Reti di Telecomunicazioni

Transport Layer

TCP e UDP

Autori

Queste slides sono state scritte da

Michele Michelotto:

michele.michelotto@pd.infn.it

che ne detiene i diritti a tutti gli effetti

Copyright Notice

Transport layer

Introduzione
Layer: Modello OSI e TCP/IP
Physics Layer
Data Link Layer
MAC sublayer
Network Layer
Transport Layer
Application Layer

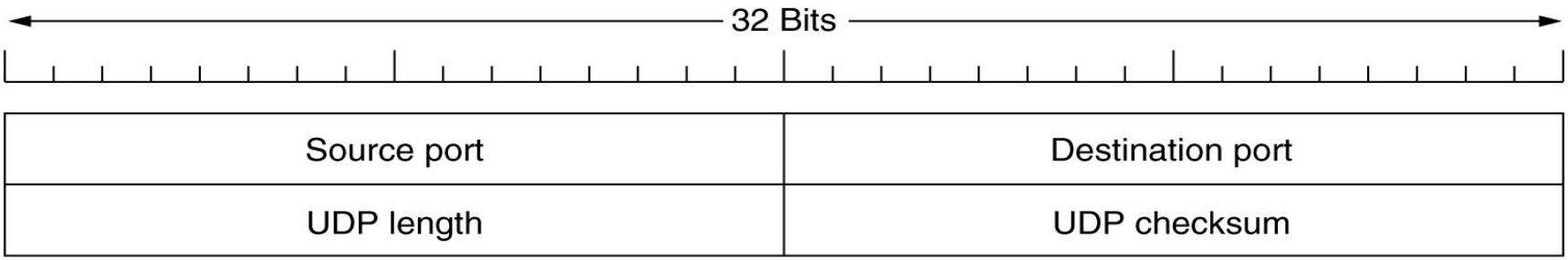
Trasporto di internet

- Internet ha due protocolli di trasporto principali
  - Uno connectionless: UDP
    - User Datagram Protocol
  - Uno connection oriented: TCP
    - Trasmissione Control Protocol

UDP – RFC 768

- Molto semplice. In pratica permette alle applicazioni di mandare datagram IP incapsulati senza stabilire una connessione
  - Header di 8 byte, seguito dal payload
  - I numeri di porta servono per identificare i punti terminali nelle macchine sorgente e destinazione
  - Quando il pacchetto arriva viene consegnato al processo legato alla porta di destinazione
  - Questo legame viene fatto con una operazione tipo BIND
  - In pratica il valore aggiunto di UDP rispetto ad usare pacchetti IP nudi è quello di avere le porte di sorgente e destinazione

UDP – RFC 768



- La porta sorgente serve quando devo mandare indietro un reply
  - Copiando il campo **Source port** del segmento entrante nel campo **Destination Port** del segmento di risposta
- Il campo **UDP length** comprende header e dati
- UDP checksum è opzionale e messo a zero se non calcolato
  - Non conviene disabilitarlo a meno che la qualità dei dati non interessi, per esempio con voce digitalizzata

UDP non fa:

- Meglio esplicitare che cosa UDP non fa:
  - Non fa flow control, controllo degli errori e ritrasmissione. Tutto questo spetta al processo dell'utente
- In pratica si limita a fornire una interfaccia a IP con il demultiplexing di diversi processi su tante porte
- Client server
  - UDP è utile nelle applicazione client server, in cui il client manda una piccola richiesta al server e si aspetta una piccola risposta indietro
  - Non importa se la richiesta o la risposta non arrivano. Eventualmente il client ripete la richiesta. Il codice è più semplice e ci si scambiano meno messaggi
  - Es Il protocollo DNS funziona in questo modo

Procedure Call

- Procedure Call
  - Mandare un messaggio ad un host remoto e avere una risposta è simile a chiamare una funzione in un sistema operativo
  - Questo porta ad implementare interazioni di **richiesta – risposta** sotto forma di chiamate a funzioni.
  - Le applicazioni di rete diventano quindi più facili da programmare, nascondendo al programmatore i dettagli della rete

RPC

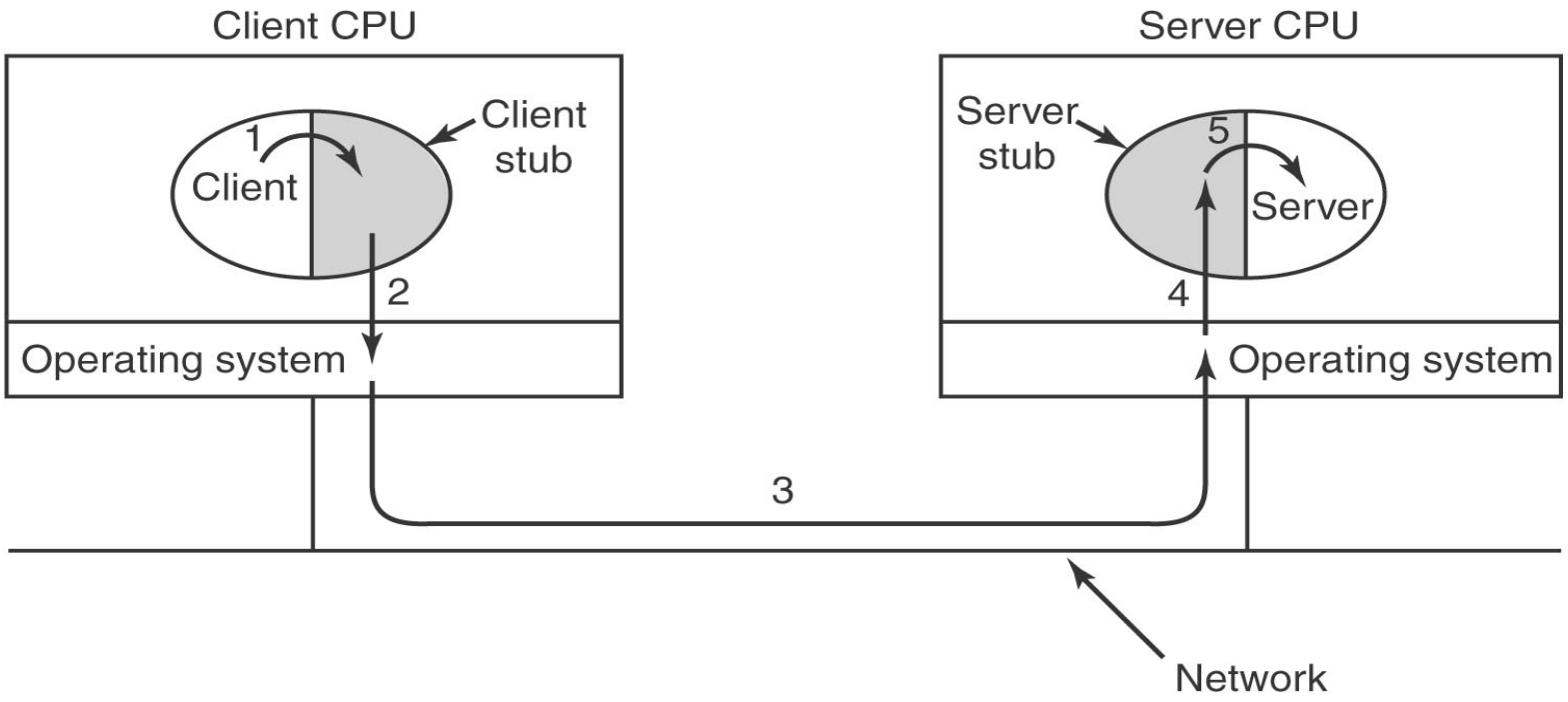
- Remote Procedure Call
  - Quando la macchina 1 chiama una funzione sulla macchina 2, il processo chiamante si blocca e l'esecuzione della funzione avviene sulla macchina 2.
  - Informazioni possono essere trasportate dal chiamante al chiamato nei parametri e possono tornare indietro come risultato della funzione (procedure)
  - Il programmatore non vede scambi di messaggi

Trasparenza

- Si vorrebbe che la chiamata remota assomigli il più possibile ad una chiamata locale,
  - Nella forma più semplice il programma è legato ad una piccola funzione di libreria chiamata **client stub** che rappresenta la funzione del server nello spazio di indirizzamento del client
  - Allo stesso modo c'è una funzione a lato server chiamata **server stub**
  - Queste funzioni nascondono il fatto che le chiamate del client alle funzioni del server non sono locali
  - Il passaggio dei parametri si chiama **marshaling**

RPC al lavoro

- Le chiamate dal client alla sua stub sono locali, nello spazio di indirizzamento locale del client
- Lo stesso per il server
- Invece di fare I/O via socket uso normali chiamate



Problemi

- L'eleganza concettuale di RPC nasconde alcuni problemi
  - Non posso passare puntatori al server perché client e server vivono in due differenti spazi di indirizzamento
  - Se devo passare un puntatore ad un intero k, la client stub potrebbe mandare k al server, quando il server rimanda indietro k, il nuovo k viene copiato sopra il vecchio
    - In pratica sostituisco una call by reference con una copy-restore
  - Questo non funziona se devo puntare ad una struttura molto complessa. Quindi ci sono limiti ai parametri che una funzione remota può accettare

Problemi

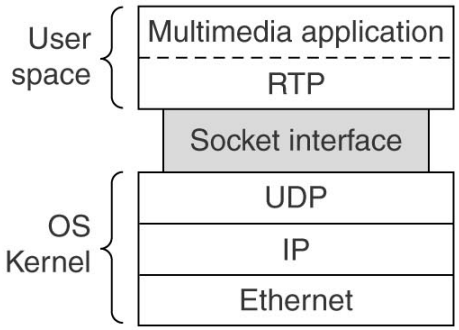
- In C posso avere una funzione che fa il prodotto scalare di due vettori senza sapere quanto è grande il vettore. Posso usare un valore particolare noto a funzione chiamante e chiamata per dire dove finisce il vettore. Qui invece è importante che lo sappia anche la stub che deve fare il marshaling dei parametri
- Inoltre non è sempre facile sapere i tipi dei paramentri. Per esempio printf può avere un numero variabile di parametri di diverso tipo
- Infine non posso usare variabili globali per comunicare (o condividere) dati tra chiamante e chiamata, oltre a comunicare via parametri. Con RPC non posso perché le variabili non sono condivise

RTP

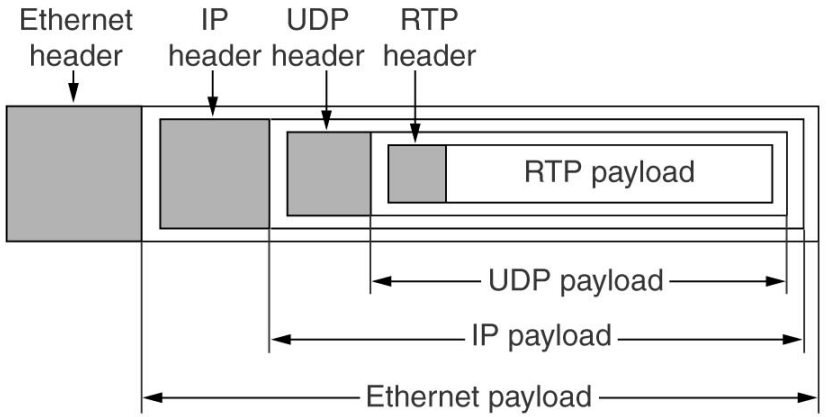
- Real-time Transport Protocol
  - RFC 1889, nato per unificare i vari protocolli di streaming di contenuti real time multimediali (internet radio, telefonia internet, videconferenza, video on demand, music on demand)
  - Normalmente RTP sta nello spazio utente e gira sopra UDP
  - Le applicazioni multimediali consistono di diversi stream audio, video o testo, che accedono ad una libreria nello user space.
  - La libreria multiplexa gli stream e li codifica in pacchetti RTP e li mette in un socket
  - All'altro lato del socket, stavolta nel kernel del sistema operativo, vengono generati pacchetti UDP

RTP a quale livello?

- Sta nello user space ed è linkato all'applicazione, quindi sembra un protocollo applicativo
- D'altra parte è un protocollo indipendente dall'applicazione che fa solo trasporto di pacchetti
- Diciamo che è un protocollo di trasporto implementato nel livello applicativo

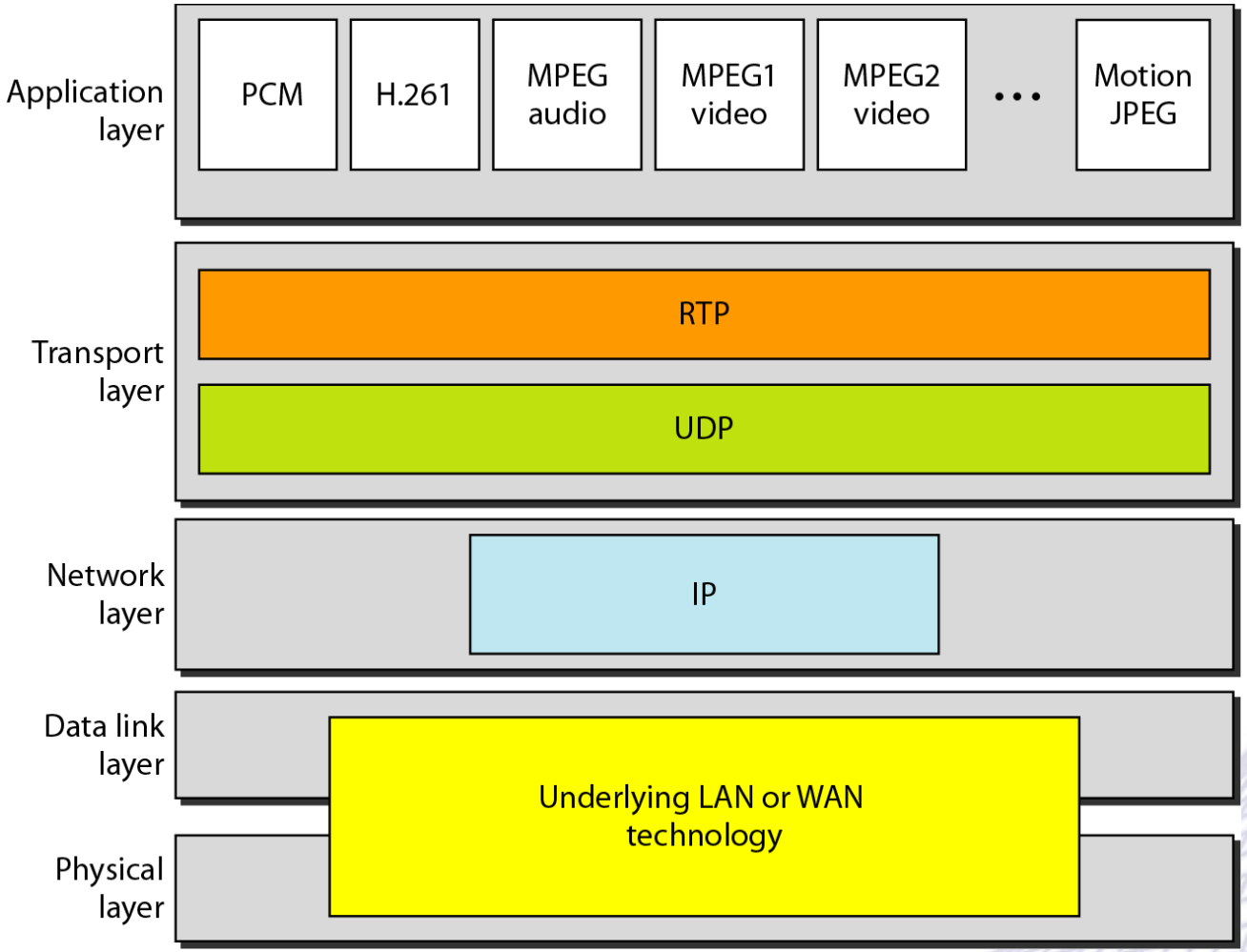


(a)



(b)

RTP e UDP



RTP su UDP

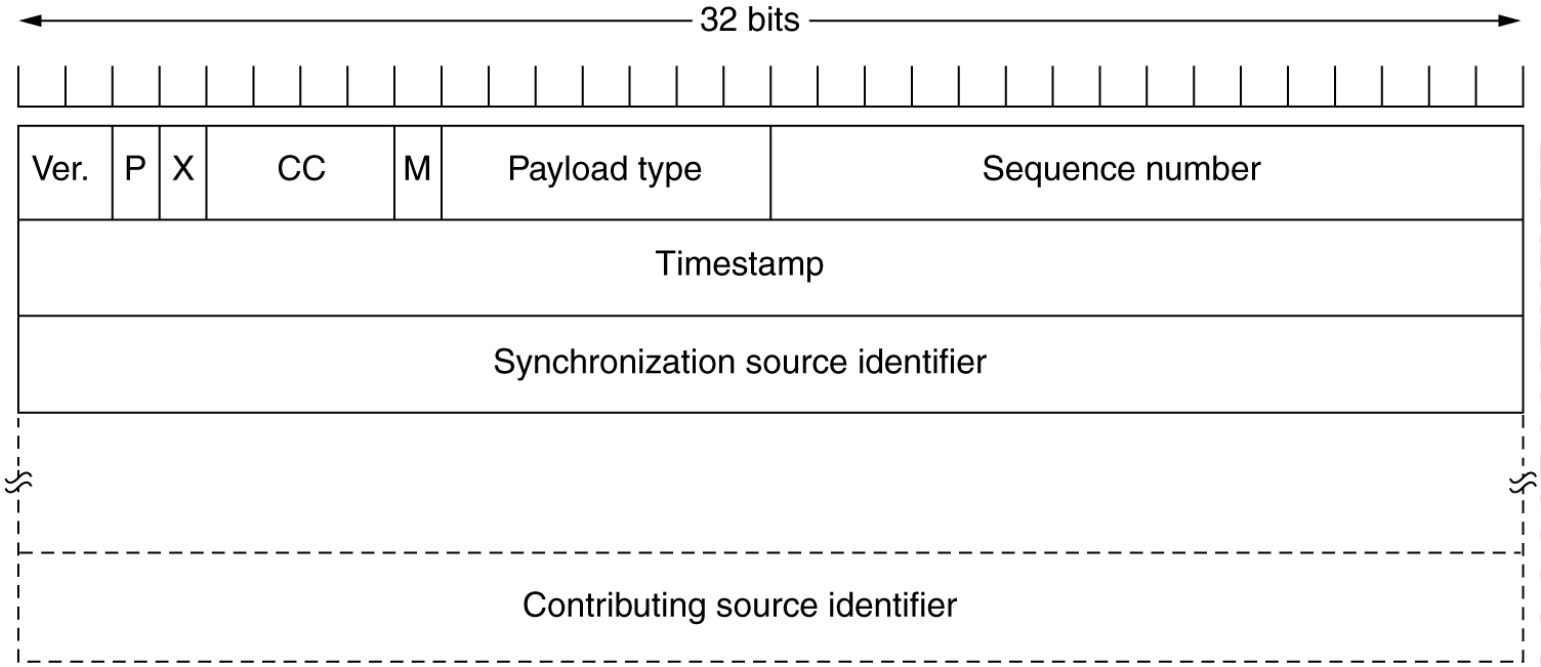
- L’idea base è di multiplexare diversi stream di dati real time in un unico stream UDP il quale può essere mandato ad un’unica destinazione (unicasting) o a multiple (multicasting)
- Ogni pacchetto ha un numero di sequenza per capire se qualcuno va perso. In tal caso è meglio cercare di approssimare il valore mancante per interpolazione.
- La ritrasmissione non sarebbe pratica. Arriverrebbe probabilmente troppo tardi per essere di qualche utilità, per cui non c’è nessun flow control, correzione di errori, acknowledgement e nessun meccanismo di ritrasmissione

RTP

- Ogni payload RTP può avere diversi campioni, ognuno con la sua codifica (PCM a 8 kHz, predictive encoding, GSM encoding, MP3 etc.)
- RTP fornisce un campo in cui la sorgente specifica l'encoding ma a parte questo non è coinvolto in come viene fatto l'encoding
- Le applicazioni real time possono avere bisogno di time-stamping, non in assoluto ma relativamente al primo campione.
- In questo modo la destinazione può fare un po' di buffering e riprodurre ogni campione dopo n millisecondi indipendentemente da quando arrivano, riducendo il jitter
- Questo permette anche di sincronizzare per esempio un flusso video con due flussi audio (es. stereo o due lingue mono diverse)

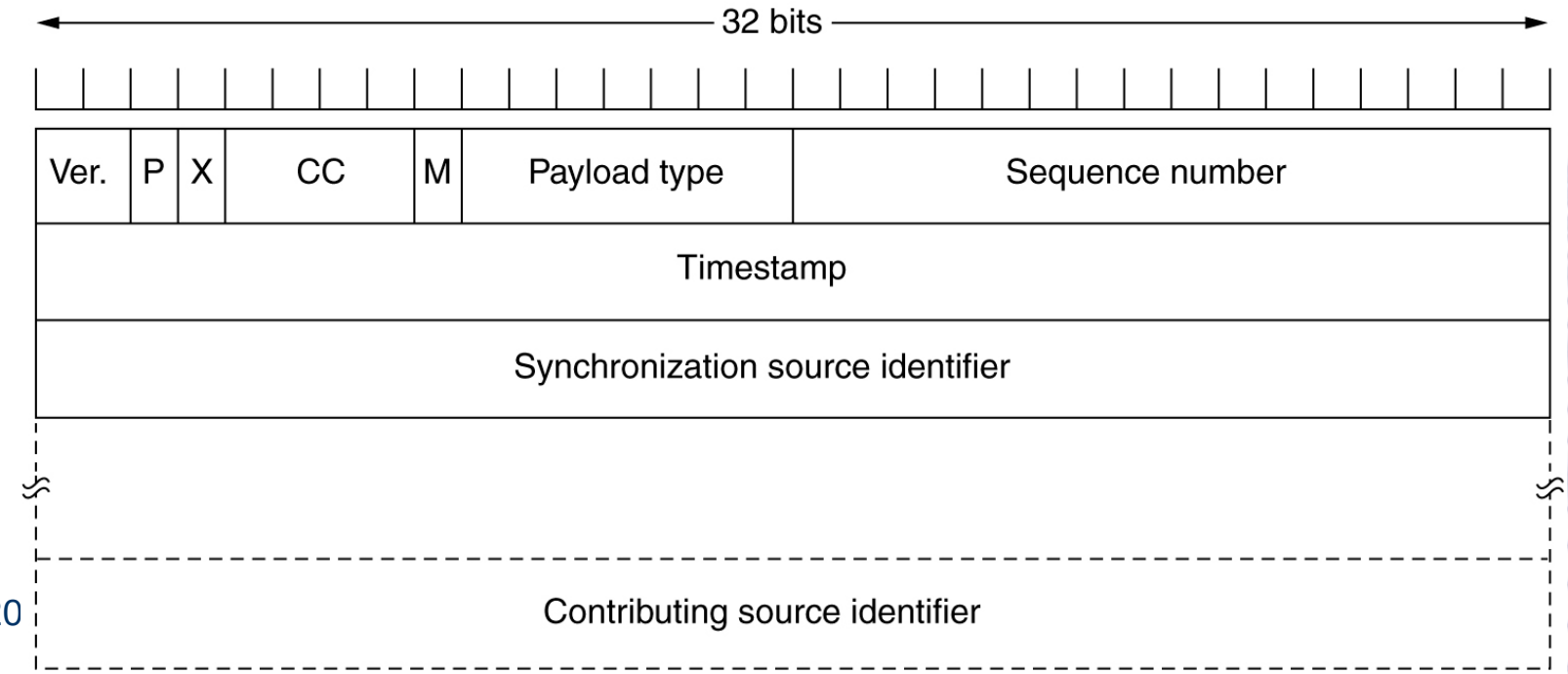
Frame RTP

- **Version** vale 2 (nb max=3 con due bit)
- **P** padded (a multipli di 4 bytes)
- **X** Extension header presente
- **CC** Quante sorgenti contribuiscono (tra 0 e 15)
- **M** Dipende dall'applicazione, per esempio marca l'inizio di un video frame in un'applicazione video o inizio di parola in una audio



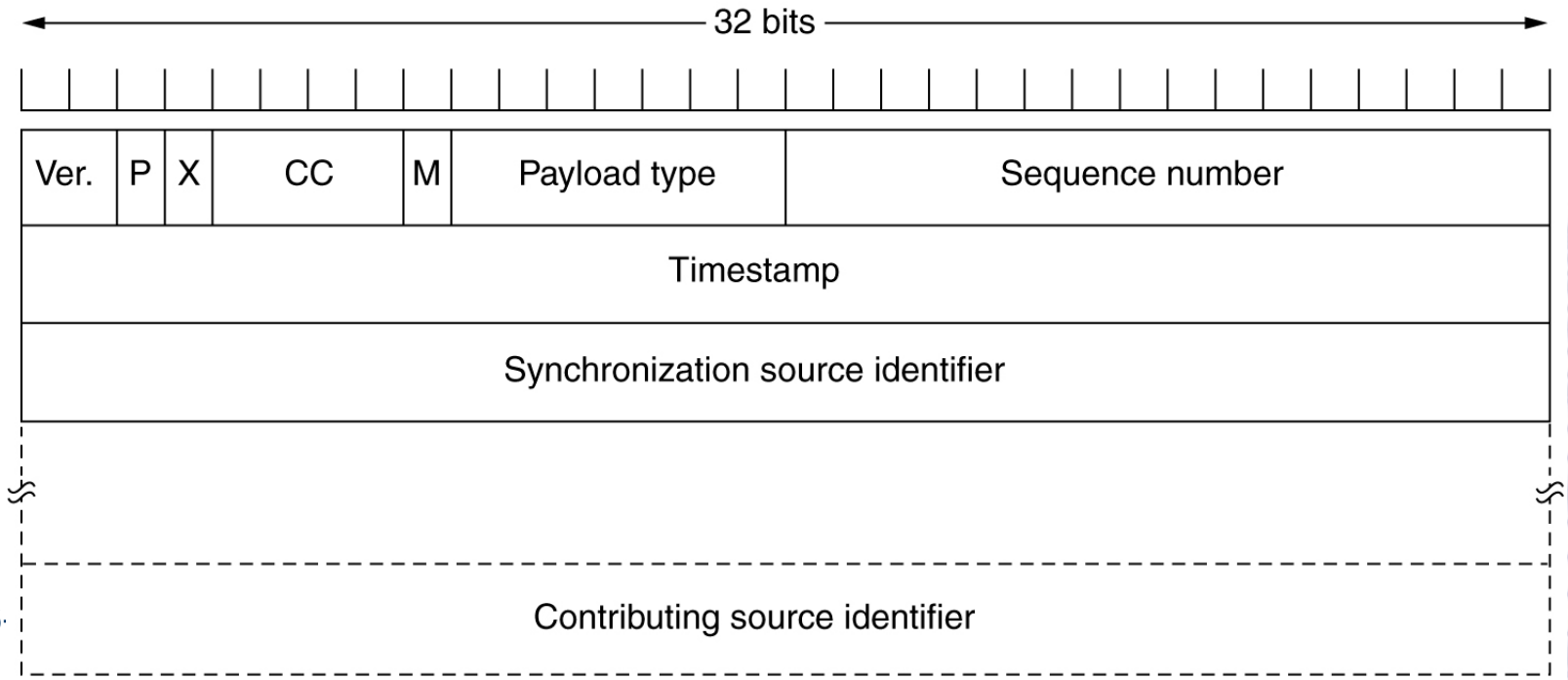
Frame RTP

- **Payload type**: quale algoritmo di encoding stiamo usando (8bit non compresso, mp3 etc…), non può cambiare durante la trasmissione
- **Sequence number**: numero incrementale per capire se stiamo perdendo pacchetti
- **Timestamp**: prodotto dalla sorgente, può essere usato per gestire il jitter, disaccopiando la riproduzione dall'arrivo dei pacchetti

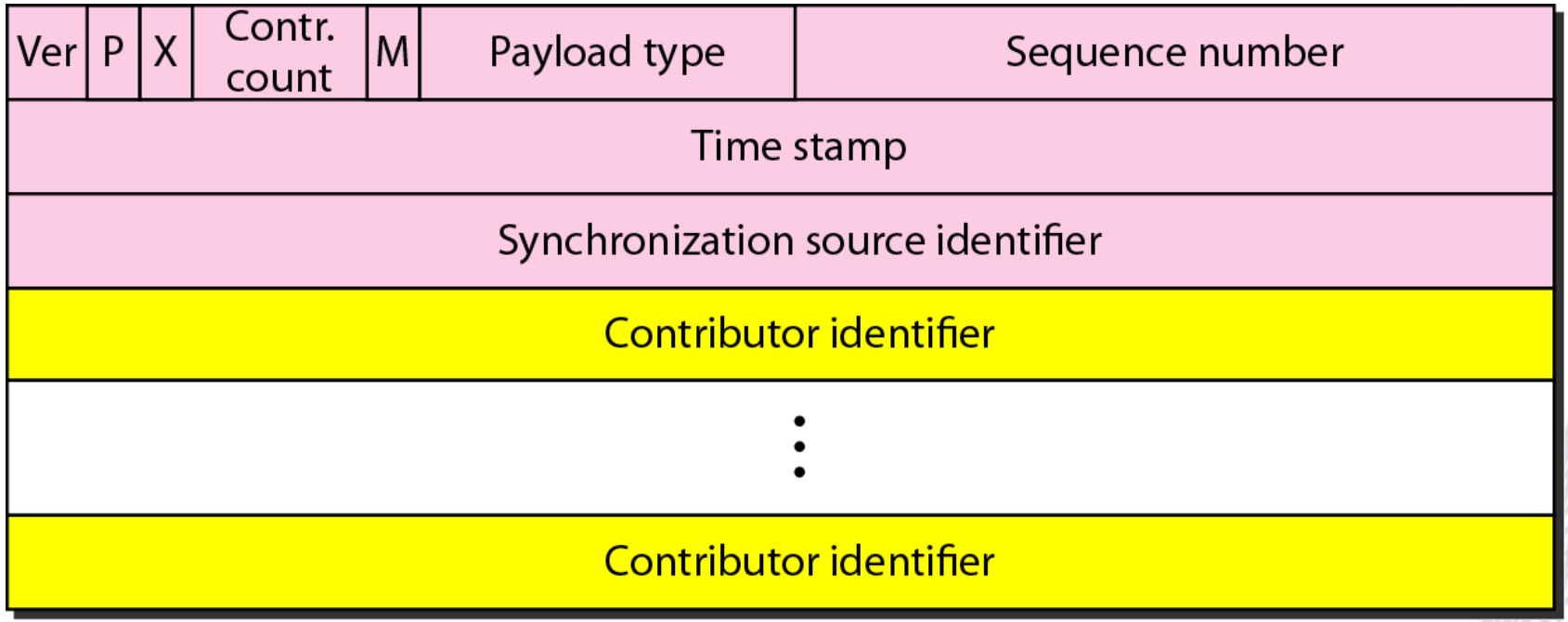


Frame RTP

- **Syncronization source identifier**. Per multiplexare o demultiplexare diversi data streams in un singolo flusso UDP
- **Contributing source identifier**, quando c'è un mixer in studio, il mixer è la sorgente che sincronizza e qui elenco i vari stream mixati



Frame RTP



Type	*Application*	*Type*	*Application*	*Type*	*Application*
0	PCMμ Audio	7	LPC audio	15	G728 audio
1	1016	8	PCMA audio	26	Motion JPEG
2	G721 audio	9	G722 audio	31	H.261
3	GSM audio	10–11	L16 audio	32	MPEG1 video
5–6	DV14 audio	14	MPEG audio	33	MPEG2 video

TCP

- Progettato fin dall'inizio per fornire un stream di byte, affidabile, end to end su di una internet inaffidabile
- Una internet differisce da una net singola perché parti diverse hanno topologie, bande, delay, packet size e altri parametri, completamente diversi
- TCP deve adattarsi dinamicamente alle diverse proprietà e deve essere così robusto da resistere a diversi tipi di problemi
- Definito nella RFC 793 ma corretto pesantemente in diversi modi dettagliati in RFC 1122. Altre estensioni in RFC 1323

Cosa fa

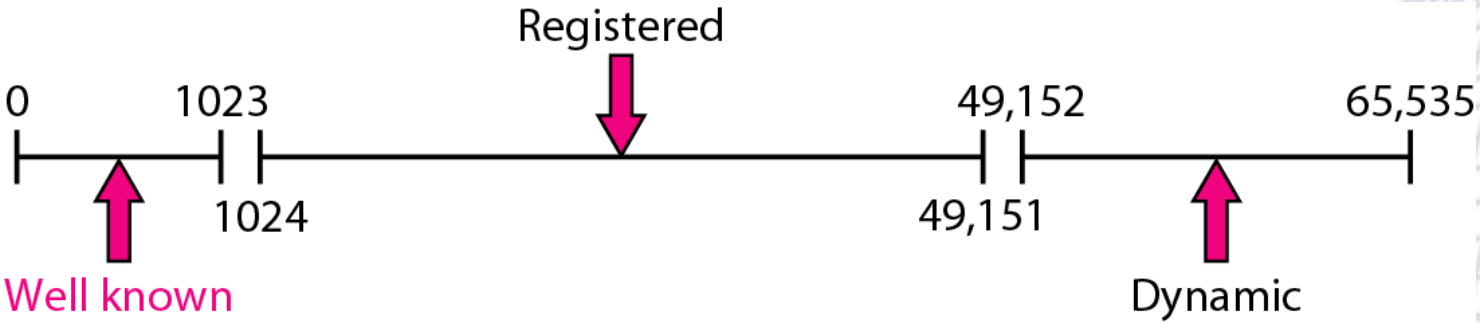
- Una macchina con TCP ha un'entità che gestisce il trasporto TCP
  - Una libreria oppure un processo utente o nel kernel
  - I dati forniti dal processo vengono spezzati in pezzi non più grandi di 64 kB (in realtà spesso 1460 Bytes per stare in un frame Ethernet) e manda ogni pezzo in un pacchetto IP

TCP Service Model

- Sender e Receiver creano due end point chiamati sockets
- L'indirizzo del socket consiste di
  - Indirizzo IP dell'host
  - Port number. Un numero a 16 bit locale all'host
- Un socket può gestire diverse connessioni identificate da socket identifier
  - Non ci sono numeri di virtual circuit o altri identificatori

Well known ports

- I numeri di porta sotto il 1024 sono chiamati well known ports e riservati per servizi standard.
  - Sono spesso servite da processi con privilegi di super user o amministratore per cui sono particolamente pericolose; un hacker che buca uno di questi processi poi riesce ad ottenere una shell con privilegi di root
- Anche le porte da 1024 a 49151 sono registrabili presso IANA, così è possibile scegliere per nuovi servizi porte non registrate da altri
  - per es. 6000 per X-Window



Alcune porte note

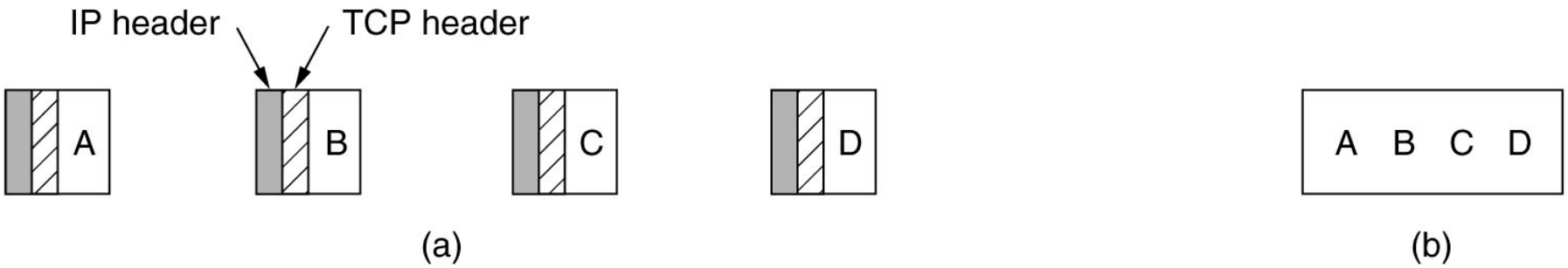
Port	Protocol	Use
21	FTP	File transfer
23	Telnet	Remote login
25	SMTP	E-mail
69	TFTP	Trivial file transfer protocol
79	Finger	Lookup information about a user
80	HTTP	World Wide Web
110	POP-3	Remote e-mail access
119	NNTP	USENET news

inetd

- Si potrebbe attaccare un daemon FTP alla porta 21 e un daemon 23 alla porta telnet al momento del boot, sprecando così molta memoria per daemon inutilizzati per la maggior parte del tempo
- Invece un singolo daemon, chiamato **inetd** in unix, si attacca a diverse porte
  - Alla prima richiesta di connessione, inetd lancia un nuovo processo (**fork**) ed esegue il daemon appropriato
- L'amministratore invece può decidere di avere daemon permanenti per servizi spesso attivi (es httpd sulla porta 80 di un web server) e gestire tutti gli altri via inetd

Connessioni TCP

- Sono full duplex e point to point
  - Full duplex: il traffico va in entrambe le direzioni nello stesso momento
  - Point to point: ci sono solo due end point, niente multicast o broadcast
- Sono byte stream non message stream
  - Non vengono salvati i confini dei messaggi.
  - Se mando 4 write da 512 bytes queste vengono consegnate come 4 pezzi da 512 (a) o 2 pezzi da 1024 o un pezzo da 2048 (b)
  - Come un file in unix, non c'è modo di sapere se un dato viene bufferizzato dal filesystem o scritto immediatamente



Flush

- Se l’applicazione vuole che i dati partano subito
  - Per esempio scrivo un comando e premo Return perché venga eseguito
  - L’applicazione usa il flag “PUSH” che dice a TCP di non ritardare la trasmissione

Protocollo TCP

- Ogni byte TCP ha un numero di sequenza di 32 bit.
- Con una linea a 56 kbps ci vuole una settimana per wrappare, al giorno d'oggi abbiamo linee più veloci di 4 o 5 ordini di grandezza (da **una settimana** a 60 o **6 secondi**)
- Le due entità TCP si scambiano segmenti dati in forma di segmenti con un **header di 20 byte** (più una parte opzionale) e un **body di zero o più bytes**
- Ogni segmento, incluso l'header TCP deve stare in un payload IP di 65536 bytes, inoltre ogni rete ha un MTU (maximum transfer unit) per cui ogni segmento deve stare in una MTU
- In pratica spesso la MTU è di 1500 byte per accordarsi con il payload di Ethernet

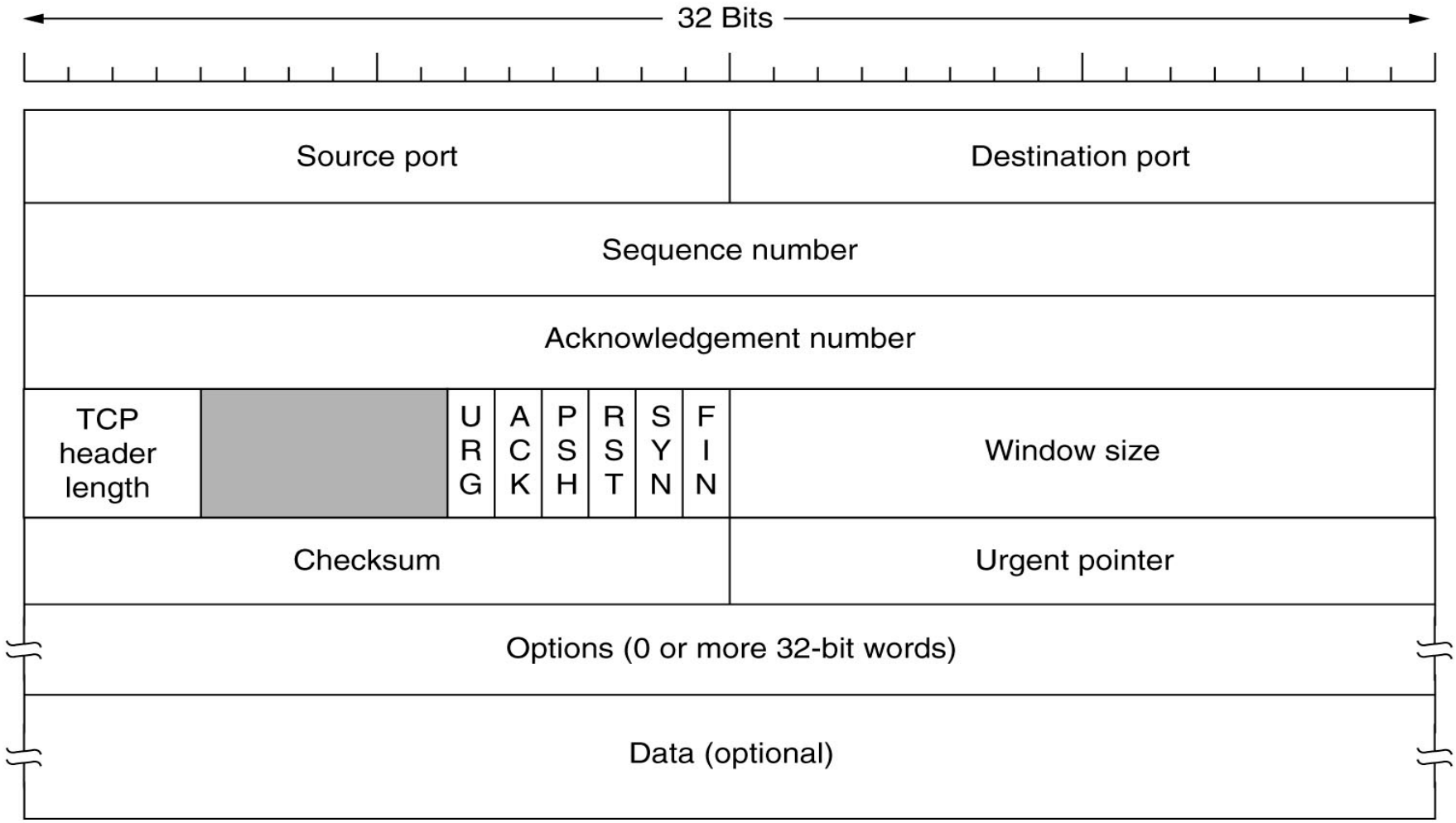
Sliding window

- Il protocollo base tra le due TCP entities è di tipo sliding window
  - Il mittente manda un segmento e fa partire un timer
  - Quando il segmento arriva il destinatario manda indietro un segmento con dati (se ne ha, altrimenti vuoto) con un numero di ack uguale al prossimo numero di sequenza che si aspetta di ricevere
  - Se il timer spira prima di ricevere un ack, il mittente rispedisce quel segmento di nuovo

Problemi di sliding window

- Sembra semplice! Ma cosa facciamo se i segmenti non arrivano in ordine? Es i byte 3072-4095 arrivano ma non possono ricevere un ack perché non sono ancora arrivati i 2048-3071?
- Oppure un segmento subisce ritardi tali che il sender va in timeout e lo rispedisce?
- Dal momento che ogni byte in uno stream ha un unico offset posso ritrasmettere solo i bytes che non hanno avuto un ack, l'importante è cercare di ottimizzare questo processo

Header TCP


32 Bits
Source port
Destination port
Sequence number
Acknowledgement number
TCP header length
U R G
A C K
P S H
R S T
S Y N
F I N
Window size
Checksum
Urgent pointer
Options (0 or more 32-bit words)
Data (optional)

Header TCP

- Dopo gli header ho fino a **65535-20-20** = 65495 byte di dati
- Il primo 20 si riferisce all'header IP e il secondo a quello TCP
- **Source port** e **destination** servono per identificare gli end point all'interno dell'host IP
- **Sequence number** e **Acknowledgement number** fanno quanto appena descritto e sono a 32 bit ciascuno
- **TCP header length** dice quante parole di 32 bit sono contenute nell'header TCP. Necessario perché il campo **Options** ha lunghezza variabile
- Poi ci sono 6 bit non ancora usati. Nota bene, non sono stati usati nei 25 - 30 anni di vita del TCP. Non ce ne è stato bisogno

6 bit importanti

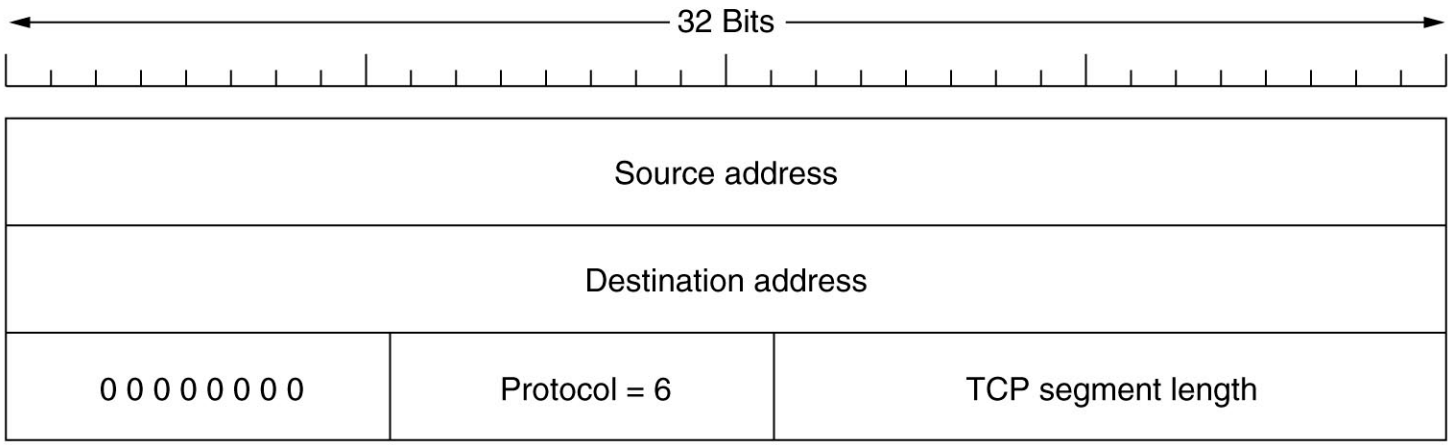
- Ora ci sono 6 campi da un bit
  - **URG=1** se viene usato lo Urgent Pointer (questo indica l'offset nel segmento dove trovare i dati urgenti, si può usare al posto di un messaggio di interrupt
  - **ACK=1** indica che il Acknowledment number è valido, =0 indica che il segmento non contiene Ack così il campo suddetto non viene considerato
  - **PSH** indica dati pushed. Il ricevente è quindi gentilmente pregato di non bufferizzare i dati all'arrivo come aveva magari intenzione di fare per maggiore efficienza
  - **RST** usato per resettare una connessione il cui stato per vari motivi è incerto. Usato anche per non accettare una connessione o per rifiutare un segmento in formato non valido
  - **SYN** per stabilire una connessione. La richiesta di connessione ha **SYN=1** e **ACK=0**, la risposta ha **SYN=1** e **ACK=1**
  - **FIN** bit per segnalare la chiusura di una connessione

Windows

- Il flow control viene gestito da una finestra di dimensione variabile
- Windows size dice quanti byte posso mandare a partire dall'ultimo byte acknowledged
  - Posso avere anche un valore **0** con cui il receiver dice che tutto è stato ricevuto fino a "**ack number -1**" ma non si desidera per il momento ricevere altro.
  - Quando poi il receiver è di nuovo in grado di ricevere può inviare di nuovo lo stesso ack number con un windows size diverso da zero

Checksum

- C’è anche un **checksum** per avere più affidabilità
- Viene fatto il controllo di header, dati e dello pseudo header presente in figura
  - Questo contiene gli indirizzi a 32 bit delle due macchine, il numero di protocollo di TCP (6) e il numero di byte nel segmento TCP compreso l’header
  - Questo accorgimento aiuta nel rivelare pacchetti consegnati male ma viola anche la gerarchia dei protocolli dal momento che gli indirizzi IP appartengono al layer IP e non a quello TCP



Options

- Il campo **Options** permette funzioni extra oltre a quelle fornite dall'header regolare
  - Una particolarmente importante è quella che permette agli host di specificare il massimo payload TCP che sono disposte ad accettare
  - Meglio usare un numero alto per efficienza, in modo da ammortizzare l'header di 20 byte su molti byte di dati ma alcuni host potrebbero non essere in grado di gestire grandi segmenti
  - Se non viene usata questa opzione per contrattare il segmento massimo, per difetto si prende 536 byte di payload

Opzioni e finestre

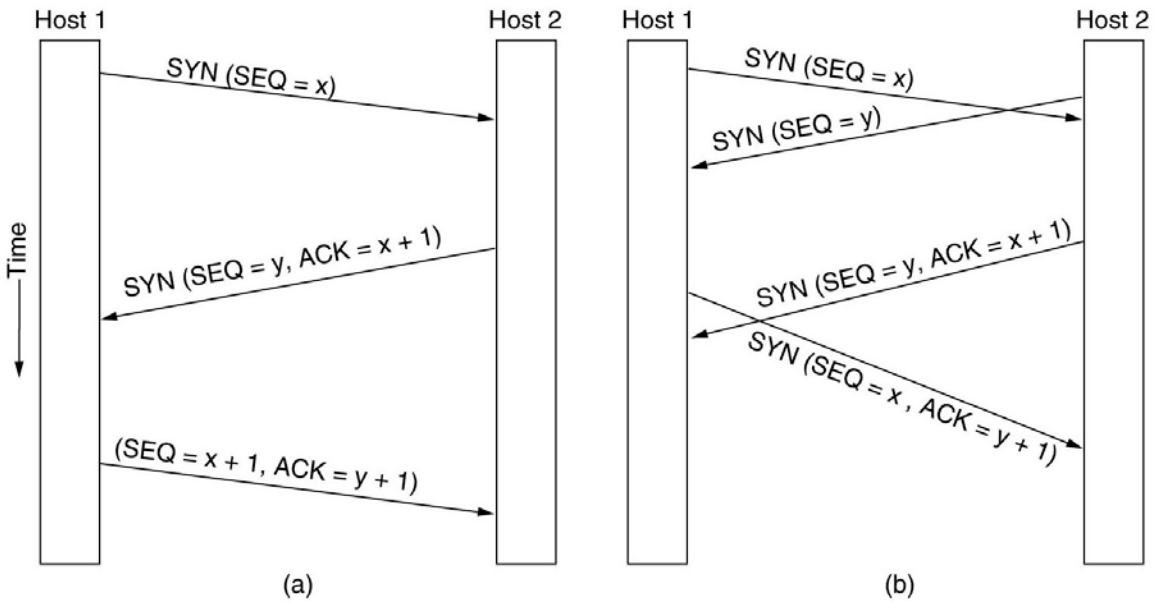
- Per linee veloci e/o con grandi delay la finestra da 64KB è un problema
- Su una linea T3 a 44.376 Mbps ci vogliono solo 12 msec per mandare fuori una finestra di 64 KB. Se il **rtt** è di 50 msec (tipica linea intercontinentale) il mittente è idle per ¾ del tempo aspettando l'ack
- Su connessione a satellite ancora peggio o su link a Gbps
- Servirebbe una finestra più grande ma come fare con solo 16 bit?
- Nella RFC 1323 si stabilì di contrattare con le options anche una Window scale. In pratica un numero che permette ad entrambi i lati di shiftare il windows size fino a 14 bit a sinistra permettendo quindi finestre grandi fino a 2^{30} byte

Selective repeat

- Altra opzione proposta in RFC 1106 ed ora spesso implementata permette di usare un "***selective repeat***" invece di un "***go back n***"
- Nel secondo caso quando ricevo un segmento corrotto e poi un grande numero di segmenti buoni dovrei al momento del time out trasmettere tutti i pacchetti senza ack, inclusi quelli ricevuti correttamente
- Invece RFC 1106 introduce un NACK in cui il receiver chiedere al mittente di ritrasmettere solo il pacchetto che non è arrivato correttamente, riducendo quindi la quantità di dati ritrasmessi

Setup connessione TCP

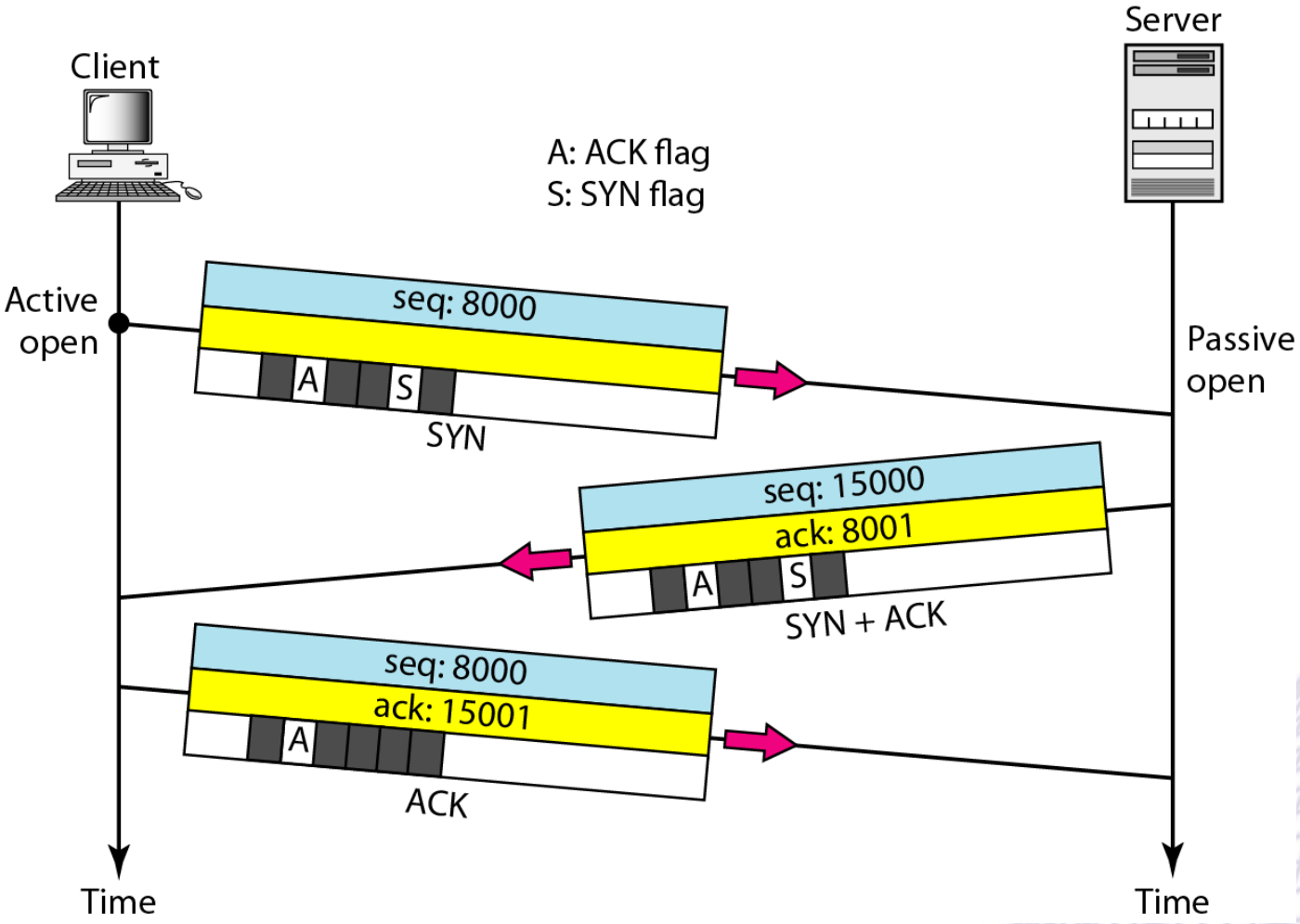
- Uso il 3way handshake già visto in precedenza
- Nel primo caso una normale connessione. Il primo segmento con il SYN usa un byte nello spazio delle sequenze
- Nel secondo caso i due host cercano di aprire una connessione simultaneamente ma una connessione viene identificata dagli end point per cui una sola connessione viene stabilita



SYN FLOODING

- Il meccanismo con il SYN espone il destinatario a subire un attacco di SYN FLOODING
- Un hacker manda tantissimi SYN, per ogni SYN il server TCP alloca le risorse e i buffer per gestire le molteplici connessioni, causando un blocco della macchina che non può servire altre richieste legittime (Denial of Service)
- Non si capisce da dove vengono se gli indirizzi IP del mittente sono fasulli (spoofed)

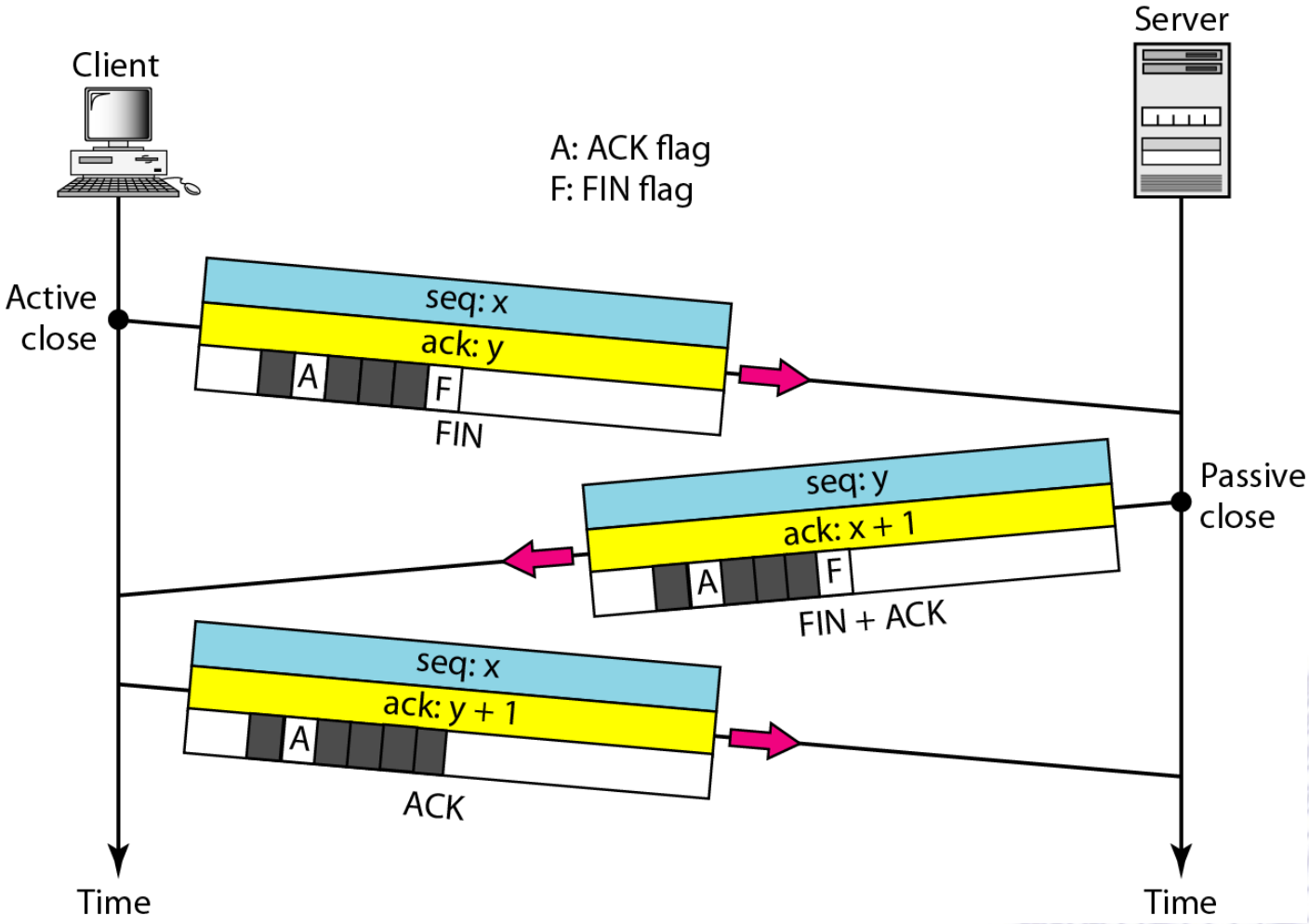
TCP connection



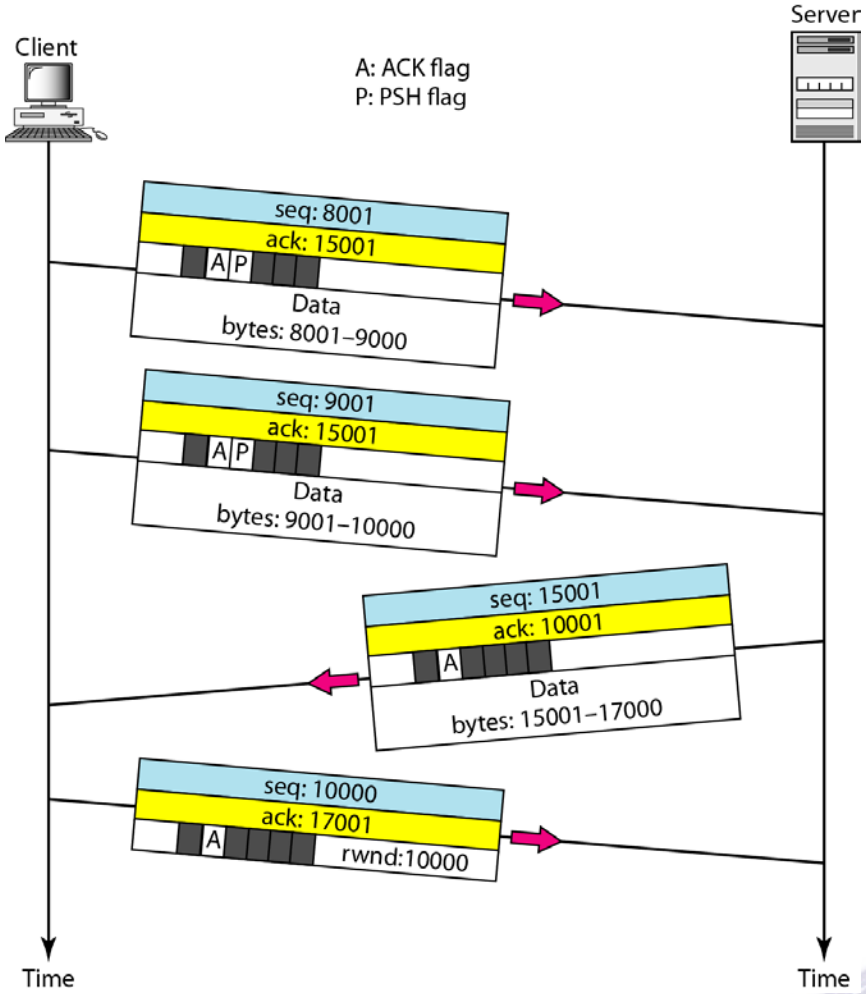
Chiusura connessione TCP

- La connessione è full duplex ma viene chiusa come una coppia di connessioni simplex
- Una delle due manda un segmento con il bit FIN (non ho più dati da trasmettere)
- Riceve il relativo ACK e a questo punto la connessione è chiusa per dati in questa direzione.
- Altri due segmenti per chiudere dall'altro lato
- Tuttavia il primo ACK e il secondo FIN possono essere messi nello stesso segmento, per cui ho solo 3 pacchetti per chiudere invece che 4
- Per evitare il problema dei due eserciti si usano dei timers. Se in risposta ad un FIN non ricevo nulla entro due max packet lifetimes il mittente chiude la connessione.

Chiusura connessione



Trasferimento dati



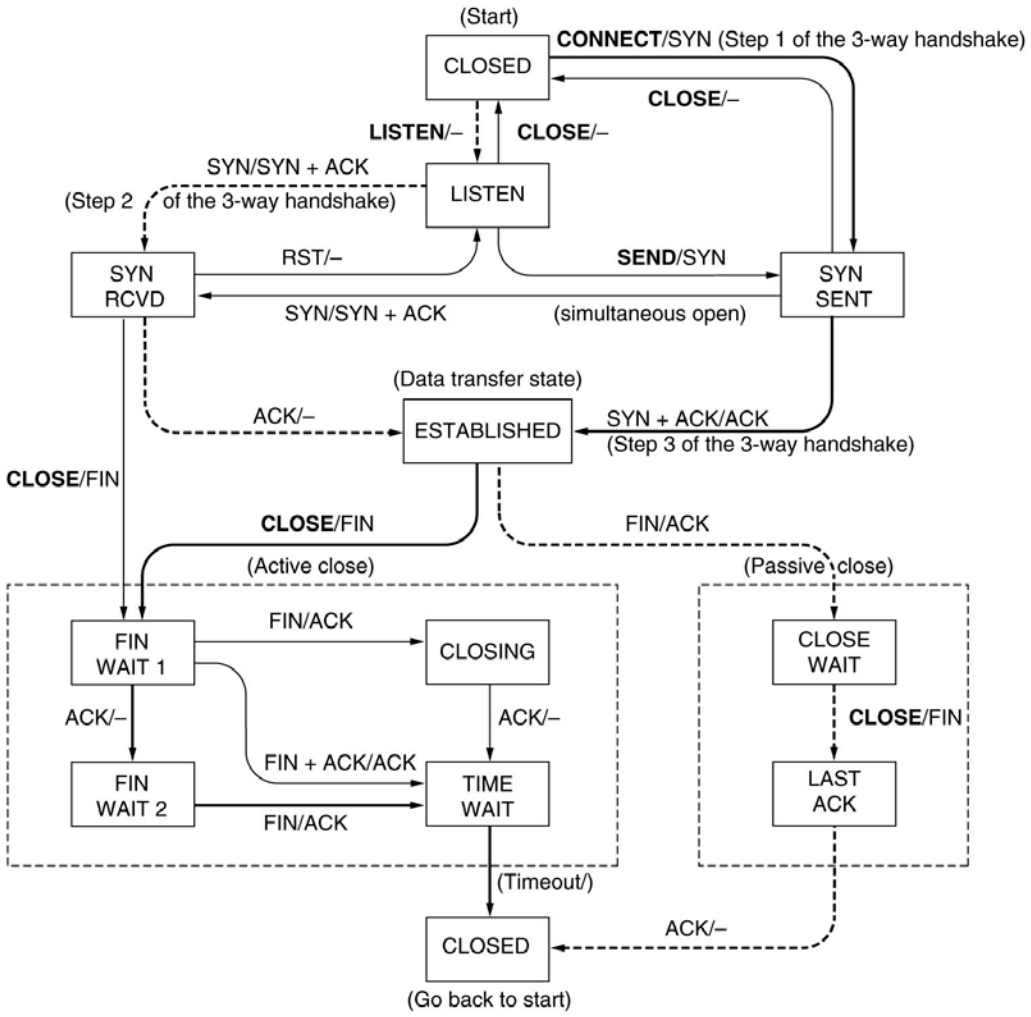
Finite state machine

- Si può descrivere il protocollo per stabilire e chiudere una connessione come una macchina a stati finiti
- Vediamo gli stati possibili

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIMED WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

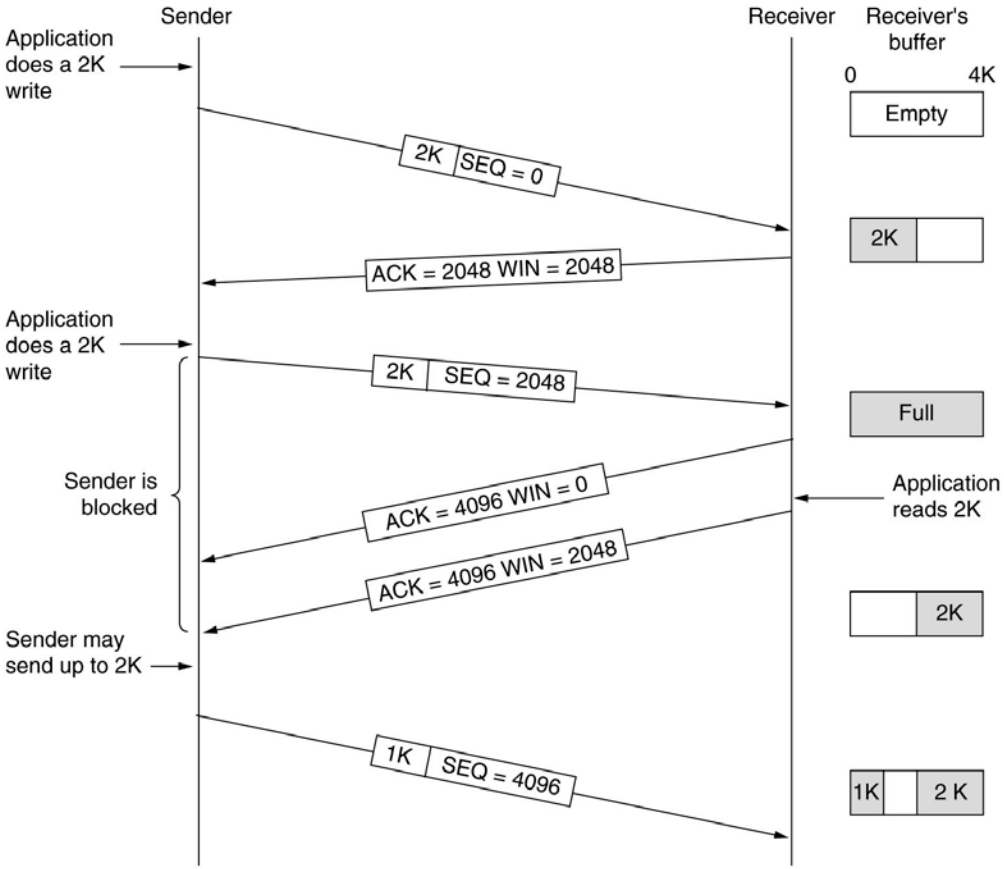
Macchina a stati finiti

- Linee continue grosse descrivono il client, quelle tratteggiate il server
- Le linee sottili descrivono eventi inusuali

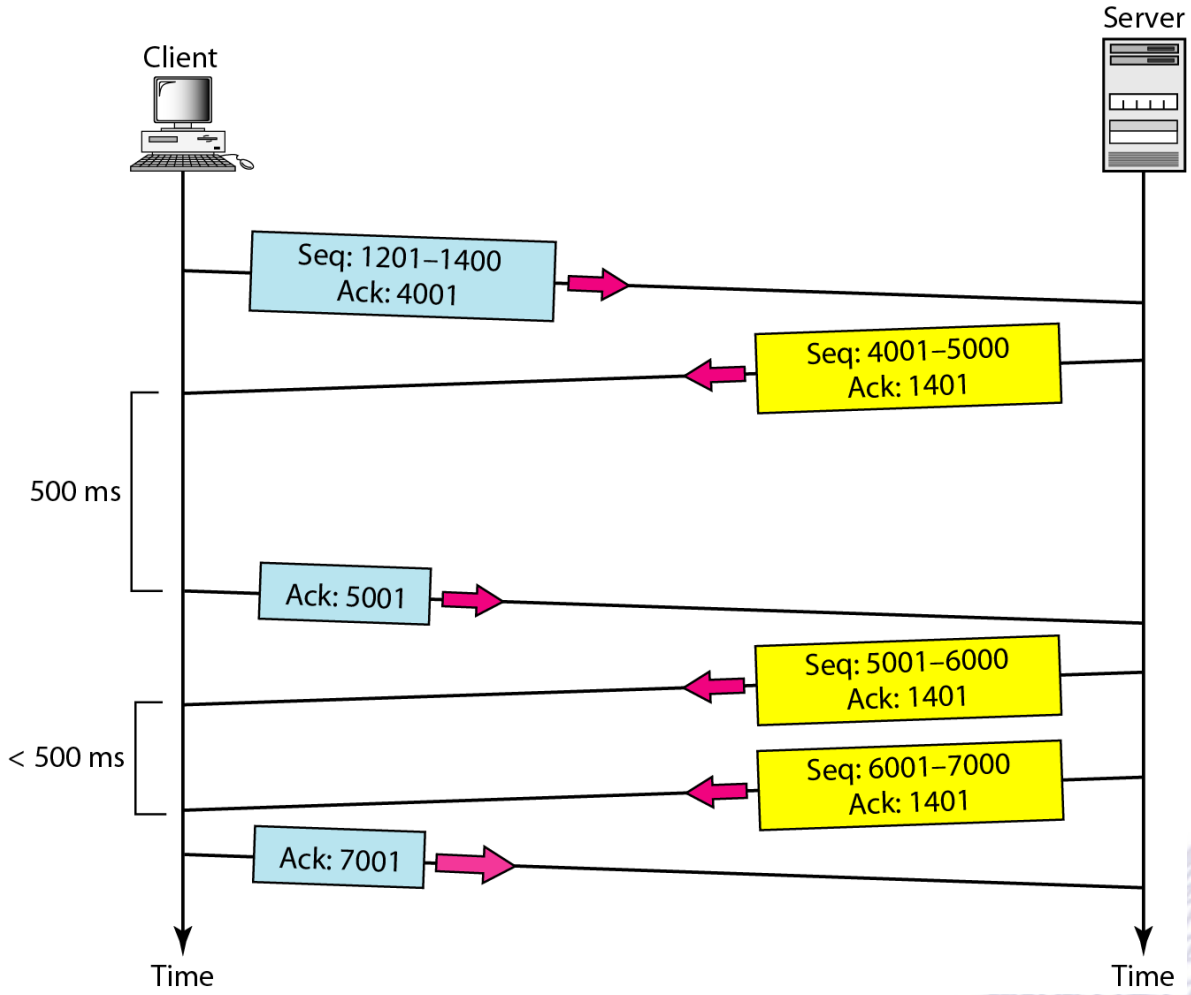


Policy di trasmissione

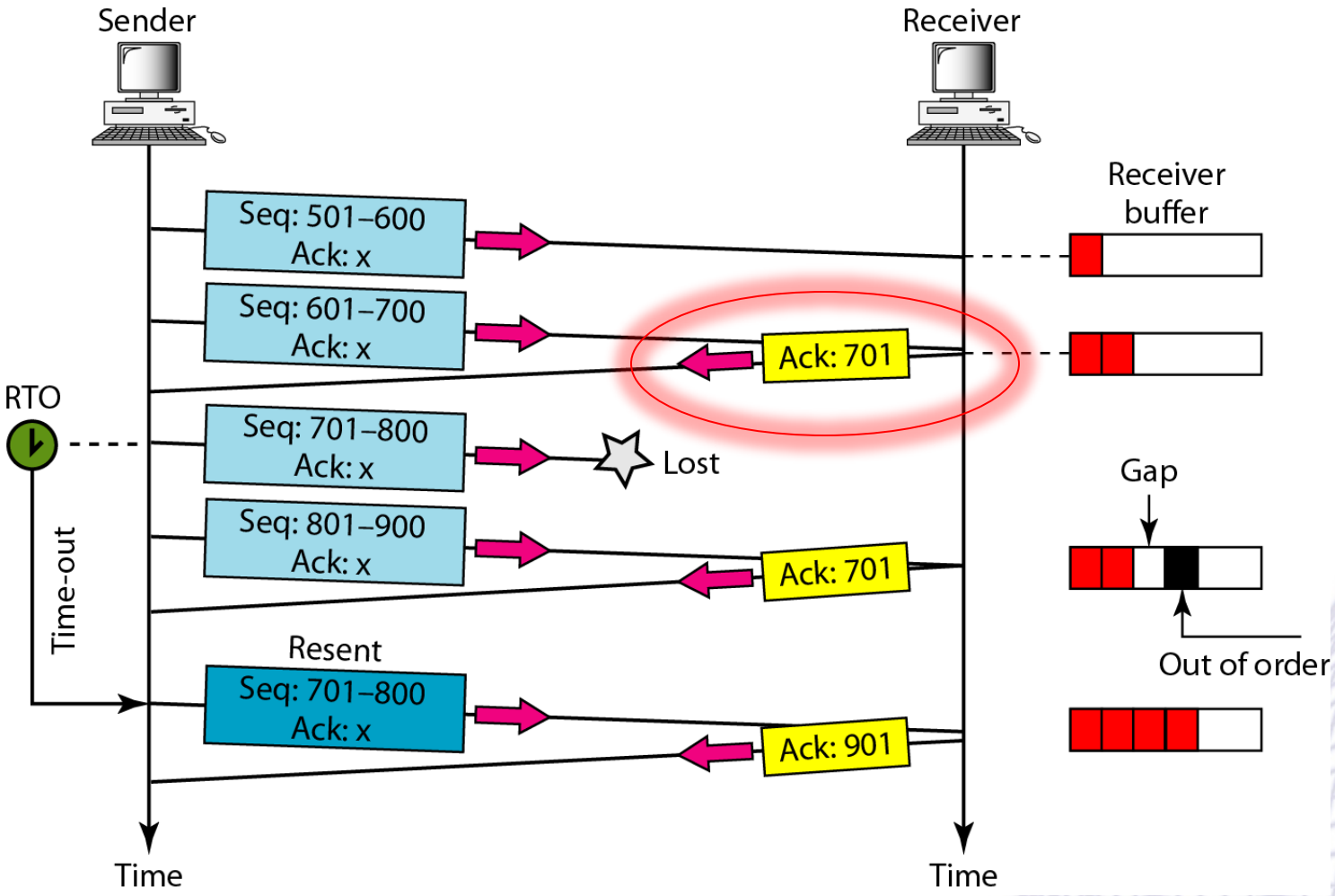
- Quando il receiver non riesce a ricevere altri dati risponde con **win=0**
- Il sender non manda più nulla con l'eccezione di **Urgent data** che possono sempre essere trasmessi, per esempio per killare un processo all'altro lato
- Il sender può rimandare un segmento di un byte per chiedere al receiver di riannunciare il prossimo byte atteso e il windows size. Lo standard prevede questo caso per prevenire un deadlock se un annuncio di finestra dovesse andare perso



Operazioni normali

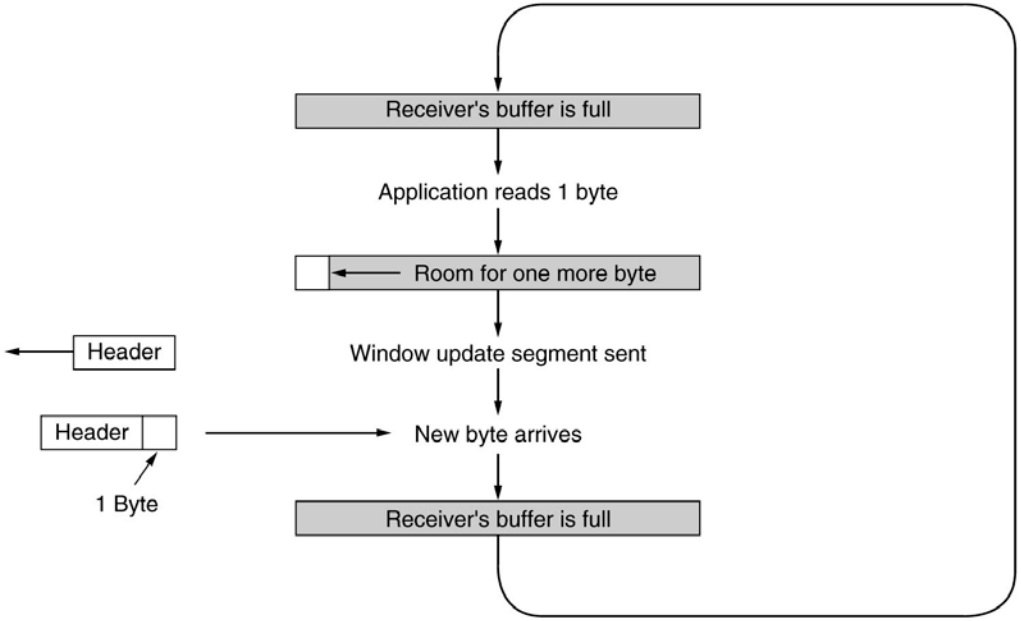


Segmento perso (dup ACK)



Silly window syndrome

- Il buffer del ricevente è pieno. Se il receiver processa un byte alla volta, ogni volta manda un pacchetto di 40 byte dicendo di mandare ancora un byte. Grande spreco di nuovo
- La soluzione di Clark (1982) è di non mandare l'update della finestra fino a quando non ha parecchio spazio libero, per es. il max segment size o mezza finestra (il più piccolo dei due)
- Approccio complementare da lato ricevente a quello di Nagle dal lato mittente



Gestione delle congestioni

- Il livello network cerca di gestire le congestioni ma il grosso del lavoro viene fatto a livello TCP perché la soluzione giusta è rallentare il data rate
- Non inserire un nuovo pacchetto fino a quando uno vecchio non se ne è andato
- TCP cerca di farlo manipolando dinamicamente la dimensione della finestra
- In passato una perdita di pacchetto poteva essere dovuta ad un errore di trasmissione o ad un pacchetto scartato da un router in congestione. Al giorno d'oggi il primo caso avviene raramente, praticamente solo se abbiamo link wireless
- Prima di vedere come reagire alla congestione vediamo come fare per prevenirle

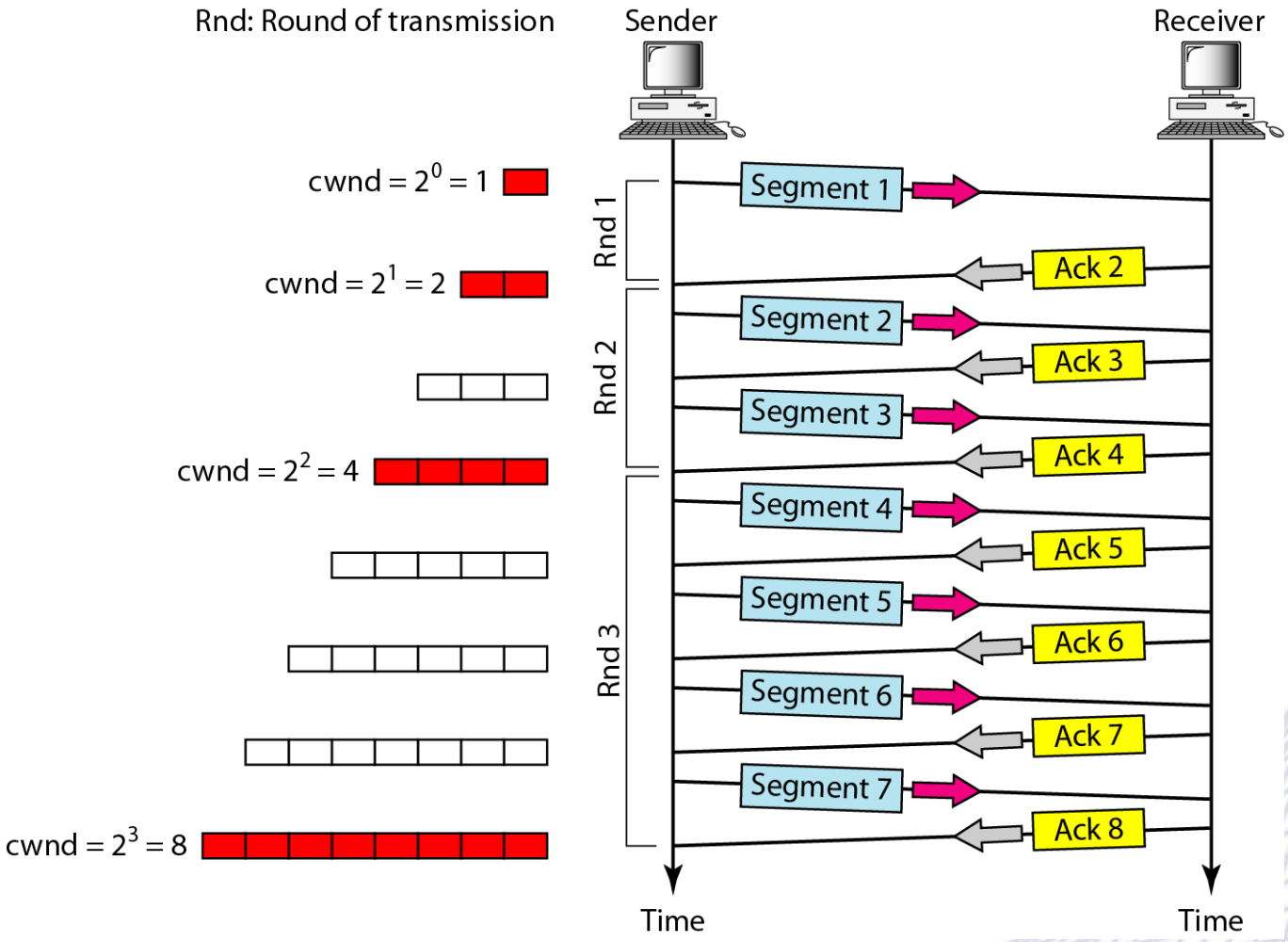
Congestion window

- Il sender mantiene due finestre
  - La finestra che il receiver gli ha concesso (**rwnd**)
  - Una finestra di congestione (**cwnd**)
- La finestra effettiva è il minimo delle due finestre
- Dimensioni di cwnd
  - All'inizio la congestion window viene inizializzata al max segment size (MSS) che può essere spedito senza frammentazione sulla connessione.
  - Se il primo segmento passa senza timeout cwnd diventa 2MSS e quindi vengono spediti due segmenti
  - Se arrivano i 2 ACK cwnd viene aumentata di 2
  - Se la finestra è di n segmenti e tutti gli n ricevono un ack in tempo, la finestra viene aumentata di altri n segmenti. Ogni ack di burst raddoppia la finestra.

Slow start

- Continua a crescere in modo esponenziale fino a quando non ricevo un timeout o raggiungo la finestra del receiver
- L'algoritmo ideato da Jacobson nel 1988 si chiama slow start
  - (ma non è per nulla slow, lo è solo nel senso che parte da valori molto bassi)
- Tutte le implementazioni TCP devono supportarlo

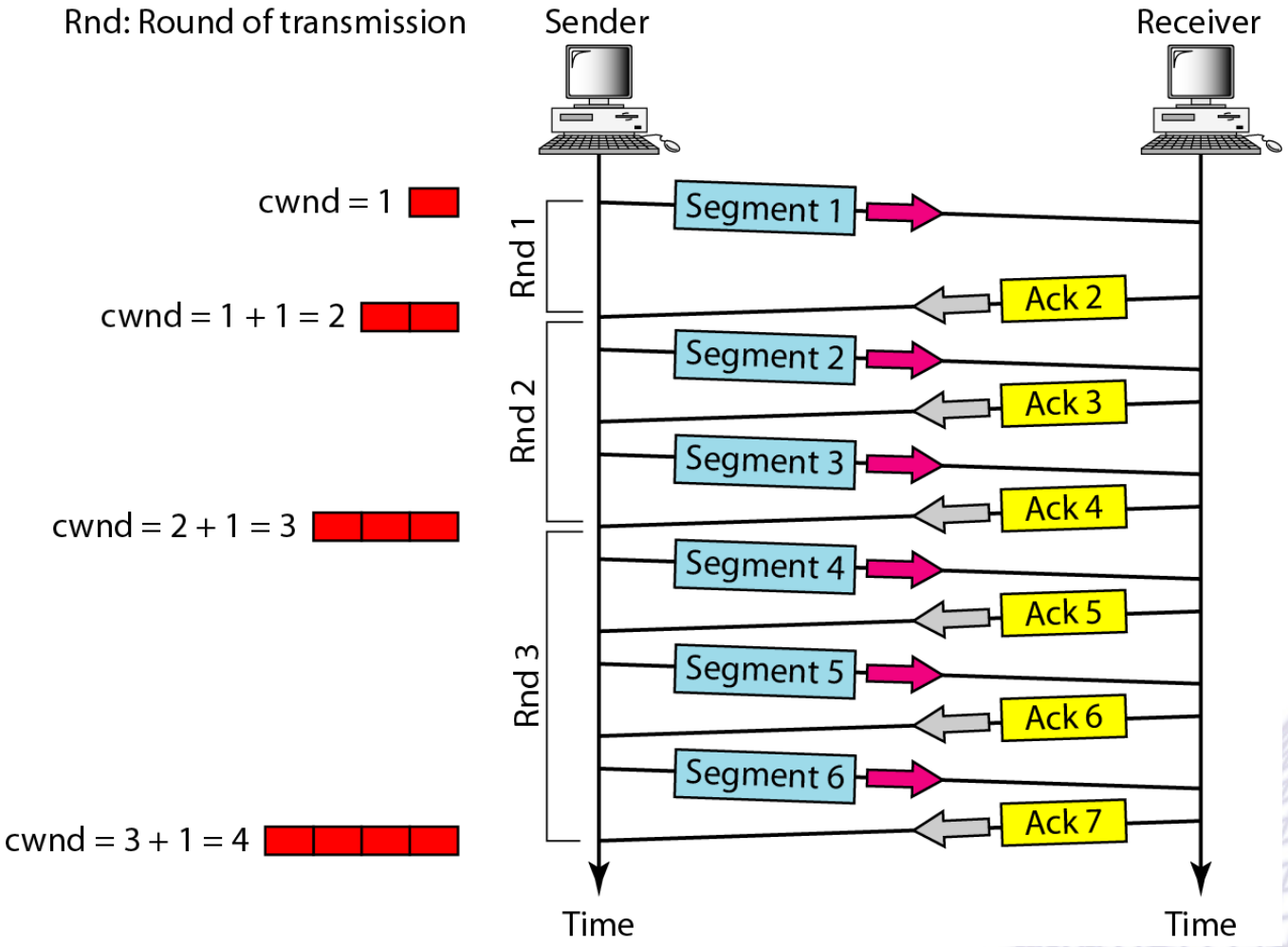
Slow start


Rnd: Round of transmission
Sender
Receiver
cwnd = 2^0 = 1
cwnd = 2^1 = 2
cwnd = 2^2 = 4
cwnd = 2^3 = 8
Rnd 1
Rnd 2
Rnd 3
Segment 1
Ack 2
Segment 2
Ack 3
Segment 3
Ack 4
Segment 4
Ack 5
Segment 5
Ack 6
Segment 6
Ack 7
Segment 7
Ack 8
Time
Time

Threshold

- Un terzo parametro chiamato threshold (**ssthresh**), di solito impostato a 65536 byte (64 KB)
- Quando arrivo a ssthresh la fase moltiplicativa dello slow start si arresta e l'aumento di cwnd diventa lineare aumentando di 1, invece che moltiplicativo
- Vediamo un esempio con ssthresh=2 MSS
- Quindi questa soglia serve per prevenire una congestione usando un incremento additivo da un certo punto in poi

Ssthresh = 2


Rnd: Round of transmission
Sender
Receiver
cwnd = 1
cwnd = 1 + 1 = 2
cwnd = 2 + 1 = 3
cwnd = 3 + 1 = 4
Rnd 1
Rnd 2
Rnd 3
Segment 1
Ack 2
Segment 2
Ack 3
Segment 3
Ack 4
Segment 4
Ack 5
Segment 5
Ack 6
Segment 6
Ack 7
Time
Time

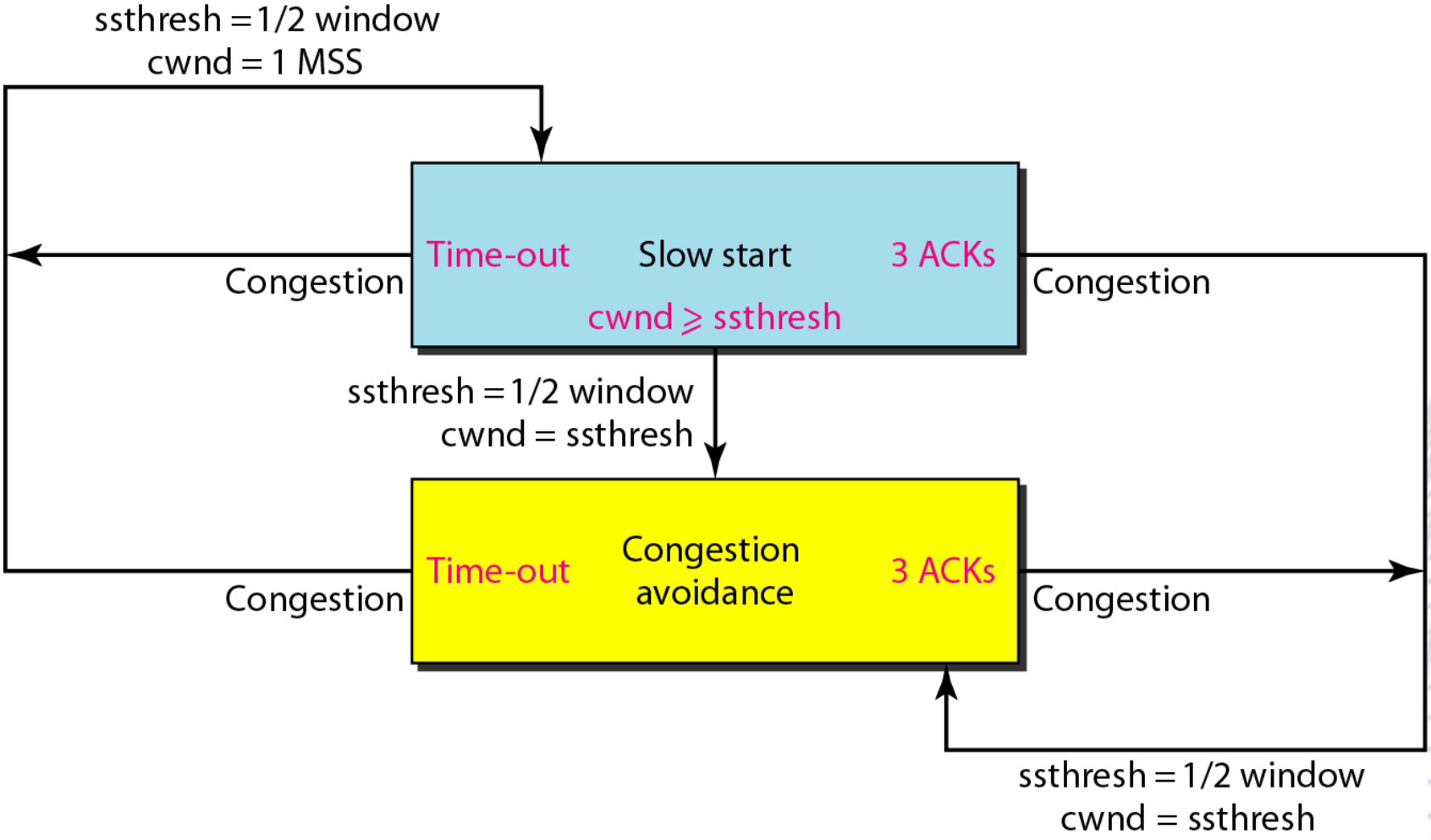
Congestione

- Quando ho un timeout devo diminuire la cwnd
- Questo viene fatto con una diminuizione moltiplicativa, la dimensione della finestra scende almeno a metà del valore precedente
- La necessità di ritrasmettere è sintomo di congestione
  - Questa è una causa molto probabile di timeout
- Oppure ritrasmette perché ha ricevuto 3 ACK duplicati. Questo tuttavia è un segnale meno forte perché gli ACK duplicati sono dovuti a segmenti successivi arrivati comunque a destinazione (anche se in ritardo)

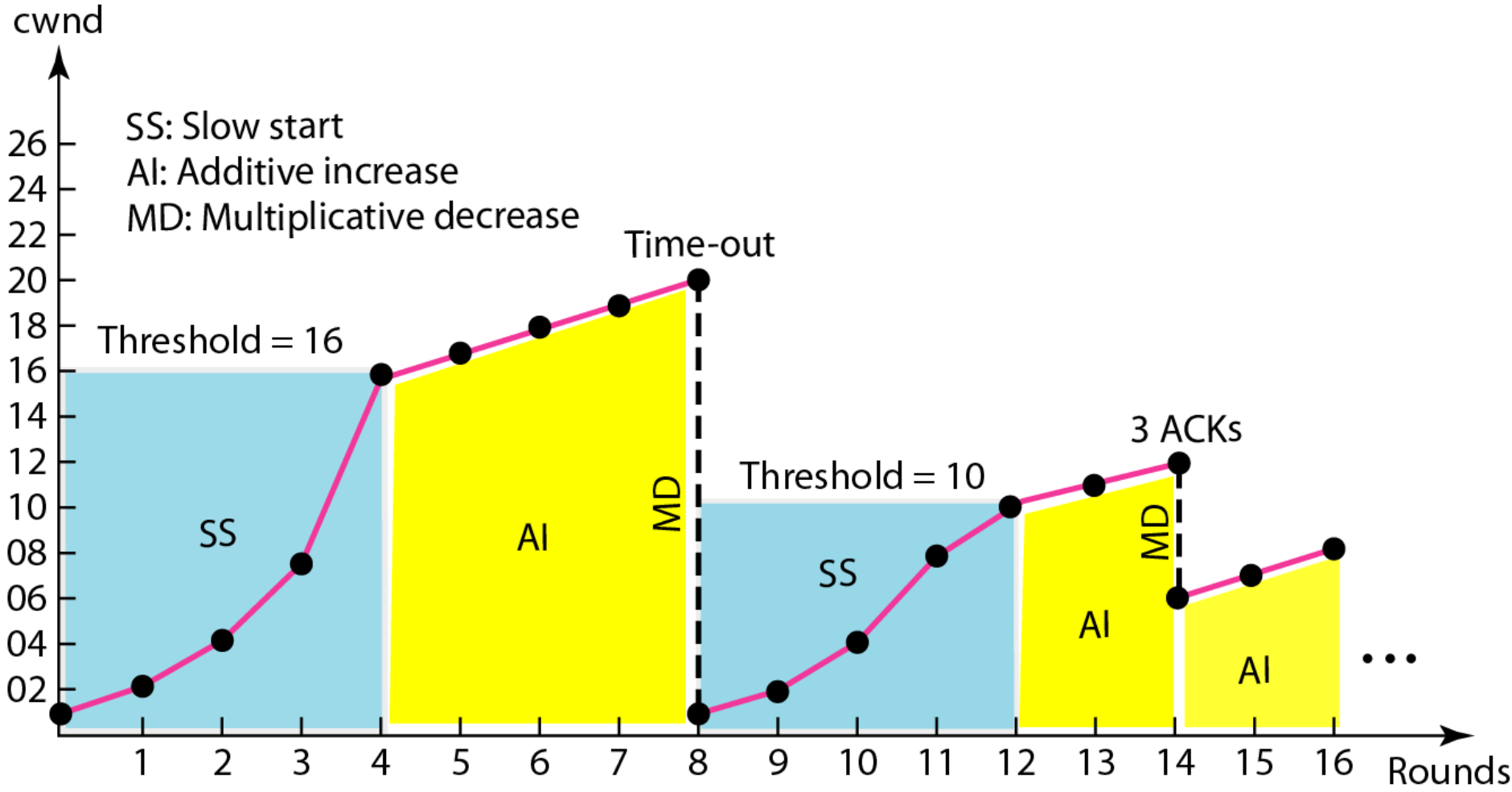
Due reazioni

- Se si ritrasmette a causa dello scadere del timeout TCP reagisce con decisione
  - **sshthresh** viene settata uguale alla metà della dimensione attuale della finestra
  - **cwnd** viene rimesso a 1 MSS
  - Si riprende lo slow start dall'inizio
- Se invece si ritrasmette per il terzo ACK duplicato
  - **sshthresh** viene settato pari alla metà della dimensione attuale della finestra
  - **cwnd** viene settato pari a sshthresh
  - Si riprende con un incremento additivo

Algoritmo di congestione



In azione

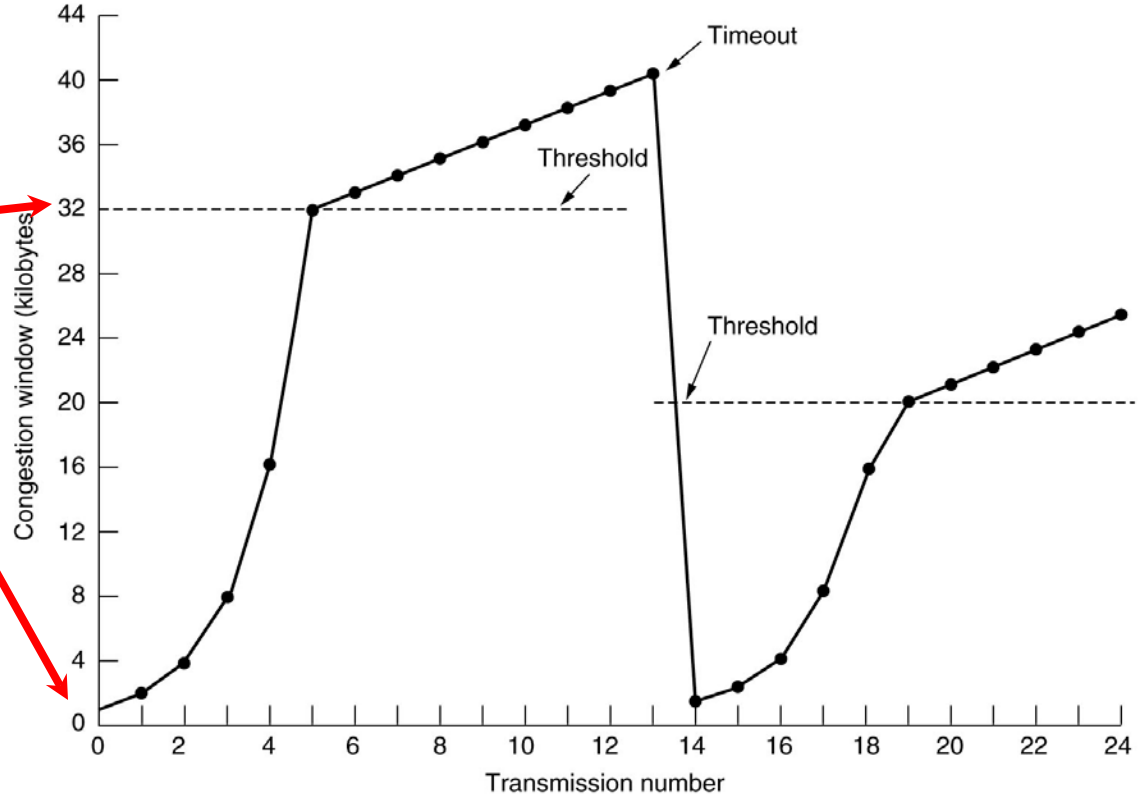

cwnd
SS: Slow start
AI: Additive increase
MD: Multiplicative decrease
Time-out
Threshold = 16
Threshold = 10
3 ACKs
SS
AI
MD
SS
AI
MD
AI
...
26
24
22
20
18
16
14
12
10
08
06
04
02
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Rounds

Congestione

- NB se arriva un pacchetto ICMP SOURCE QUENCH e viene passato a TCP, questo evento viene trattato come un timeout

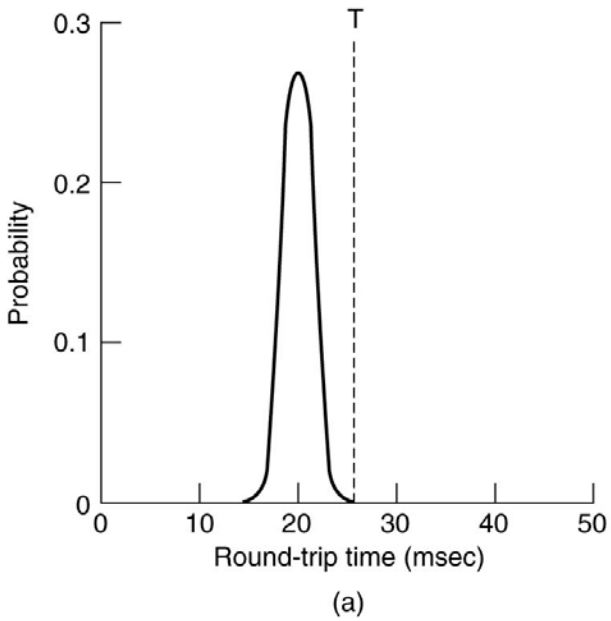
Algoritmo in azione

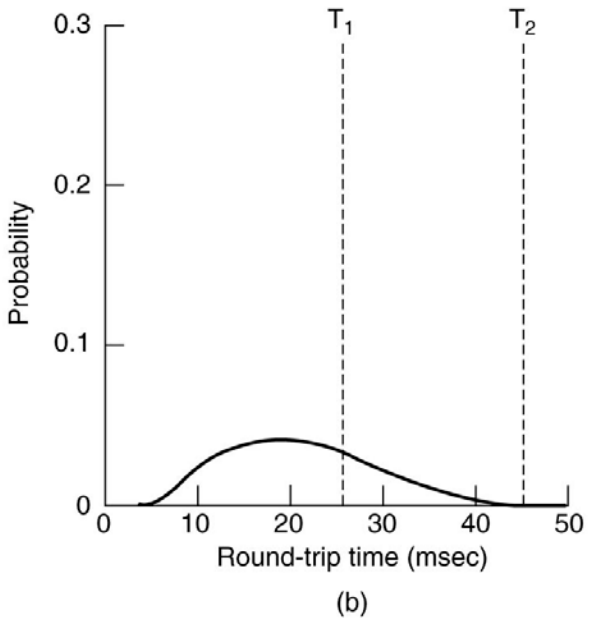
- Max segment size 1024 byte
- Congestion windows a 64KB
- Timeout a t=0 per cui Threshold va a 32 KB e congestion windows a 1KB
- Cresco esponenzialmente fino a 32 KB, poi linearmente fino a 40 e di nuovo a t=13 un nuovo timeout
- Nuova threshold a 20KB e ricomincio con lo slow start



Retrasmission timer

- Come setto il timeout entro cui aspetto un ack prima di ritrasmetterlo?
- In un datalink layer gli ack arrivano tutti vicini con una distribuzione stretta, quindi mi metto appena dopo la distribuzione
- A livello di trasporto la distribuzione è molto larga per cui se sto appena dopo la media vado troppo spesso in timeout (T1) ma se sto troppo lontano soffro di delay di ritrasmissione troppo lunghi (T2)
- Inoltre media e varianza cambiano rapidamente in pochi secondi, al formarsi e risolversi delle congestioni





Variazione del timeout

- Devo quindi calcolare dinamicamente il timeout
- L'algoritmo di Jacobson (1988) calcola un RTT che è la migliore stima del round trip time verso la destinazione in questione
- Quando un segmento parte il timer non solo controlla che non vada in timeout ma anche misura il tempo d'arrivo dell'ACK (*M*)
- Ci si calcola il nuovo $RTT = \alpha RTT + (1 - \alpha)M$
  - dove α è un fattore di smoothing che determina quanto pesa il vecchio valore di *RTT.*
  - Di solito $\alpha = 7/8$

Variazione del timeout

- A questo punto conosco un buon valore di RTT per cui aspetto βRTT ma il trucco è scegliere β.
- Inizialmente β vale 2 ma un valore costante non risponde bene ad aumenti della varianza
- Jacobson propone di renderlo proporzionale alla deviazione standard della distribuzione dei tempi di arrivo, per cui mi devo calcolare la deviazione media

$$D = \alpha D + (1 - \alpha) |RTT - M|$$

- come approssimazione della deviazione standard

Variazione del timeout

$$D = \alpha D + (1 - \alpha) |RTT - M|$$

- Dove α può essere lo stesso o diverso dal precedente
- Molte implementazioni di TCP settano il timeout pari a *RTT* + 4 x *D*
- Il numero 4 è arbitrario ma è facile da moltiplicare (due shift a sx) e minimizza timeout e ritrasmissioni dal momento che meno dell 1% dei pacchetti arriva dopo 4 standard deviations in ritardo

Persistence Timer

- **Persistence Timer**: Serve per prevenire il seguente deadlock
  - Il **retrasmission timer** è il più importante ma ce ne sono altri
  - Il receiver manda un ack con windows size=0 (come dire **“fermati!”**), in seguito manda un secondo pacchetto con la nuova windows ma questo secondo pacchetto viene perso
  - Ora sender e receiver stanno aspettandosi l’un l’altro
  - Quando il persistent timer si azzera il sender manda un pacchetto per sondare il ricevente. La risposta è la nuova windows size. Se è ancora zero si rimette in funzione il persistent timer e il ciclo ricomincia
  - Se non è zero posso invece mandare i miei dati

Keepalive timer

- Quando una connessione è idle da molto tempo questo timer può andare a zero
- Questo spinge uno dei due lati a controllare se l'altro è ancora in ascolto
- Se non risponde la connessione viene terminata.
- Questa è una funziona controversa perché aggiunge overhead e può terminare una connessione in salute per colpa di una problema di partizionamento di rete momentaneo
- L'ultimo timer è quello usato nello stato TIMED WAIT al momento della chiusura. Dura due volte il max packet lifetime per sincerarsi che quando una connessione è chiusa tutti i suoi pacchetti siano spirati

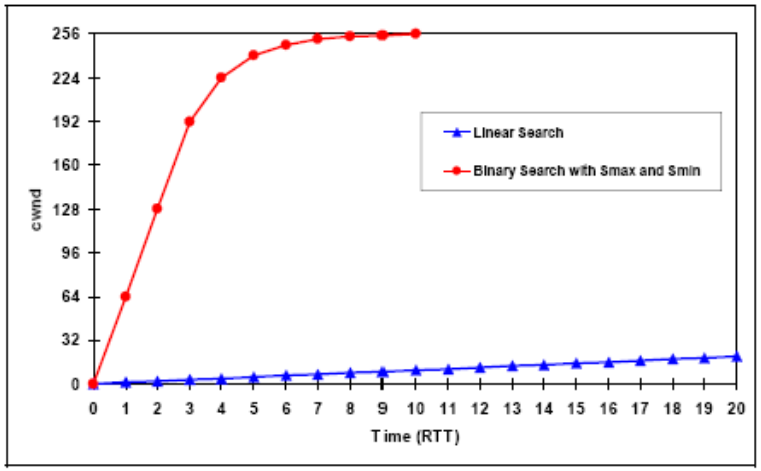
Diverse Versioni

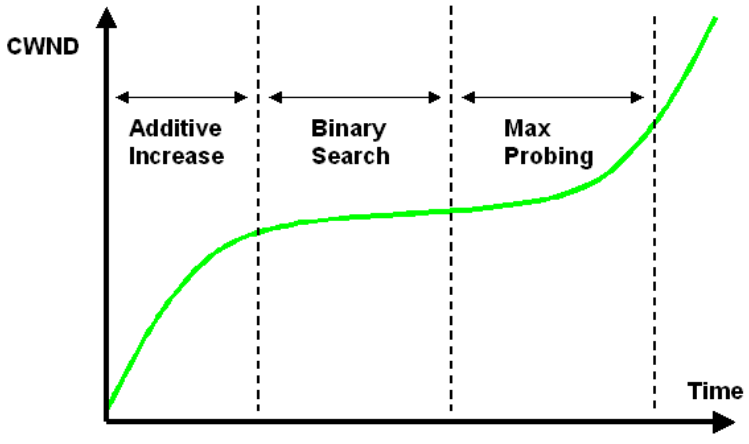
- Il meccanismo di gestione delle congestioni che abbiamo visto si trova nelle versioni Tahoe/Reno
- Ci sono altre versioni con meccanismi diversi, in particolare per le reti LFN (Large Fat Network) in cui il prodotto Banda * Latenza è molto grande (quindi link transatlantici a Gigabit)
- BIC (binary Increase Congestion) usato di default nei kernel Linux da 2.6.8 a 2.6.18
- CUBIC che è un po' meno aggressivo e che è di default nei linux dalla 2.6.19

TCP BIC

- Crescita logaritmica del parametro cwnd
- Algoritmo di binary search

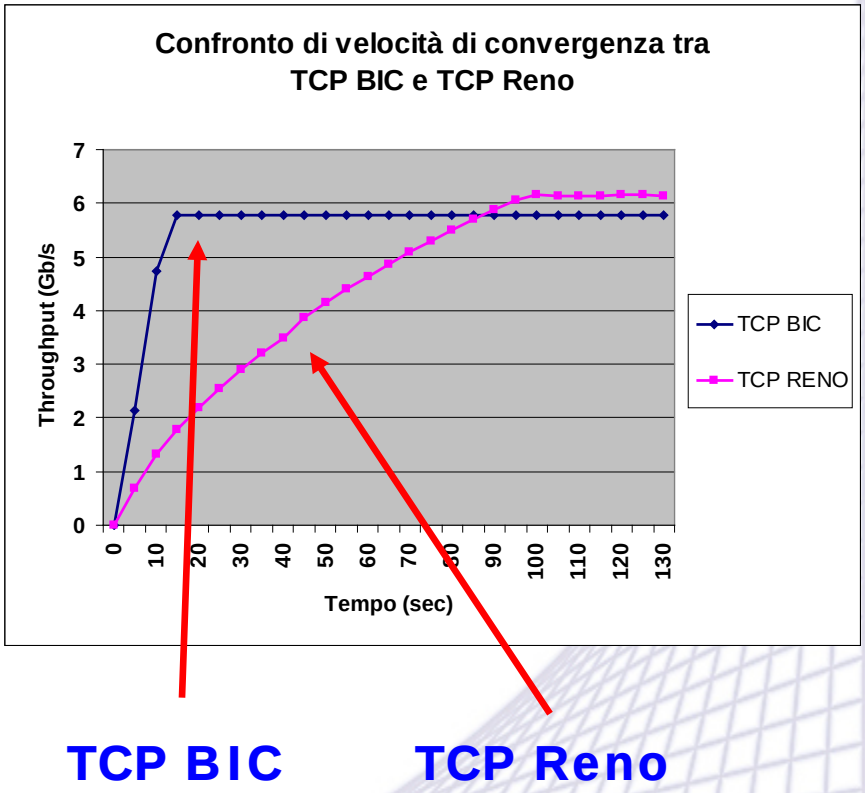
TCP BIC (Binary Increase Congestion control)



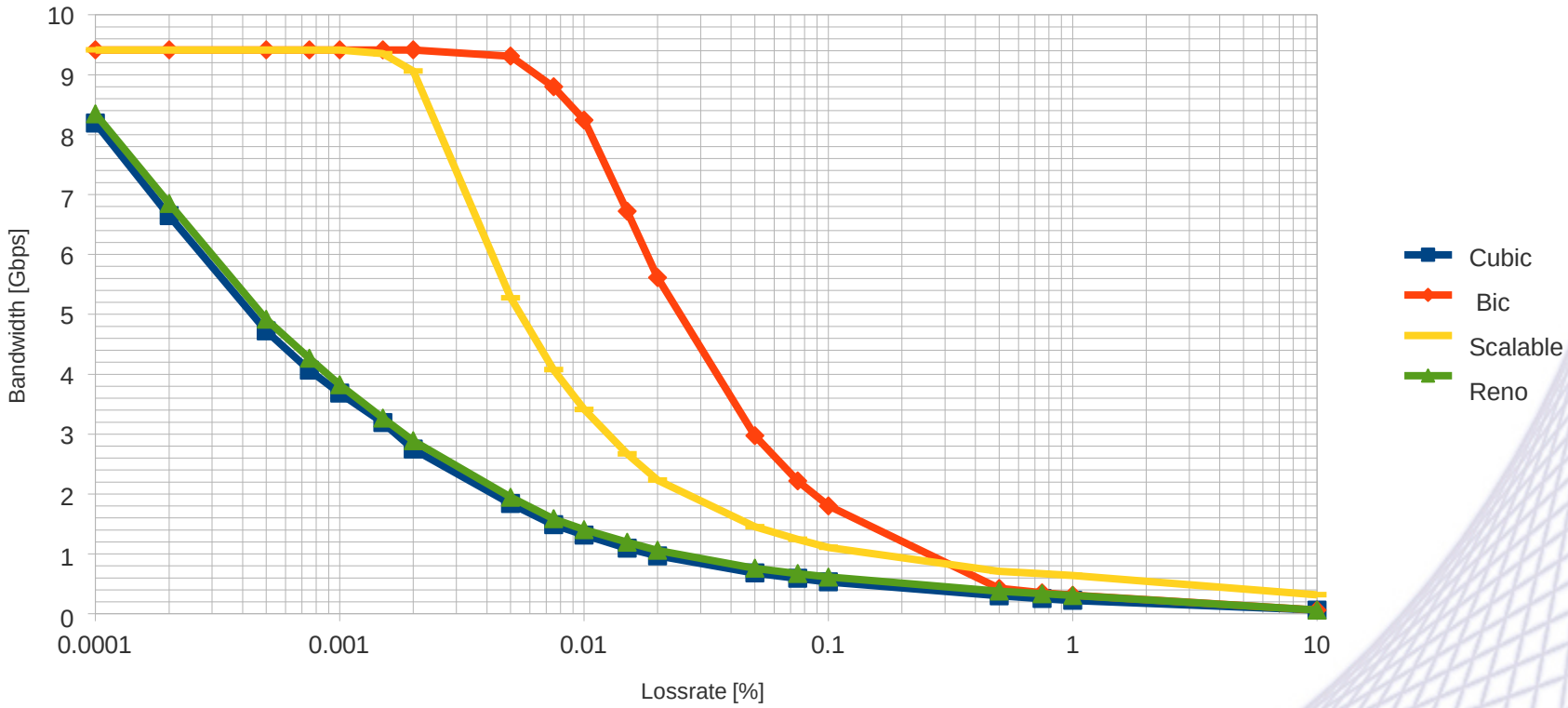


Confronto TCP BIC vs RENO

- Confronto Kernel 2.6 e kernel 2.4
  - 2.4 implementa di default il protocollo **TCP Reno**
  - 2.6 implementa **TCP BIC** e una serie di nuovi algoritmi di scheduling di processi
- TCP BIC rispetto a TCP Reno ha un incremento logaritmico della finestra di congestione TCP invece che lineare, ciò garantisce:
  - Maggiore rapidità di convergenza
  - TCP Fairness: trattamento equo di flussi distinti con stesso RTT
  - RTT Fairness: trattamento equo per flussi con RTT diverso

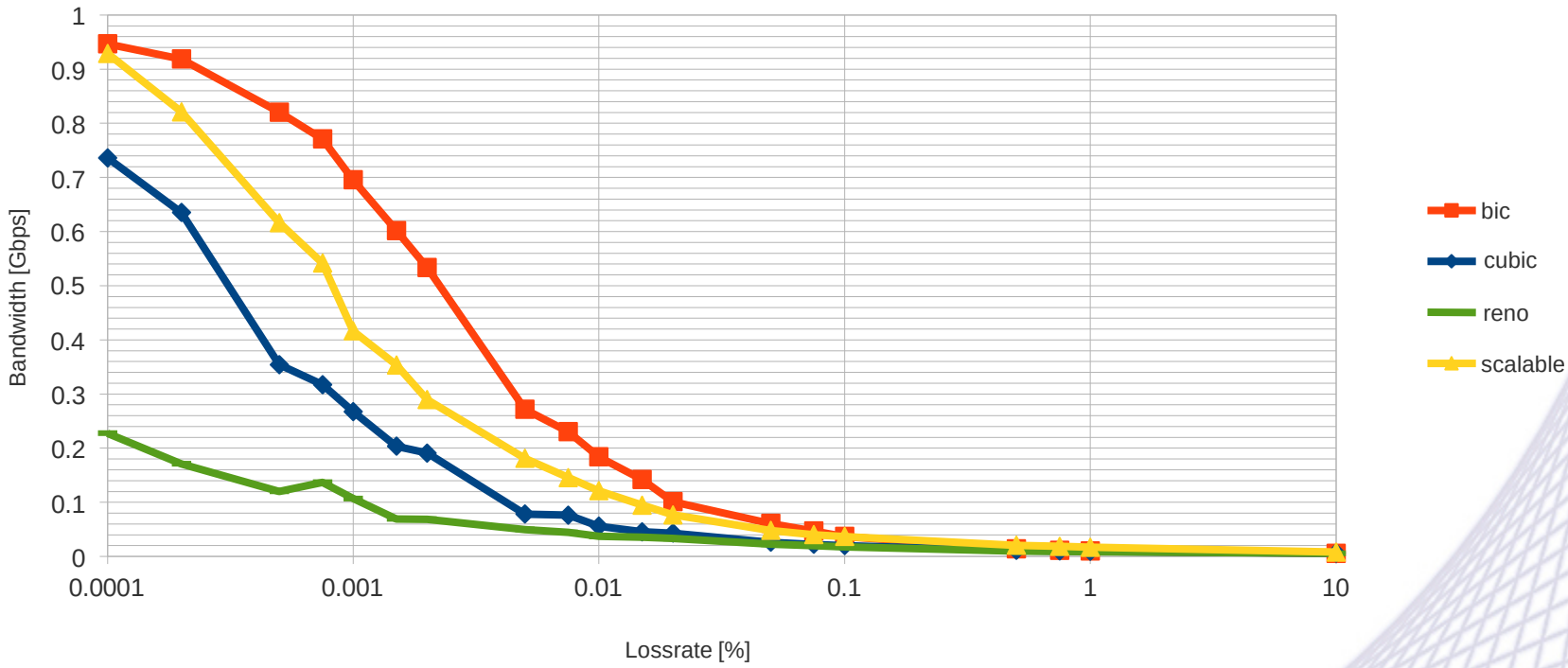


10 Gbps - 0 ms delay



25ms delay, default settings

PROBLEM



Tuning dei parametri

- $W_{25} = B * RTT = 10\text{Gps} * 25\text{ms} \approx 31.2\ MB$

default settings:

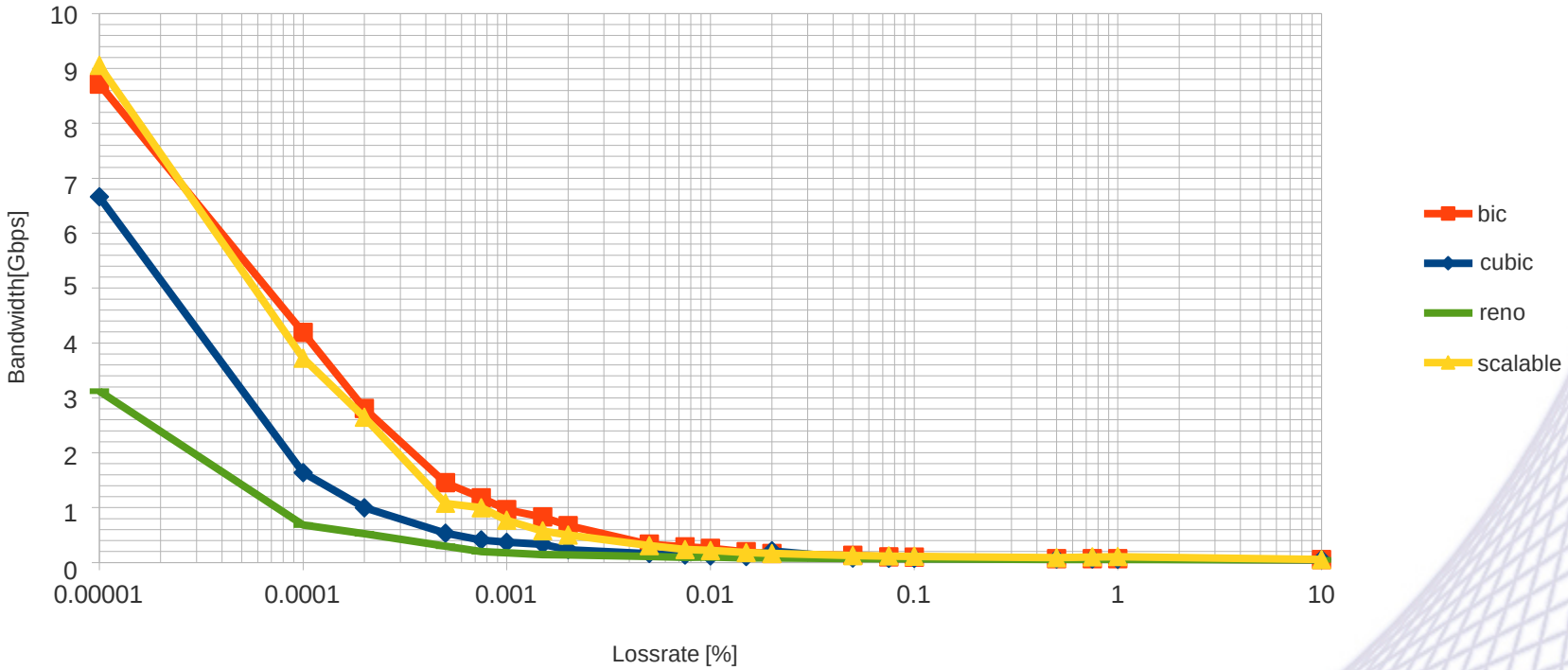
net.core.rmem_max = 124K
net.core.wmem_max = 124K
net.ipv4.tcp_rmem = 4K 87K 4M
net.ipv4.tcp_wmem = 4K 16K 4M
net.core.netdev_max_backlog = 1K

We tune **TCP settings** on **both server and client**.

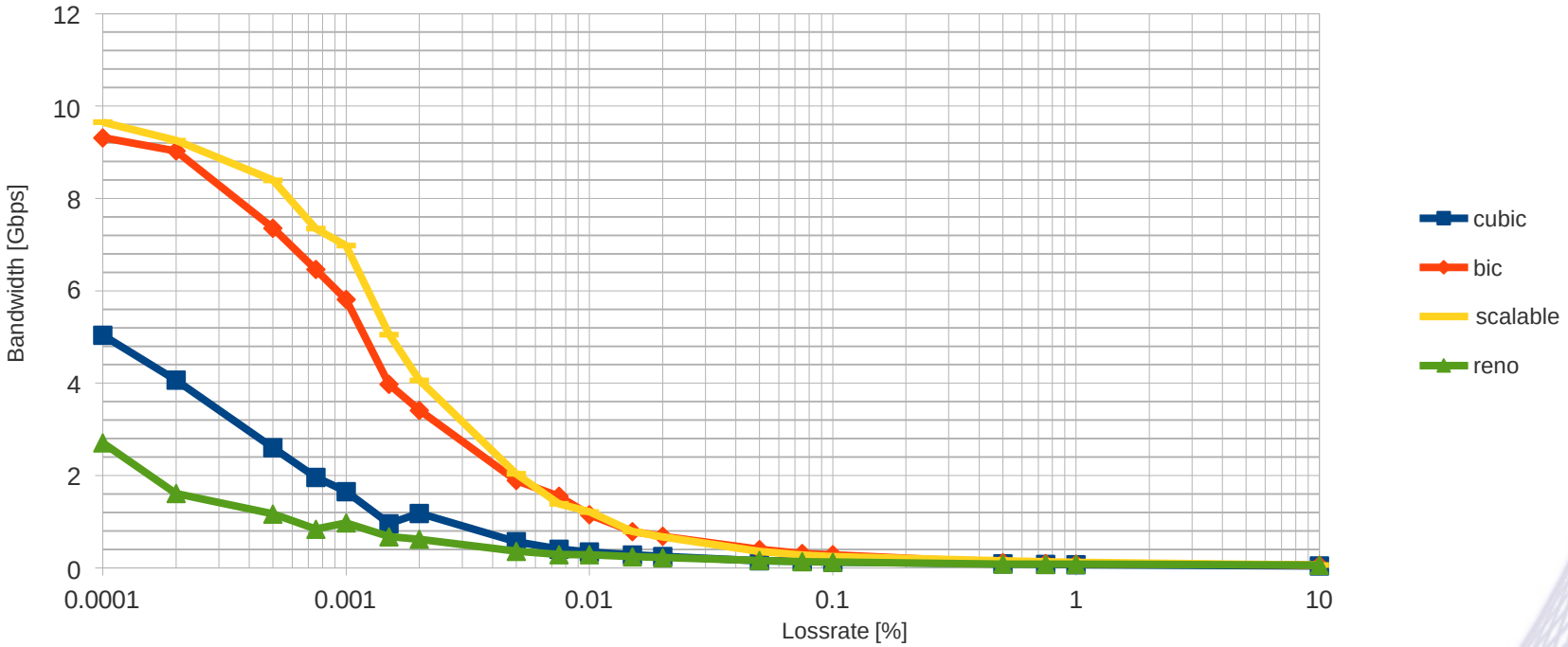
Tuned settings:

net.core.rmem_max = 67M
net.core.wmem_max = 67M
net.ipv4.tcp_rmem = 4K 87K 67M
net.ipv4.tcp_wmem = 4K 65K 67M
net.core.netdev_max_backlog = 30K

25 ms delay, tuned settings



25 ms delay, tuned + jumbo

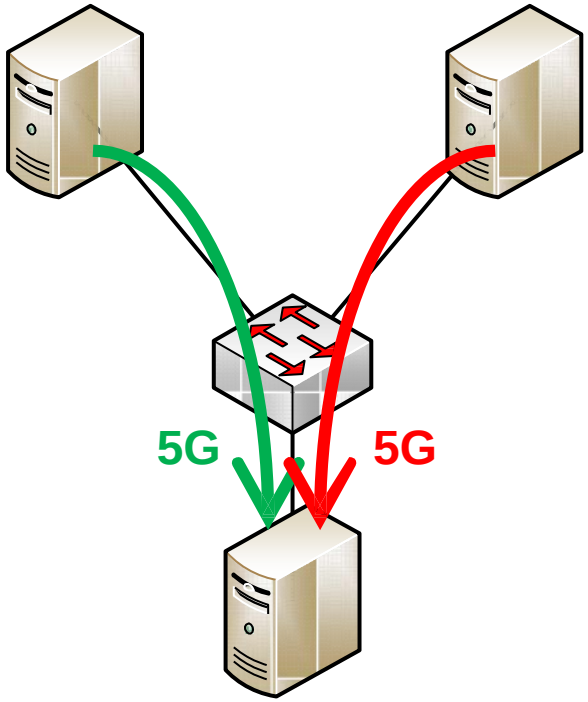


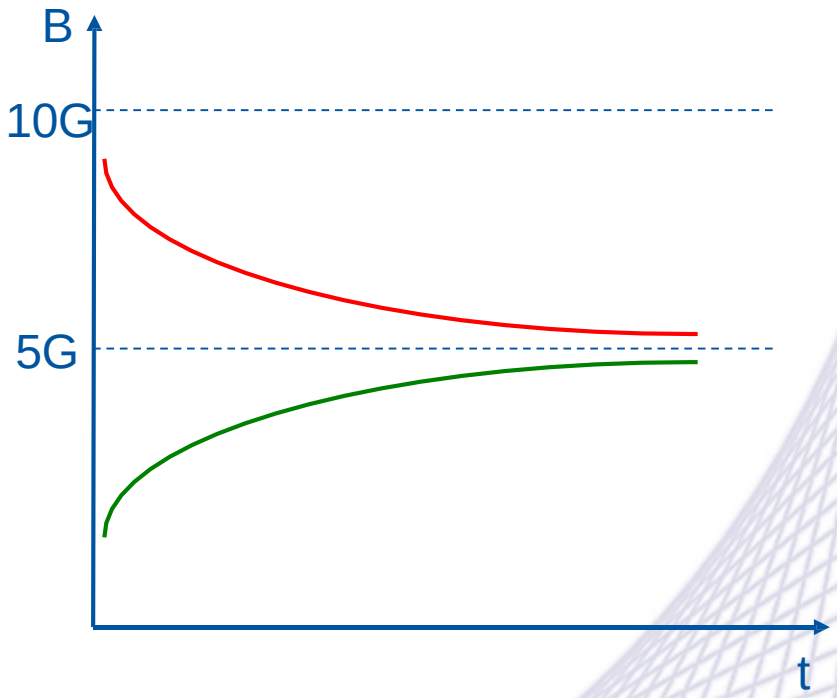
Better performance with jumbo frames

TCP window growth function is boosted by bigger MTU
CUBIC is the default algorithm in Linux kernel since 2.6.19
BIC was the default before but was changed with this commit msg:
”Change default congestion control used from BIC to the newer CUBIC which is the successor to BIC but has better properties over long delay links.”

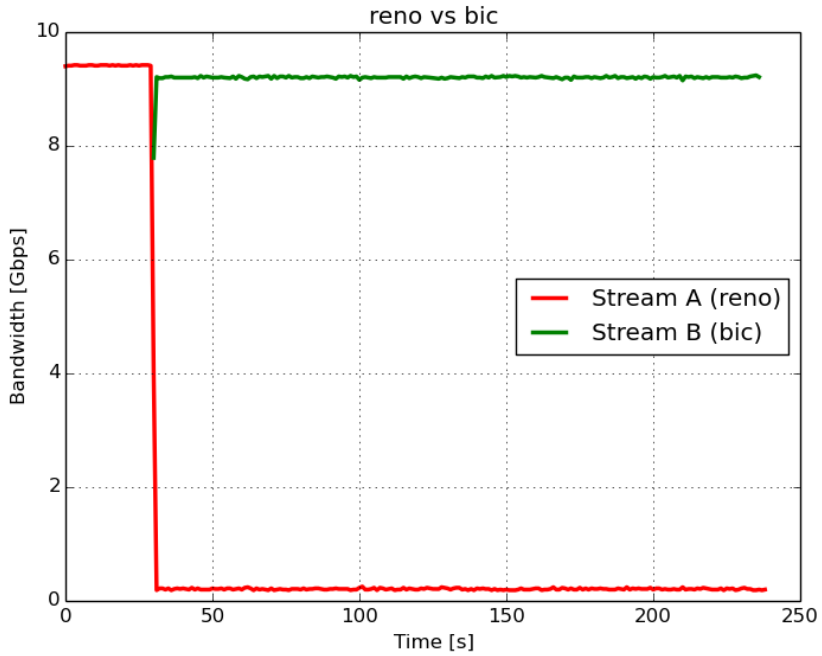
Bandwith fairness

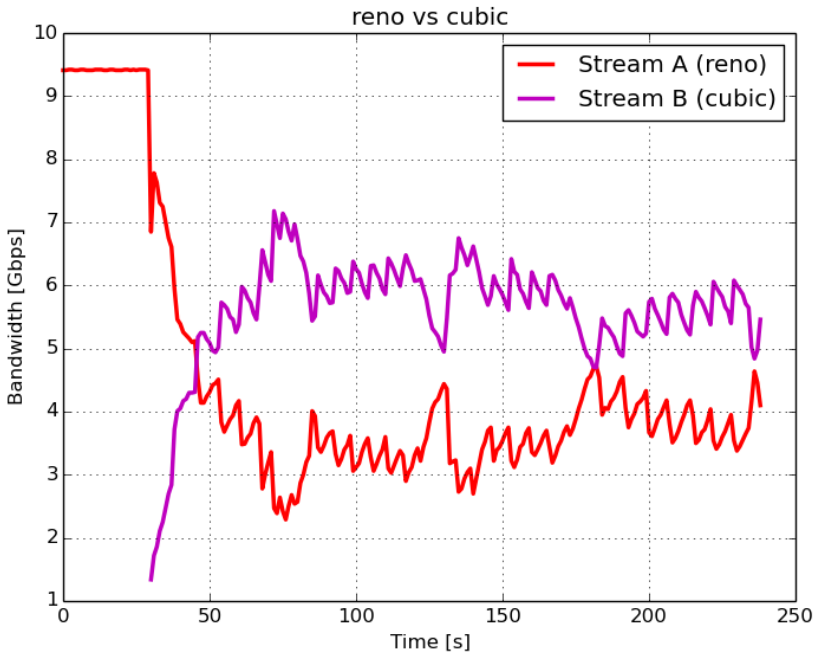
- **Goal:** Each TCP flow should get a **fair share of available bandwidth**





Fairness





BIC stream is unfair to CUBIC stream

BIC has better performance on lossy links, but it is more aggresive

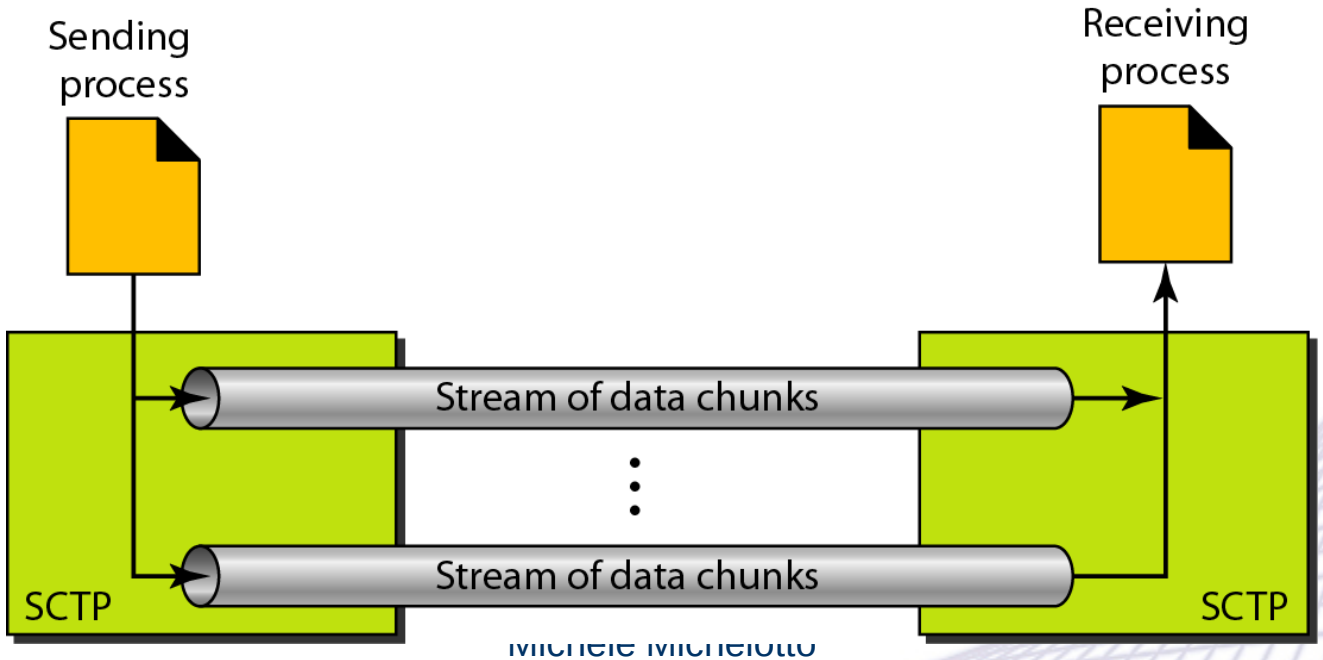
SCTP

- Nuovo protocollo di trasporto di internet per applicazioni che richiedono servizi non forniti da UDP o TCP
- Come TCP ha connessione, rileva dati mancanti, dati duplicati, controllo del flusso e della congestione

Protocol	*Port Number*	*Description*
IUA	9990	ISDN over IP
M2UA	2904	SS7 telephony signaling
M3UA	2905	SS7 telephony signaling
H.248	2945	Media gateway control
H.323	1718, 1719, 1720, 11720	IP telephony
SIP	5060	IP telephony

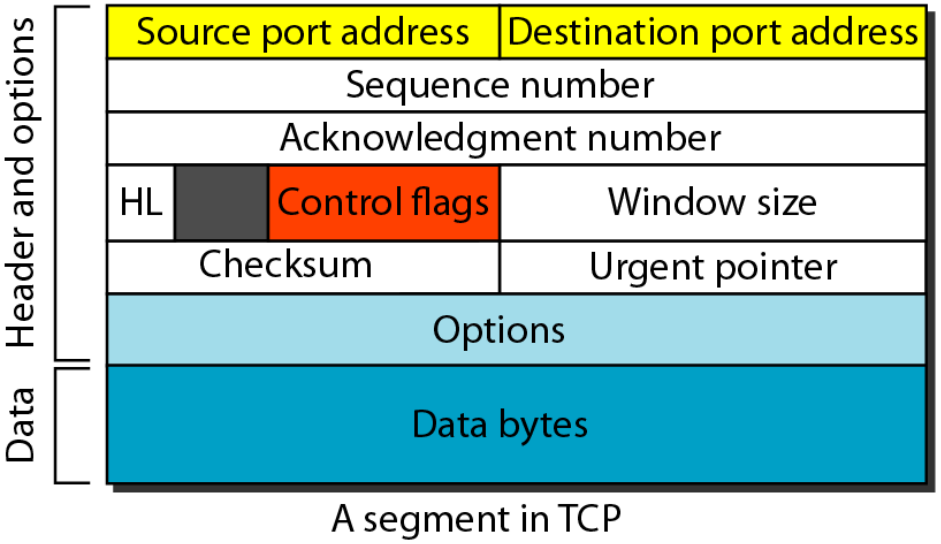
Flussi

- Mentre TCP permette un flusso di dati per ogni connessione SCTP permette flussi multipli per ogni associazione (l'equivalente della connessione)
- Se un flusso viene bloccato da un errore gli altri possono continuare a funzionare

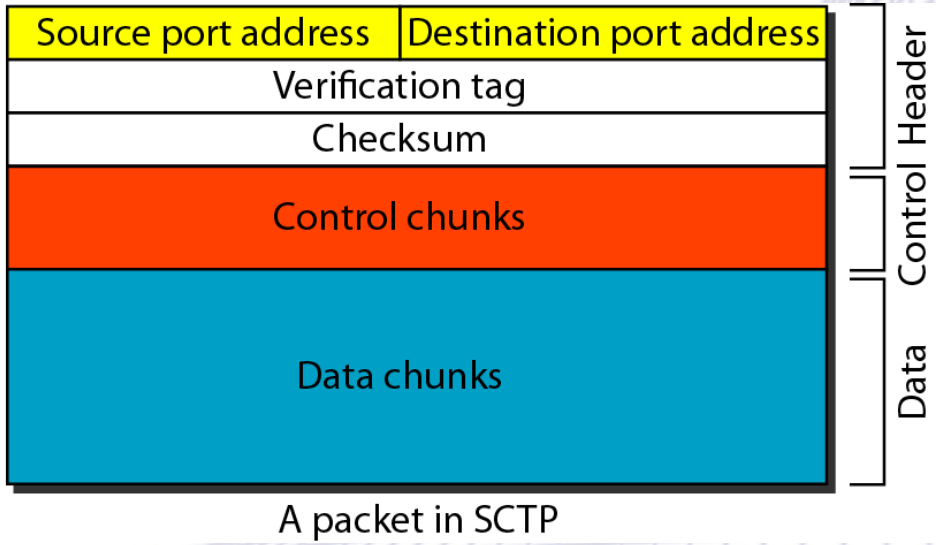


Pacchetti e segmenti

- Le unità di dati si chiamano pacchetti mentre in TCP si chiamavano segmenti
  - Un pacchetto SCTP può contenere diversi chunk di dati, ognuno dei quali viene numerato con un TSN
  - Ci sono chunks di controllo e chunks di dati



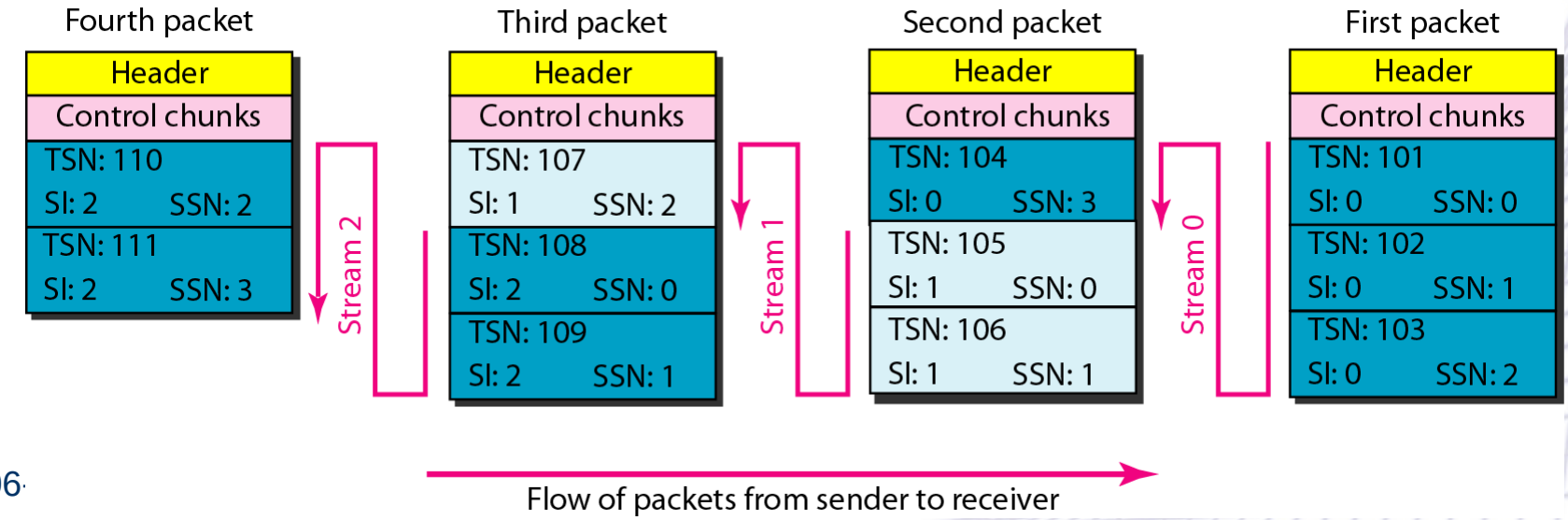
A segment in TCP



A packet in SCTP

Pacchetti e segmenti

- Per distinguere tra i diversi flussi (stream) di una stessa associazione viene assegnato un SI (stream identifier)
- Per mantenere l'ordinamento dei chunk di uno stream, SCTP utilizza numeri di sequenza di stream (SSN)
- Quindi un chunk di dati viene identificato da tre numeri TSN, SI e SSN, il TSN relativo all'intera associazione mentre il SSN è relativo al flusso



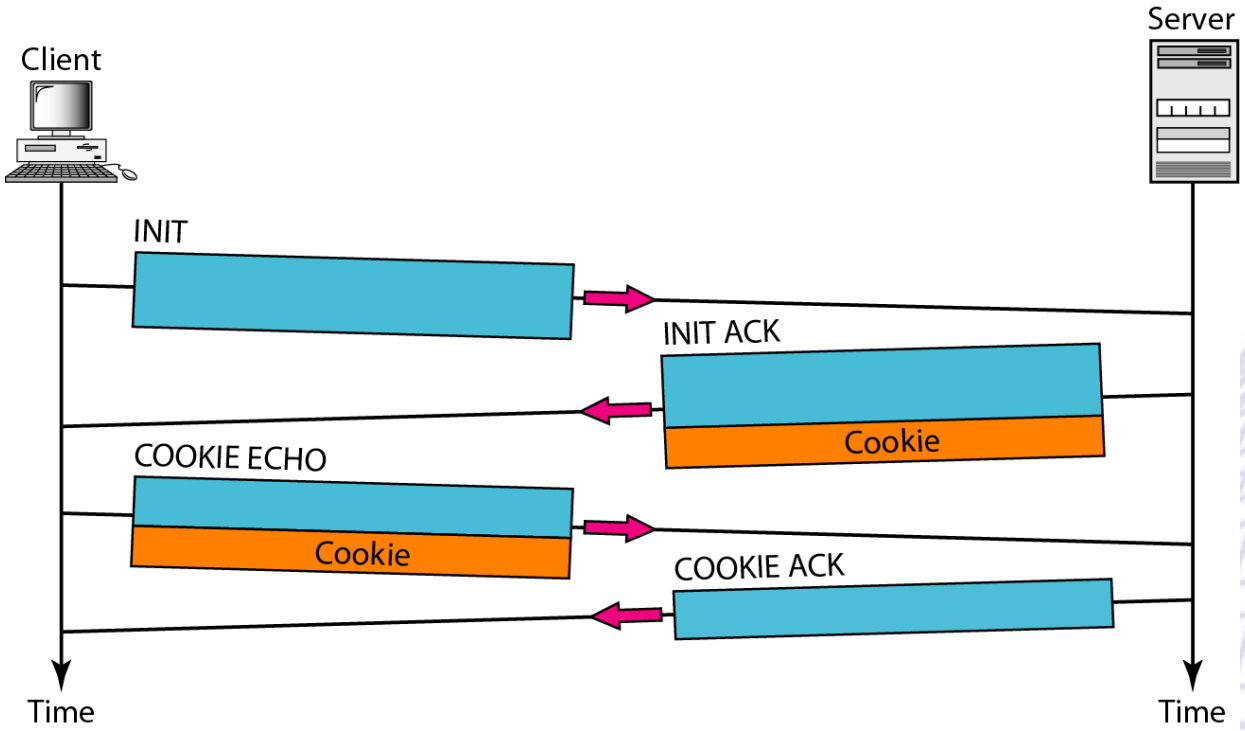
ACK

- I numeri per gli ACK vengono utilizzati per dare l'aknowledgement per i chunk dei dati
- I chunk di controllo vengono riscontrati con chunk di controllo
- Prima ci sono i chunk di controllo e poi quelli di dati

Type	*Chunk*	*Description*
0	**DATA**	User data
1	**INIT**	Sets up an association
2	**INIT ACK**	Acknowledges INIT chunk
3	**SACK**	Selective acknowledgment
4	**HEARTBEAT**	Probes the peer for liveliness
5	**HEARTBEAT ACK**	Acknowledges HEARTBEAT chunk
6	**ABORT**	Aborts an association
7	**SHUTDOWN**	Terminates an association
8	**SHUTDOWN ACK**	Acknowledges SHUTDOWN chunk
9	**ERROR**	Reports errors without shutting down
10	**COOKIE ECHO**	Third packet in association establishment
11	**COOKIE ACK**	Acknowledges COOKIE ECHO chunk
14	**SHUTDOWN COMPLETE**	Third packet in association termination
192	**FORWARD TSN**	For adjusting cumulative TSN

4 way handshaking

- Init e Init Ack non possono avere dati
- Cookie e Cookie Echo invece possono
- Le risorse non vengono allocate fino a quando non c'è lo scambio di cookie, in questo modo non è esposto ad attacchi SYN FLOODING



Chiusura 3way handshake