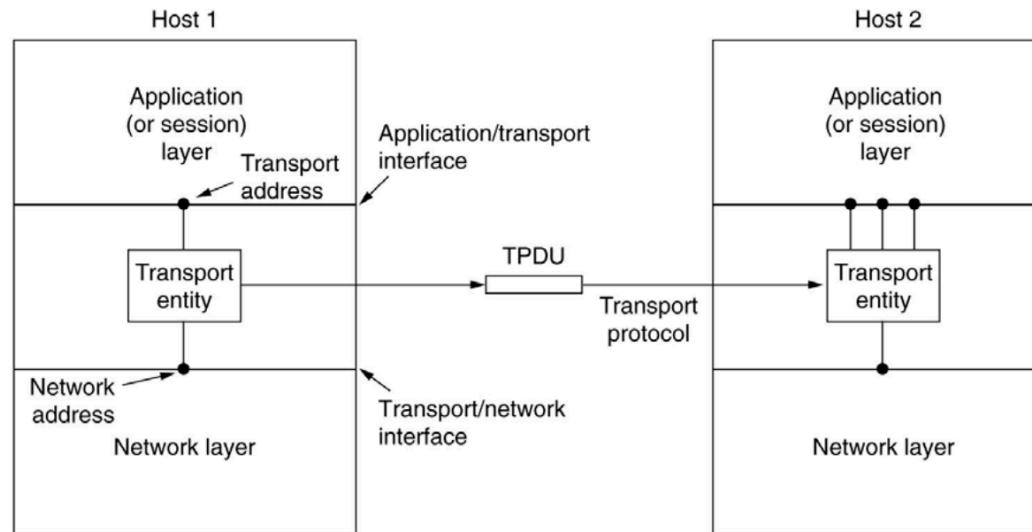


Il Livello Transport

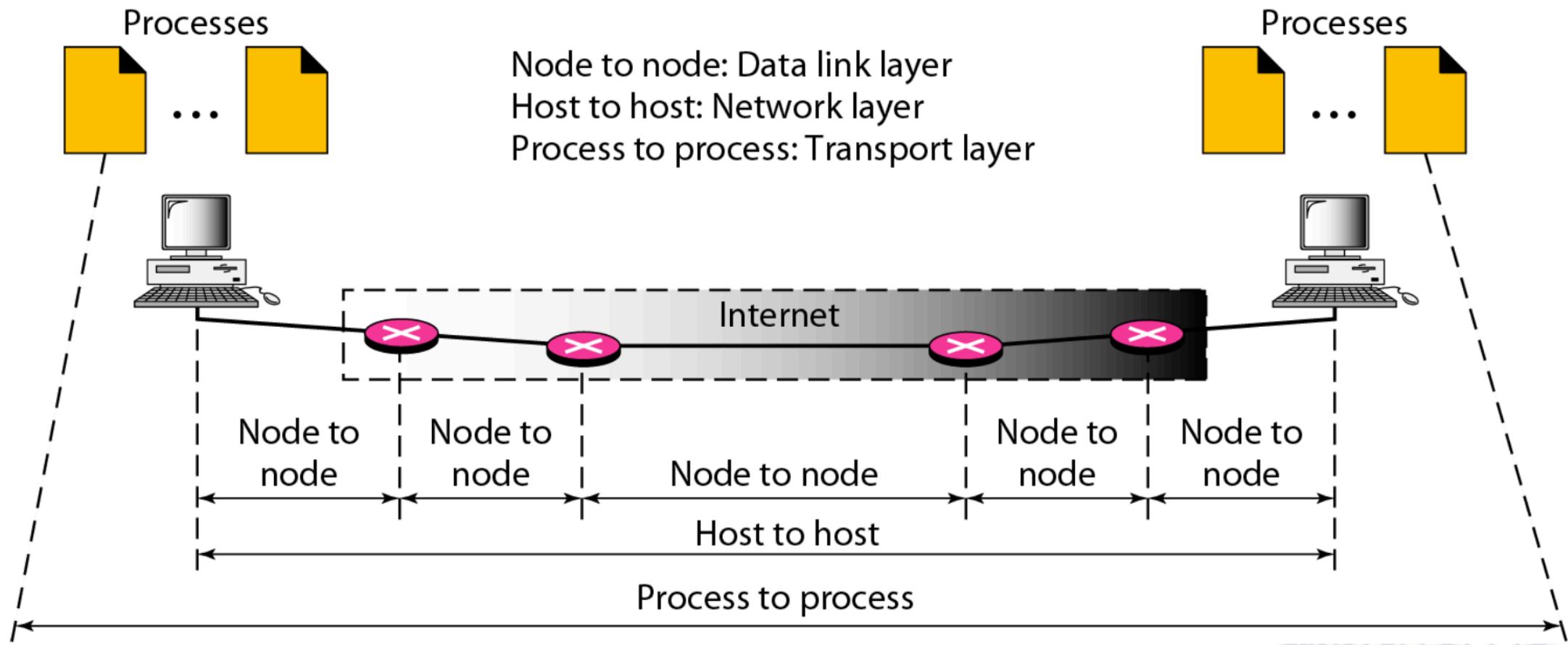
A cosa serve?

- Deve fornire un servizio di trasporto efficiente ed affidabile ai suoi clienti, normalmente processi del livello applicativo
- Si serve dei servizi forniti dal livello di network
- Solitamente è implementato nel **kernel** del sistema operativo o in processo separato dello spazio **user** in una libreria del sistema operativo.



NOTA: Nei sistemi operativi, lo spazio utente è uno spazio per applicazioni eseguite in modalità utente esterno al kernel e protetto attraverso la separazione dei privilegi. Si può riferire a un set di librerie per effettuare operazioni di input/output o interagire con il kernel dalle applicazioni. Si può anche riferire a componenti di sistema che non fanno parte del kernel come le shell o altre utilità per manipolare gli oggetti all'interno del file system collettivamente indicate col termine userland.

Da processo a processo



Due tipi di servizio

A livello trasporto ci sono due tipi di servizio:

- **Connection Oriented**
- **Connectionless**
- Il servizio di trasporto connection oriented somiglia molto a quello equivalente a livello 3 OSI:
 - connessione in tre fasi: setup, trasferimento dati, disconnessione
 - Indirizzamento, flow control
- Anche il livello connectionless esegue le stesse funzioni del livello connectionless a livello 3 OSI.

Perchè anche il livello Transport?

- Il livello di trasporto gira sulla macchina dell'utente, mentre il livello network gira anche nei router che sono gestiti dagli operatori della rete
- Cosa succede se una rete offre servizi non adeguati? Es: se perde pacchetti, o i router cadono spesso
- L'utente non ha controllo sulla rete, l'unica cosa che si può fare localmente è mettere un altro livello che migliora la qualità del servizio fornito dalla rete

Il livello Transport permette di avere un servizio più affidabile della rete sottostante.

Perchè anche il livello Transport?

- Quindi Transport si occupa di pacchetti persi, o rovinati, di come scoprirli e di correggerli o richiederne la trasmissione
- Le funzioni base del livello di trasporto sono chiamate a funzioni di libreria per renderle indipendenti dalle funzioni base del sistema di network.
- Queste ultime possono variare da rete a rete (tipicamente quelle di una rete LAN connectionless sono diverse da quelle di una WAN connection oriented)
- Queste differenze a livello di network sono nascoste da un insieme di funzioni base (primitives), per cui cambiare i servizi di rete comporta solo chiamare un insieme diverso di librerie

Provider e User

- I tre layer inferiori al transport si possono vedere come Transport Service **Provider**
- I restanti layer superiori sono Transport Service **User**

Primitive

Il livello di trasporto come abbiamo detto deve essere affidabile anche se la rete sottostante non lo è.

Prendiamo come esempio due processi connessi da una pipe Unix:

- I due processi assumono che la connessione tra i due sia perfetta
- Non vogliono sapere nulla di pacchetti persi, ack, congestione etc
- Vogliono solo che i dati che entrano da una parte escano dall'altra

Facilità d'uso

- Le **primitive** di trasporto sono usate dai normali programmatori mentre pochi utenti si scrivono le proprie funzioni di trasporto.
- Le **primitive** di trasporto devono quindi essere facili da usare.

Esempio

Esempio di primitive di trasporto, ridotte all'osso

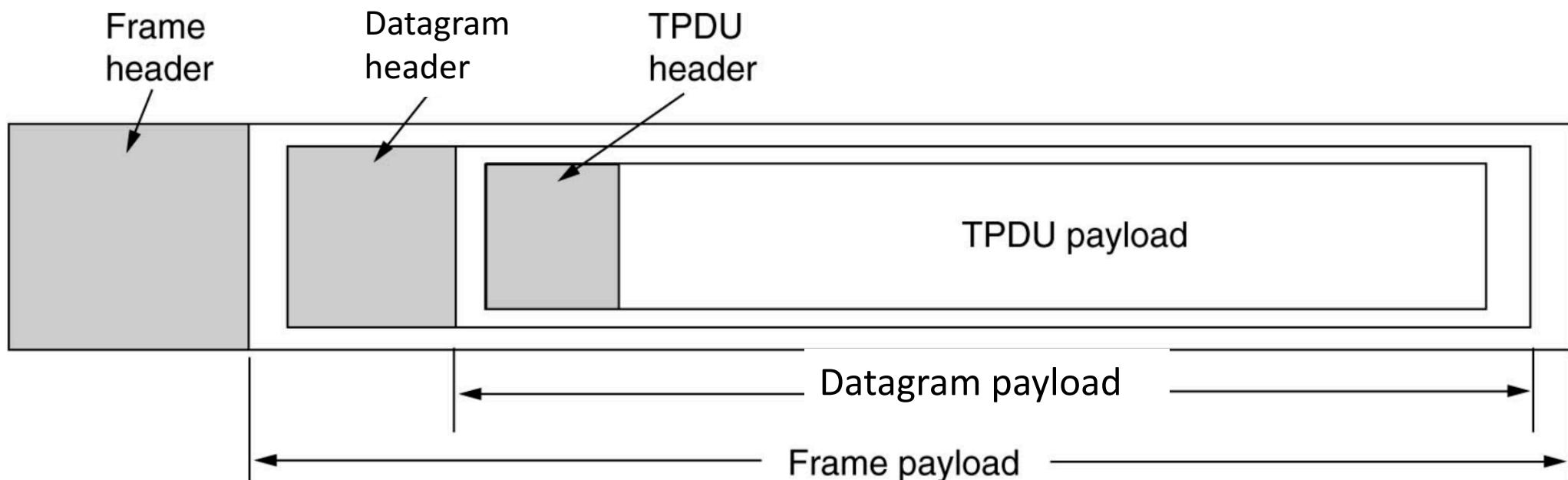
Prendiamo un server e un client.

- Il server chiama una LISTEN, procedura di libreria che fa una system call e blocca il server fino a quando un client non si fa vivo
- Il client esegue una CONNECT, eseguita bloccando il chiamante e mandando un pacchetto al server. Dentro il payload di questo pacchetto c'è il messaggio di transport layer per l'entità di trasporto del server.

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

TPDU

Chiamiamo TPDU, Trasport Protocol Data Unit il pacchetto di livello Transport:



Connessione

Quando il client chiama la CONNECT, una TPDU di CONNECTION REQUEST arriva al server.

L'entità di trasport del server vede se il server è in ascolto, cioè bloccato su una LISTEN, e in caso positivo, sblocca il server e manda indietro una CONNECTION ACCEPTED TPDU, questa sblocca il client e la connessione risulta stabilita.

Trasferimento dati

Ora client e server si possono mandare dati, usando le primitive SEND e RECEIVE.

Es: uno dei due fa una chiamata bloccante a RECEIVE aspettando che l'altro faccia una SEND.

Quando la TPDU arriva, il ricevente viene sbloccato, quindi può processare la TPDU e mandare un reply.

Tale approccio è semplice e funziona bene solo se entrambe le parti sono d'accordo sui turni....

Complicazioni

Notiamo che a livello transport anche un semplice scambio di dati unidirezionale è molto più complicato che a livello network.

- Ogni pacchetto di dati riceve un acknowledgement, implicitamente o esplicitamente. Anche i pacchetti che portano le TPDU di controllo vengono ACKed
- Questi ACK vengono gestiti dall'entità di trasporto, che fa uso del protocollo di network, non visibile agli utenti del trasporto
- **Per gli utenti del livello di trasporto la connessione è come una bit pipe, si infilano i bit da un lato e questi riappaiono per magia all'altro lato.**
- Ci pensano le transport entities a gestire timer e ritrasmissioni.

Disconnessione

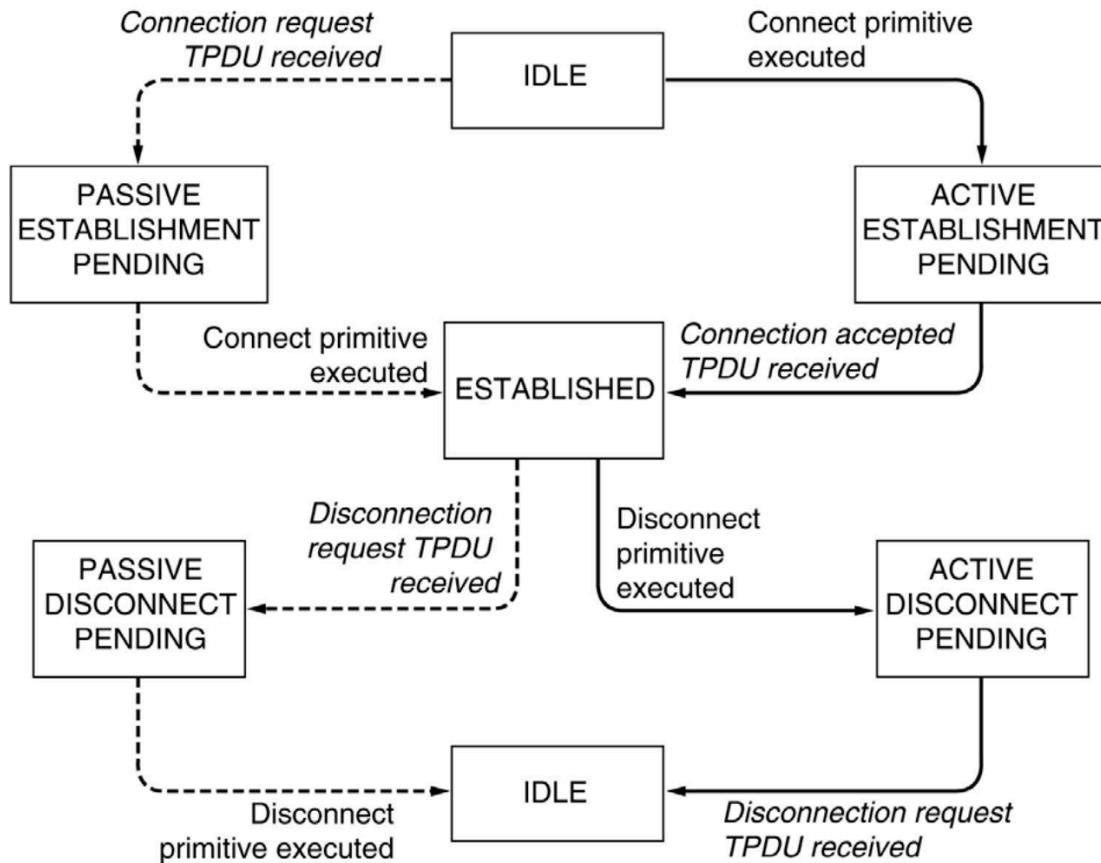
Asimmetrica:

- Uno dei due manda una **DISCONNECT TPDU** all'altro e considera la connessione chiusa
- All'arrivo della TPDU la connessione è chiusa anche per l'altro

Simmetrica:

- Ogni direzione viene chiusa in modo indipendente.
- Quando uno dei due non vuole trasmettere manda una **DISCONNECT** ma è ancora disposto a ricevere dati dall'altro.
- In questo modello la connessione è chiusa **solo quando entrambi** hanno fatto una **DISCONNECT**

Diagramma di stato



- La **linea continua** rappresenta la sequenza del **client**
- La **linea tratteggiata** quella del **server**
- Transizioni in *corsivo* sono causate dall'*arrivo* di un **pacchetto**

Berkeley Socket

- Sono l'insieme “reale” di primitive, molto usate per programmazione in ambiente Internet
- Molto simili a quelle “basic” già viste nel primo modello ma qui abbiamo tutti i dettagli e la flessibilità necessari.

Per il momento trascuriamo la TPDU che vedremo più avanti quando studieremo TCP.

Primitive del socket Berkeley TCP

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

SOCKET

- **SOCKET.** Crea un end point e alloca spazio nel server
- I parametri indicano il formato di indirizzamento da usare, il tipo di servizio desiderato (es. reliable data stream) e il protocollo
- Una chiamata eseguita con successo restituisce un *file descriptor* da usare nelle call successive
- In questo senso si comporta come una **OPEN** unix su di un file

Parentesi: file descriptor

In Unix and related computer operating systems, a file descriptor (FD, less frequently fildes) is an abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket.

File descriptors form part of the POSIX application programming interface.

A file descriptor is a non-negative integer, generally represented in the C programming language as the type int (negative values being reserved to indicate "no value" or an error condition).

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

BIND

- Un socket appena creato non ha un indirizzo di rete. Questo viene assegnato dalla **BIND**
- Solo a questo punto si può fare la connessione
- Il SOCKET non si crea direttamente l'indirizzo perché alcune applicazioni possono volersi scegliere l'indirizzo, mentre per altre è irrilevante o inutile.

LISTEN

- La **LISTEN** alloca dello spazio per accodare chiamate entranti, nel caso diversi client vogliano connettersi allo stesso istante
- Nei Socket Berkeley la **LISTEN non è bloccante!**

ACCEPT

- Per bloccarsi in ascolto il server chiama una **ACCEPT**
- Quando arriva una TPDU il server crea un nuovo socket con le stesse proprietà di quello originale e ritorna un file descriptor per esso
- Il server può creare (con una *fork*) un nuovo processo o un nuovo thread per gestire la connessione sul nuovo socket e tornare ad aspettare la prossima connessione

Lato client

- Anche dal lato cliente deve prima essere creato un **SOCKET** ma **BIND** non è richiesto, l'indirizzo usato non interessa al server
- La **CONNECT** blocca il chiamante e attiva l'inizio del processo di connessione
- Quando la connessione è completa (ha ricevuto l'appropriata TPDU dal server) il processo client viene sbloccato.

Client e Server

- Ora entrambi possono usare **SEND** e **RECV** per trasmettere e ricevere dati sulla connessione full-duplex
- Si possono anche usare normali chiamate Unix **READ** e **WRITE** se non c'è bisogno di usare le opzioni speciali di SEND e RECV
- La chiusura della connessione è **simmetrica**, solo quando entrambe hanno chiamato la **CLOSE** la connessione è chiusa

Transport vs Host to Network

A prima vista un protocollo di trasporto assomiglia ad un protocollo host to network

- Entrambi devono gestire controllo degli errori, sequenze, flow control tra le altre cose
- Tuttavia mentre a livello host to network due router comunicano direttamente sul canale fisico, **a livello di trasporto al posto del canale fisico c'è una intera subnet!**

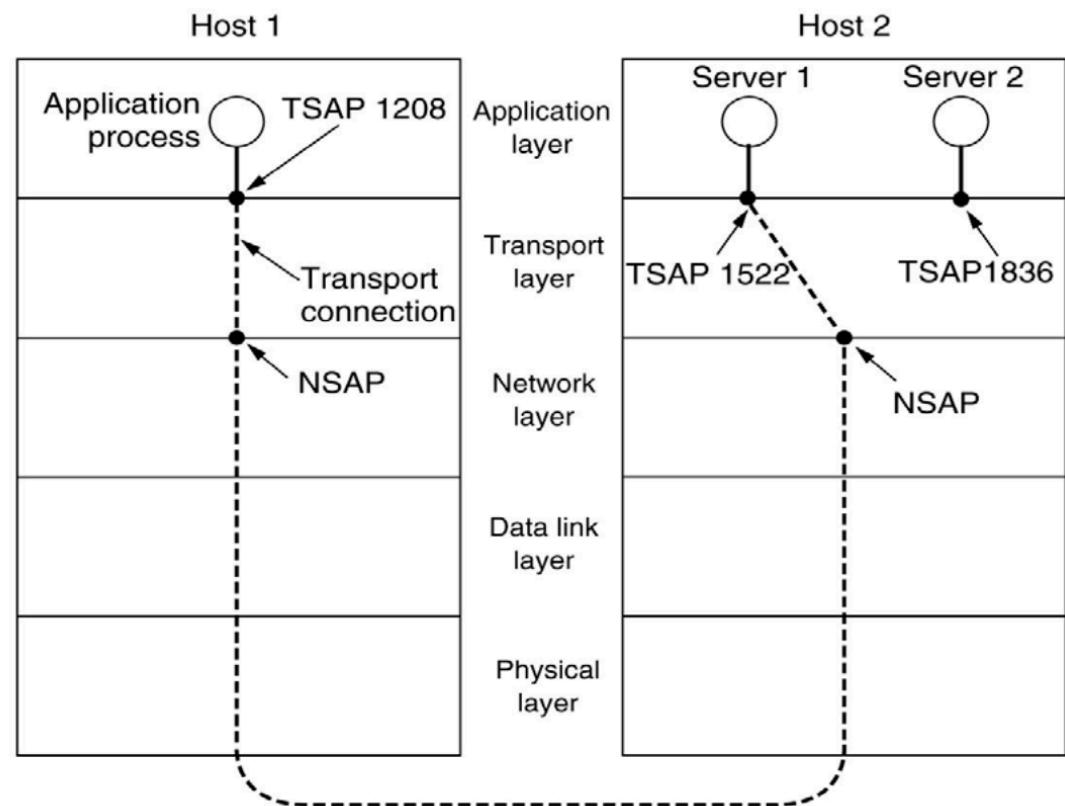
Indirizzamento

Indirizzi di trasporto:

- Quando un processo di un'applicazione vuole stabilire una connessione con un processo di un'applicazione remota deve sapere a quale connettersi
- Lo stesso nel caso connectionless, a chi mando il datagram?
- Si definiscono degli indirizzi di trasporto che sono chiamati **port** (o porte) in TCP/IP o genericamente **TSAP** (Transport Service Access Point)
- A livello di rete allora gli indirizzi IP li chiamiamo **NSAP** per distinguerli dai TSAP (NSAP = indirizzo IP)
- Il TSAP serve per distinguere quale degli end point di trasporto voglio raggiungere quando tutti condividono la stessa NSAP!!!

TSAP e NSAP

1. Un processo Server 1 “time of day” su Host 2 che fornisce l’ora esatta si attacca al TSAP 1522 in attesa di una call (tipo una LISTEN)
2. Un processo su Host 1 vuole sapere che ore sono e fa una CONNECT dalla TSAP 1208 alla TSAP 1522
3. L’applicazione manda la richiesta
4. Il time server risponde con l’ora esatta
5. La connessione viene chiusa

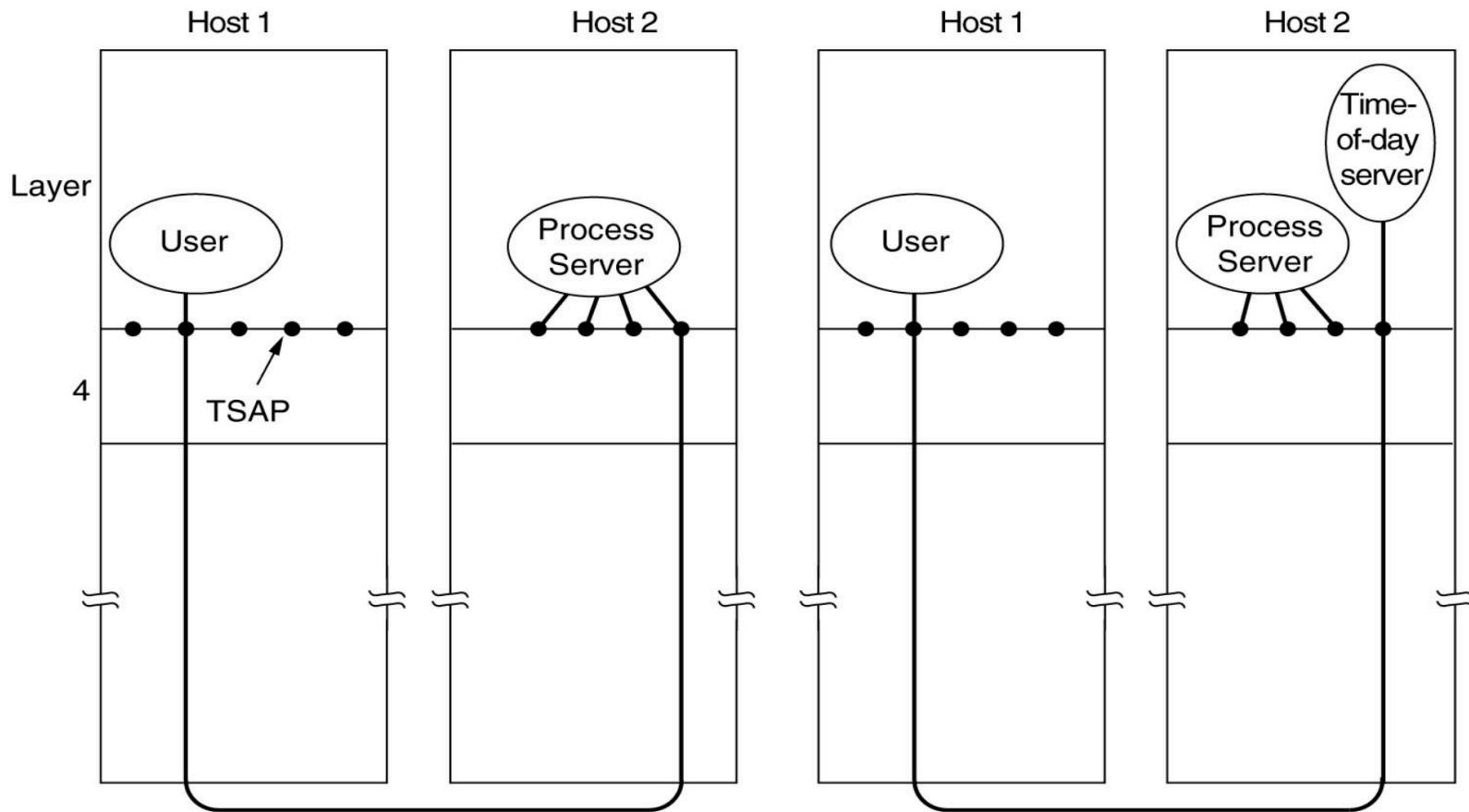


Come trovo la porta?

Come faccio a sapere che devo usare la TSAP 1522 ?

- Il server “time of day” è da anni sulla TSAP1522 e quindi tutti lo sanno (come tutti sanno che il web è sulla TCP:80). Sono i cosiddetti well known services (1 - 1023)
- Ma se voglio parlare con un processo che esiste solo per poco tempo? Posso usare un process server che mi fa da proxy: quando arriva la CONNECT con l’indirizzo della TSAP se non c’è un server in attesa viene connesso al process server. Questo fa partire un nuovo processo con il server richiesto, dopo di che si rimette in ascolto per nuove richieste
- –Se il server non può essere creato ogni volta ci potrebbe essere un name server (directory server) a cui i server si registrano fornendo il nome del servizio che offrono e il loro TSAP. Il client contatta il name server che è ad una TSAP ben nota e chiede di un certo servizio. Il server risponde con la TSAP

Process server



Fare una connessione...

.....sembra facile

- Uno manda una CONNECTION REQUEST TPDU
- L'altro risponde con CONNECTION ACCEPTED

Ma se la rete perde, ritarda o duplica pacchetti?

- Es: Mi connetto alla banca. Mando un messaggio in cui dico di versare un sacco di soldi ad una persona non completamente fidata, infine chiudo la connessione
- Per sfortuna il messaggio arrivano duplicato: la banca come fa a sapere che non sono due operazioni ma una sola? La banca pensa che si tratti di un secondo nuovo mandato di pagamento.

Evitare le duplicazioni

Allora potrei numerare le connessioni (un numero sequenziale)

- Dopo ogni chiusura di connessione tengo traccia di quelle obsolete e se arriva un nuova richiesta posso controllare se appartiene ad una connessione già chiusa
- Questo implica che ogni macchina tenga in memoria queste informazioni per un tempo indefinito. Inoltre dopo un crash non saprei più quali connessioni sono chiuse.

Per cui abbiamo bisogno di un meccanismo per impedire che i pacchetti vivano per sempre nella rete. I pacchetti vecchi devono morire.

A morte i pacchetti

Potrei usare uno o più di uno dei seguenti metodi

- Impedisco in qualche modo ai pacchetti di andare in loop e cerco di mettere un limite al delay di congestione
- Metto un contatore di hop nel pacchetto e dopo ogni hop lo decremento. Quando arrivo a zero butto via il pacchetto
- Ogni pacchetto ha un'etichetta con la data di creazione per cui butto tutti i pacchetti più vecchi di una certa data. Questo metodo richiede però che tutti i router siano sincronizzati tra di loro con un GPS o un segnale radio...

Sequenza dal clock

In pratica dobbiamo assicurarci che non solo un pacchetto sia morto ma che anche sia passato un tempo T tale che tutti gli ack relativi a quel pacchetto siano morti!

Metodo di Tomlinson (1975) migliorato da Sunshine e Dalal (1978) con diverse varianti di cui una è usata da TCP:

- Ogni macchina deve avere un orologio che misura l'ora con una certa precisione ma che non deve essere sincronizzata con gli orologi delle altre macchine. In pratica è un contatore binario che si incrementa uniformemente nel tempo, con un numero di bit superiore a quello dei numeri di sequenza
- Importante: il clock deve funzionare anche quando la macchina si spegne.
- Scopo: non dobbiamo mai avere due TPDU “vive” con lo stesso numero di sequenza

Tempo e sequenza

- Quando si stabilisce una connessione, i **k** bit più bassi del clock vengono usati come numero iniziale della sequenza (dunque di **k bit**)
- Quindi ogni connessione inizia con un numero diverso nella sequenza, chiaramente devo avere abbastanza numeri che quando il numero “*wrappa*” (torna allo stesso numero dopo un integer overflow) le vecchie TDPU con quel numero siano ormai già morte da un pezzo
- Dunque il numero iniziale di sequenza aumenta linearmente nel tempo

Setup di connessioni

Come si creano nuove connessioni?

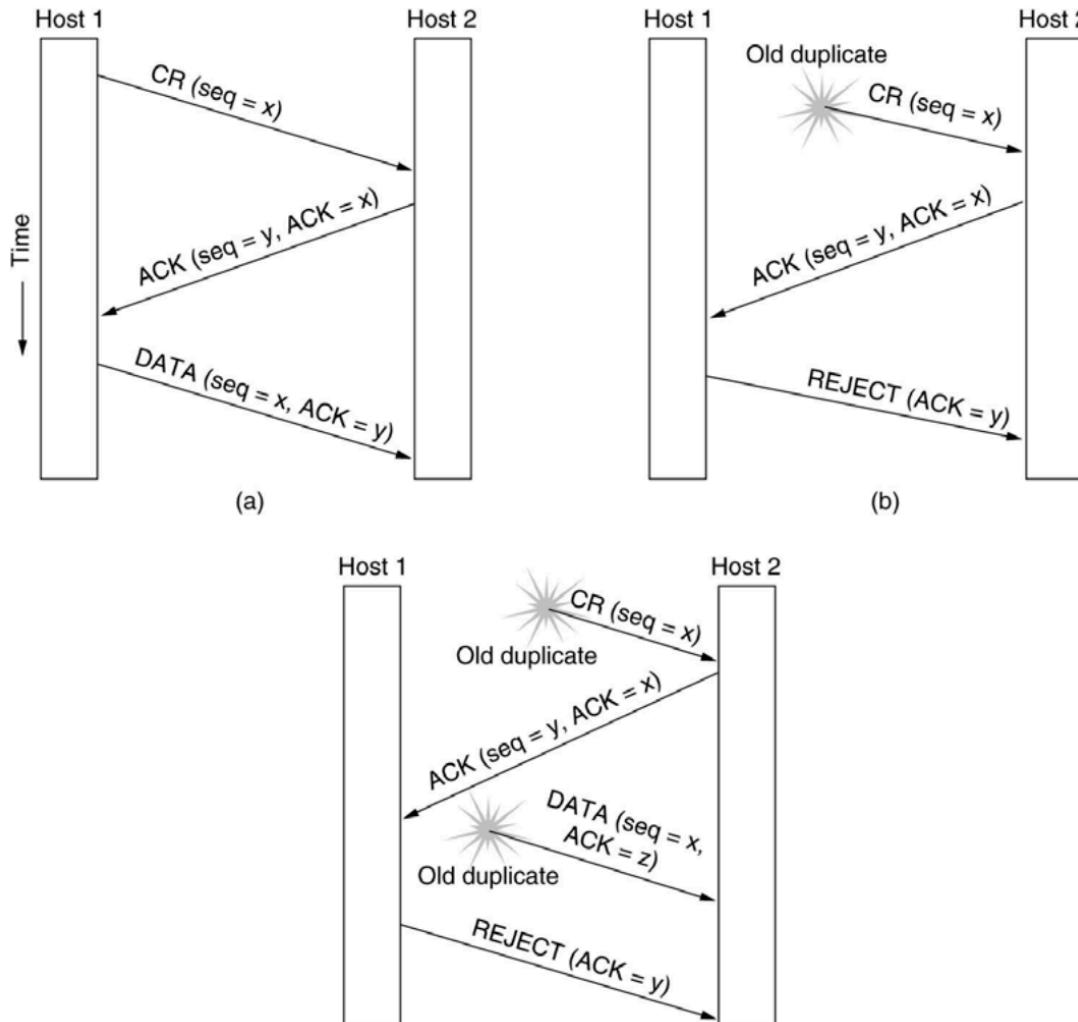
- Anche le TPDU di controllo possono essere ritardate, quindi bisogna fare in modo che entrambe le parti si accordino per un numero di sequenza iniziale.
- Es: host1 manda una CONNECTION REQUEST TPDU a host2 con il numero di sequenza iniziale proposto. L'host risponde con un CONNECTION ACCEPTED TPDU che però va perso ma intanto arriva all'host2 una nuova CONNECTION REQUEST ritardata duplicata, allora siamo nei guai!

Three-way handshake

Tomlinson ha risolto il problema nel 1975 con il three-way handshake

- Non richiede che entrambi i lati usino lo stesso numero di sequenza, quindi si può usare anche quando non c'è un clock globale
- La procedura normale nella figura seguente.
 - Host1 manda una CR con seq=x
 - Host2 risponde con un ACK verso Host1 con il suo proprio seq=y
 - Infine Host1 manda un ACK a Host2 confermando di avere ricevuto il numero y

Three-way handshake



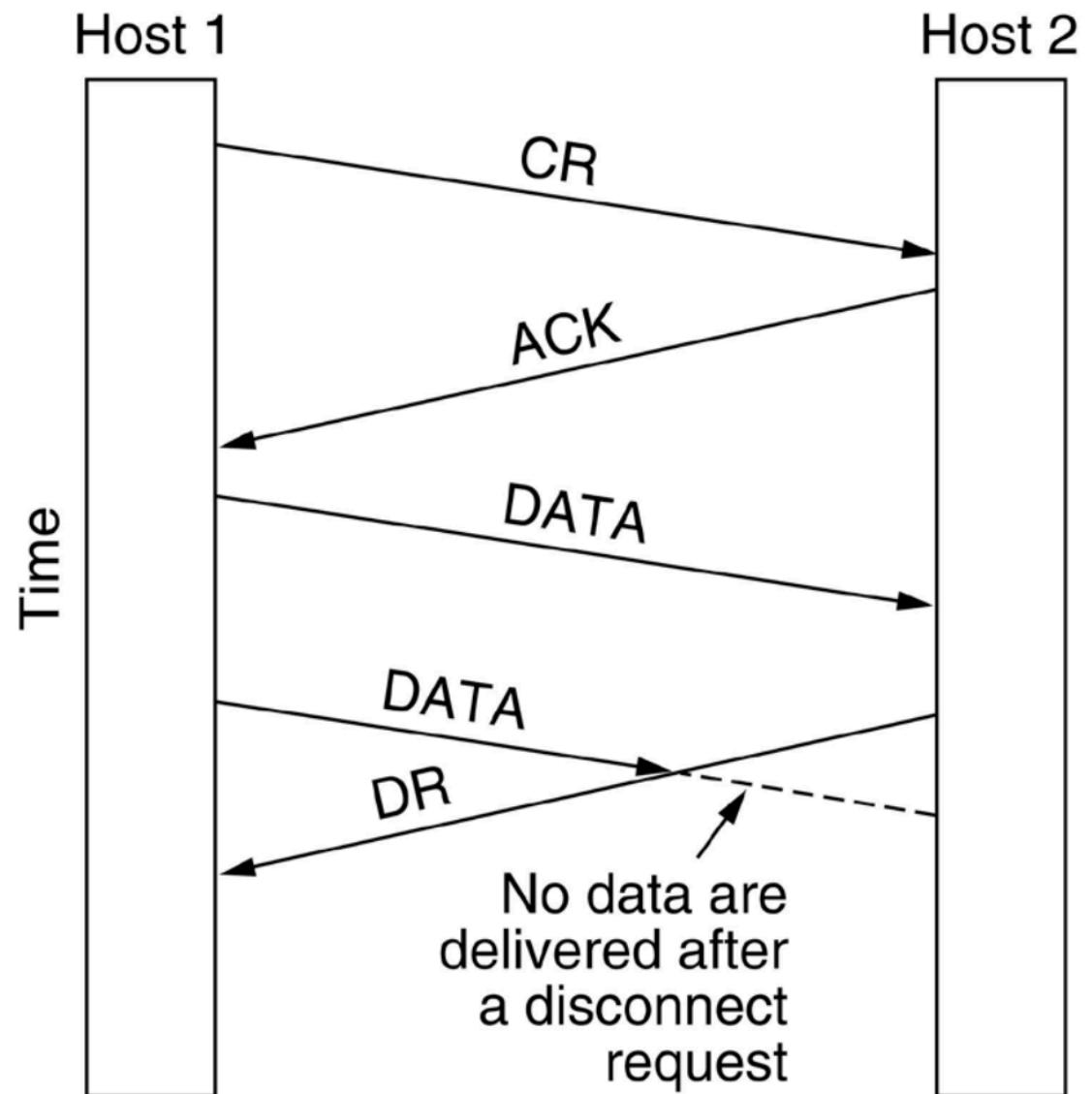
- (a) three-way handshake normale
- (b) quando appare una vecchia duplicata TPDU di CR
- (c) quando appare sia una vecchia CR duplicata che una ACK duplicata (NB, host2 manda la ACK con y sapendo molto bene che non esistono vecchi TPDU o ACK con y . Per cui quando arriva il secondo duplicato con z capisce subito che è un duplicato)

Chiusura connessione

- Chiudere una connessione è più facile che aprirla ma ci sono comunque dei tranelli
- Chiusura asimmetrica: come al telefono, uno dei due attacca e la connessione è rottata
- Chiusura simmetrica: la connessione viene considerata come due connessioni unidirezionali separate ognuna delle quali deve essere chiusa separatamente.

Chiusura asimmetrica

- Molto brutale, può portare a perdita di dati
- Host2 manda una disconnect request prima che la seconda TPDU arrivi e quindi viene persa

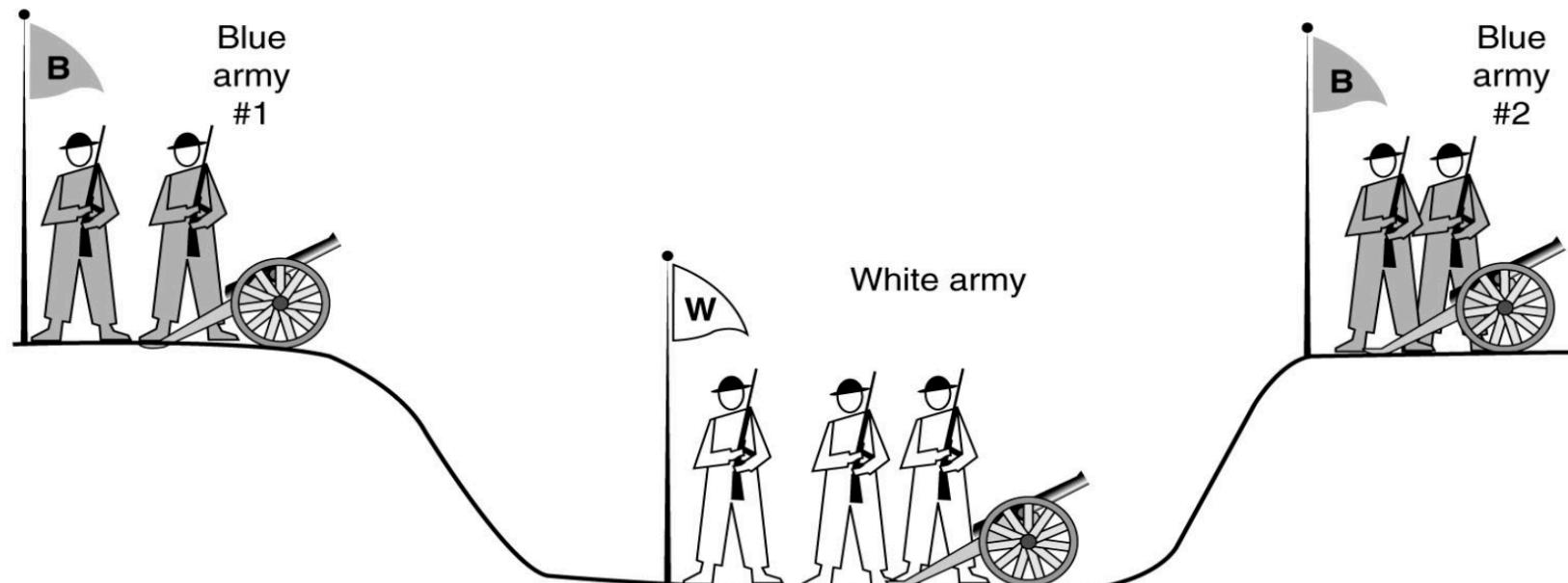


Chiusura simmetrica

- La connessione viene chiusa in modo indipendente per cui un host rimane in ascolto anche dopo aver mandato la DR TPDU
- Quindi un host dice “ho finito, tu hai finito?” e l’altro risponde “ho finito anche io” allora la connessione si può chiudere
- Ma questo semplice protocollo non sempre funziona.
- Vediamo l’esempio dei due eserciti

I due eserciti...

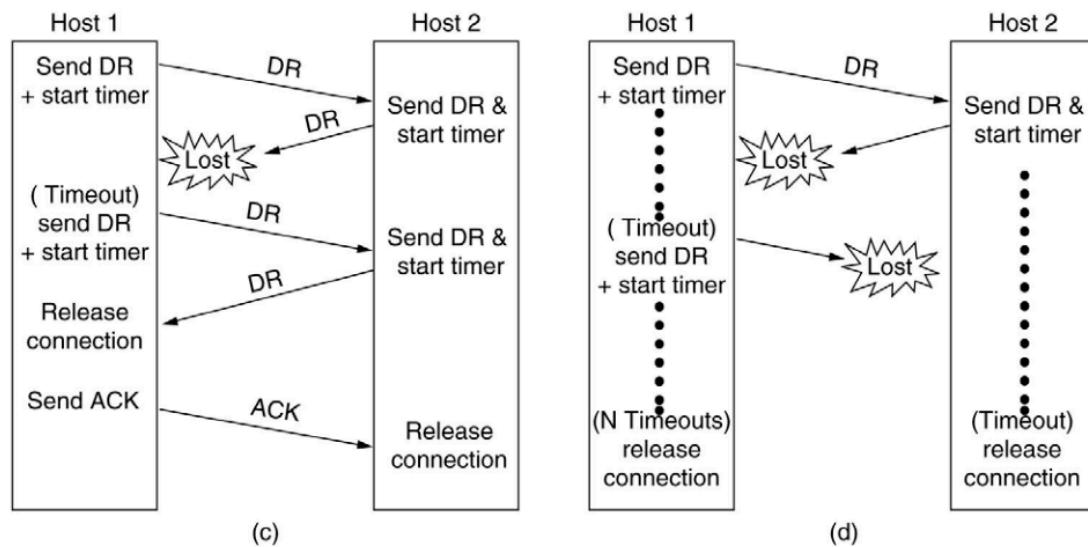
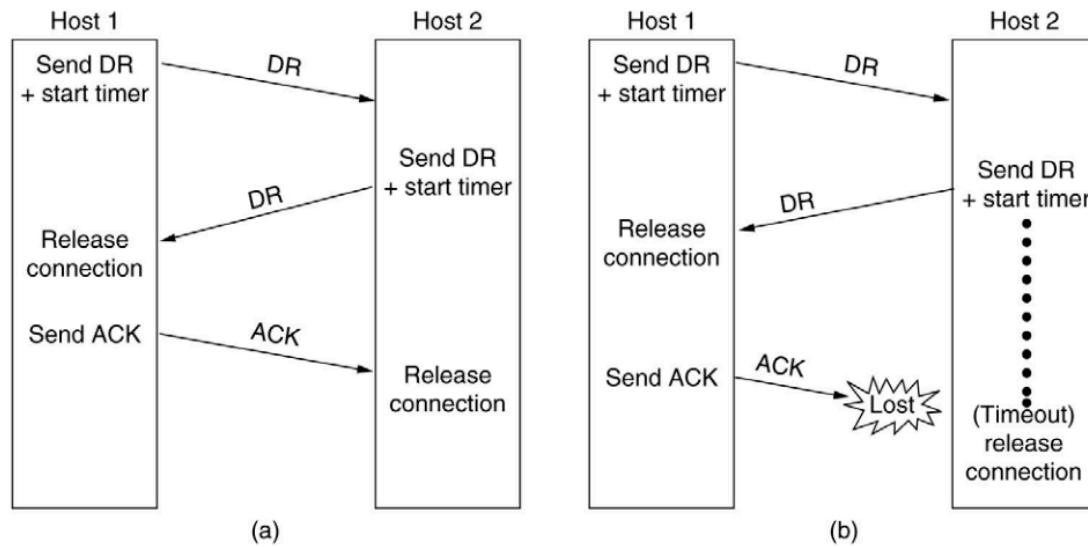
- I blue vincono se attaccano insieme. Se invece non attaccano insieme vincono i bianchi. Come fanno i blue a sincronizzarsi se il messaggero viene catturato nella valle?



Quanti handshake?

- Ogni “**disconnect**” che viene mandato da uno dei due eserciti ha bisogno di un ACK per avere la certezza che sia arrivato e dare il via all’attacco
- Sostituendo “**disconnect**” ad “**attacco**” abbiamo lo stesso problema. Se nessuno dei due si disconnette fino a quando non è convinto che l’altro si sia pronto a sconnettersi, allora nessuno si disconnette
- In pratica uno tende a rischiare di più quando chiude una connessione rispetto a quando attacca un esercito nemico...

Casi



Timeout

- Potremmo evitare il problema dicendo che il mittente non deve rinunciare dopo N tentativi ma deve continuare per sempre
- Il ricevente continua a non ricevere nulla per cui dobbiamo imporre una regola che se non arriva nulla dal mittente entro N secondi la connessione viene chiusa automaticamente
- Quindi ogni entità di trasporto deve avere un timer che viene fatto ripartire ogni volta che una TPDU viene mandata. Se il timer arriva a zero viene mandata una TPDU dummy, solo per tenere su la connessione

Multiplexing

A livello di trasporto il multiplexing è importante

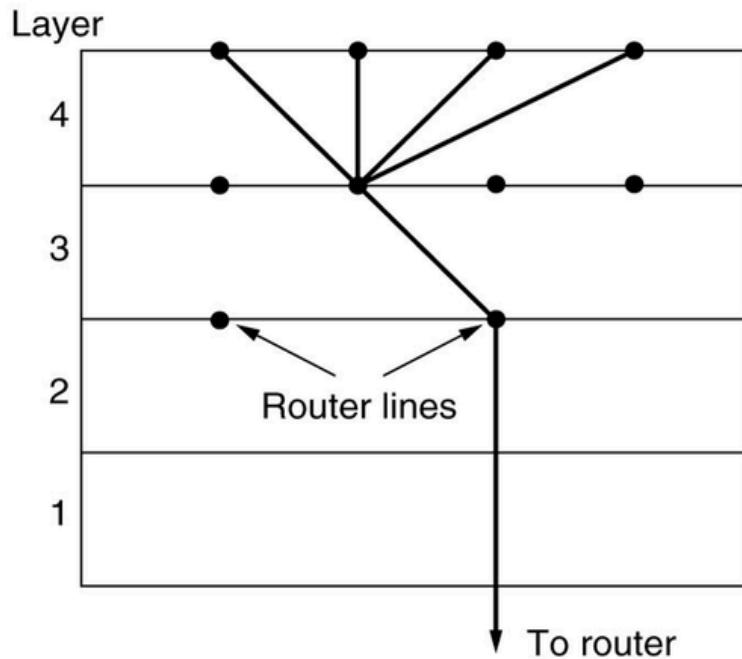
- Una macchina usa un unico indirizzo di rete e tutte le connessioni di trasporto devono usare quell'indirizzo
- Quando arriva un TPDU ci vuole un modo per stabilire a quale processo vada consegnata.
- Questo situazione si chiama upward multiplexing
- In figura a (sotto) quattro diverse connessioni di trasporto usano la stessa connessione di rete (indirizzo IP)

Downward multiplexing

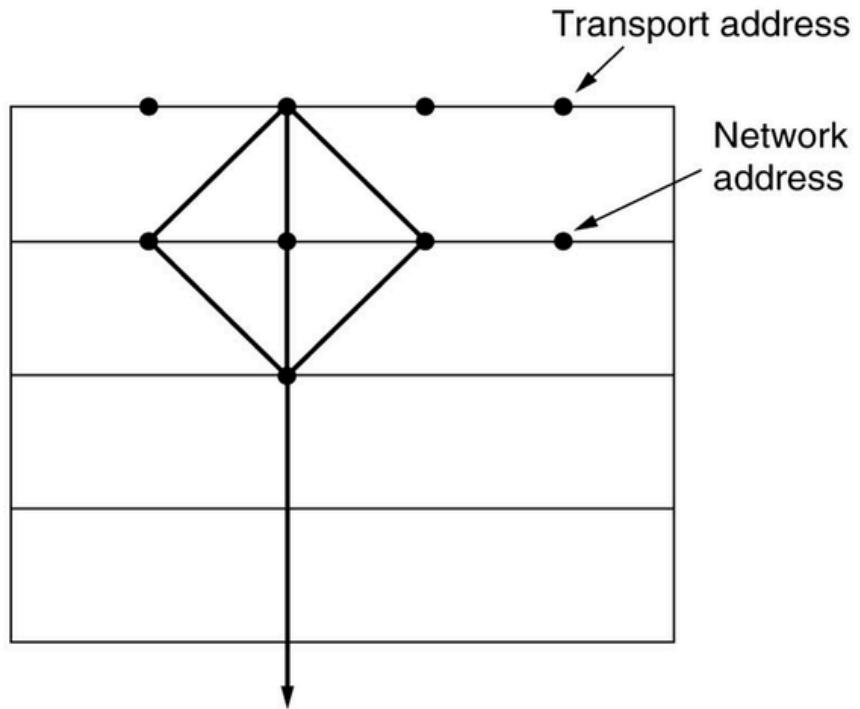
Al contrario se una connessione usa diversi circuiti virtuali, ognuno con un suo data rate limitato:

- Se un utente ha bisogno di banda maggiore può aprire diverse connessioni di rete e distribuire il traffico su di esse, per es. con un algoritmo round robin
- Con k connessioni aumenta la banda di un fattore k
- Si fa per esempio per avere un collegamento a 128 kbps quando ho due connessioni separate da 64 kbps ciascuna

Upward e Downward



(a)



(b)

(a) upward multiplexing (b) downward multiplexing

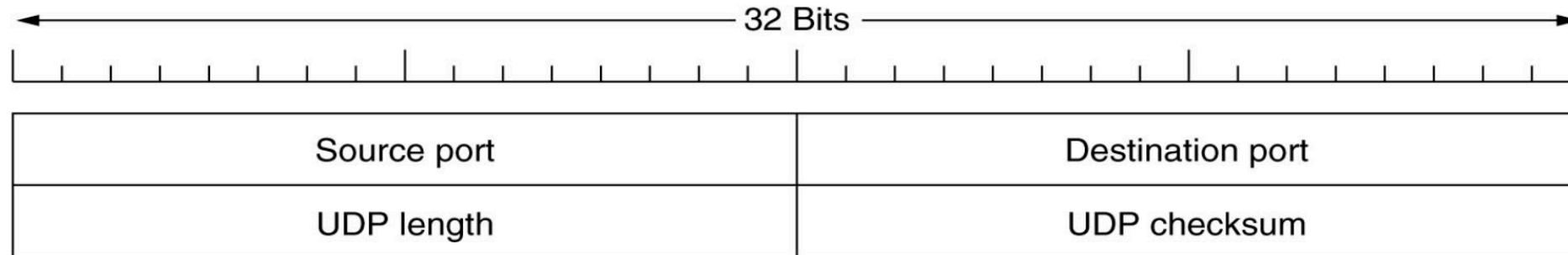
UDP

Trasporto in internet

Internet ha due protocolli di trasporto principali:

- uno connectionless: UDP (User Datagram Protocol)
- uno connection oriented: TCP (Transmission Control Protocol o anche Transfer Control Protocol)

UDP e RFC 768



- La **Source port** serve quando devo mandare indietro un reply:
 - Copiando il campo **Source port** del segmento entrante nel campo
- **Destination port** indica la porta di destinazione
- 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

DOMANDA: Cosa fa un socket UDP?

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

Utilizzo di UDP in internet

- DNS (Domain Name System) traduzioni nomi – indirizzi IP
- NFS (Network File System) dischi di rete
- SNMP (Simple Network Management Protocol) gestione apparecchiature di rete (router, switch, ...)
- molte applicazioni di streaming audio e videoin è importante la bassa latenza ed è accettabile la perdita di alcuni dati

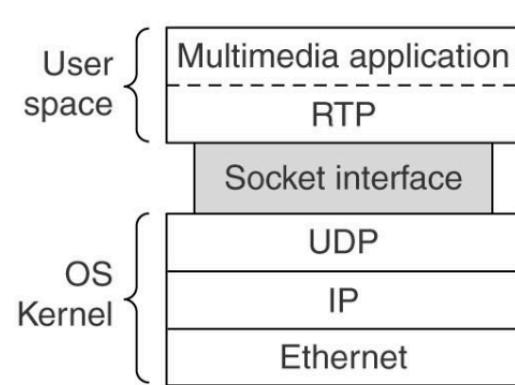
RTP

Real-time Transport Protocol:

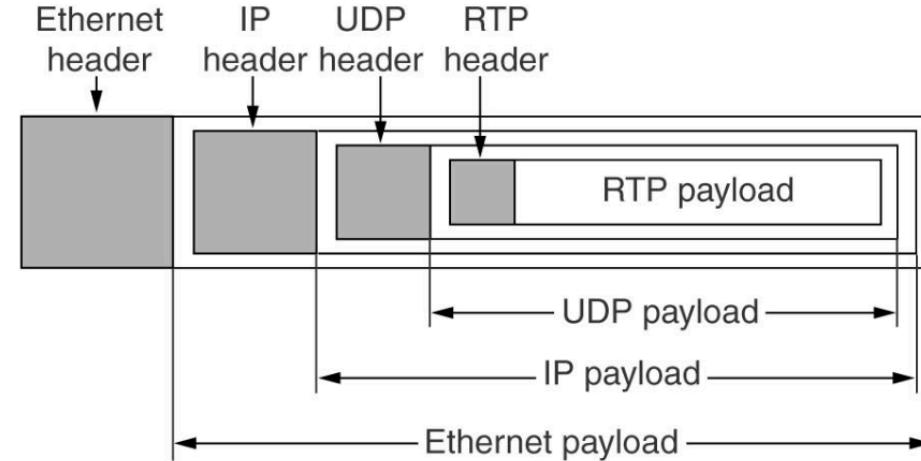
- RFC 1889, nato per unificare i vari protocolli di streaming di contenuti real time multimediali (internet radio, telefonia internet, videconferenza, video on demand, music on demand)
- Normalmente RTP sta nello spazio utente e gira sopra UDP
- Le applicazioni multimediali consistono di diversi stream audio, video o testo, che accedono ad una libreria nello spazio utente.
- 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

Di che livello è RTP?

- 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

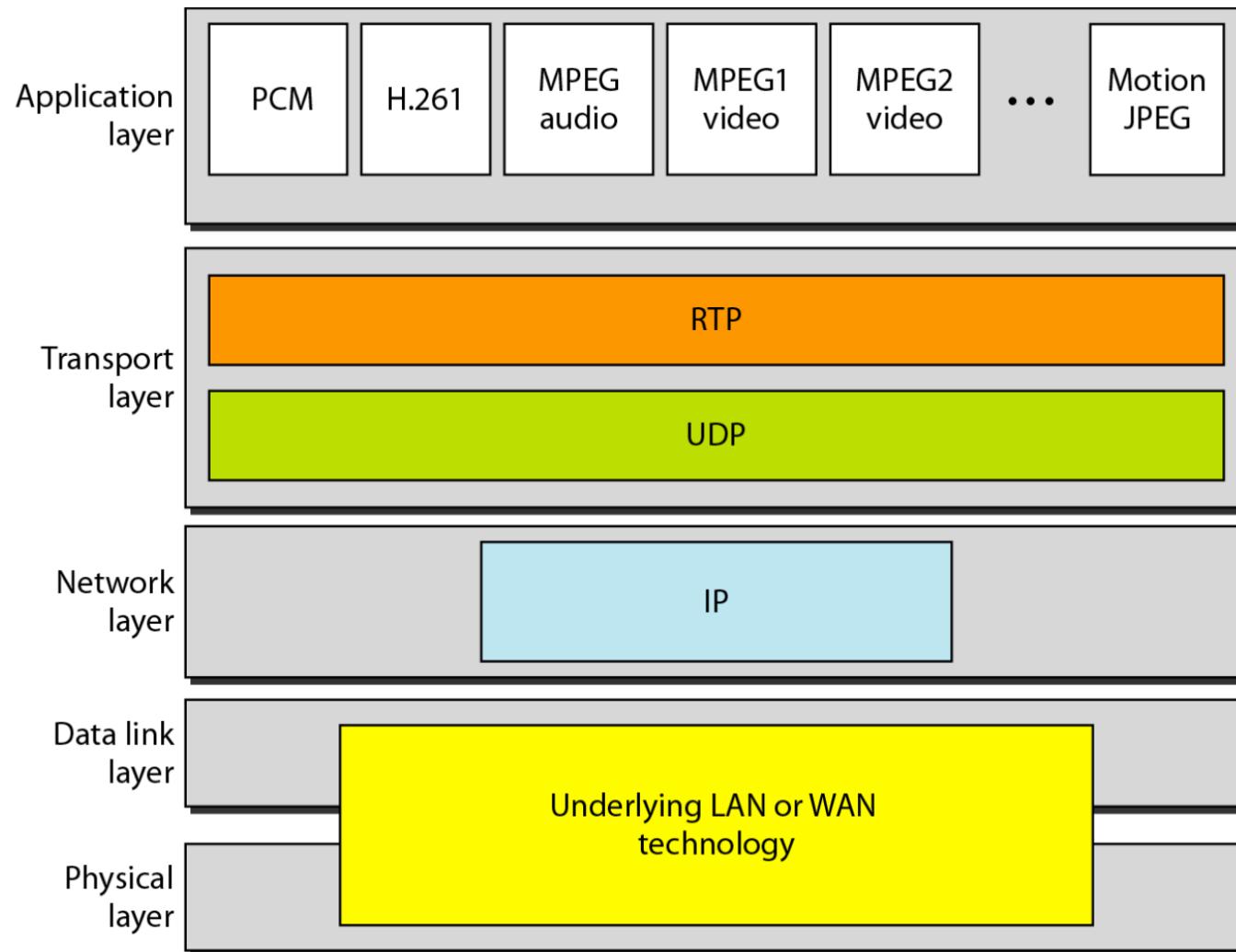


(a)



(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. Arriverebbe 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.

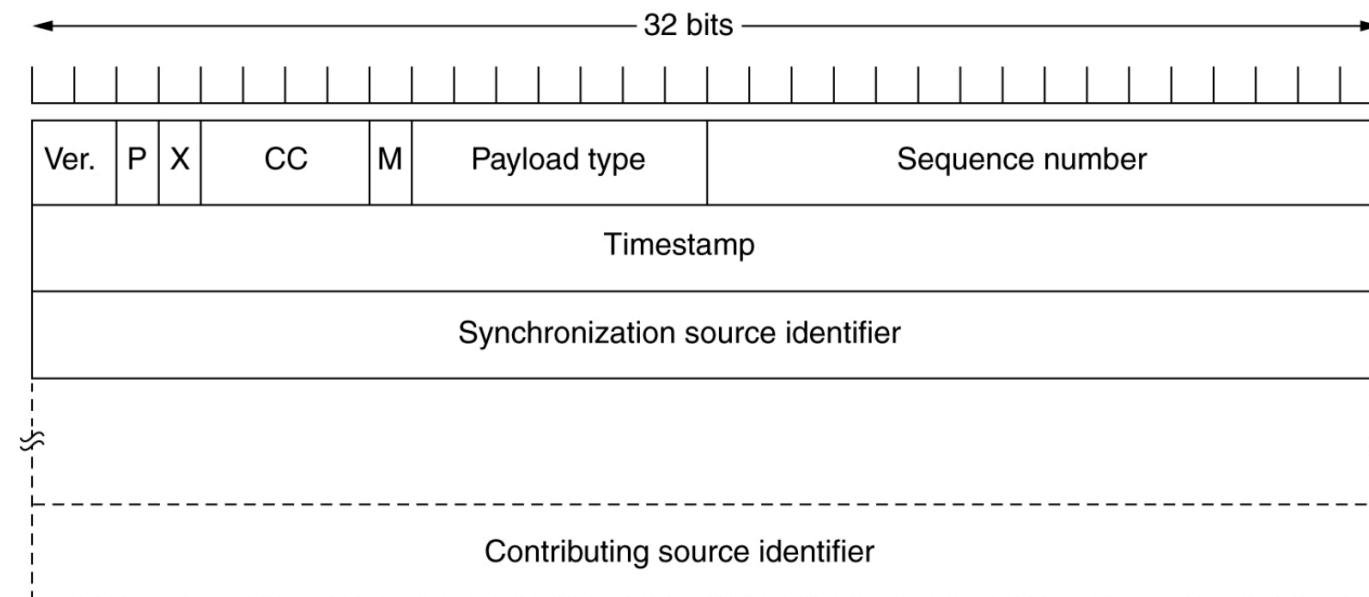
RTP

- Ogni payload RTP può avere diversi campioni, ognuno con la sua codifica (PCM (Pulse-code modulation) a 8 kHz, predictive 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 assoluto ma relativo 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)

NOTA: In electronics and telecommunications, jitter is the deviation from true periodicity of a presumably periodic signal, often in relation to a reference clock signal. In clock recovery applications it is called timing jitter. Jitter is a significant, and usually undesired, factor in the design of almost all communications links.

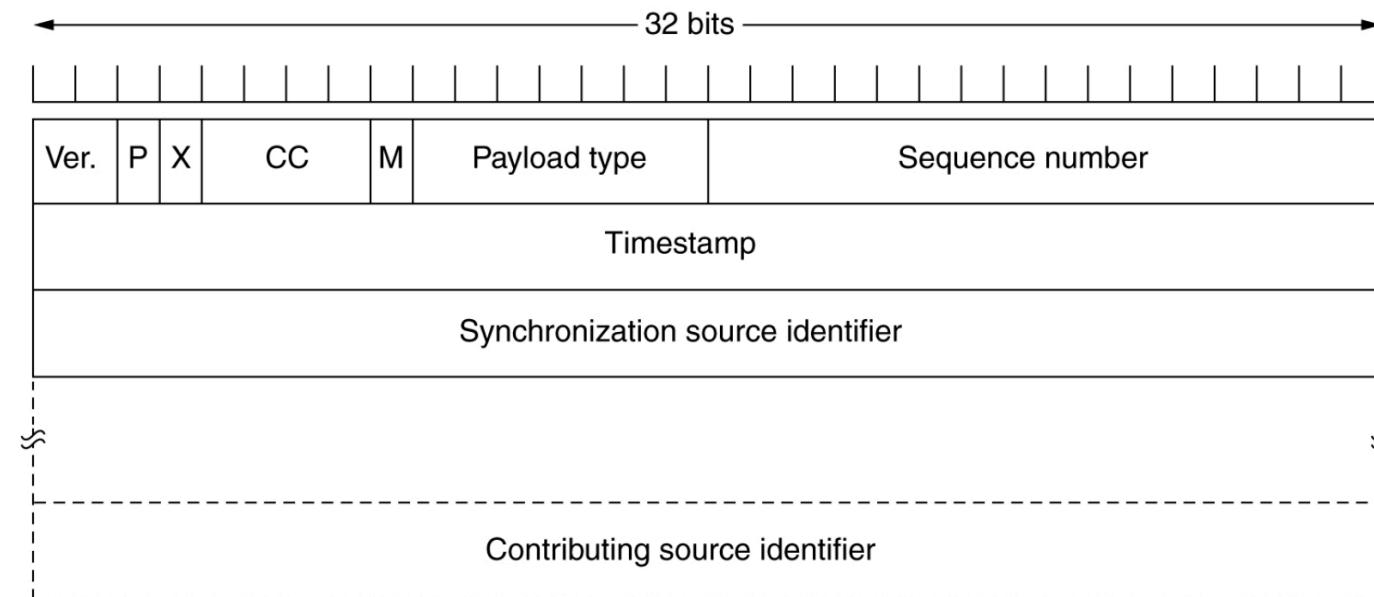
Frame RTP

- **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



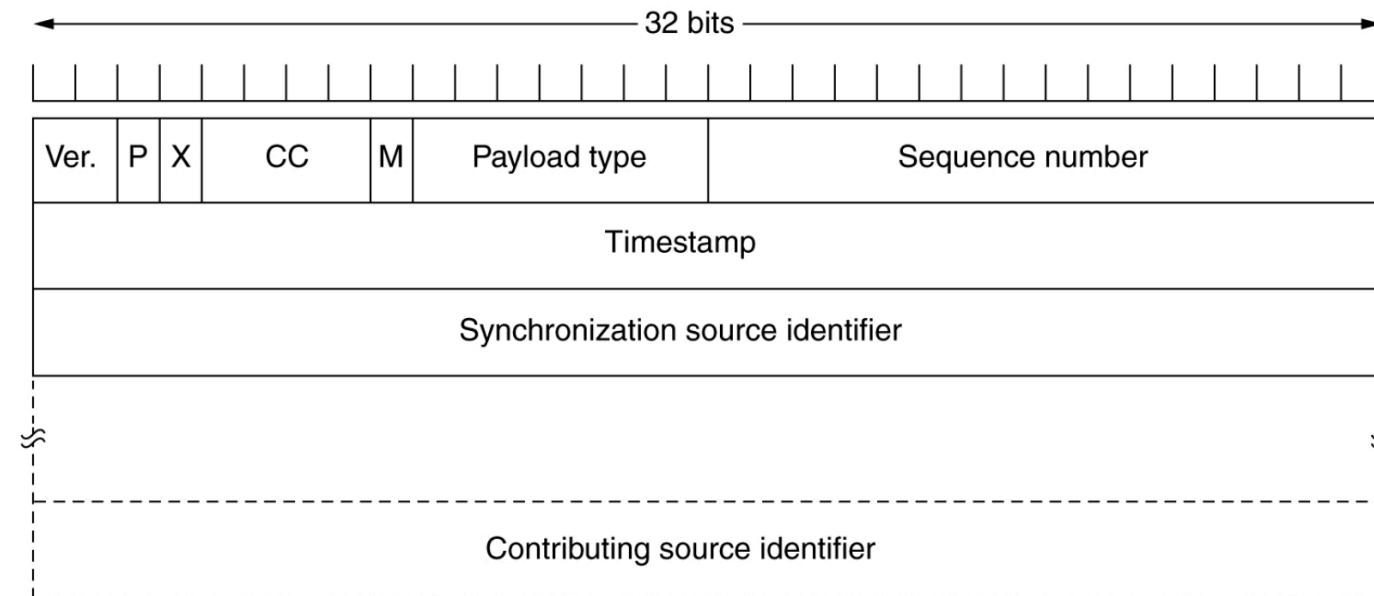
Frame RTP

- **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, disaccoppiando la riproduzione dall'arrivo dei pacchetti

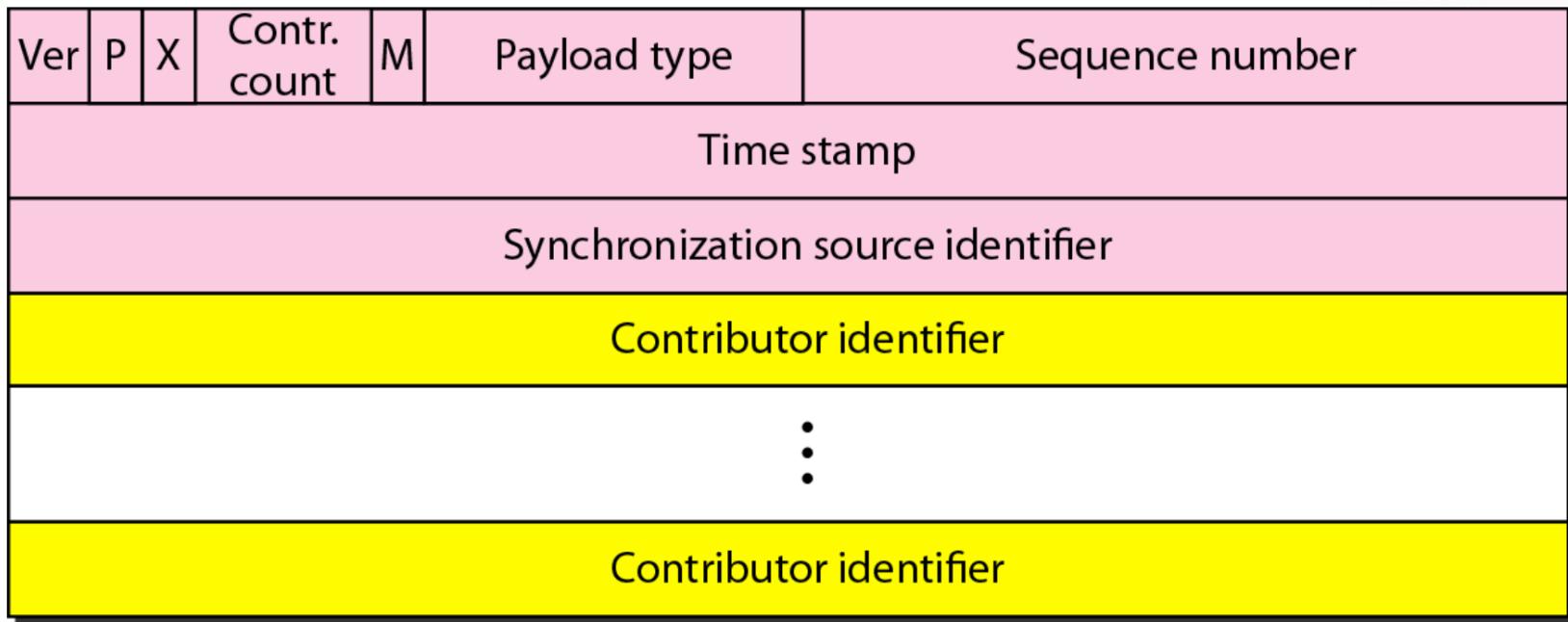


Frame RTP

- **Synchronization source identifier.** Per multiplexare o demultiplexare diversi data streams in un singolo flusso UDP
- **Contributing source identifier,** quando c'è un mixer, il mixer è la sorgente che sincronizza i vari stream questo campo contiene l'elenco dei vari stream.



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

TCP

- Progettato fin dall'inizio per fornire un stream di byte, affidabile, end to end su 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

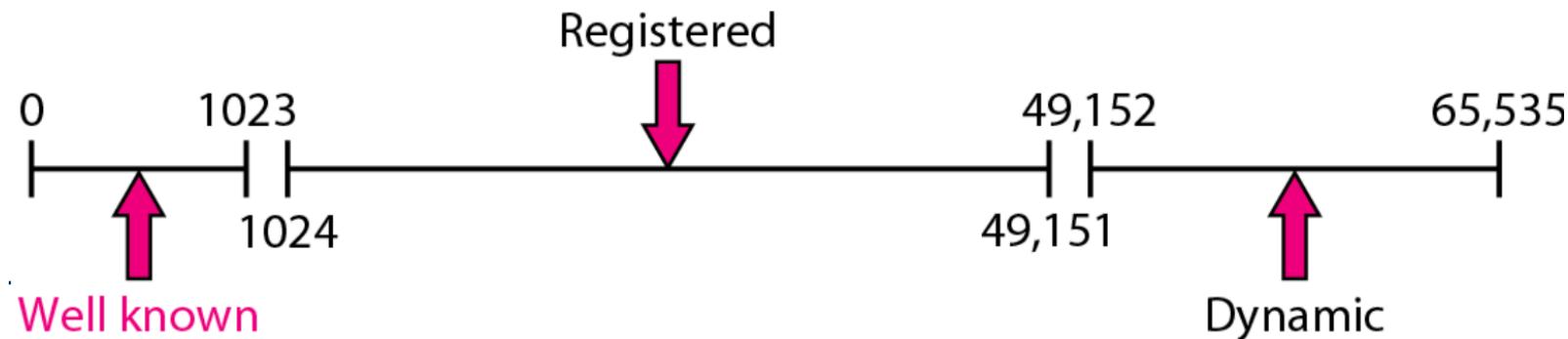
- Una macchina con TCP ha un'entità che gestisce il trasporto TCP
 - Una libreria oppure un processo utente o nel kernel
 - I dati forniti dal processo vengono spezzati in pezzi non più grandi di 64 kB (in realtà spesso 1460 Bytes per stare in un frame Ethernet) e manda ogni pezzo in un pacchetto IP

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 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 particolarmente 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 (Internet Assigned Numbers Authority), così è possibile scegliere per nuovi servizi porte non registrate da altri
- per es: 6000 per X-Window (gestore grafico UNIX)



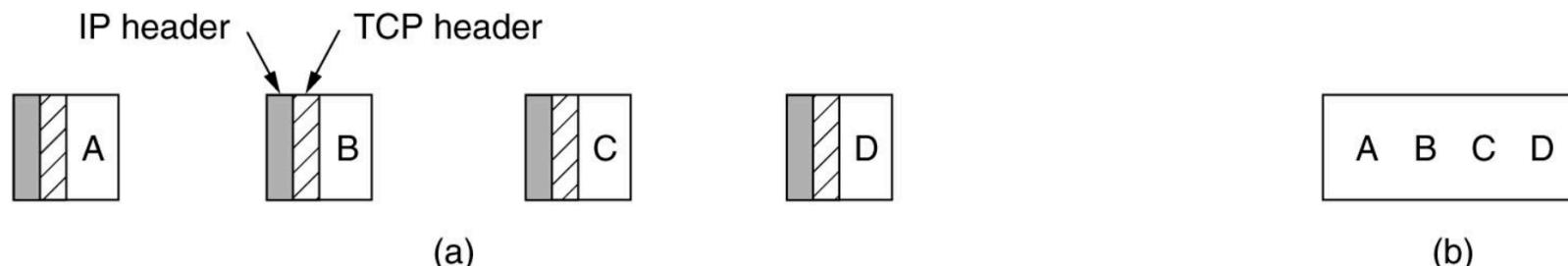
Connessioni TCP

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 inizio e fine dei messaggi.
- Se mando 4 blocchi da 512 bytes questi vengono consegnati 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

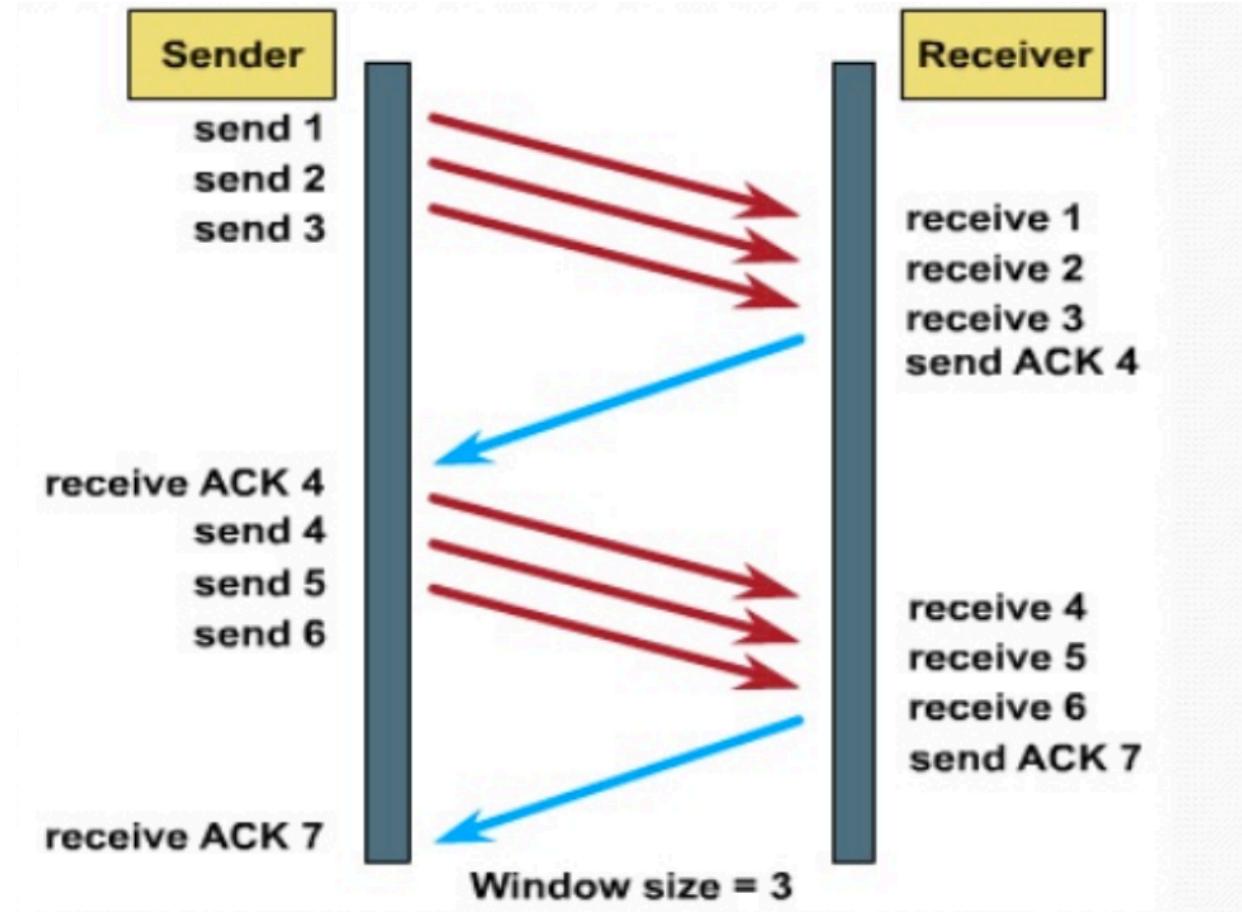
Protocollo TCP

- 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 TCP ha un numero di sequenza di 32 bit.
- 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

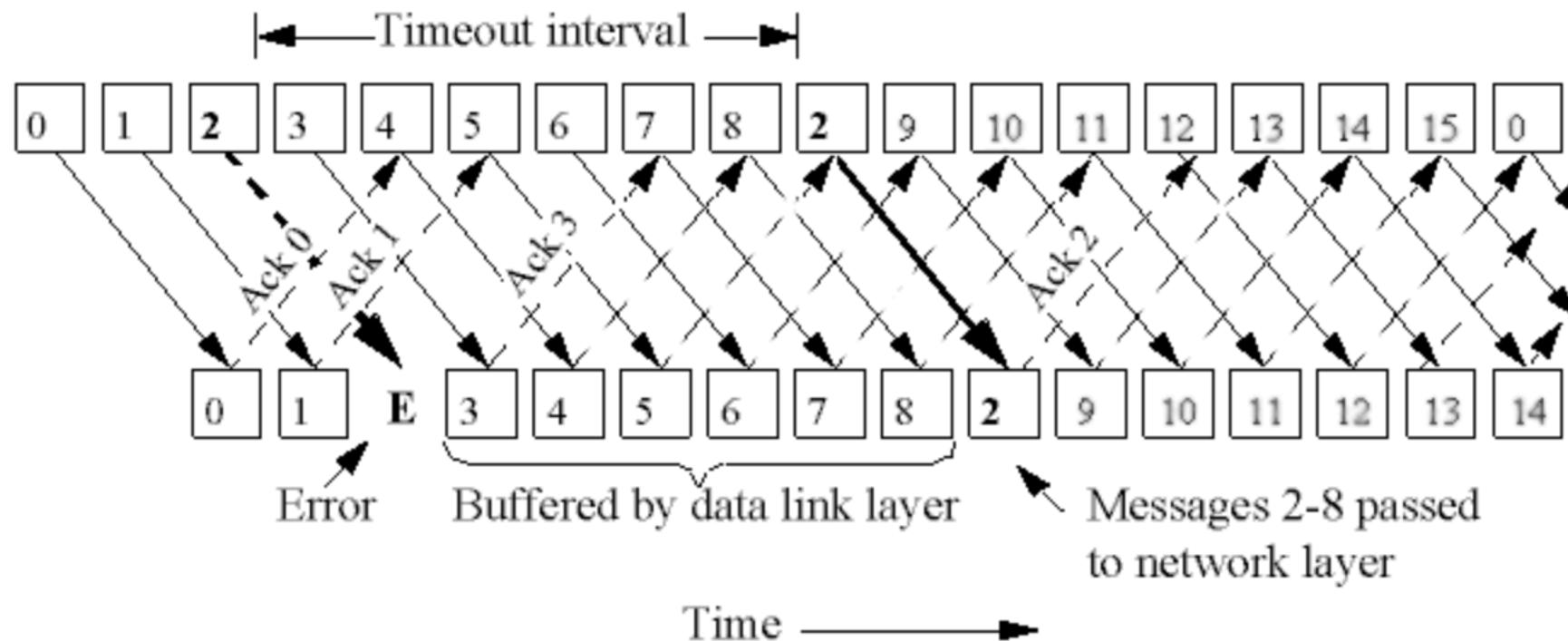
Sliding window



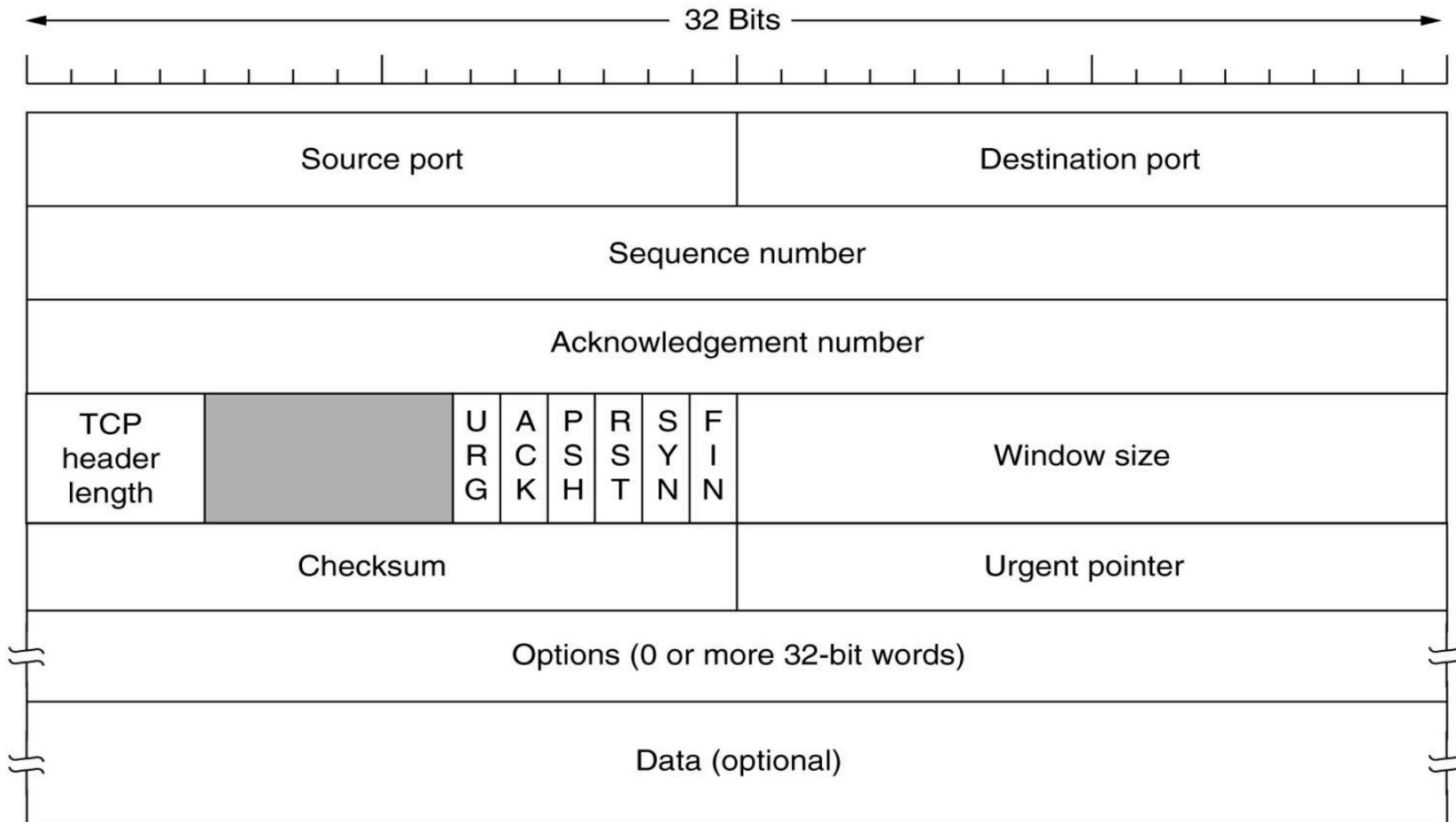
Problemi di sliding window

- 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

Problemi di sliding window



Header TCP



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 20 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 di 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

- **URG=1** indica che il ricevente deve iniziare a leggere il campo dati a partire dal numero di byte specificato in **Urgent pointer**. Viene usato se si inviano comandi che danno inizio ad eventi asincroni “urgenti”. Ad esempio il comando Control-C in una sessione Telnet.
- **ACK=1** indica che il Acknowledgment 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

Window size

- 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

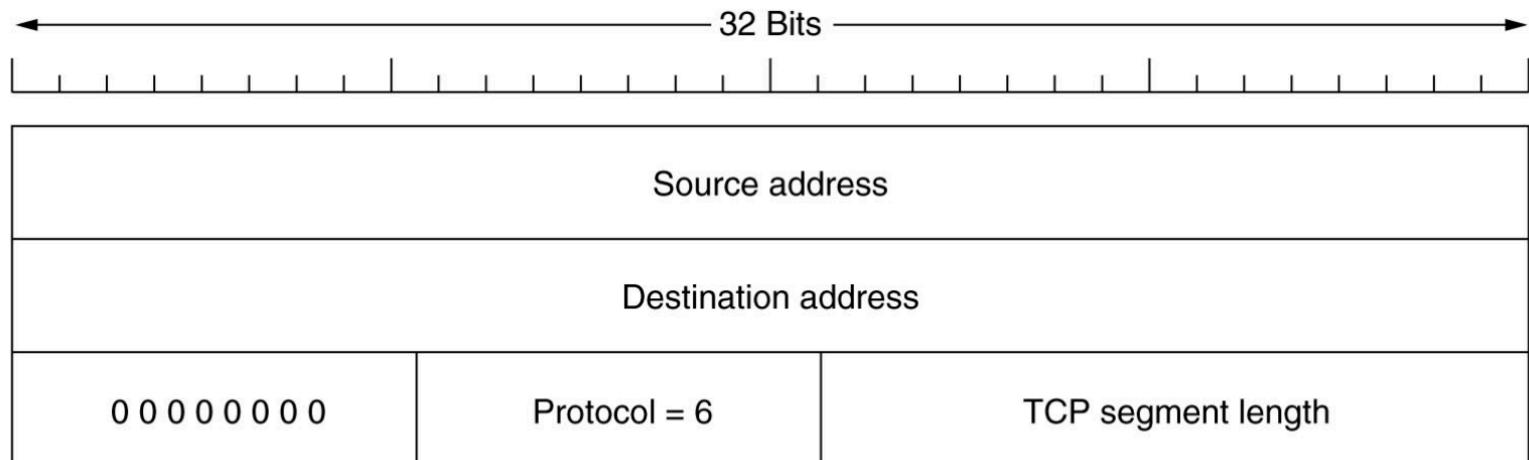
Checksum

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

Lo pseudo header è
creato prima del calcolo
del checksum. Esso
contiene informazioni
importanti prese
dall'header TCP e dal
datagramma IP che
conterrà il segmento
TCP.



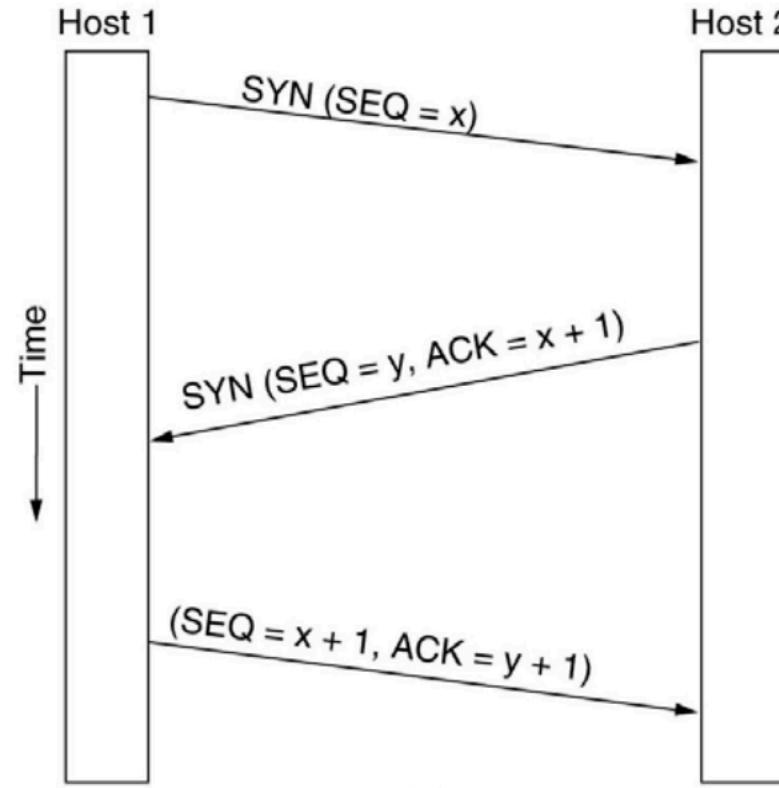
Options

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.

Setup connessione TCP

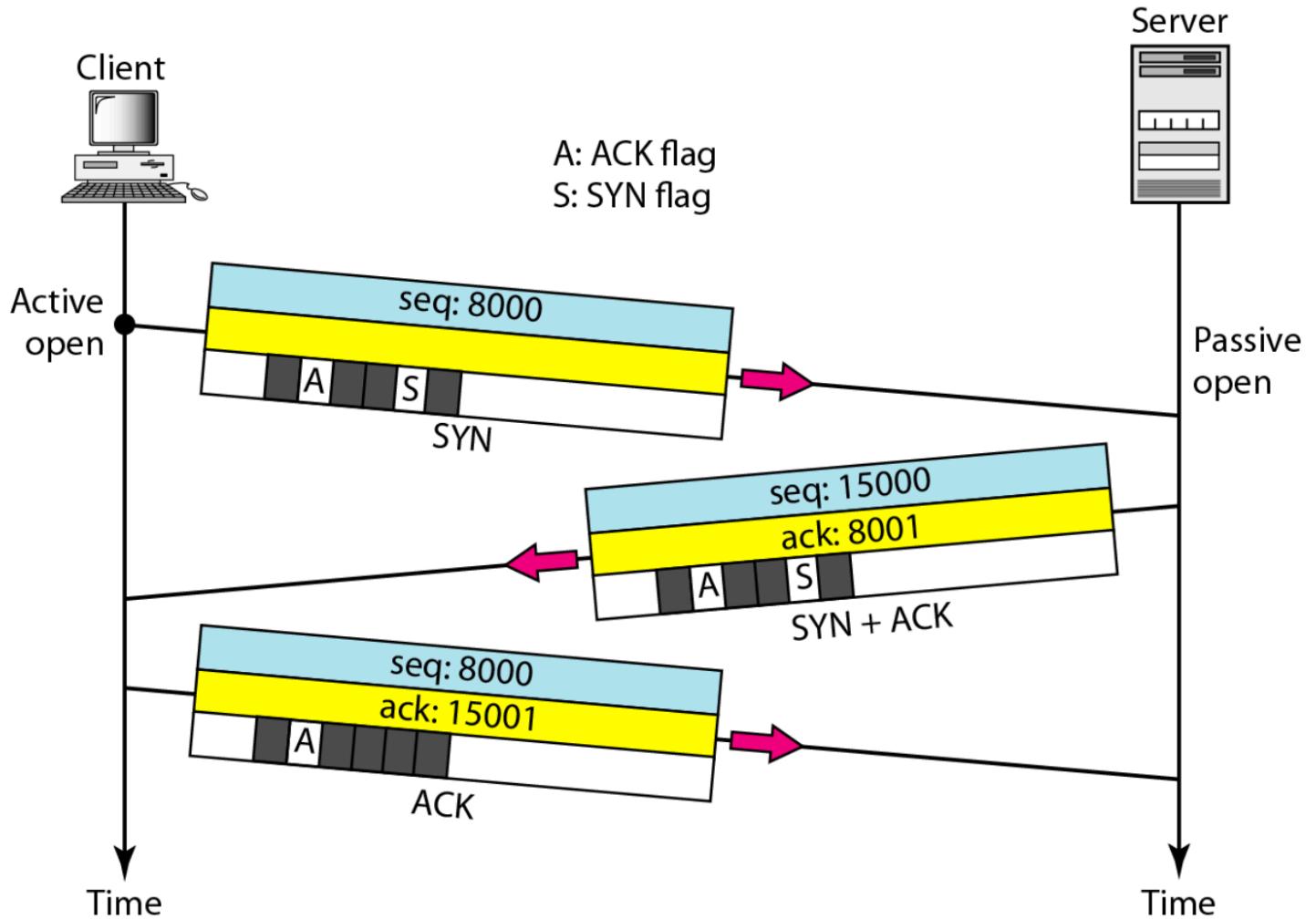
- Uso il 3way handshake già visto in precedenza



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)

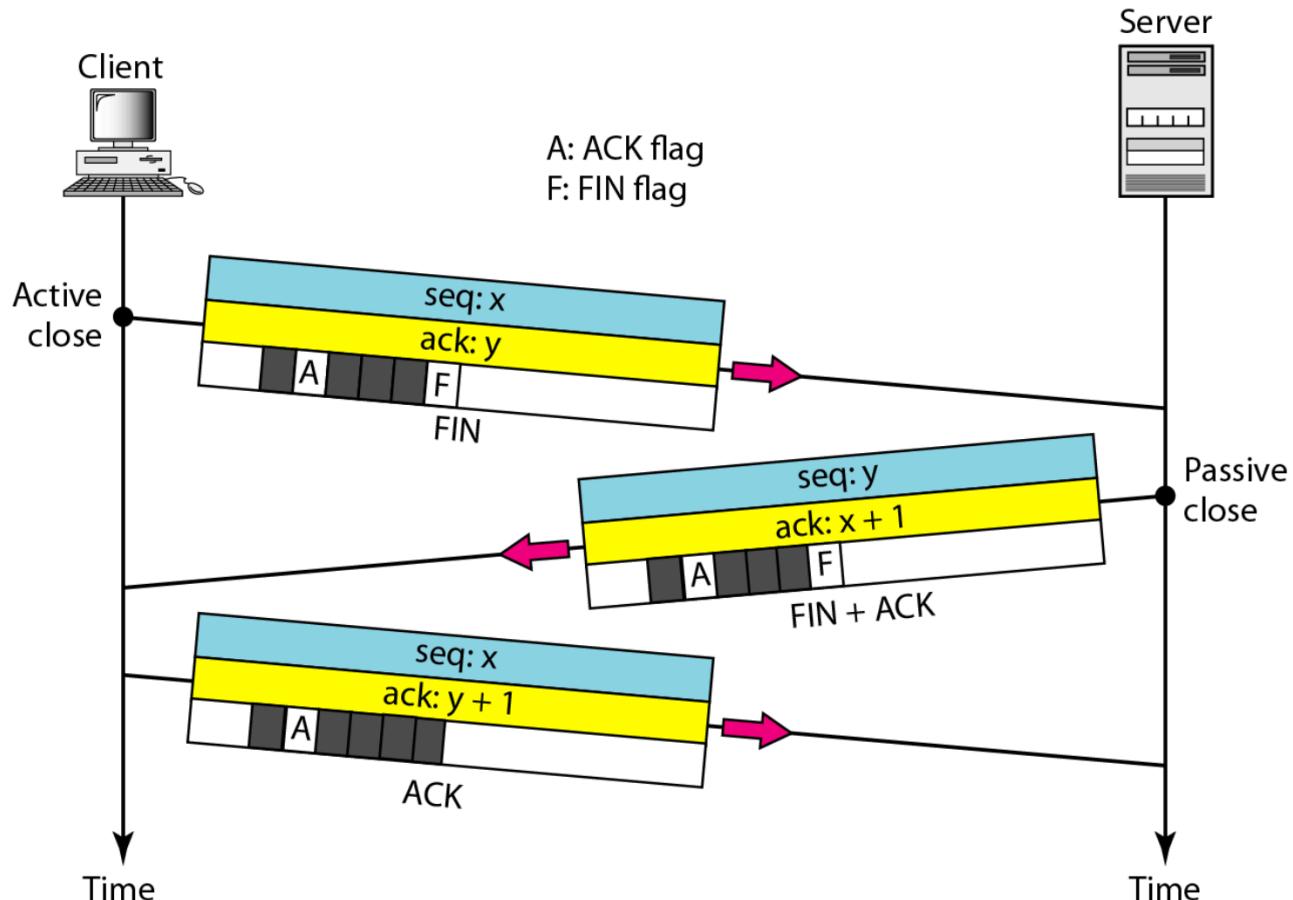
Rivediamo la connessione TCP



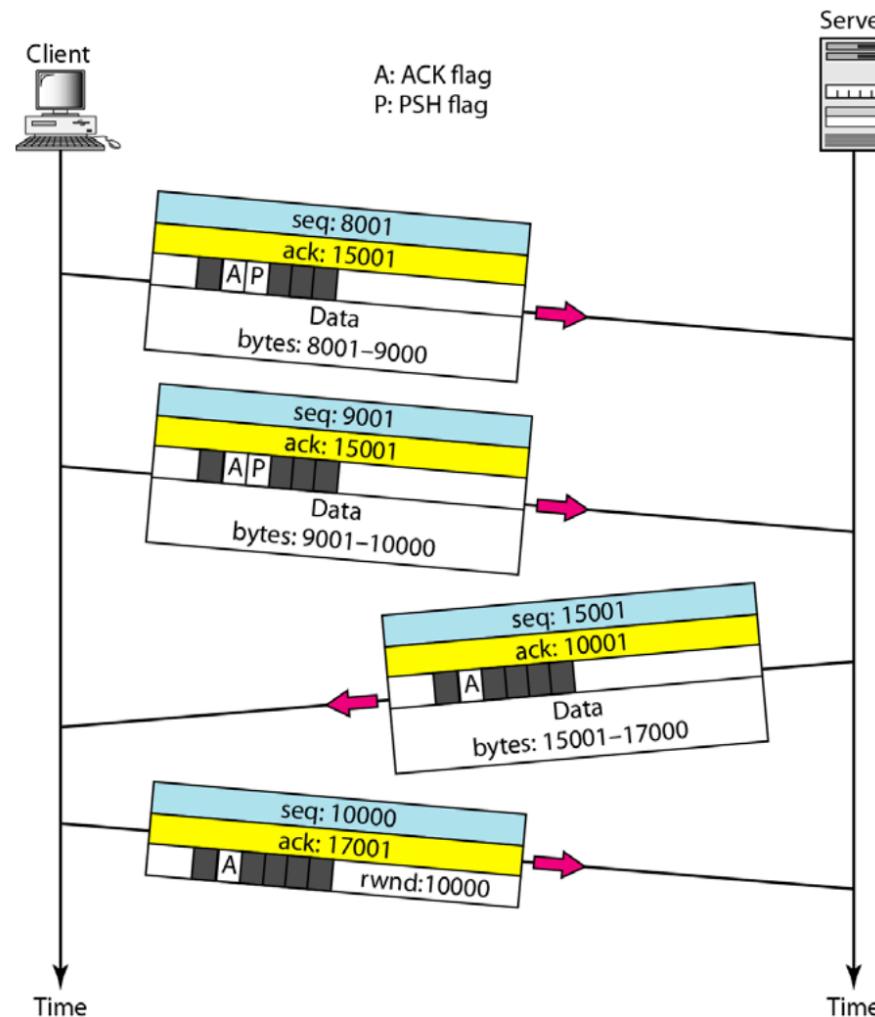
Chiusura connessione TCP

- La connessione è full duplex ma viene chiusa come una coppia di connessioni simplex
- Una delle due parti manda un segmento con il bit FIN (non ho più dati da trasmettere)
- Essa riceve il relativo ACK e a questo punto la connessione è chiusa per dati in questa direzione.
- Servono 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 TCP



Trasferimento dati



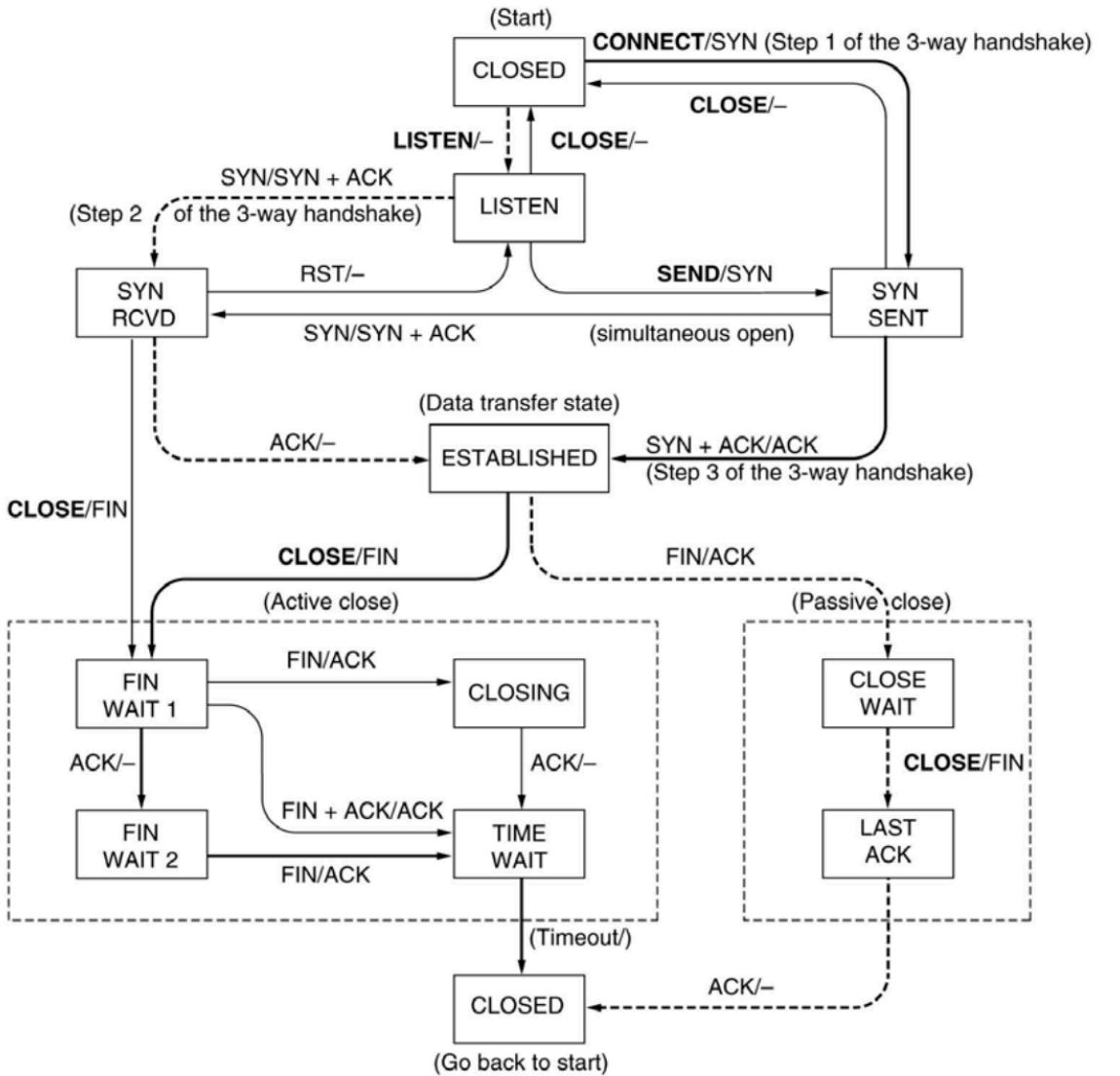
Macchina a stati finiti

- 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

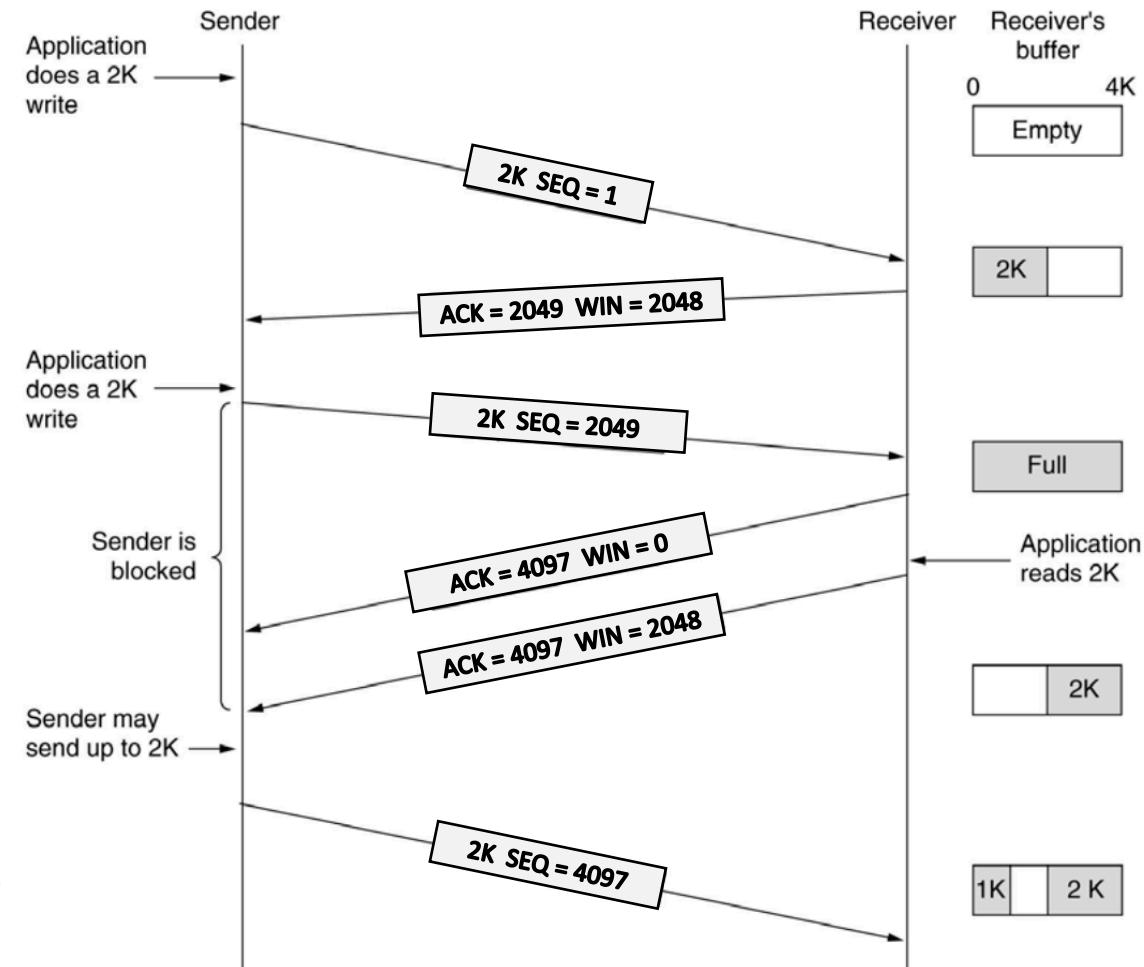
Macchina a stati finiti

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

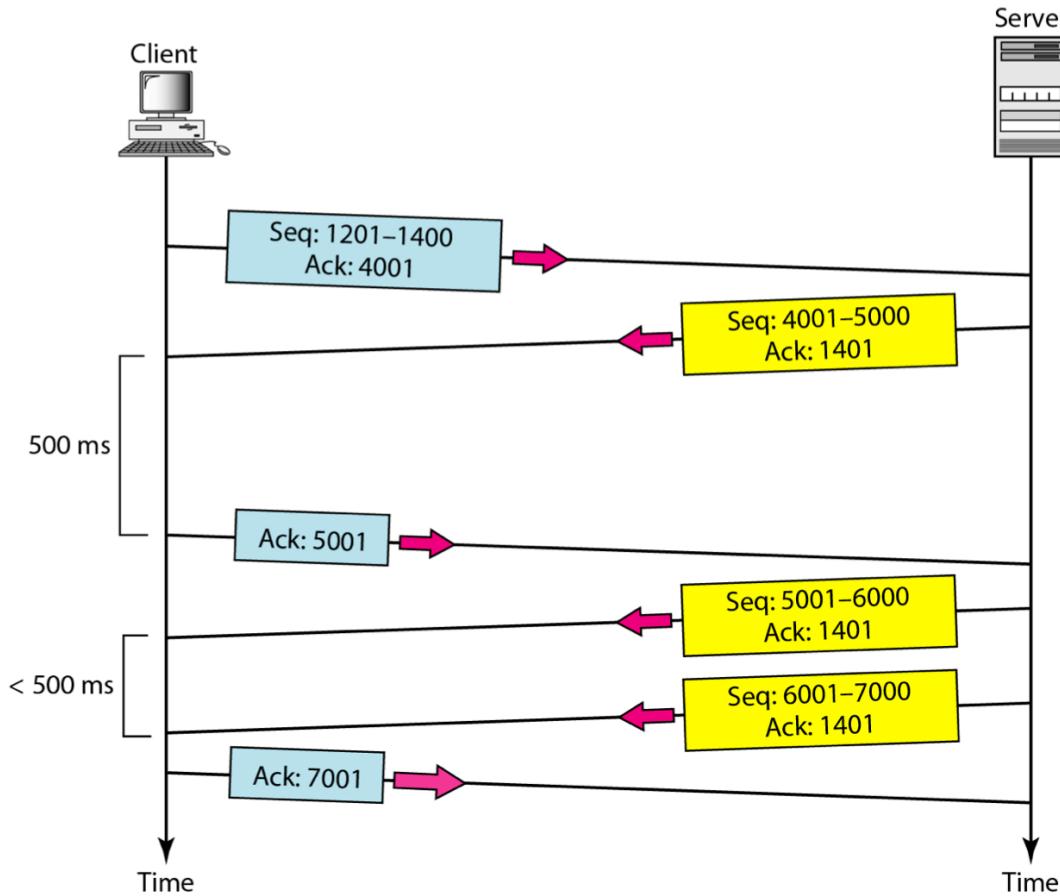


Buffer di ricezione

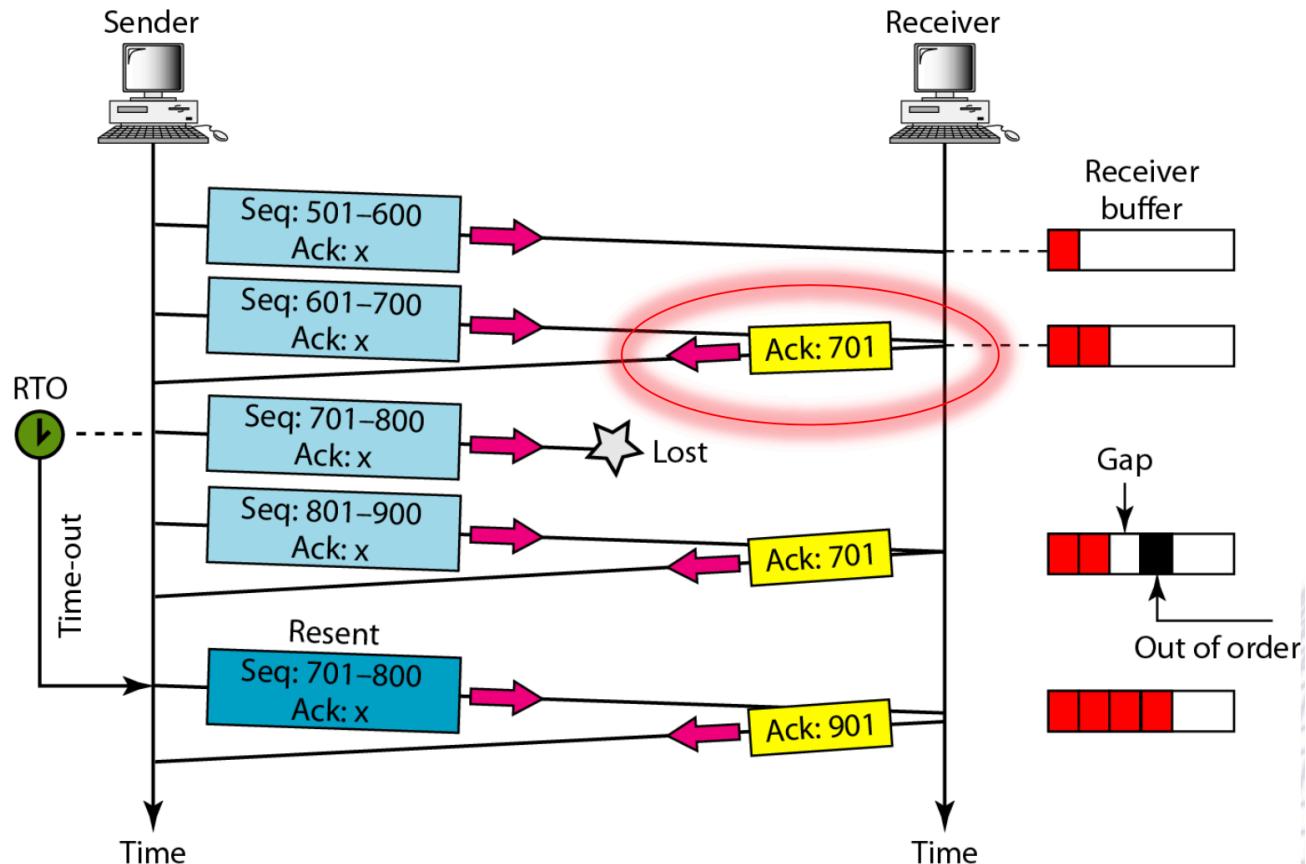
- Quando il receiver non riesce a ricevere altri dati risponde con **win=0**
- Il sender non manda più nulla con l'eccezione di **Urgent data** che possono sempre essere trasmessi, per esempio per killare un processo all'altro lato
- Il sender può rimandare un segmento di un byte per chiedere al receiver 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



Caso normale



Caso di segmento perso (ACK duplicato)



Silly window sindrome

- 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!
- La soluzione di Clark (1982) è di non mandare l'update della finestra fino a quando il receiver non ha parecchio spazio libero

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 prevenirla

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

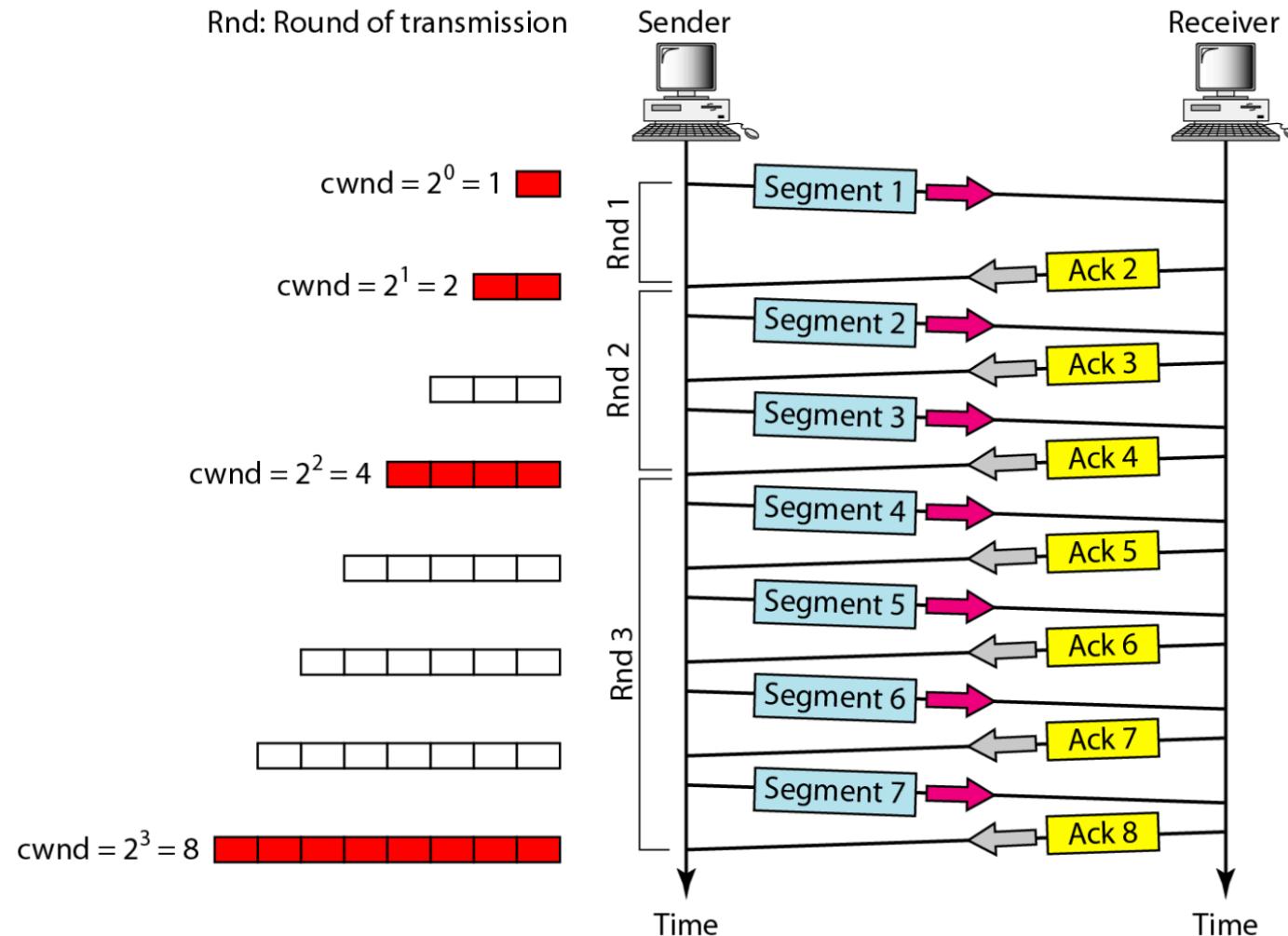
Dimensioni di cwnd:

- 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 $2 * \text{MSS}$ e quindi vengono spediti due segmenti
- Se arrivano i 2 ACK, cwnd viene aumentata di 2 segmenti
- Se la finestra è di n segmenti e tutti gli n ricevono un ack in tempo, la finestra viene aumentata di altri n segmenti.

Slow start

- In questa maniera cwnd 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

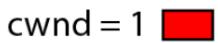


Threshold

- 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

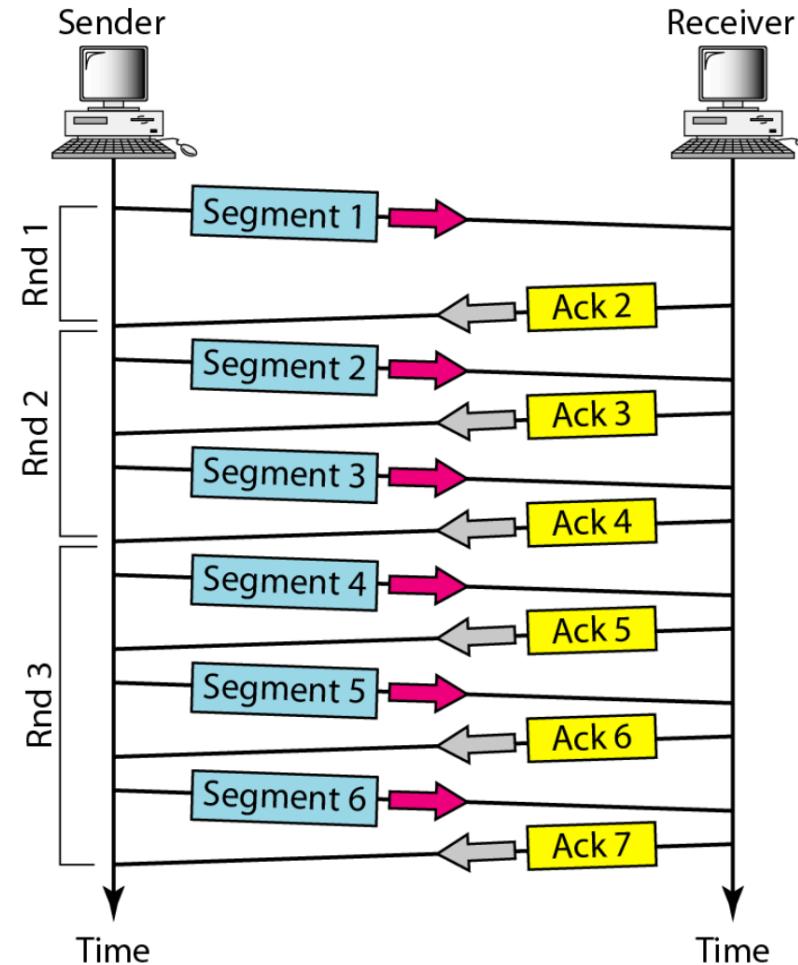
Rnd: Round of transmission

cwnd = 1 

cwnd = 1 + 1 = 2 

cwnd = 2 + 1 = 3 

cwnd = 3 + 1 = 4 



Congestione

- Quando ho un timeout devo diminuire la cwnd
- Questo viene fatto con una diminuzione 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 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 casi

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