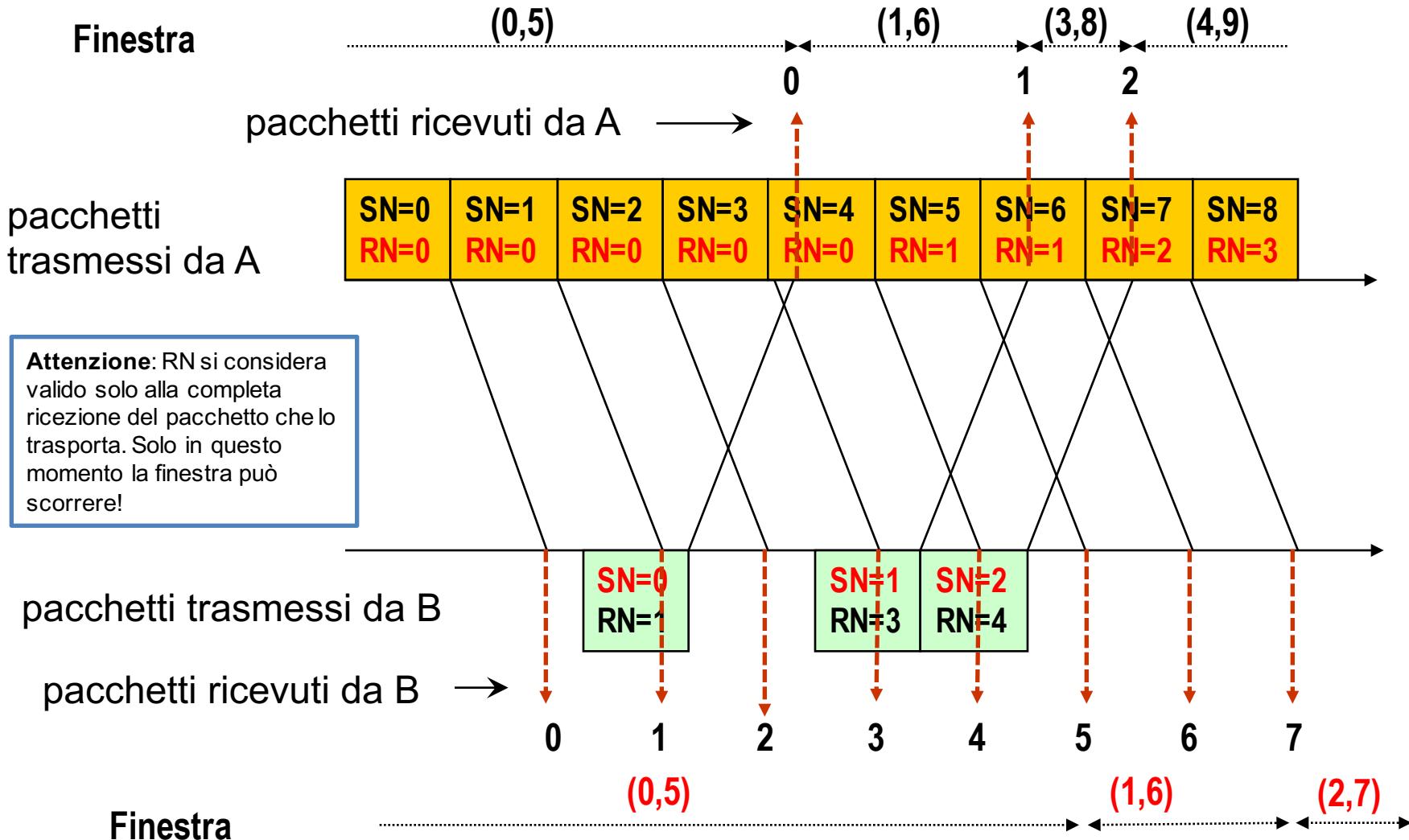


Regole Go-back-N

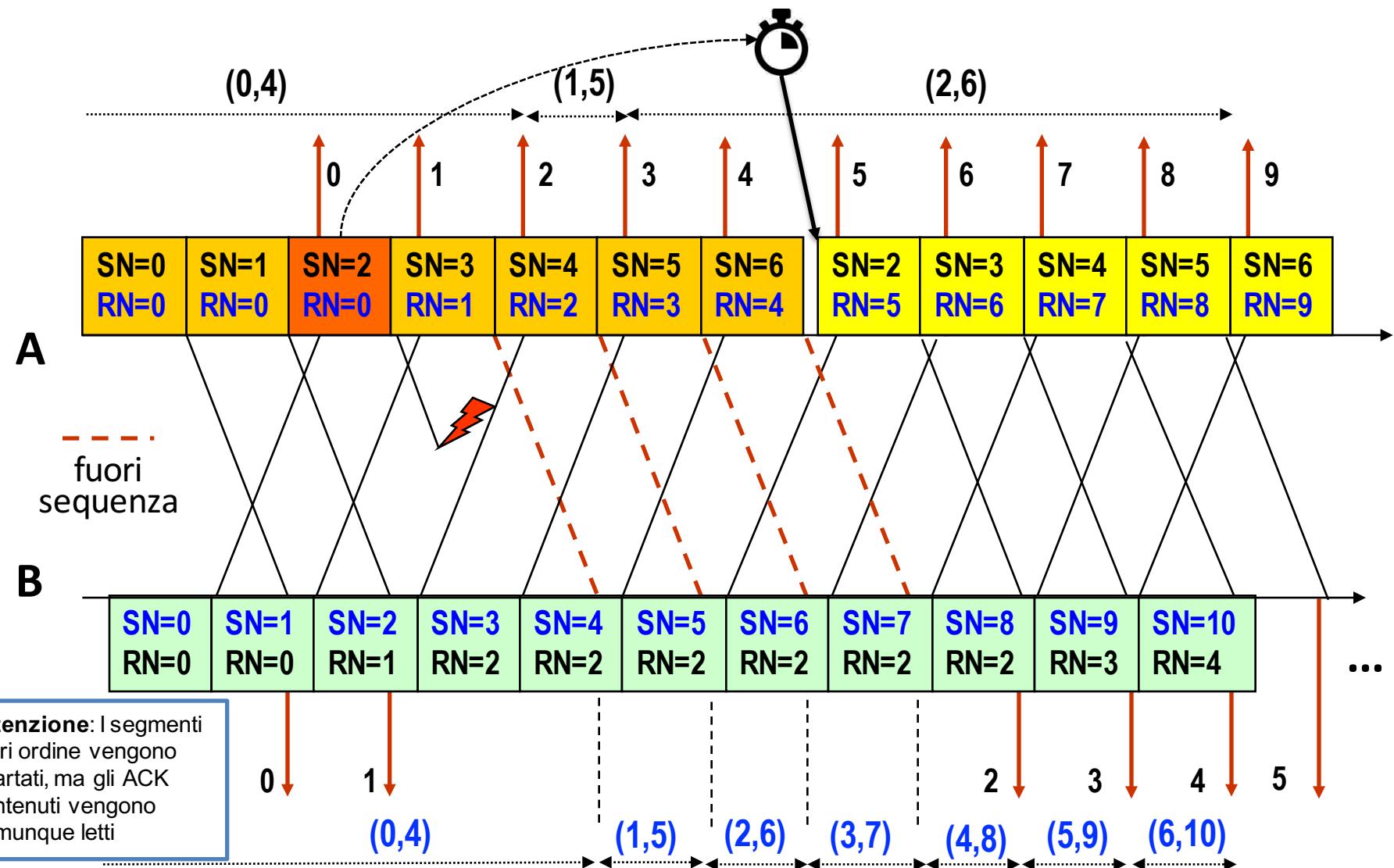
- Ricevitore:
 - *RN: stato dei riscontri corrente (SN che il ricevitore si aspetta di ricevere)*
 - **Regole:**
 - Se viene ricevuto correttamente un pacchetto con $SN=RN$, questo viene inoltrato ai livelli superiori e si pone $RN:=RN+1$;
 - Ad istanti arbitrari ma con ritardo finito, RN viene trasmesso al mittente utilizzando i pacchetti in direzione opposta.



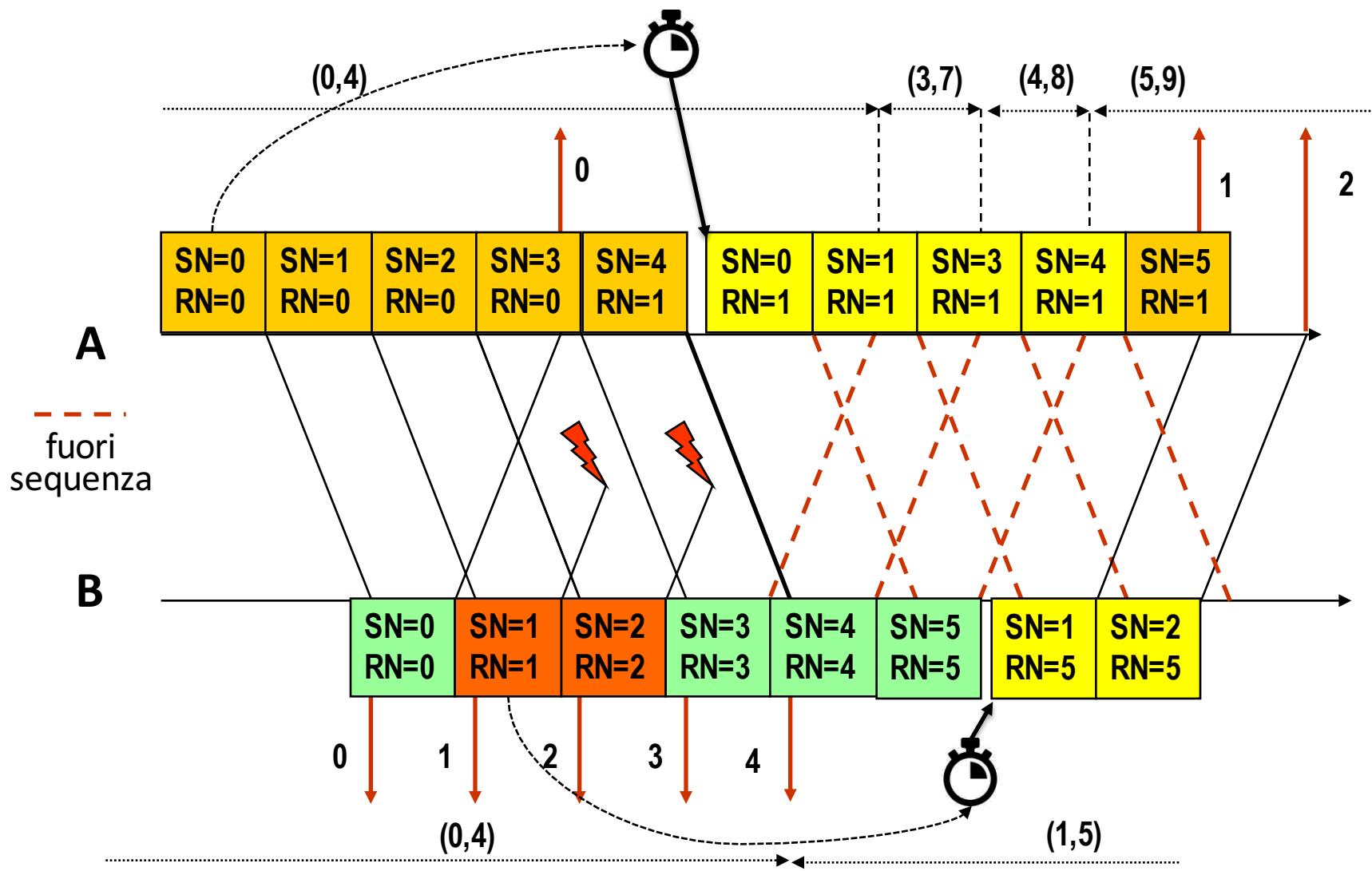
Go-back-N full duplex senza errori



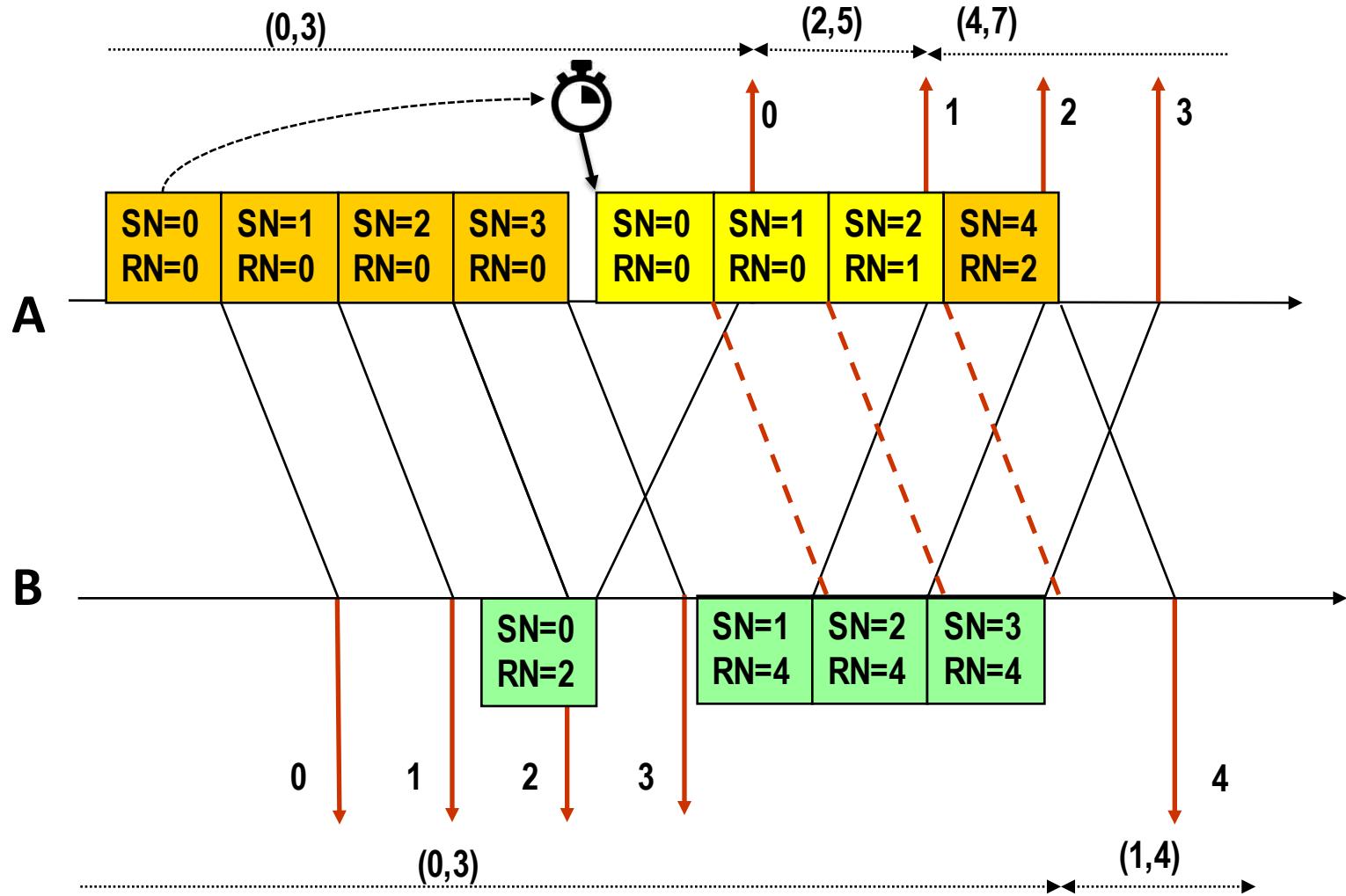
Go-back-N full duplex con errore



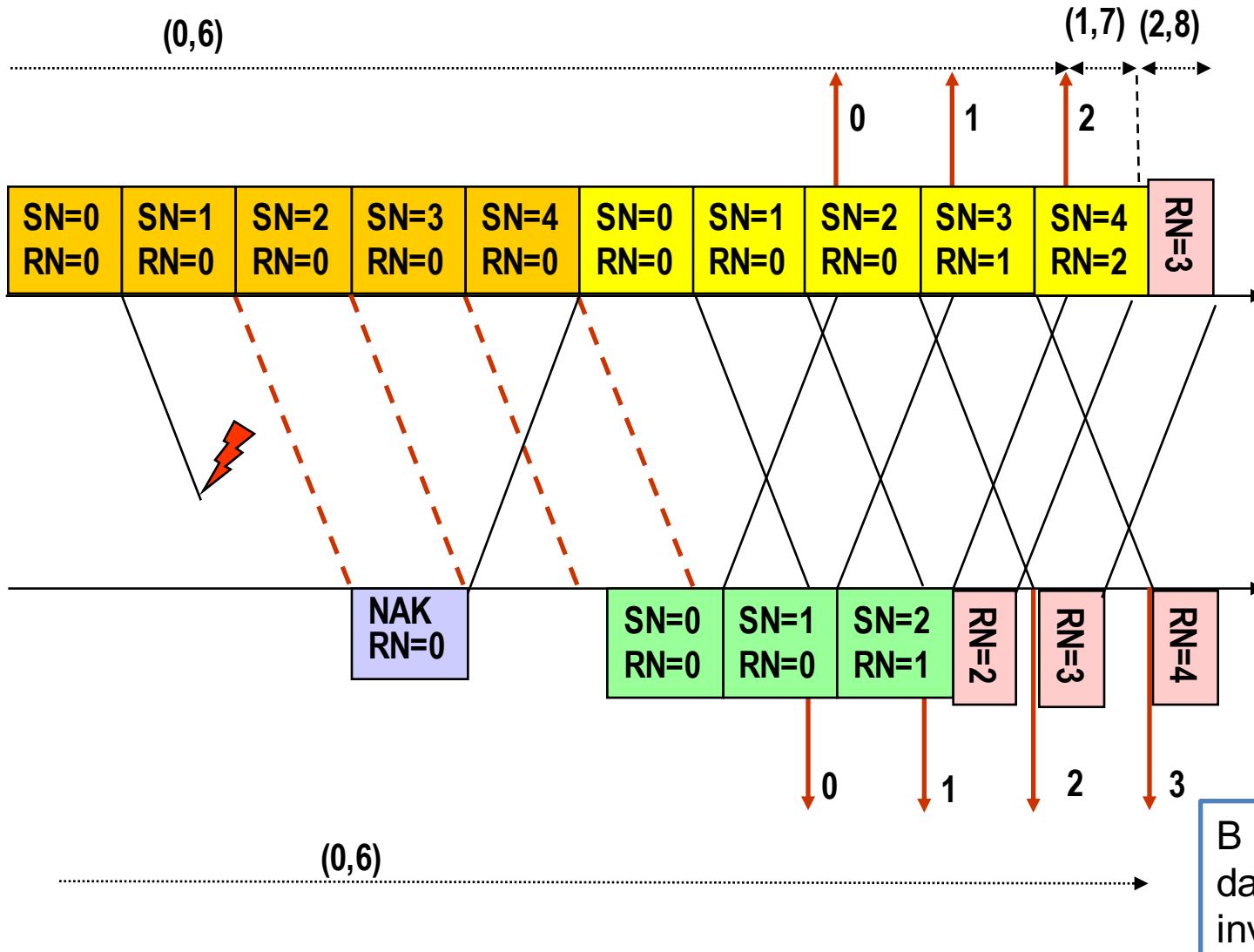
Go-back-N full duplex con errore



Go-back-N full con ACK ritardati



Go-back-N con NAK



Osservazioni sul controllo d'errore

- Necessità di inizializzare il protocollo
 - I numeri SN e RN devono essere inizializzati
 - Deve esistere un momento di inizio non equivocabile in cui scambiare l'informazione per l'inizializzazione

Occorre un meccanismo a connessione che stabilisca l'istante $t=0$



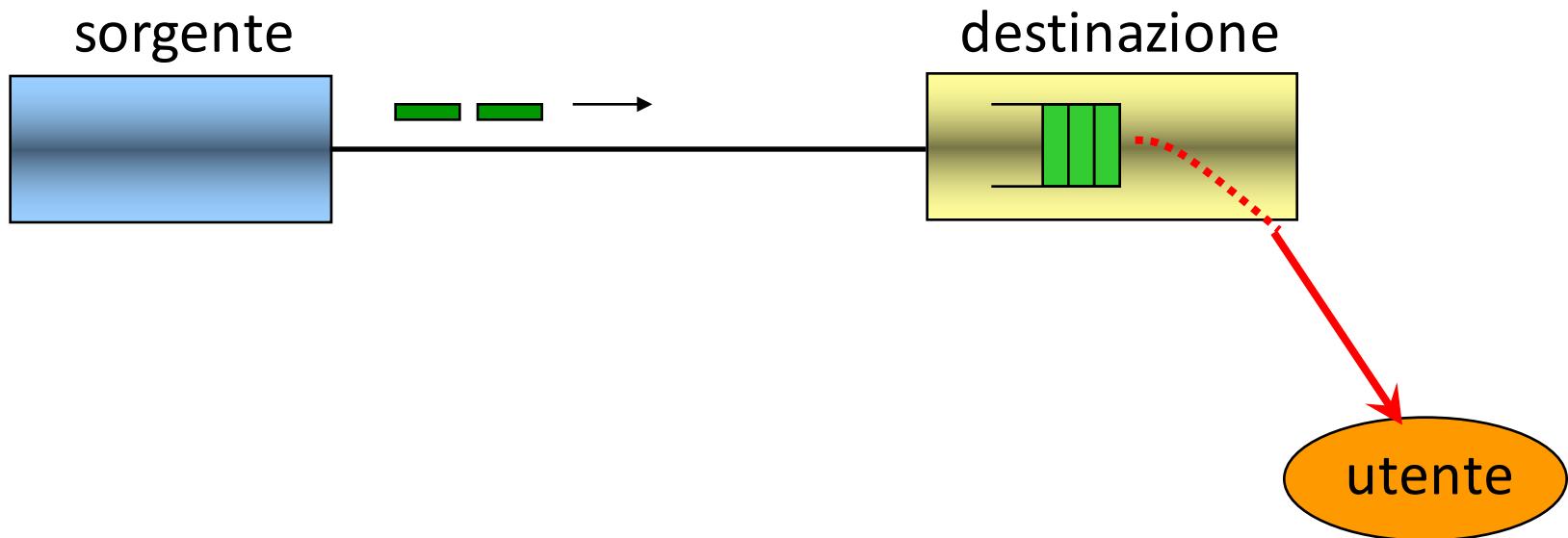
Livello di Trasporto

- Introduzione
- Protocollo UDP
- Trasporto affidabile
 - Protocolli di ritrasmissione
 - Controllo di flusso a finestra mobile
- Protocollo TCP
 - Generalità
 - Formato e connessioni
 - Controllo di flusso
 - Controllo d'errore
 - Controllo di congestione



Controllo di flusso

- Buffer di ricezione limitato a **W** posizioni
- Ritmo di assorbimento dell'utente arbitrario
- Obiettivo: regolando il ritmo di invio, evitare che pacchetti vadano persi perché all'arrivo trovano il buffer pieno



Sliding window flow control

- Controllo di flusso a finestra mobile
- E' possibile usare un meccanismo come quello del *Go-Back-N*
- La sorgente non può inviare più di W trame (stessa funzione del parametro N) senza aver ricevuto il riscontro
- I riscontri vengono inviati dal ricevitore solo quando i pacchetti vengono letti (**tolti dal buffer**) dal livello superiore



Problema delle ritrasmissioni

- Se il ricevitore ritarda molto l'invio dei riscontri a causa del **livello superiore lento**, il trasmettitore inizia la ritrasmissione perché **scade il time-out**
 - Controllo di flusso a finestra mobile e controllo d'errore a finestra mobile sono fortemente legati
- Aumentare troppo il time-out non è ovviamente una soluzione in quanto l'aumento del time-out aumenta i ritardi in caso di errore

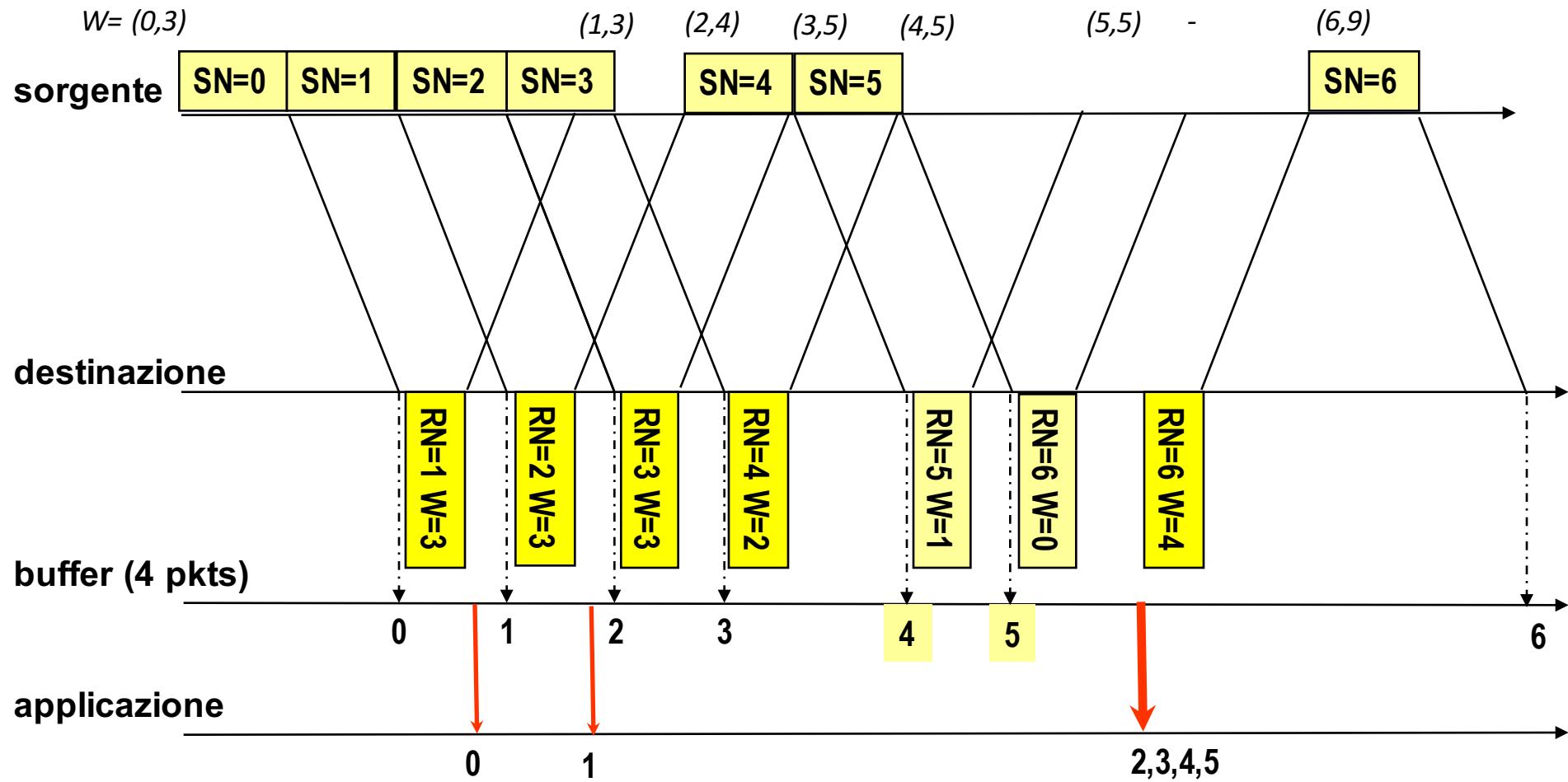


Uso del campo W

- Il problema può essere risolto in modo radicale separando i meccanismi di controllo d'errore e di controllo di flusso a finestra
- Si inserisce nei riscontri (o nell'*header* dei pacchetti in direzione opposta) un campo finestra W (oltre a quella del Go-Back-N)
 - Il ricevitore invia i riscontri sulla base dell'arrivo dei pacchetti
 - E usa il campo W per indicare lo spazio rimanente nel buffer



Uso del campo W



Uso del campo W

- Gestione della finestra del controllo di flusso
 - Non è necessario che il ricevitore dica la “verità” sullo spazio restante R
 - Può tenersi un margine di sicurezza ($W=R-m$)
 - Dicho il buffer vuoto anche se non lo è veramente
 - Robustezza nel caso arrivino altri segmenti prima che il nuovo valore di W sia ricevuto
 - Può aspettare che il buffer si sia svuotato per una frazione (ad es. $W=0$ se $R < \text{buffer}/2$ ed $W=R$ altrimenti)
 - Aspetto che il buffer si riempia significativamente prima di comunicarlo a trasmettitore
 - Evito l'invio di segmenti troppo brevi
 - Può usare dei meccanismi adattativi
- Successivamente vedremo degli esempi



Livello di Trasporto

- Introduzione
- Protocollo UDP
- Trasporto affidabile
 - Protocolli di ritrasmissione
 - Controllo di flusso a finestra mobile
- **Protocollo TCP**
 - Generalità
 - Formato e connessioni
 - Controllo di flusso
 - Controllo d'errore
 - Controllo di congestione



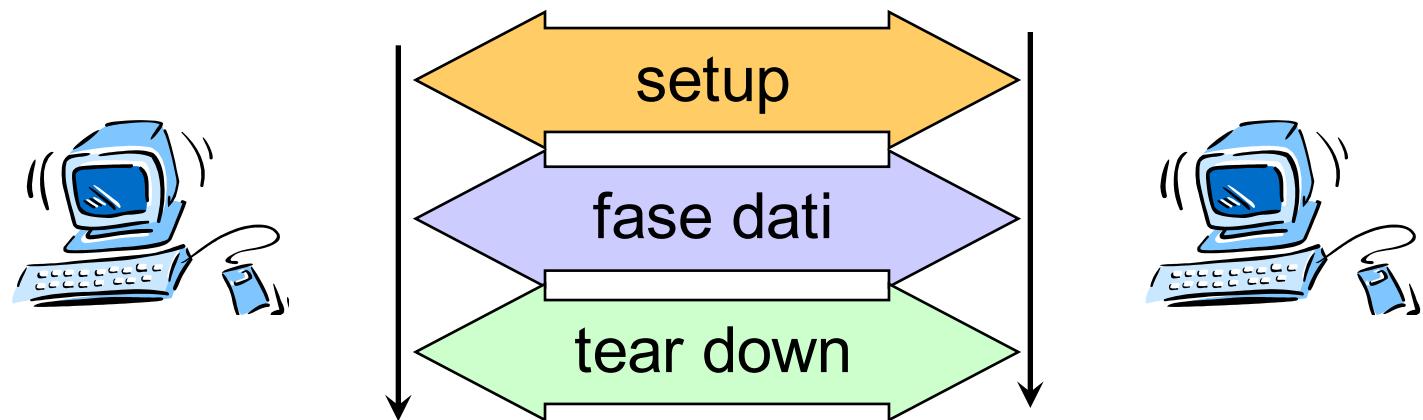
Transmission Control Protocol (TCP) – RFC 793 et al.

- Il TCP è un protocollo di trasporto che assicura il trasporto affidabile
 - In corretta sequenza
 - Senza errori/perdite dei dati
- Mediante TCP è possibile costruire applicazioni che si basano sul trasferimento di file senza errori tra *host* remoti (web, posta elettronica, ecc.)
- E' alla base della filosofia originaria di Internet: servizio di rete semplice e non affidabile, servizio di trasporto affidabile
- Il TCP effettua anche un controllo di congestione *end-to-end* che limita il traffico in rete e consente agli utenti di condividere in modo equo le risorse



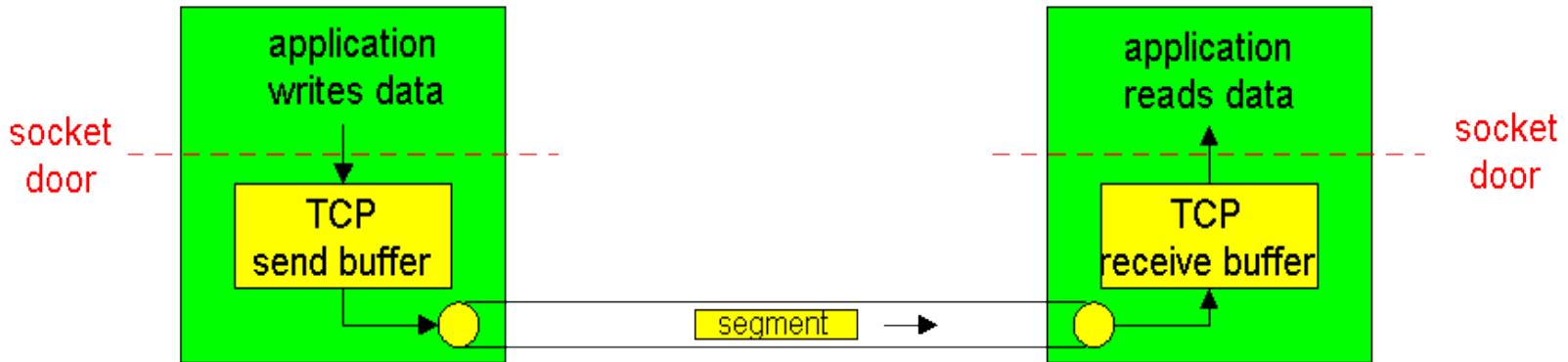
TCP: *connection oriented*

- Il TCP è orientato alla connessione (*connection oriented*):
 - Prima del trasferimento di un flusso dati occorre instaurare una connessione mediante opportuna segnalazione
 - Le connessioni TCP si appoggiano su una rete *connectionless* (*datagram*)
 - Le connessioni TCP sono di tipo *full-duplex* (esiste sempre un flusso di dati in un verso e nel verso opposto, anche se questi possono essere quantitativamente diversi)



TCP: flusso dati

- Il TCP è orientato alla trasmissione di flussi continui di dati (stream di byte)
- Il TCP converte il flusso di dati in *segmenti* che possono essere trasmessi in rete
- Le dimensioni dei segmenti sono variabili
- L'applicazione trasmittente passa i dati (byte) a TCP e TCP li accumula in un buffer.
- Periodicamente, o quando avvengono particolari condizioni, il TCP prende una parte dei dati nel buffer e forma un segmento
- La dimensione del segmento è critica per le prestazioni, per cui il TCP cerca di attendere fino a che un ammontare ragionevole di dati sia presente nel buffer di trasmissione



TCP: numerazione byte e riscontri

- Il TCP adotta un meccanismo per il controllo delle perdite di pacchetti di tipo **Go-Back-N**
- Sistema di numerazione e di riscontro dei dati inviati
 - TCP numera ogni byte trasmesso, per cui ogni byte ha un numero di sequenza
 - I segmenti sono formati da gruppi di byte
 - Le dimensioni delle finestre sono espresse in byte
 - Nell'*header* del segmento TCP è trasportato il numero di sequenza del primo byte nel segmento stesso
 - Il ricevitore deve riscontrare i dati ricevuti inviando il numero di sequenza del prossimo byte che ci si aspetta di ricevere.
 - Se un riscontro non arriva entro un dato *timeout*, i dati sono ritrasmessi

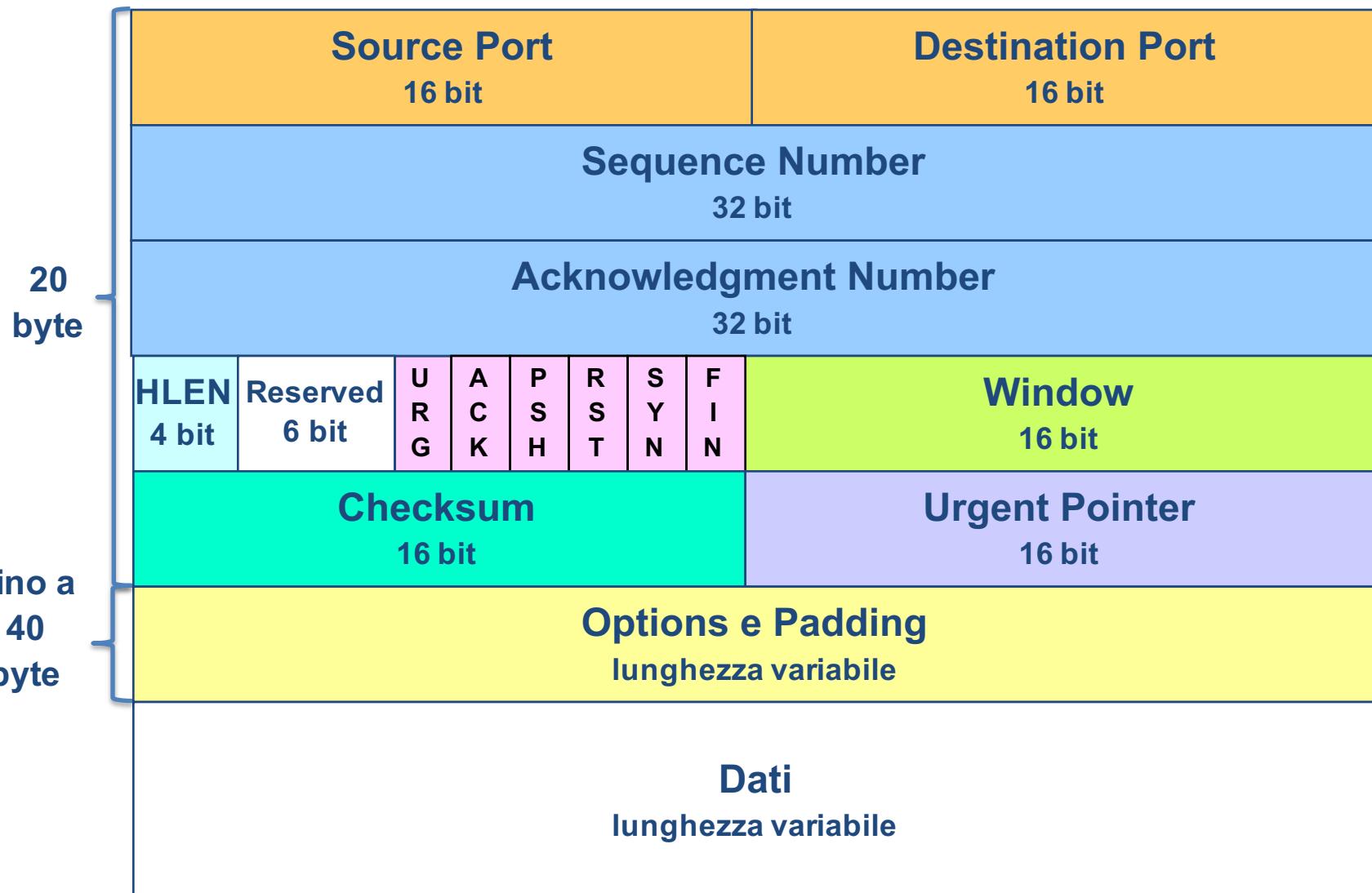


Livello di Trasporto

- **Introduzione**
- **Protocollo UDP**
- **Trasporto affidabile**
 - Protocolli di ritrasmissione
 - Controllo di flusso a finestra mobile
- **Protocollo TCP**
 - Generalità
 - Formato e connessioni
 - Controllo di flusso
 - Controllo d'errore
 - Controllo di congestione



Segmento TCP



Header Segmento TCP (1)

- **Source port, Destination port:** indirizzi di porta sorgente e porta destinazione di 16 bit
- **Sequence Number:** il numero di sequenza del primo byte nel payload
- **Acknowledge Number:** numero di sequenza del prossimo byte che si intende ricevere (numero valido solo se flag ACK valido)
- **HLEN (4 byte words):** contiene la lunghezza complessiva dell'header TCP, che DEVE essere un multiplo intero di 32 bit
- **Window:** contiene il valore della finestra di ricezione come comunicato dal ricevitore al trasmettitore
- **Checksum:** il medesimo di UDP, calcolato in maniera uguale



Header Segmento TCP (2)

- Flags:
 - **URG**: vale 1 se vi sono dati urgenti e quindi il TCP deve passare in modalità urgente; in questo caso *urgent pointer* punta all'ultimo byte dei dati all'interno del flusso oltre il quale TCP può tornare in modalità normale
 - **ACK**: vale 1 se il pacchetto è un ACK valido; in questo caso *l'acknowledge number* contiene un numero valido
 - **PSH**: vale 1 quando il trasmettitore intende usare il comando di PUSH; il ricevitore può anche ignorare il comando (dipende dalle implementazioni)
 - **RST**: reset, resetta la connessione senza un *tear down* esplicito
 - **SYN**: *synchronize*; usato durante il *setup* per comunicare i numeri di sequenza iniziale
 - **FIN**: usato per la chiusura esplicita di una connessione
- **Options and Padding**: riempimento (fino a multipli di 32 bit) e campi opzionali, es., durante il setup per comunicare il MSS (il valore di default è 536 byte)



Opzioni

- Le opzioni sono aggiunte all'header TCP
- Opzioni di 1 byte (no-operation, end-of-option, ...):
 - Servono come riempimento per avere un header multiplo di 32 bit
- Opzioni lunghe:
 - Maximum segment size (MSS)
 - Fattore di scala della finestra
 - ...
 - <http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml>



Opzioni: Maximum Segment Size (MSS)

- Definisce la dimensione massima del segmento che verrà usata nella connessione TCP
- La dimensione è decisa dal mittente durante la fase di *setup*
- Valore di *default* è 536 byte, il valore massimo 65535 byte

Code (00000010)	Length (00000100)	MSS 16 bit
---------------------------	-----------------------------	----------------------



Opzioni: Fattore di scala della finestra

- Definisce il fattore di scala della finestra
- Il valore di *default* è 1
- L'opzione fa sì che venga moltiplicato il valore del campo *Window* di un fattore pari a 2 elevato al valore contenuto nel campo *fattore di scala*

Code (00000010)	Length (00000011)	Fattore di scala 8 bit
---------------------------	-----------------------------	----------------------------------



Servizi e porte

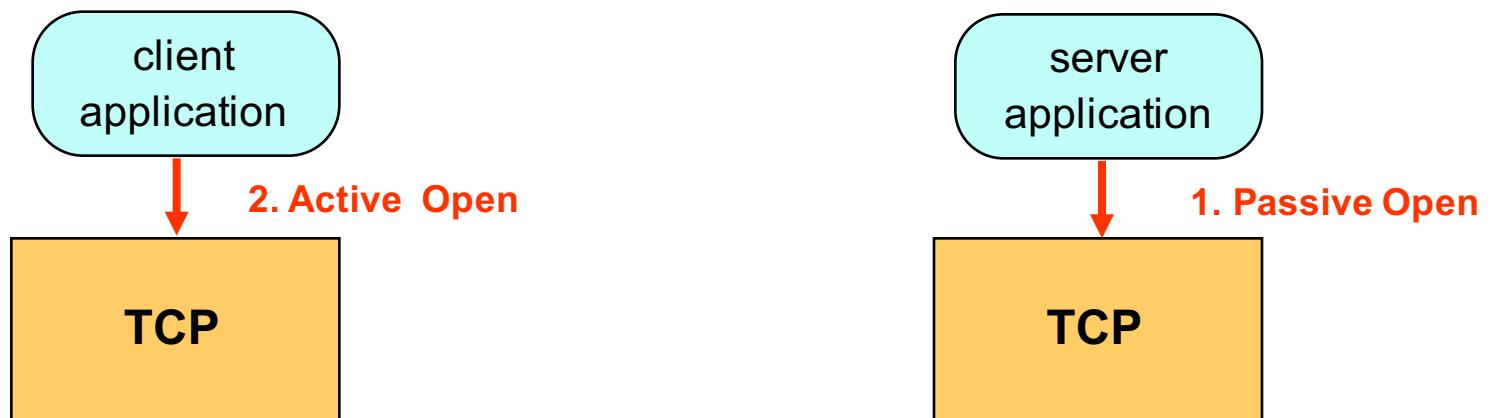
- La divisione tra porte note, assegnate e dinamiche è la stessa che per UDP
- Alcuni delle applicazioni più diffuse:

22	SSH
21	FTP signalling
20	FTP data
23	telnet
25	SMTP
53	DNS
80	HTTP
110	POP
143	IMAP



Setup delle connessioni

- Prima del *call setup* le applicazioni dal lato *client* e dal lato server devono comunicare con il software TCP
 - 1. Il server fa una *Passive Open*, che comunica al TCP locale che è pronto per accettare nuove connessioni
 - 2. Il *client* che desidera comunicare fa una *Active Open*, che comunica al TCP locale che l'applicativo intende effettuare una connessione verso un dato socket



Setup delle connessioni

- 3. Il client TCP estrae a caso un numero di sequenza iniziale (67803) e manda un messaggio di SYNchronize (flag SYN=1) contenente questo numero di sequenza. Eventualmente indica anche i parametri della connessione (MSS, Windows Scale)
- L'estrazione del numero iniziale serve a evitare problemi nel caso in cui il setup non va a buon fine a causa della perdita di pacchetti e un nuovo setup viene iniziato subito dopo



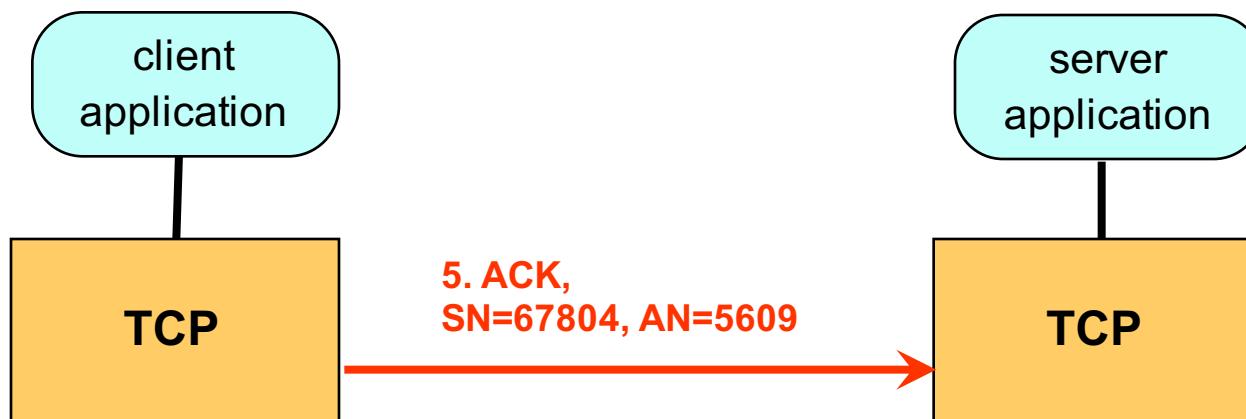
Setup delle connessioni

- Quando riceve il SYN, il TCP server estrae a caso un numero di sequenza iniziale (5608) e manda un segmento SYN/ACK (flag SYN=1, flag ACK=1) contenente anche un *acknowledgment number* uguale a 67804, per riscontrare il numero di sequenza iniziale precedentemente inviato dal TCP client.



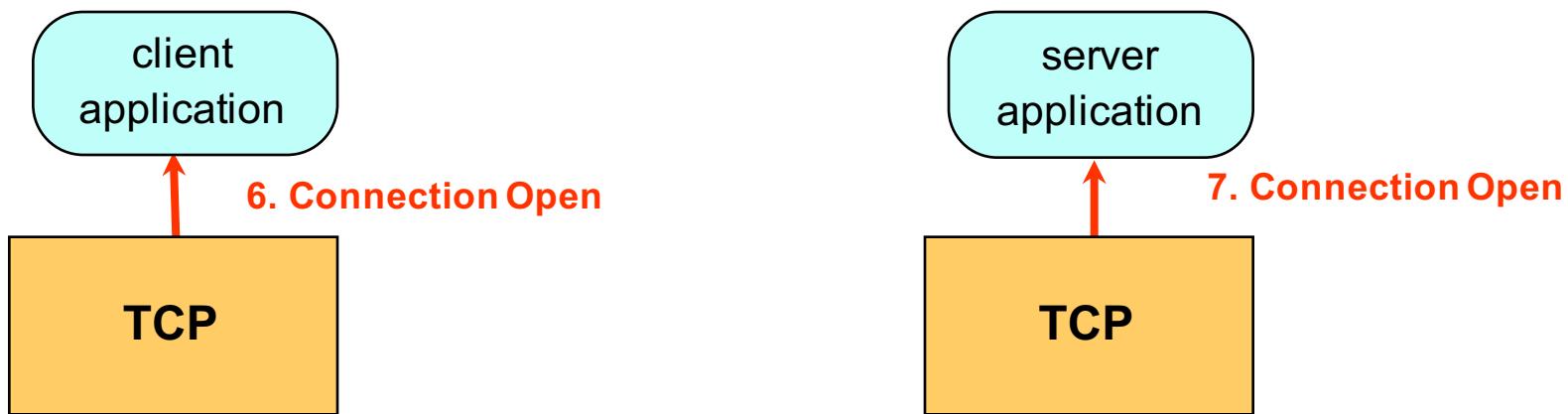
Setup delle connessioni

- Il TCP *client* riceve il messaggio SYN/ACK del server, e invia un ACK per il 5609. Nel *payload* inserisce i primi dati della connessione con numero di sequenza del primo byte pari a 67804. Inserisce anche la dimensione della finestra del server.

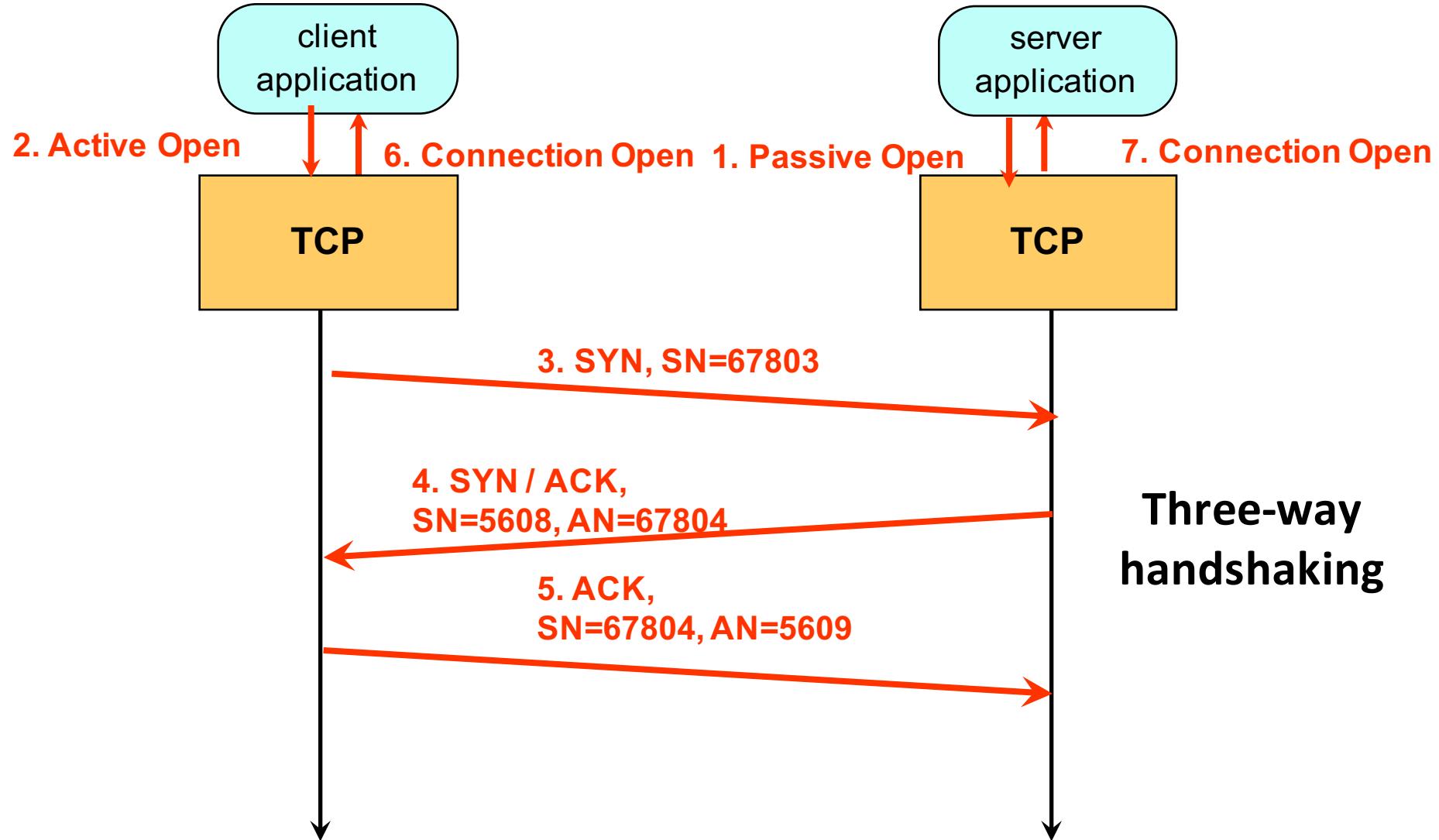


Setup delle connessioni

- Il TCP client notifica all'applicazione che la connessione è aperta
- Quando il TCP server riceve l'ACK del TCP client, notifica al suo applicativo che la connessione è aperta



Setup delle connessioni (sommario)



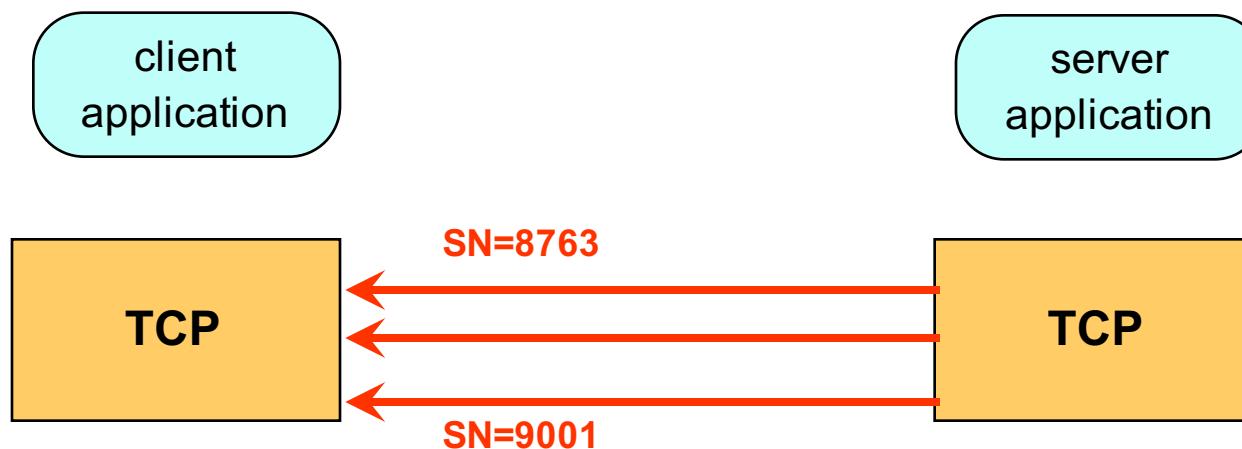
Tear down (chiusura) delle connessioni

- 1. Il TCP che chiude la connessione invia un messaggio di FIN (flag FIN=1) con gli ultimi dati
- 2. Il TCP dall'altra parte invia un ACK per confermare



Tear down (chiusura) delle connessioni

- La connessione rimane comunque aperta nell'altra direzione e quindi il TCP dall'altra parte può continuare ad inviare dati

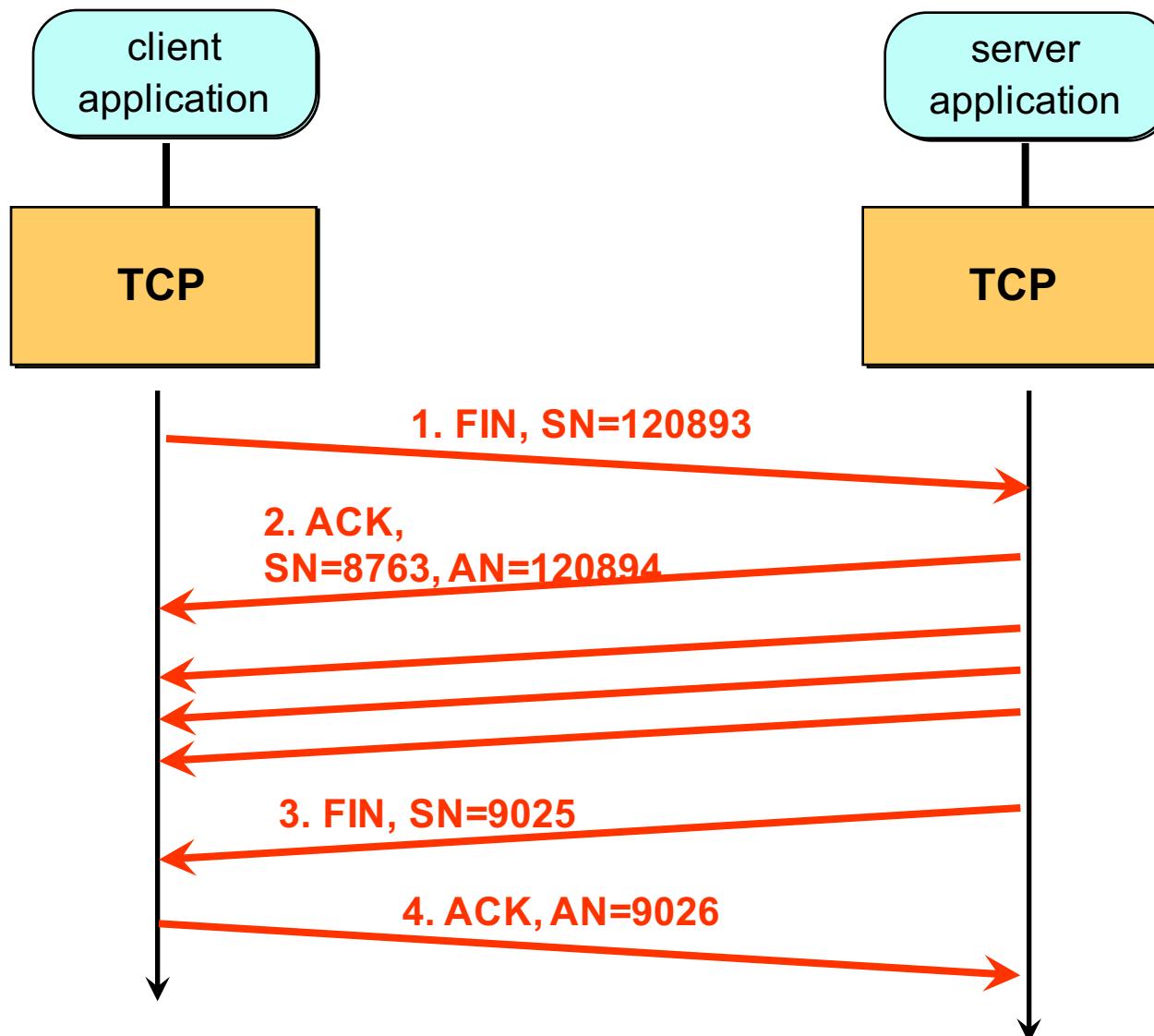


Tear down (chiusura) delle connessioni

- 3. Infine, il TCP dall'altra parte chiude la connessione invia un messaggio di FIN (flag FIN=1)
- 4. Il TCP che aveva già chiuso la connessione in direzione opposta invia un ACK finale per confermare

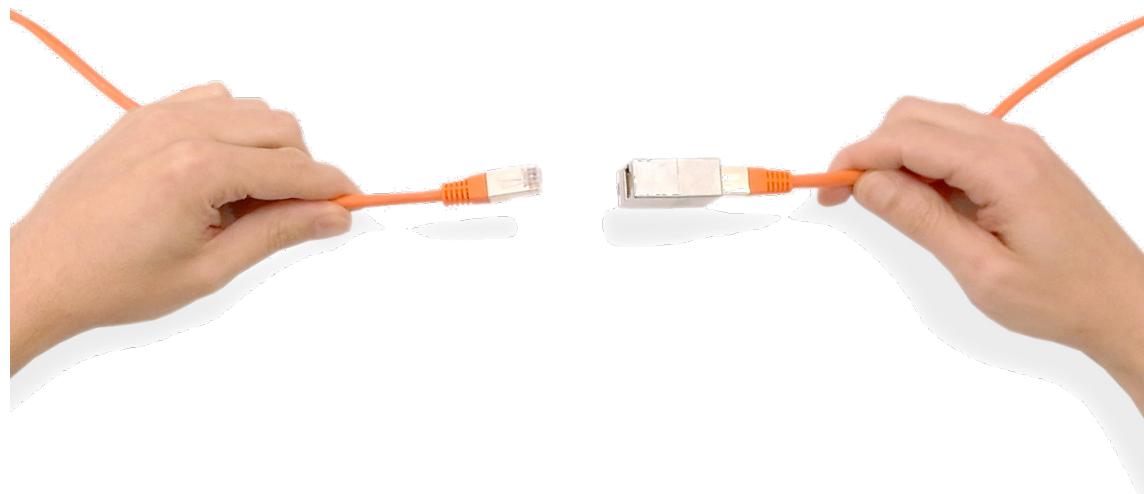


Tear down delle connessioni



Reset della connessione

- La connessione può anche essere chiusa senza scambio di messaggi nei due versi
- E' possibile infatti settare il *flag* di RESET nel segmento e interrompere la connessione in entrambe le direzioni
- Il TCP che riceve un RESET chiude la connessione interrompendo ogni invio di dati



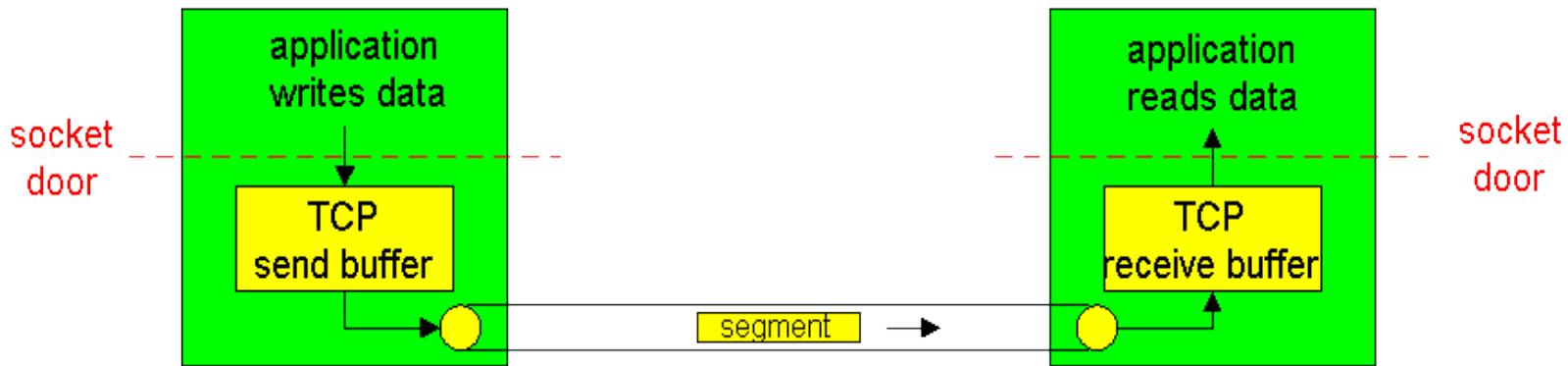
Livello di Trasporto

- Introduzione
- Protocollo UDP
- Trasporto affidabile
 - Protocolli di ritrasmissione
 - Controllo di flusso a finestra mobile
- Protocollo TCP
 - Generalità
 - Formato e connessioni
 - Controllo di flusso
 - Controllo d'errore
 - Controllo di congestione



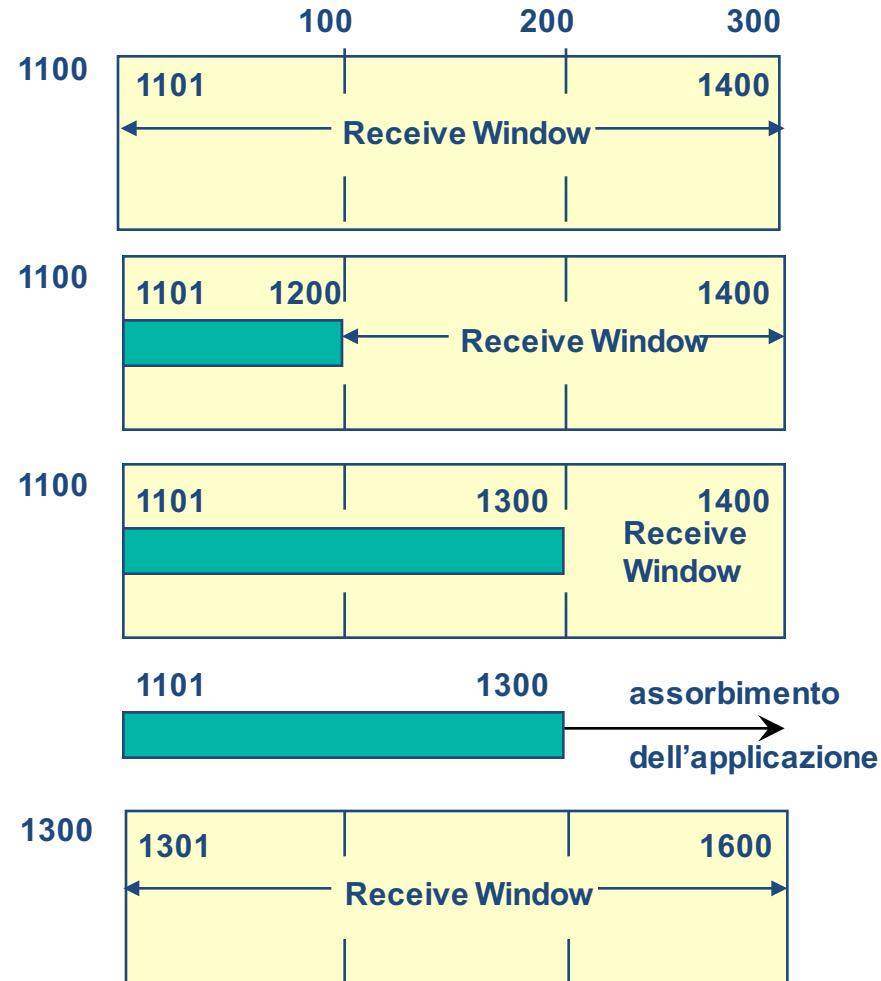
Implementazione del controllo di flusso

- Il TCP ricevente controlla il flusso di quello trasmittente
- Lato ricevitore:
 - Buffer di ricezione: accumula i byte ricevuti e non ancora assorbiti dall'applicazione
- Lato trasmettitore:
 - Buffer di trasmissione: accumula i byte in attesa di essere trasmessi



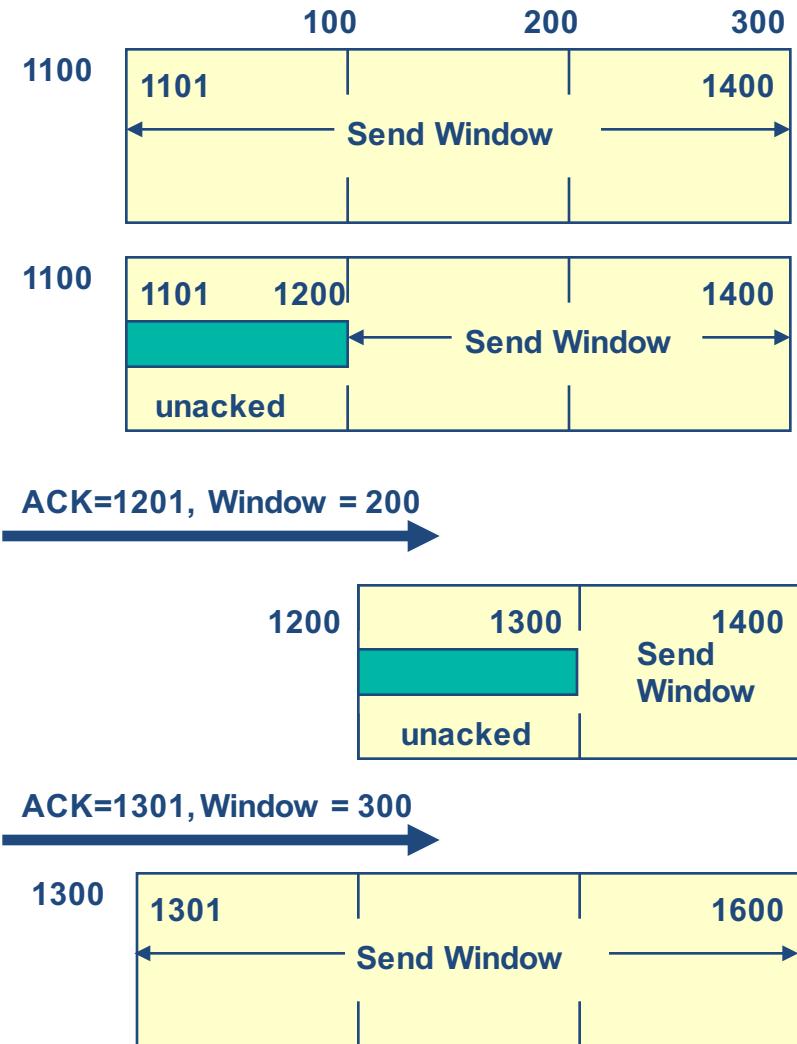
Controllo di flusso: lato ricevitore

- *Receive Window (RCWND)*: spazio del buffer in ricezione disponibile per ricevere nuovi dati
- RCWND si estende dall'ultimo byte inoltrato all'applicazione fino alla fine del buffer
- Il buffer di ricezione può riempirsi, per esempio, a causa di congestione nel sistema operativo del ricevitore
- La dimensione di *RCWND* è segnalata in ogni segmento inviato dal ricevitore al trasmettitore



Controllo di flusso: lato trasmettitore

- *Send Window (SNDWND)*: parte inutilizzata del buffer, rappresenta i byte che possono essere trasmessi senza attendere ulteriori riscontri
- Il trasmettitore mantiene un buffer di trasmissione che tiene traccia di
 - Dati che sono stati trasmessi ma non ancora riscontrati
 - Dimensione della finestra di ricezione del partner
- Il buffer di trasmissione si estende dal primo byte non riscontrato all'estremo a destra della finestra di ricezione del ricevitore



Problemi con la finestra

- *Silly window syndrome* - lato ricevitore:
 - Il ricevitore svuota lentamente il buffer di ricezione
 - Invia segmenti con finestra molto piccola
 - Il trasmettitore invia segmenti corti con molto overhead
- Soluzione (algoritmo di Clark)
 - Il ricevitore “mente” al trasmettitore indicando una finestra nulla sino a che il suo buffer di ricezione non si è svuotato per metà o per una porzione almeno pari al MSS

Finestra = max(1/2 Receive_Buffer_Size, Maximum_Segment_Size)



Problemi con la finestra

- *Silly window syndrome* - lato trasmettitore:
 - L'applicazione genera dati lentamente
 - Invia segmenti molto piccoli man mano che vengono prodotti
- Soluzione (algoritmo di Nagle)
 - Il TCP sorgente invia la prima porzione di dati anche se corta
 - Gli altri segmenti vengono generati e inviati solo se
 - Il buffer d'uscita contiene dati sufficienti a riempire un MSS
 - Oppure, quando si riceve un *acknowledgement* per un segmento precedente.



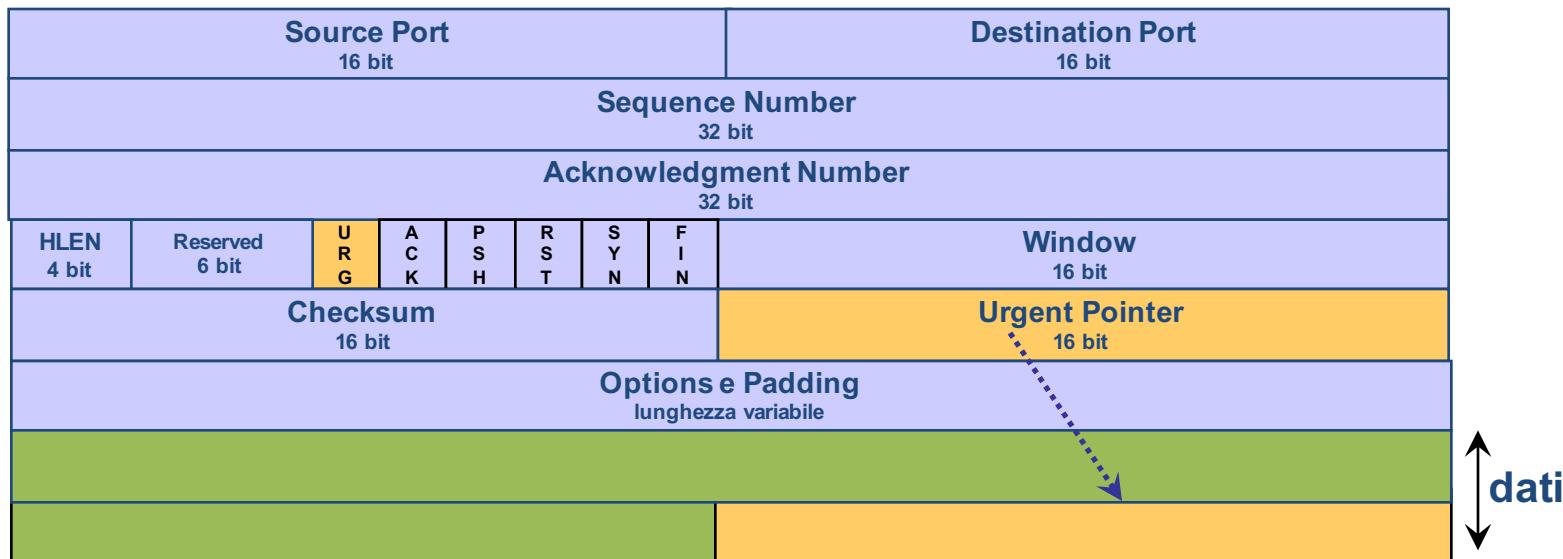
Funzione Push

- TCP originariamente prevedeva una gestione "speciale" per i dati che richiedono di essere **immediatamente** consegnati all'applicazione ricevente
- Per ottenere un inoltro immediato dei dati da parte del TCP ricevente all'applicazione ricevente, l'applicazione inviante può inviare un comando di PUSH
 - Viene settato il flag di PUSH nel segmento
- In verità l'uso del PUSH **non è generalmente implementato** nel comando *send* delle interfacce TCP offerte dai linguaggi di programmazione
 - Viene automaticamente settato da TCP (sistema operativo) nell'ultimo segmento che svuota il buffer



Dati URGENT

- Alternativamente, la modalità di funzionamento di TCP può essere cambiata in URGENT
- Confusione negli RFC
 - RFC 793 (1981): puntatore all'inizio del primo byte dopo i dati urgenti
 - RFC 1122 (1989): puntatore all'inizio dell'ultimo byte dei dati urgenti
- Nonostante RFC successivi ribadiscano il contenuto di RFC 1122, molte implementazioni di TCP seguono ancora il vecchio RFC
 - Inoltre, sistemi operativi differenti implementano URGENT in maniera differente
 - **Consiglio:** semantica e implementazione dubbie, meglio non usare.



Livello di Trasporto

- Introduzione
- Protocollo UDP
- Trasporto affidabile
 - Protocolli di ritrasmissione
 - Controllo di flusso a finestra mobile
- Protocollo TCP
 - Generalità
 - Formato e connessioni
 - Controllo di flusso
 - Controllo d'errore
 - Controllo di congestione



Controllo d'errore

- Il meccanismo di controllo d'errore del TCP serve a recuperare pacchetti persi in rete
- La causa principale della perdita è l'*overflow* di una delle code dei *router* attraversati a causa della congestione
- Il meccanismo di ritrasmissione è di tipo **Go-back-N con Timeout**
 - Differenze:**
 - TCP mantiene nel buffer anche i segmenti fuori ordine
 - Quando arrivano i segmenti mancanti la finestra scorre in avanti fino al primo segmento non riscontrato in quelli ricevuti fuori ordine
 - Viene mandato un ACK che riscontra collettivamente anche i segmenti fuori ordine



- La finestra di trasmissione (valore di N) dipende dal meccanismo di controllo di flusso e di congestione



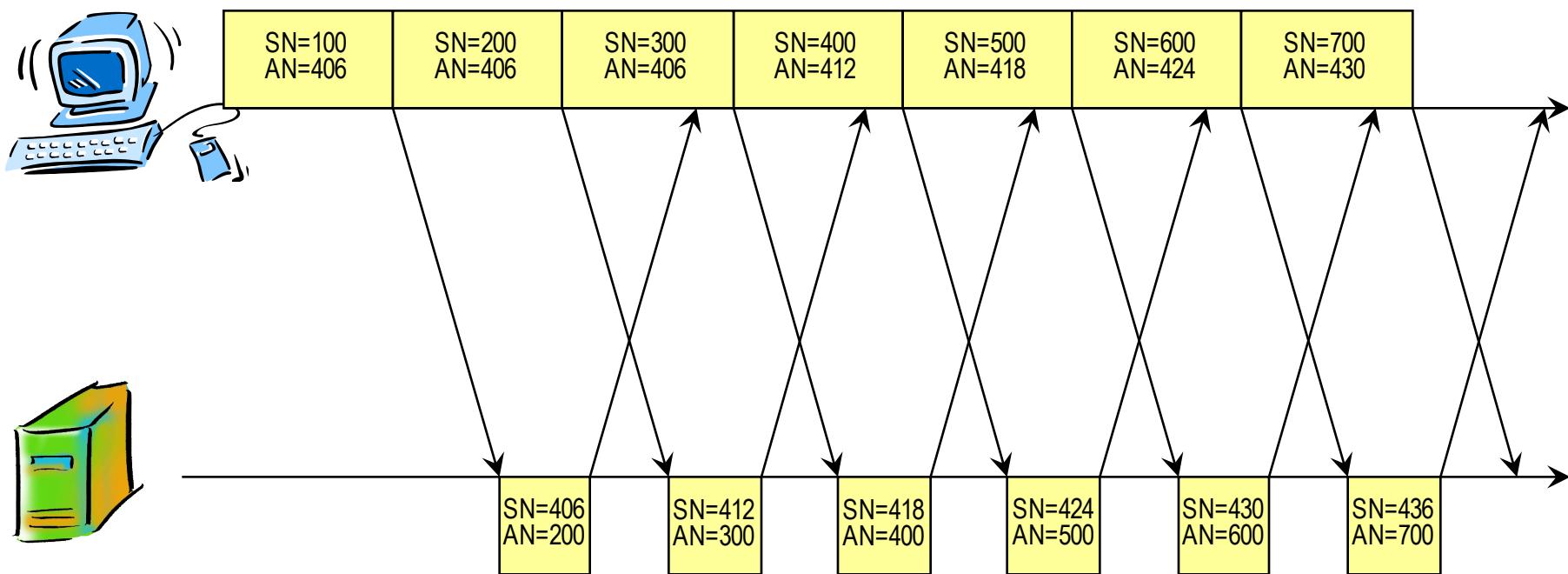
Controllo d'errore

esempio 1: senza errori

MSS = 100 byte

ACK = 6 byte

Window = 4 MSS

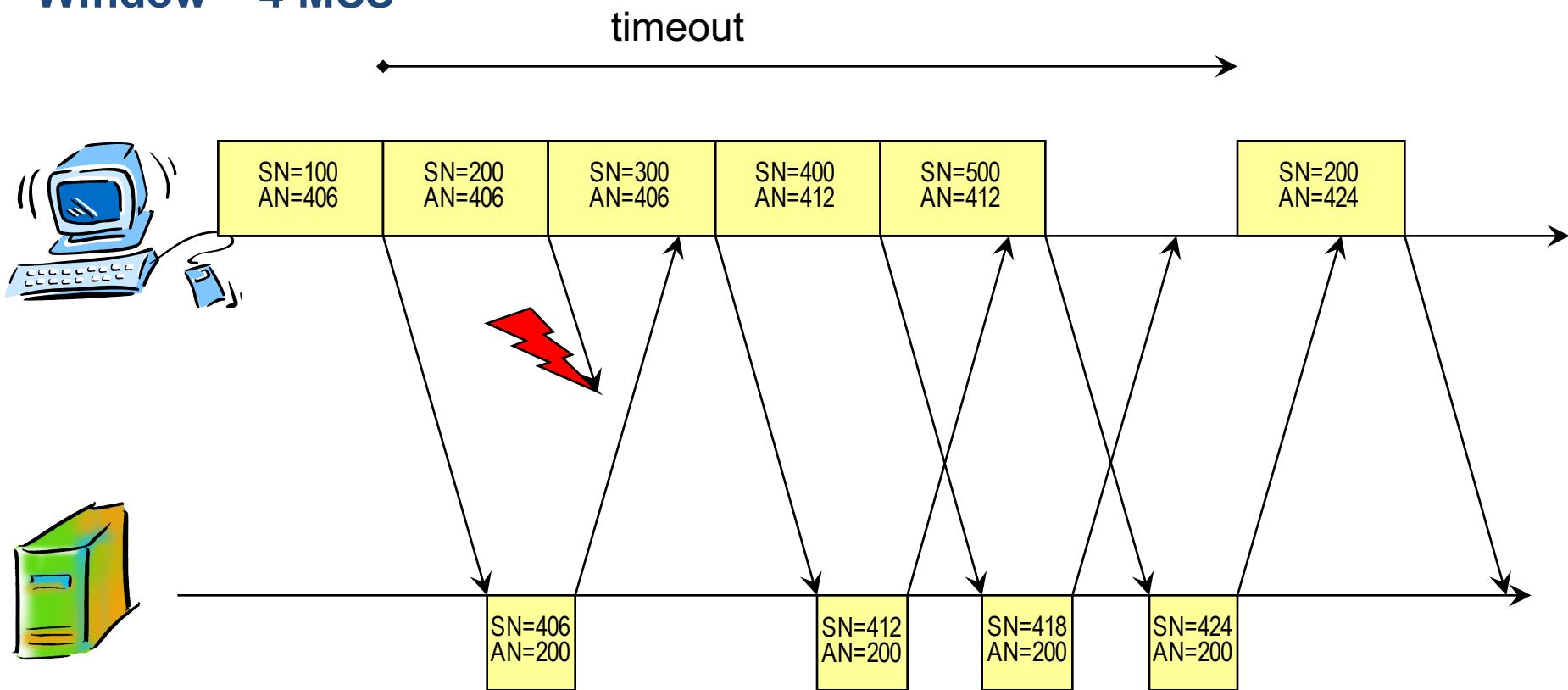


Controllo d'errore

esempio 2: errore nei dati

MSS = 100 byte

Window = 4 MSS

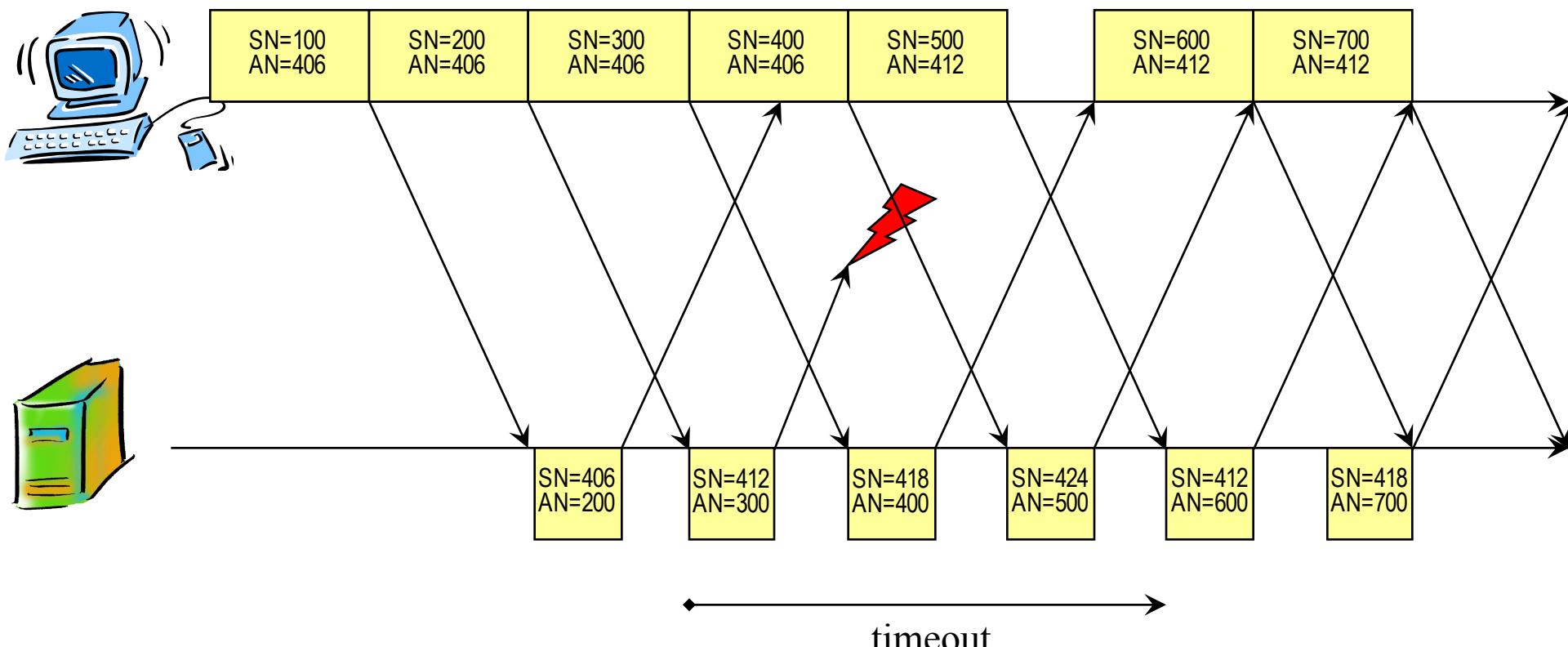


Controllo d'errore

esempio 3: errore nell'ACK

MSS = 100 byte

Window = 4 MSS



Wireshark: Protocollo TCP (1)

- File cattura : `tcp-ethereal-trace-1`
- Attività:
 - Quale è il segmento di apertura SYN?
 - Come lo riconosco ? Che SN ha?
 - Quale è il segmento di apertura SYN-ACK?
 - Come lo riconosco? Che SN ha? Cosa contiene il campo ACK?
 - Esaminare i pacchetti dal numero 4 al numero 9
 - Che relazione c'è tra SN e AN ?
 - Indicare la lunghezza dei primi 6 segmenti
 - Ci sono ritrasmissioni?
 - Usare *Statistics->TCP Stream Graph- > Time- Sequence-Graph (Stevens)* evidenziando il segmento SYN

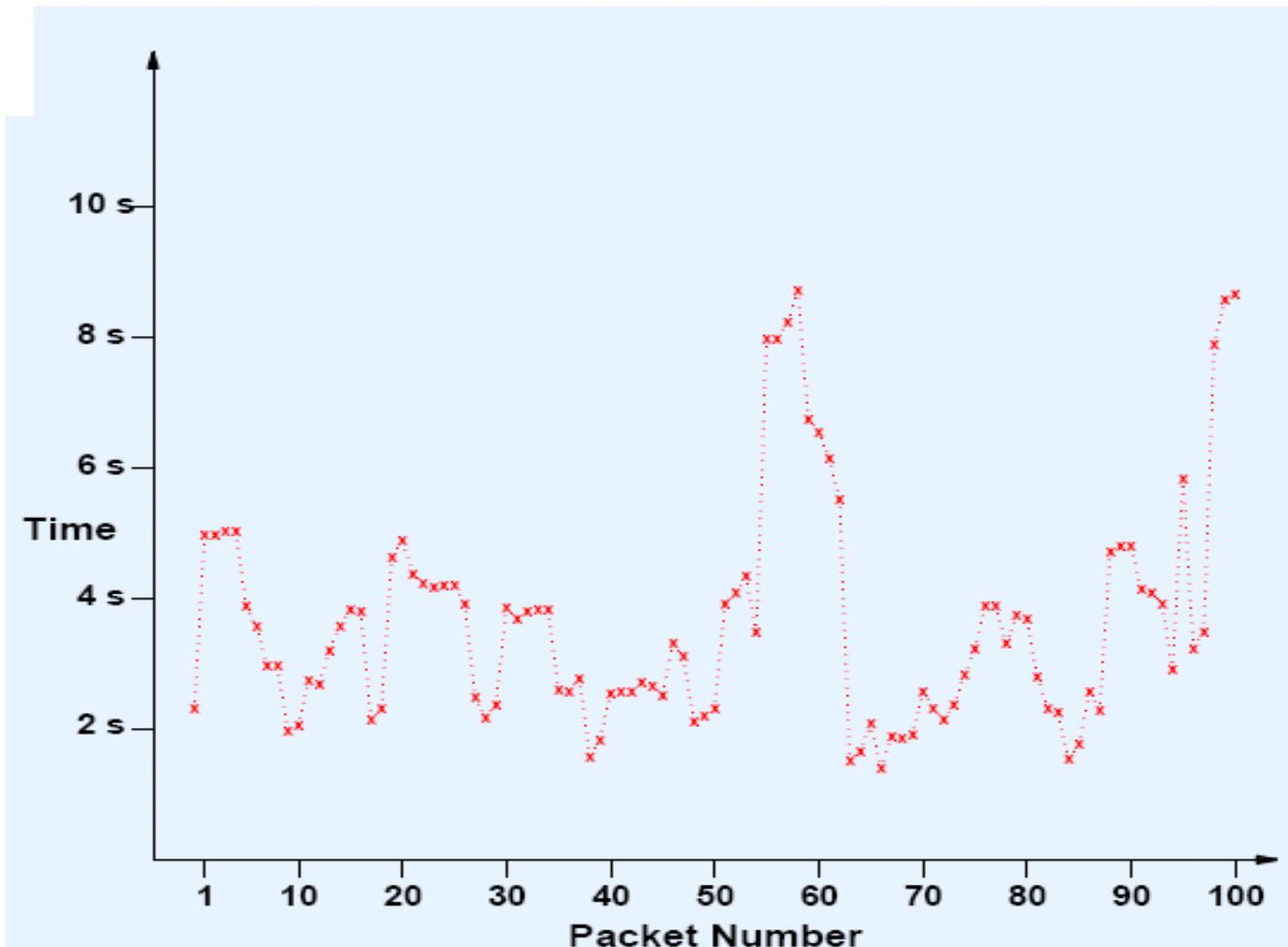


Gestione del Time-Out

- Uno dei problemi è stabilire il valore ottimo del *timeout*:
 - *Timeout* troppo breve, il trasmettitore riempirà il canale di ritrasmissioni di segmenti,
 - *Timeout* troppo lungo impedisce il recupero veloce di reali errori
- Il valore ottimale dipende fortemente dal ritardo in rete (rete locale o collegamento satellitare?)
- Il TCP calcola dinamicamente un valore opportuno per il *timeout* stimando il RTT (*Round Trip Time*)



Variabilità RTT



Stima del RTT

- Il TCP adatta il timeout di trasmissione alle condizioni reali della rete tramite gli algoritmi di **Karn e Jacobson**
- I campioni di round-trip-time $\{RTT^{(i)}\}$ sono definiti come il tempo che passa tra la trasmissione di un segmento e la ricezione del relativo riscontro

Stima del valor medio

- Sulla base delle misure il *sender* TCP calcola lo *Smoothed Round Trip Time* (SRTT) tramite l'algoritmo di Jacobson

$$SRTT^{(i)} = (1-\alpha) SRTT^{(i-1)} + \alpha RTT^{(i)}.$$

- Con α compreso tra 0 e 1 (tipicamente 1/8)



Stima del RTT

Stima della deviazione standard

- Oltre al valor medio viene anche stimata la deviazione standard dei RTT

$$DEV = |RTT^{(i)} - SRTT^{(i-1)}|$$

- Anche delle deviazione standard viene calcolato un valore filtrato (*smoothed*):

$$SDEV^{(i)} = 3/4 \ SDEV^{(i-1)} + 1/4 \ DEV$$



Calcolo del Time Out

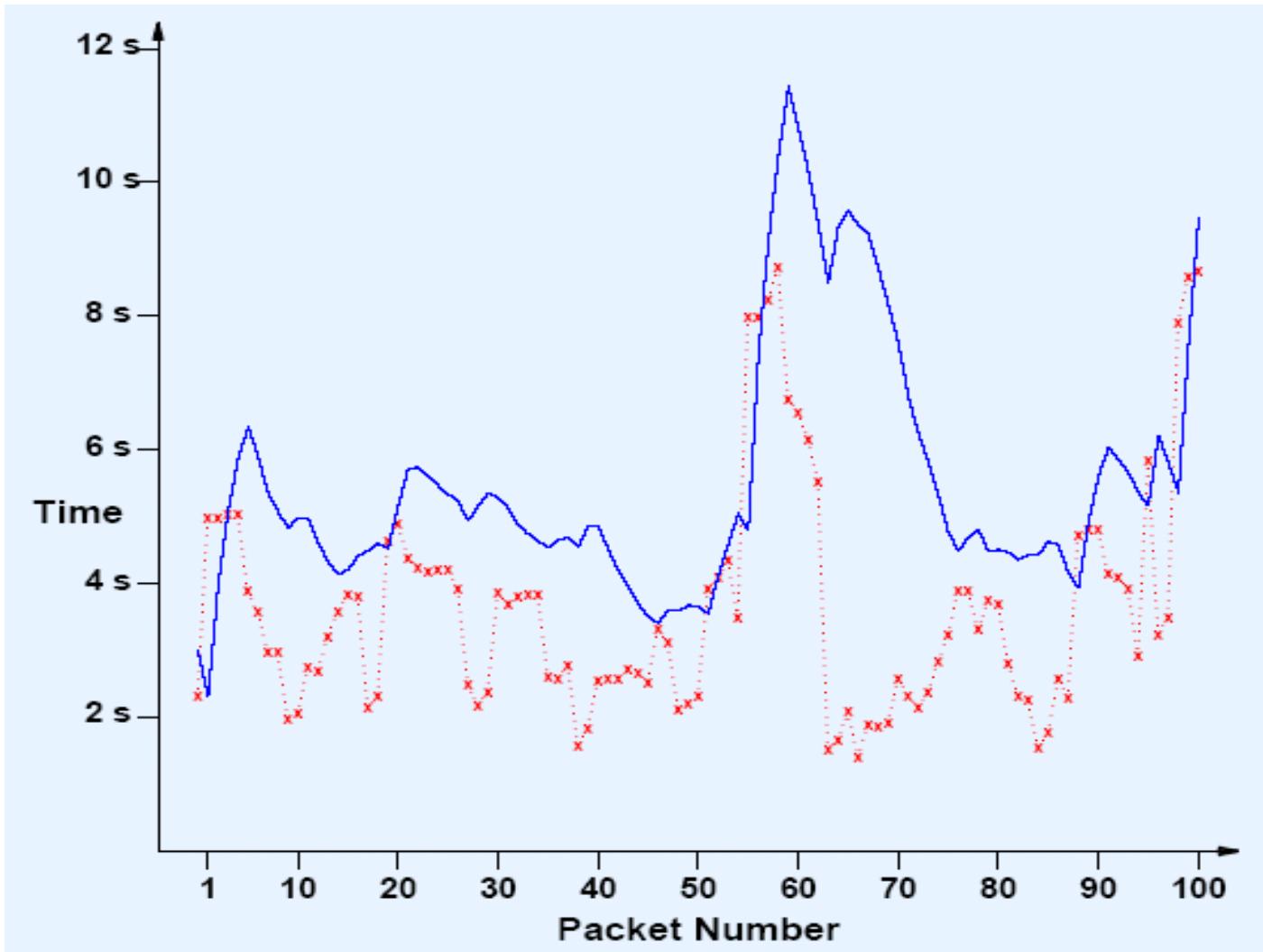
- Sulla base dei valori stimati il timeout è calcolato come

$$\text{TIMEOUT} = \text{SRTT} + 4 \text{ SDEV}$$

- All'inizio il timeout parte da 1 s
- A seguito di una ritrasmissione è meglio passare all'algoritmo di Karn:
 - RTT non viene aggiornato
 - Il timeout è moltiplicato per un fattore fisso (tipicamente 2)
 - Il *timeout* cresce fino ad un valore massimo
 - Dopo un numero massimo di ritrasmissioni la connessione viene chiusa



Qualità della stima RTT



Persistenza

- Se il destinatario fissa a zero la finestra di ricezione la sorgente TCP interrompe la trasmissione
- La trasmissione riprende quando il destinatario invia un ACK con una dimensione della finestra diversa da zero
- Nel caso in cui questo ACK andasse perso la connessione rimarrebbe bloccata
- Per evitare questa situazione si usa un *timer di persistenza* che viene attivato quando arriva un segmento con finestra nulla
- Se il timer di persistenza scade (valore di timeout uguale a quello di ritrasmissione) viene inviato un piccolo segmento di sonda (probe)
- Se viene ricevuto un ACK si esce dallo stato critico altrimenti al nuovo scadere del timeout si invia un altro probe



Livello di Trasporto

- Introduzione
- Protocollo UDP
- Trasporto affidabile
 - Protocolli di ritrasmissione
 - Controllo di flusso a finestra mobile
- Protocollo TCP
 - Generalità
 - Formato e connessioni
 - Controllo di flusso
 - Controllo d'errore
 - Controllo di congestione



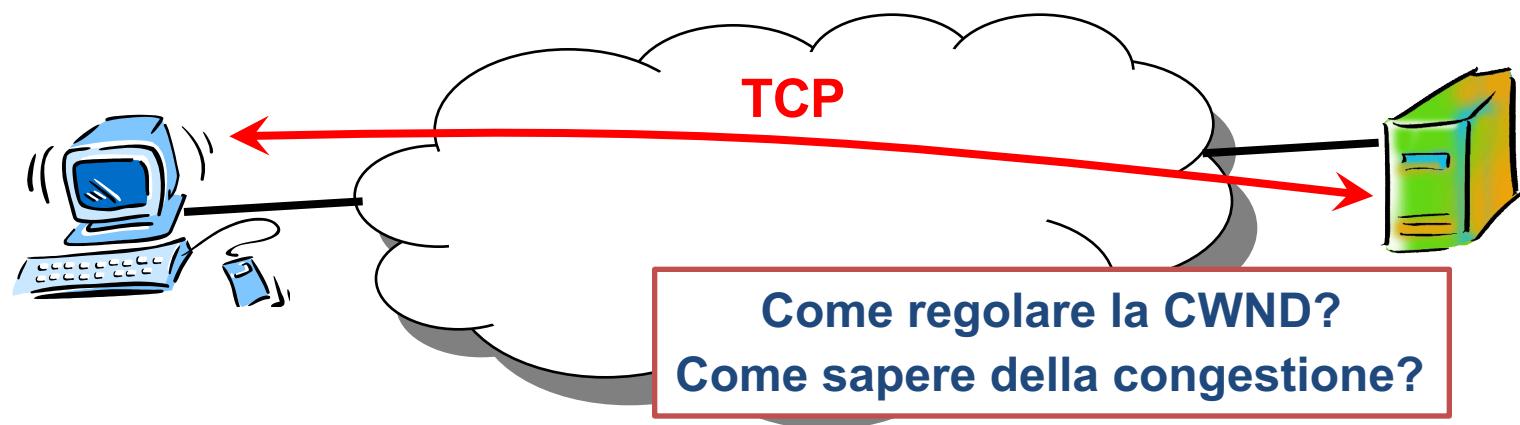
Controllo di congestione

- Il controllo di flusso
 - Dipende **solo** dalla “capacità” del ricevitore
 - Non è sufficiente ad evitare la congestione nella rete
- Nella rete INTERNET attuale non ci sono meccanismi sofisticati di controllo di congestione a livello di rete (come ad esempio meccanismi di controllo del traffico in ingresso)
- Il controllo di congestione è delegato al TCP!!!
 - Se il traffico in rete porta a situazioni di congestione il TCP deve ridurre velocemente il traffico in ingresso
- Essendo il TCP implementato solo negli *host* il controllo di congestione è di tipo *end-to-end*



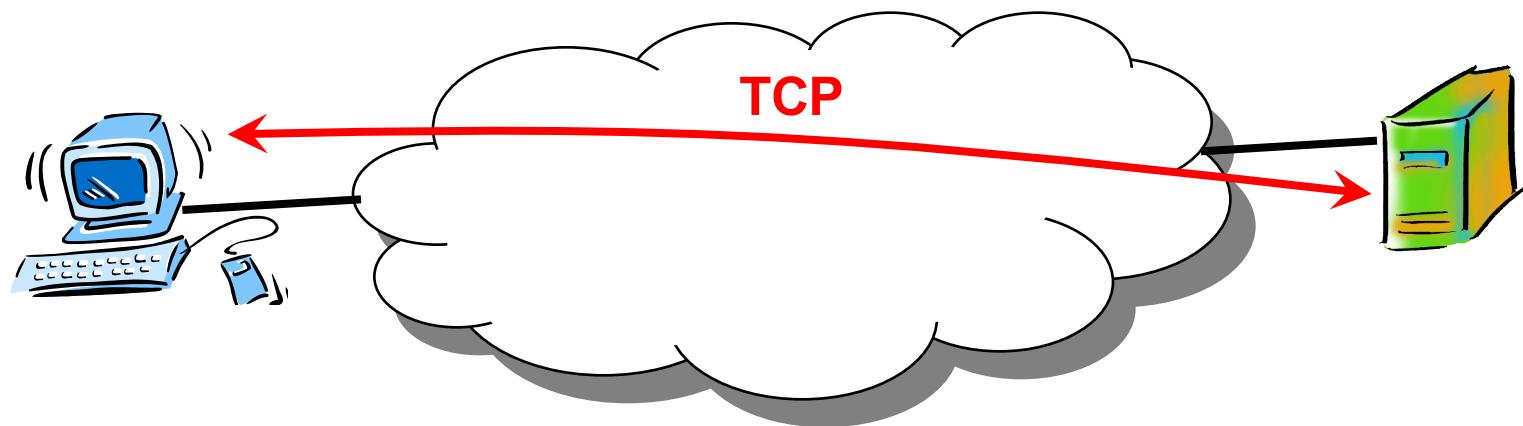
Controllo di congestione

- Il modo più naturale per controllare il ritmo di immissione in rete dei dati per il TCP è quello di regolare la finestra di trasmissione
- Il trasmettitore mantiene una *Congestion Window* (CWND) che varia in base agli eventi che osserva (ricezione ACK, timeout)
- Il trasmettitore non può trasmettere più del minimo tra RCVWND e CWND



Controllo di congestione

- L'idea base del controllo di congestione del TCP è quella di interpretare la perdita di un segmento, segnalata dallo scadere di un timeout di ritrasmissione, come un evento di congestione
- La reazione ad un evento di congestione è quella di ridurre la finestra (*CWND*)



Slow Start & Congestion Avoidance

- Il valore della finestra CWND viene aggiornato dal trasmettitore TCP in base ad un algoritmo
- Il modo in cui avviene l'aggiornamento dipende dalla fase (o stato) in cui si trova il trasmettitore
- Esistono due fasi fondamentali:



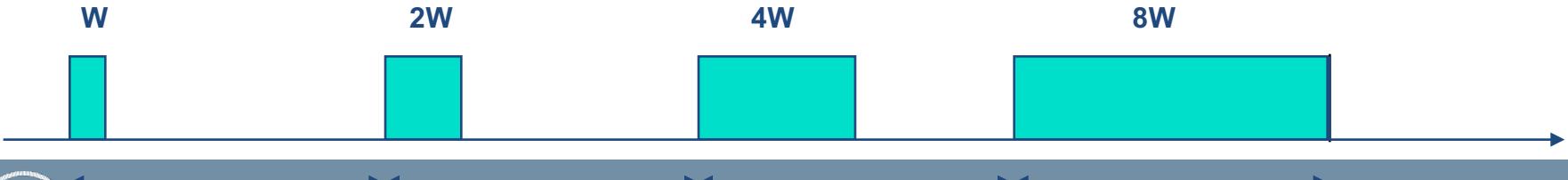
- Slow Start
- Congestion Avoidance

- La variabile STHRESH è mantenuta al trasmettitore per distinguere le due fasi:
 - ➔ se $CWND < STHRESH$ si è in *Slow Start*
 - ➔ se $CWND > STHRESH$ si è in *Congestion Avoidance*



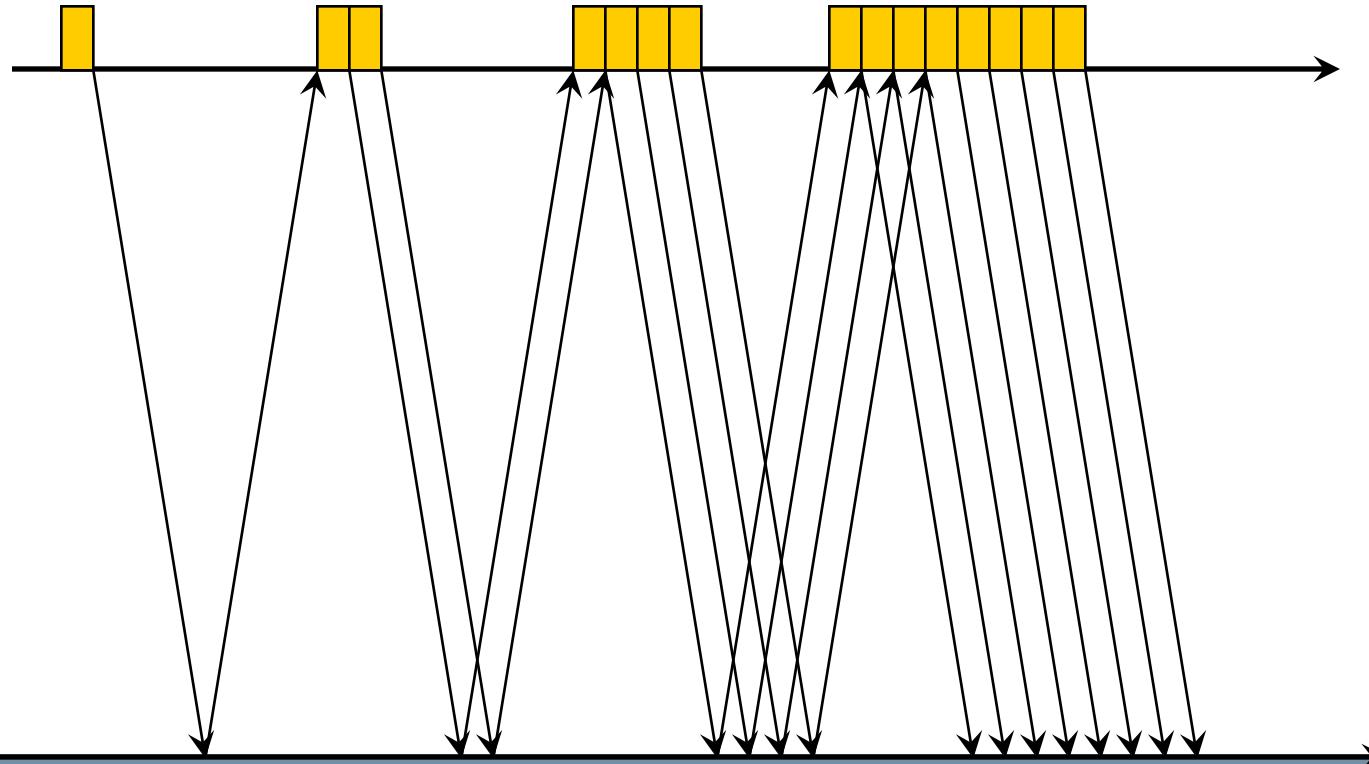
Slow Start

- All'inizio, il trasmettitore pone la CWND a 1 segmento (MSS) e la SSTHRESH ad un valore di *default* molto elevato
- Essendo CWND < SSTHRESH si parte in *Slow Start*
- *In Slow Start:*
 - La CWND viene *incrementata di 1 per ogni ACK ricevuto*
- Si invia un segmento e dopo RTT si riceve l'ACK, si pone CWND a 2 e si inviano 2 segmenti, si ricevono 2 ACK, si pone CWND a 4 e si inviano 4 segmenti, ...



Slow Start

- Al contrario di quanto il nome faccia credere l'incremento della finestra avviene in modo esponenziale (raddoppia ogni RTT)



Slow Start

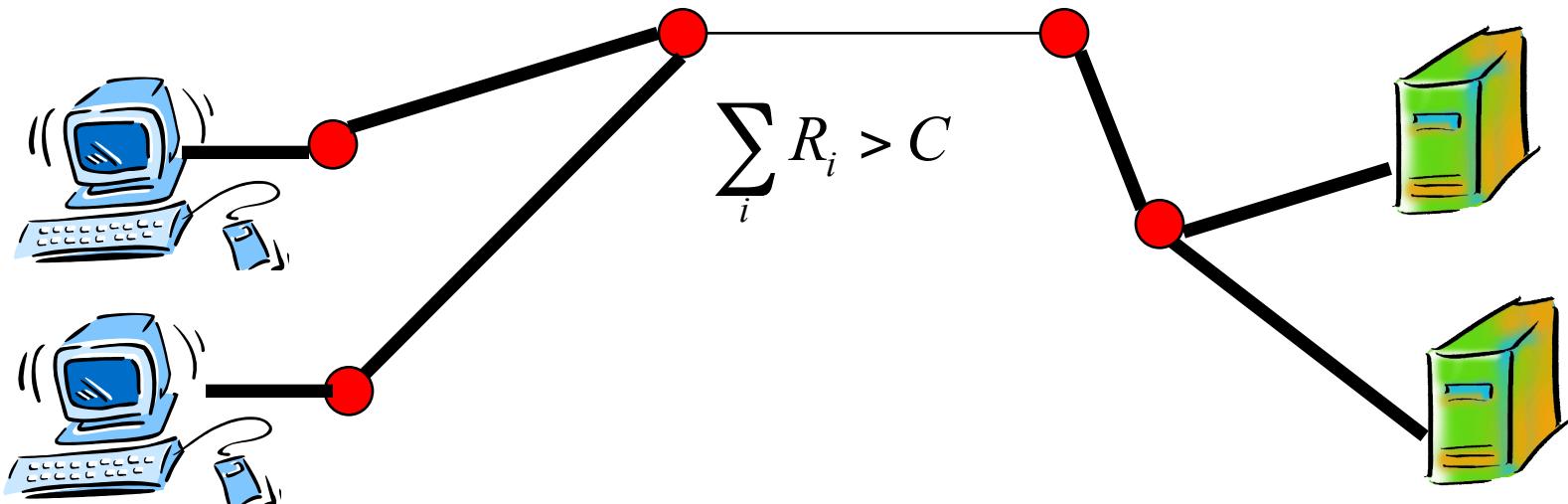
- L'incremento può andare avanti fino
 - Primo evento di congestione
 - Fino a che CWND < SSTHRESH
 - CWND < RCWND
- Insieme alla finestra aumenta il ritmo (o rate) di trasmissione che può essere stimato come:

$$R = \frac{CWND}{RTT} \text{ [bit/s]}$$



Evento di Congestione

- Un evento di congestione si verifica quando il ritmo di trasmissione porta in congestione un link sul percorso in rete verso la destinazione
- Un link è congestionato quando la somma dei ritmi di trasmissione dei flussi che lo attraversano è maggiore della sua capacità



Evento di congestione

- Scade un timeout di ritrasmissione
 - Il TCP reagisce ponendo SSTHRESH uguale alla metà dei “byte in volo” (byte trasmessi ma non riscontrati); più precisamente
 - E ponendo CWND a 1MSS
- Si noti che di solito i “byte in volo” sono con buona approssimazione pari all’ultima CWND



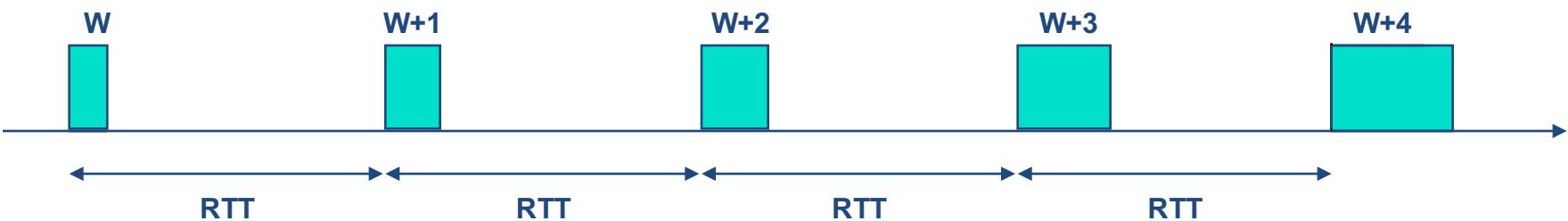
Evento di congestione

- Come risultato:
 - CWND è minore di SSTHRESH e si entra nella fase di Slow Start
 - Il trasmettitore invia un segmento e la sua CWND è incrementata di 1 ad ogni ACK
- Il trasmettitore trasmette tutti i segmenti a partire da quello per cui il timeout è fallito (politica Go-Back-N)
- Il valore a cui è posta la SSTHRESH è una stima della finestra ottimale che eviterebbe futuri eventi di congestione



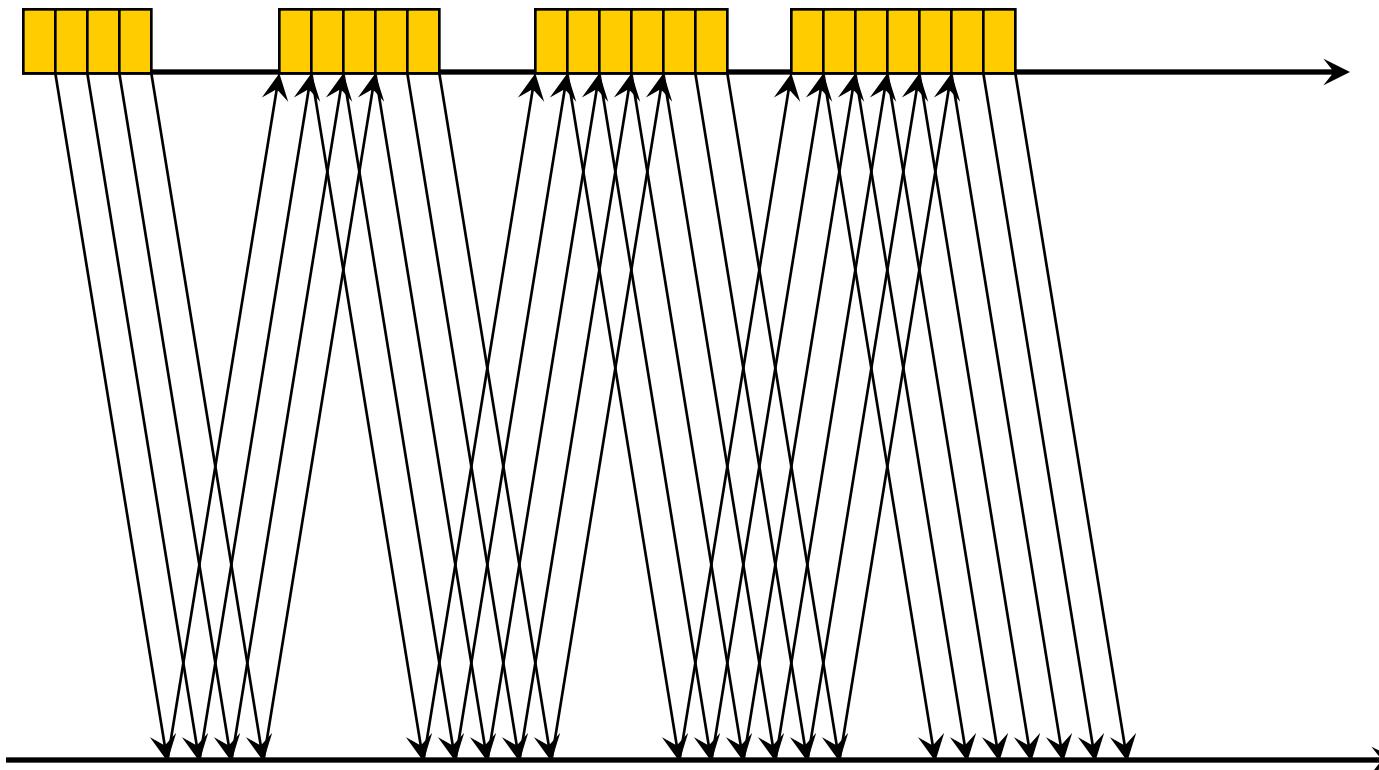
Congestion Avoidance

- Lo slow start continua fino a che CWND diventa grande come SSTHRESH e poi parte la fase di *Congestion Avoidance*
- Durante il *Congestion Avoidance*:
 - Si incrementa la CWND di $1/\text{CWND}$ ad ogni ACK ricevuto
- Se la CWND consente di trasmettere N segmenti, la ricezione degli ACK relativi a tutti gli N segmenti porta la CWND ad aumentare di 1 segmento
- In *Congestion Avoidance* si attua un incremento lineare della finestra di congestione

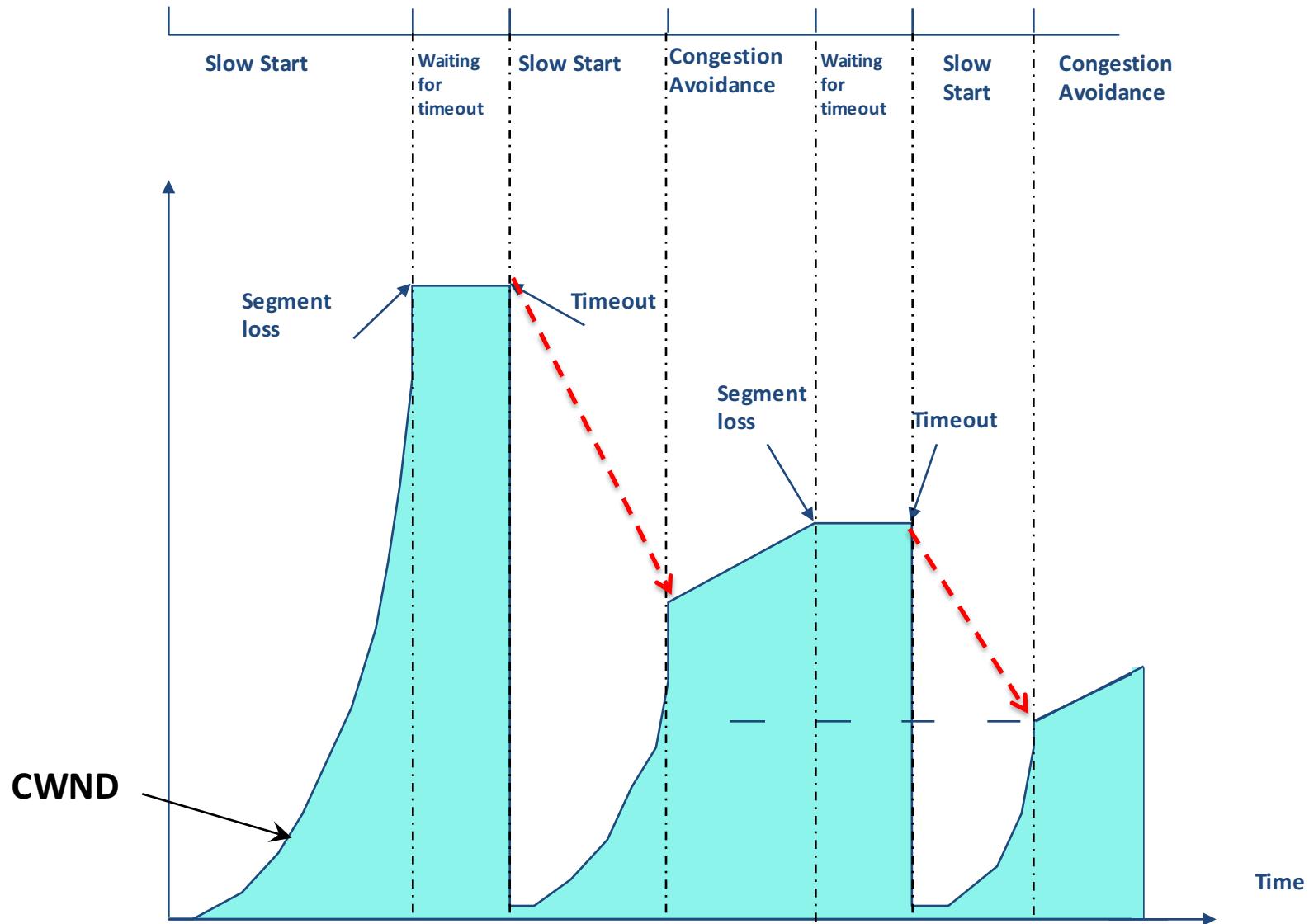


Congestion Avoidance

- Dopo aver raggiunto STHRESH la finestra continua ad aumentare ma molto più lentamente



Esempio di funzionamento



Wireshark: Protocollo TCP (2)

- File cattura : `tcp-ethereal-trace-1`
- Attività:
 - Guardare i valori di RTT tramite il grafico *Statistics->TCP Stream Graph- >Round Trip Time Graph*
 - Analizzare lo stream scambiato dalla connessione TCP usando *Analyze -> Follow TCP Stream* evidenziando il segmento SYN
 - Confrontarlo poi con il payload del pacchetto numero 4
 - Possiamo vedere le fasi di *Slow Start* e *Congestion Avoidance*?
 - C'è qualche vincolo sulla lunghezza della finestra?



Condivisione equa delle risorse

- Si può mostrare che in condizioni ideali il meccanismo di controllo del TCP è in grado di
 - Limitare la congestione in rete
 - Consentire di dividere in modo equo la capacità dei link tra i diversi flussi
- Le condizioni ideali sono alterate tra l'altro da
 - Differenti RTT per i diversi flussi
 - Buffer nei nodi minori del prodotto banda-ritardo

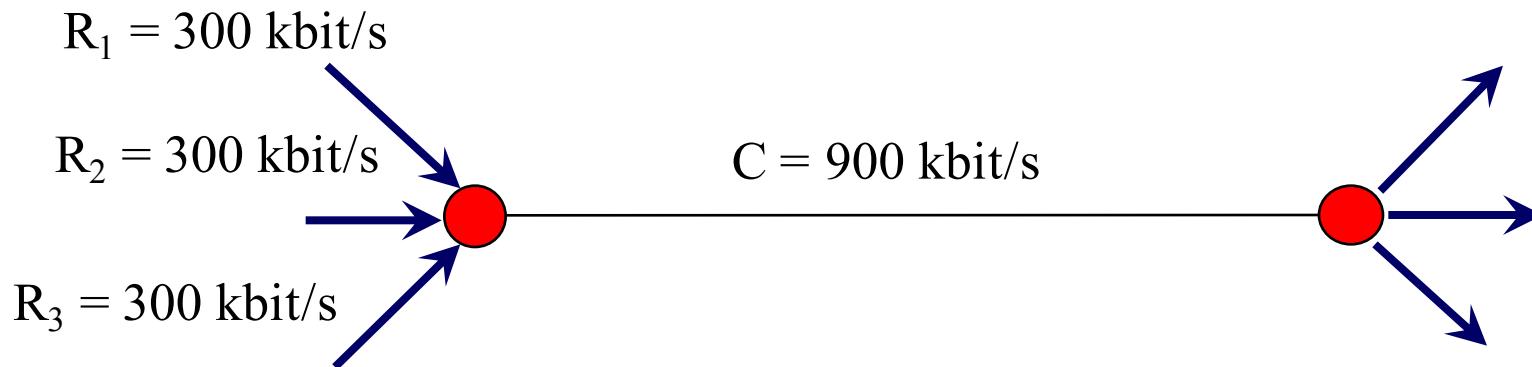


Applet TCP Congestion Control
http://media.pearsoncmg.com/aw/aw_kurose_network_4/applets/fairness/index.html

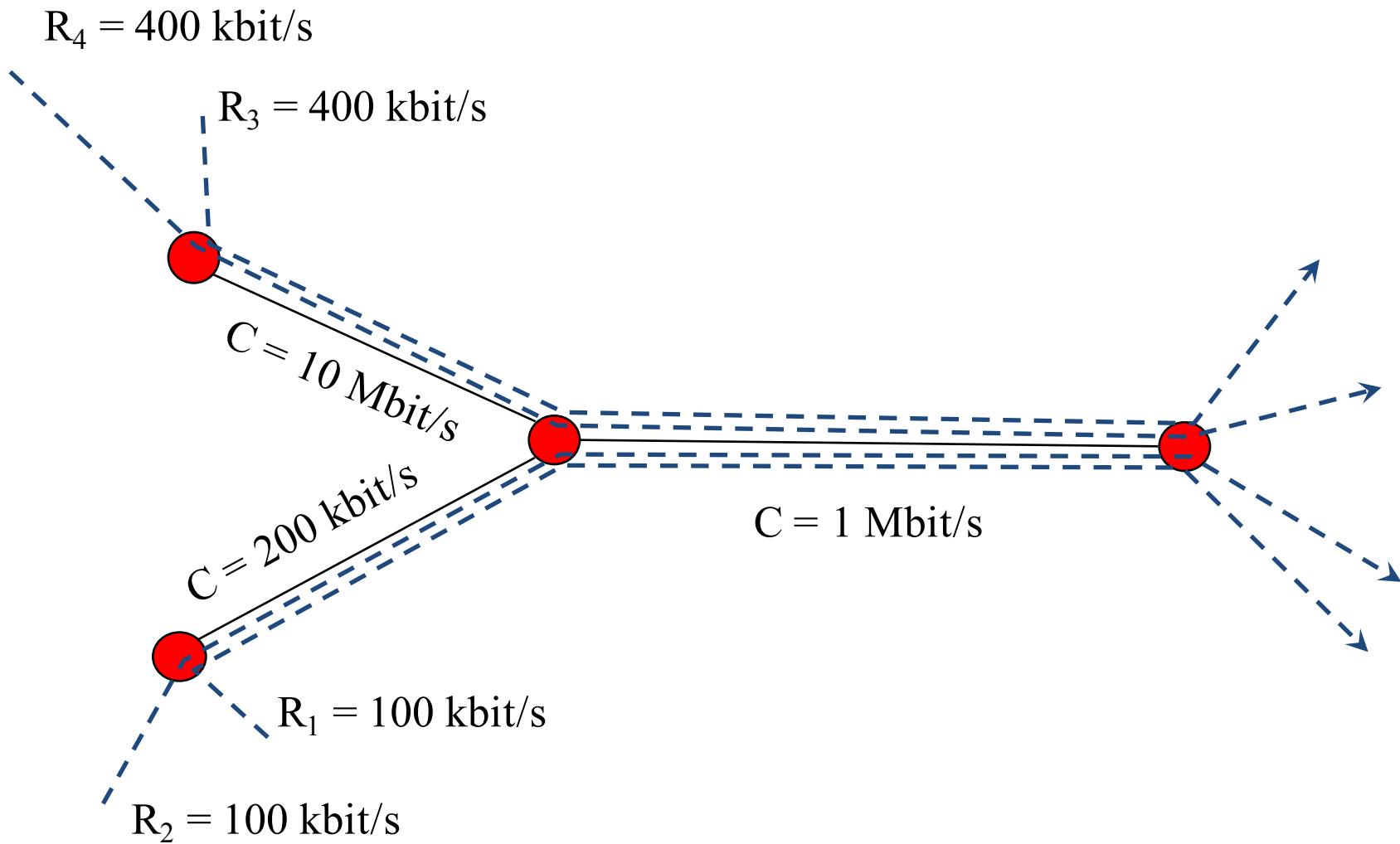


Condivisione equa delle risorse

- I valori dei rate indicati sono solo valori medi e valgono solo in condizioni ideali
- Il ritmo di trasmissione in realtà cambia sempre e in condizioni non ideali la condivisione può non essere equa

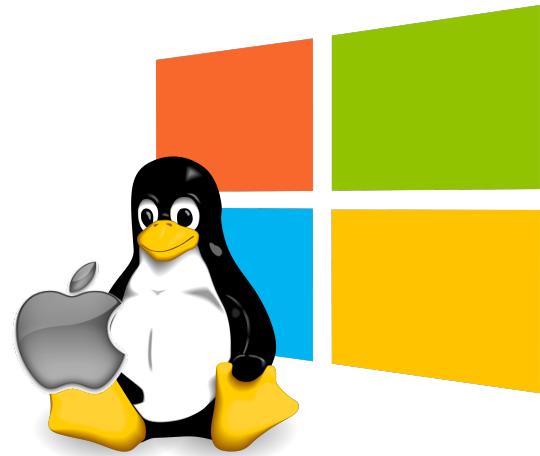


Condivisione equa delle risorse



TCP e sistemi operativi

- Esistono diverse versioni di protocollo TCP
- TCP è implementato nel sistema operativo
- Versione base è TCP Tahoe, quella vista



Famiglia Windows

- *Reno / New Reno: fast retransmit/fast recovery*
- SACK - Selective ACK: accetto e riscontro fuori ordine
- ECN - *Explicit Congestion Notification*: ECN-capable router marchiano pacchetti gestiti in situazioni di congestione
- F-RTO - *Forward RTO-Recovery*: Cerca di limitare l'effetto della scadenza di *timeout* spuri, non dovuti a congestione, guardando gli ACK successivi alla scadenza di un *timeout*
- CTCP - Compound TCP: Nuovo TCP ottimizzato per connessioni con grandi prodotti banda-ritardo, utilizza la variazione del RTT per individuare l'inizio di una congestione



TCP e sistemi operativi (cont'd)

Famiglia Linux

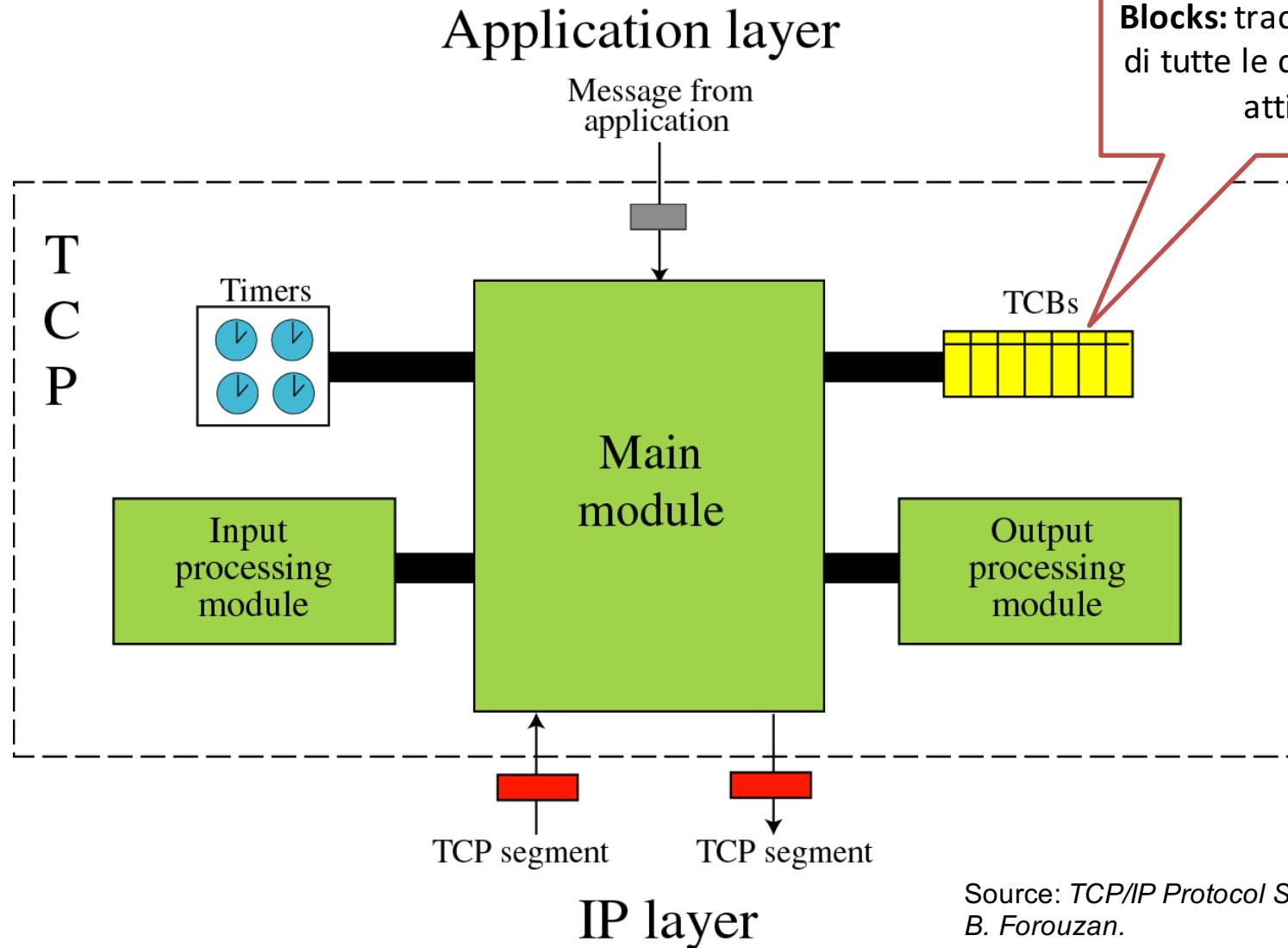
- BIC - *Binary Increase Congestion* control (fino a 2.6.18)
 - TCP ottimizzato per grandi prodotti banda ritardo, cerca di trovare la finestra migliore con una ricerca binaria dopo una congestione
- CUBIC - *Enhanced version of BIC* (fino a 2.6.19 a 3.1)
 - TCP ottimizzato per grandi prodotti banda ritardo, finestra è una funzione cubica del tempo trascorso dall'ultimo evento di congestione, non dipende da ACK
- PRR - *Proportional Rate Reduction* (da 3.2)
 - Nuovo meccanismo di *fast recovery*

Famiglia MacOS

- Protocollo proprietari: MacTCP (fino a 7.5.1) e Open Transport (7.5.2 – 9.2.2)
- CUBIC da 10.10 in poi
- Dalla 10.11 è abilitato di *default* anche ECN



Un modulo TCP



Source: *TCP/IP Protocol Suite*,
B. Forouzan.

