UNIVERSITÀ DI PARMA

DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE ED INFORMATICHE
Corso di Laurea in Informatica

Il Livello Trasporto

RETI DI CALCOLATORI - a.a. 2022/2023

Roberto Alfieri

Il Livello Trasporto: sommario

PARTE I

- Scopo del livello Trasporto
- L'indirizzamento
- Il modello client/server
- Il protocollo UDP
- I servizi orientati alla connessione
- Il protocollo TCP

PARTE II

- Congestione, Qualità del Servizio
- Algoritmi Slow start, Tahoe, Fast Recovery
- QoS e controllo del traffico
- Servizi differenziati e integrati

RIFERIMENTI

- *Reti di Calcolatori, A. Tanenbaum, ed. Pearson*
- *Reti di calcolatori e Internet, Forouzan , Ed. McGraw-Hill*

Scopo del livello di Trasporto

- Fornire al **livello applicazione** paradigmi astratti per la comunicazione tra 2 processi: flusso di byte, scambio di messaggi, chiamata a funzione, ecc.

- Offre al livello applicativo una **interfaccia indipendente** dalle diverse tecnologie dello strato di rete (es IPv4, IPv6).

- Per assolvere le sue funzioni utilizza i **servizi** dello **strato di rete**

- **Presupposti della rete sottostante**:
  - I pacchetti possono andare perduti
  - arrivare in ordine modificato
  - consegnati in più copie
  - con ritardi indefiniti

Servizi del livello di Trasporto

I servizi di trasporto di Internet si basano sulla Socket Library introdotta dall'Univ. di Berkeley nel 1982

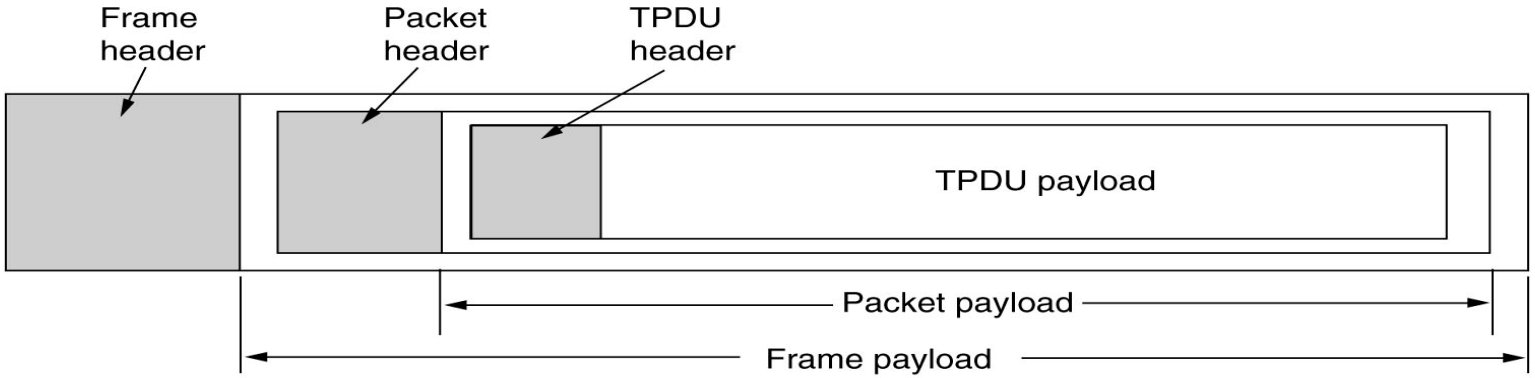
Fornisce 2 tipi di servizi di Trasporto:

- **Servizio affidabile orientato alla connessione: stream sockets (TCP)**
  - Garanzia di integrità, completezza e ordine
  - Gli utenti TCP vedono la connessione come una Pipe
- **Scambio (inaffidabile) di Datagrammi: datagram sockets (UDP)**
  - Ogni datagramma viene inviato senza garanzia di consegna
  - Eventuali ordinamenti nell'invio di una successione di datagrammi devono essere gestiti dall'applicazione.

Processi di ricezione e di trasmissione

1) Nel processo di **ricezione** dei pacchetti dal livello rete (**Demultiplexing)** il livello di trasporto gestisce un indirizzamento (porta) che serve per associare il pacchetto IP in arrivo al processo applicativo a cui è destinato: analizza la porta di destinazione indicata nel pacchetto e smista il pacchetto processo applicativo corretto.

2) Nel processo di **spedizione** (**Multiplexing**) il dato viene eventualmente ridotto in **segmenti** (detti anche TPDU – Transport Protocol Data Unit) che vengono imbustati nell'header di trasporto con l'indicazione della porta di destinazione.



Demultiplexing: Le porte di Berkeley Socket Library

La porta è un identificativo numerico (16 bit , 64K porte) che rappresenta il punto di arrivo di una connessione su di un host. La coppia (IPaddr, Port) identifica quindi univocamente un estremo di una connessione ed è detta **Socket**.

Una connessione tra gli host A e B è identificata da una coppia di socket

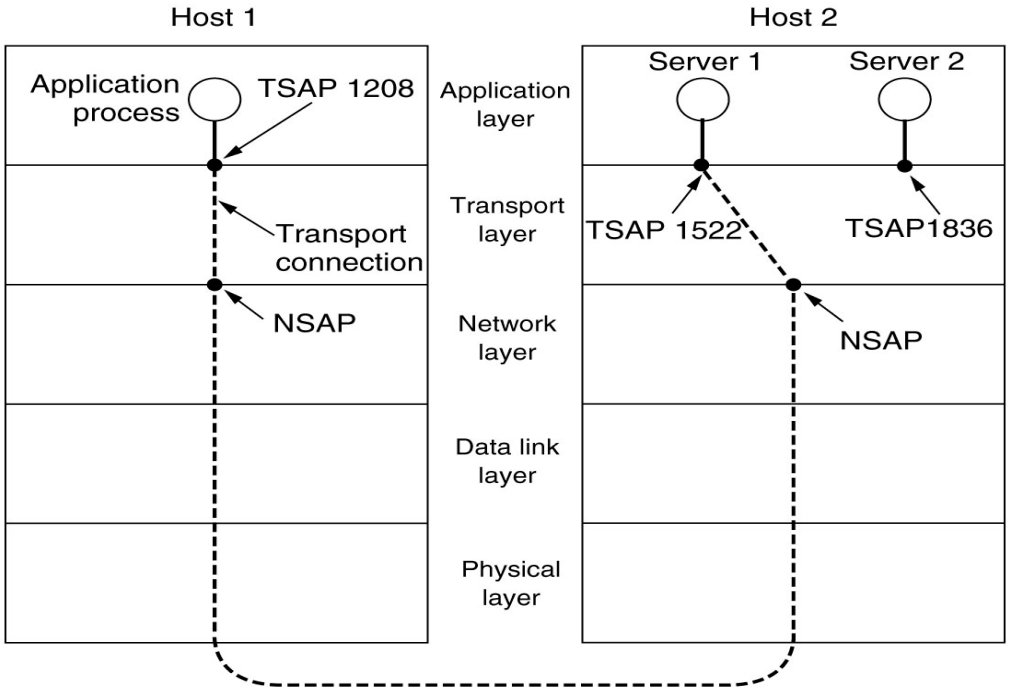
IPaddrA,PortA – IPaddrB, portB

Le porte inferiori a 1024 sono dette "**Well-Known-Port**" e vengono universalmente associate alle principali applicazioni server da **IANA**, per agevolare l'identificazione del Socket server. Vedi http://www.iana.org/assignments/port-numbers

In diversi sistemi operativi le Well-Known-Ports possono essere assegnate solo da processi con privilegi.

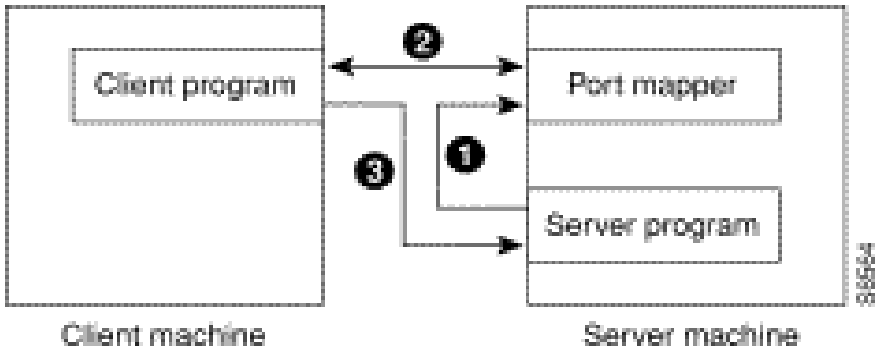
I processi server creati da utenti senza privilegi possono usare le porte non privilegiate (da 1025 a 32768)

Le porte da 32768 a 61000 sono dette **effimere**, assegnate dinamicamente ai processi client.



Come trovo la porta?

- Se il **servizio è standard** il server utilizza una “Well known port”, che tutti conoscono, o una porta non privilegiata.
- Per **servizi di rete dinamici** si può utilizzare un Name Server (Directory Server) con un servizio di PortMapper in ascolto su una Well Known Port, su cui i servizi di rete registrano la porta di ascolto (1) . Il client interroga il Portmapper per conoscere la porta del Server (2), quindi contatta il Server (3). Questo meccanismo è utilizzato dal protocollo RPC.



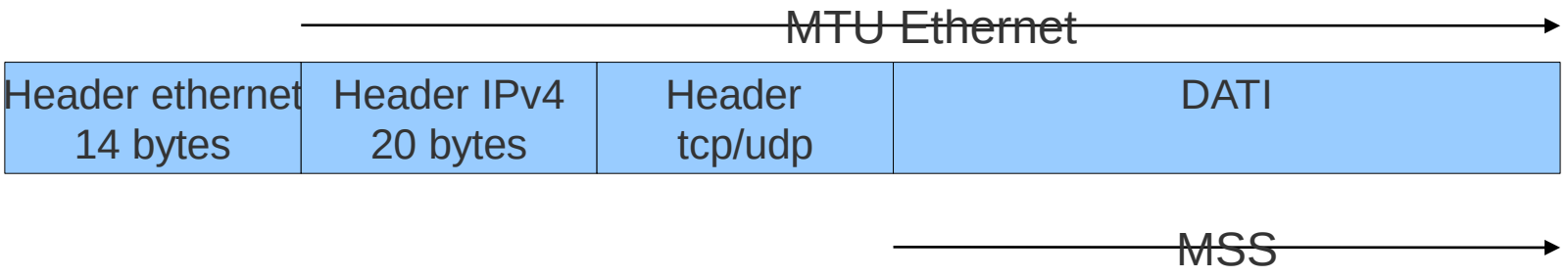
Segmentazione

Il mittente fraziona il flusso dell'applicazione in segmenti che avranno una dimensione massima detta MSS (Maximum Segment Size).

I segmenti vengono consegnati al layer Network (IP) il quale si occuperà della consegna all'host di destinazione.

Se durante il tragitto viene incontrato un Link con MTU inferiore alla dimensione del pacchetto il protocollo IP frammenterà il pacchetto in 2 o più parti, per poi ricomporle a destinazione.

Per evitare la frammentazione normalmente viene definito l'MSS in base al MTU dell'interfaccia locale (meno i byte dell'header TCP/UDP e i byte dell'header IP).



Il modello client/server

La Berkeley Socket Library utilizza un modello di comunicazione di tipo **Client/Server:**

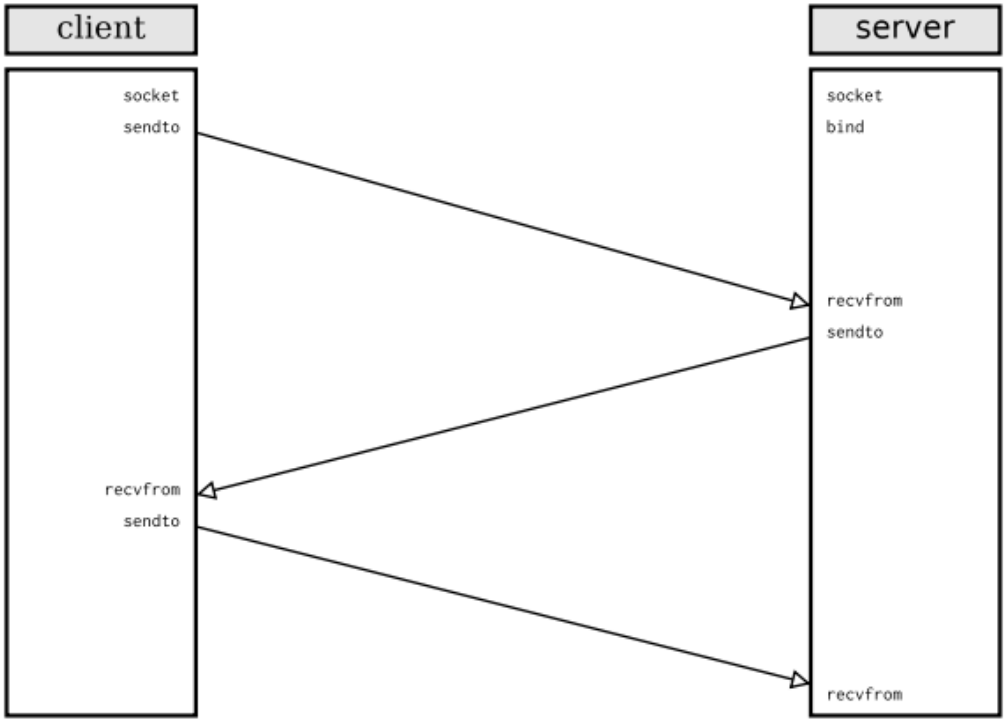
- un estremo "**Server"** è sempre in ascolto su una porta stabilita.
  - La primitiva **bind()** assegna un indirizzo locale (Porta) ad un socket.
- L'altro estremo **"Client"** prenderà contatto con il server specificandone il socket.
  - Il client per poter contattare il server deve quindi conoscerne indirizzo IP e porta.
  - La porta utilizzata dal client apparirà al server nell'intestazione di trasporto, quindi la porta del client non deve essere nota a priori. Generalmente viene determinata dinamicamente dal sistema operativo al momento della richiesta di connessione.

Programmazione Servizi a Datagrammi

Quando **il client** invia il messaggio con **sendto()** riceve dal sistema operativo un numero di porta dinamico.

Al successivo (eventuale) **recvfrom()** il client si mette in ascolto sulla stessa porta, nota al server poiché contenuta nel messaggio inviato dal client.

Il server inizia con un **recvfrom()** e la sua porta di ascolto deve essere stabilita dall'applicazione mediante la primitiva **bind().**



https://gapil.gnulinux.it/fileshare/gapil.pdf

Programmazione Servizi Connection-Oriented

Per i servizi **“Connection Oriented” (TCP)** la libreria fornisce:

- la primitiva **listen()** predispone le code di attesa per i processi client che accederanno contemporaneamente al servizio

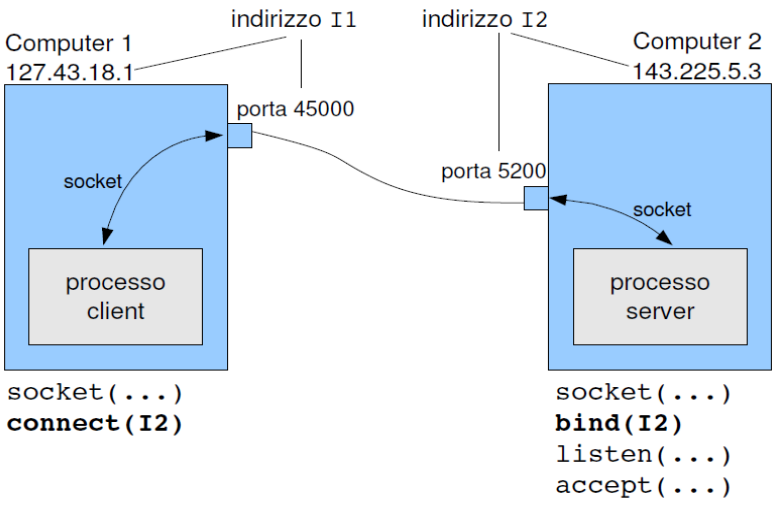
- **accept() è** una primitiva bloccante che consente al server di mettersi in ascolto sulla porta. Quando arriva una TPDU il server crea un nuovo socket con le stesse proprietà di quello originale e ritorna un file descriptor per esso. Il server può creare un nuovo processo (fork) o un nuovo thread per gestire la connessione sul nuovo socket e tornare ad aspettare la prossima connessione.

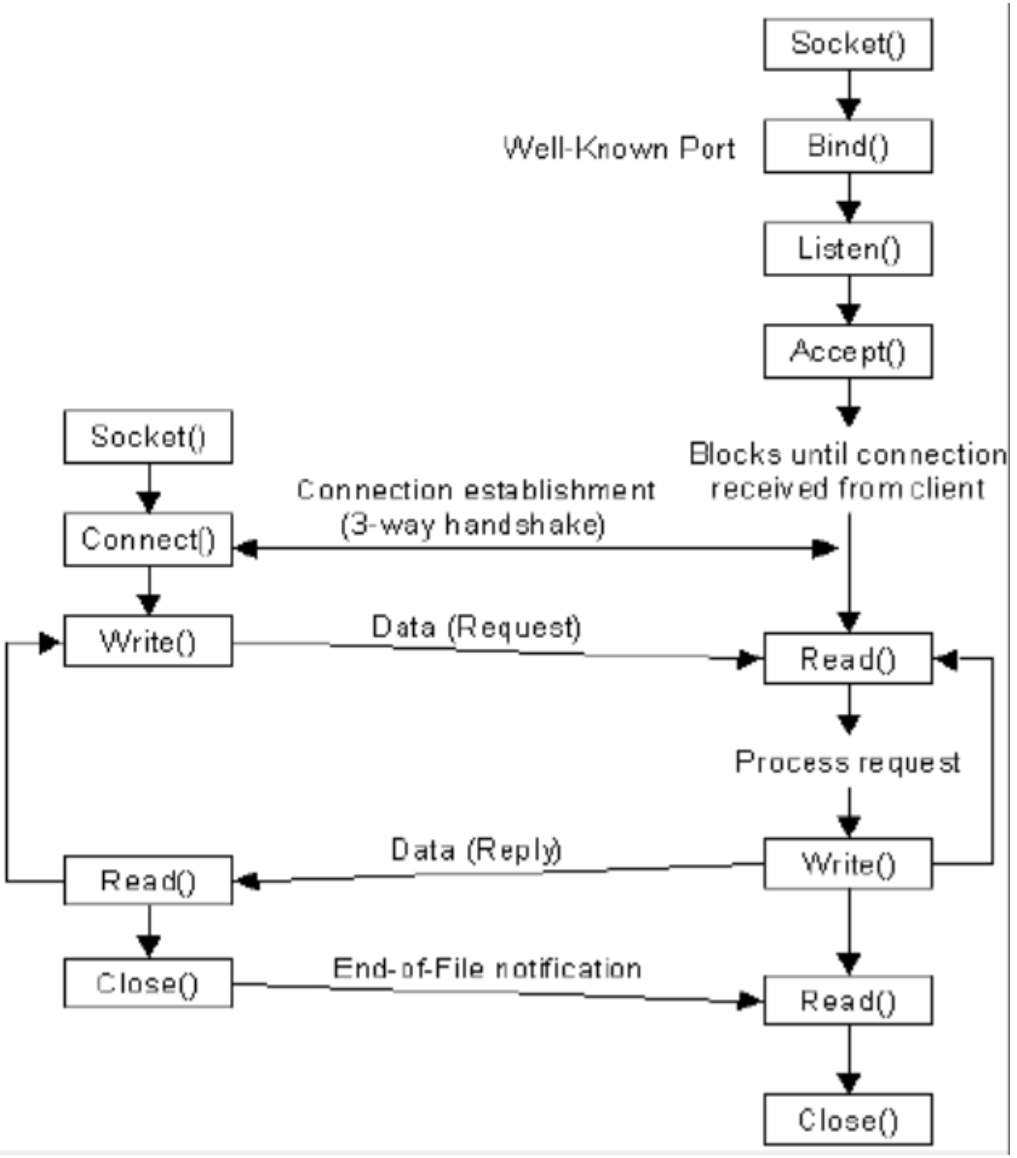
- **connect()** è utilizzata dal client per aprire una connessione.

- Quando la connessione è instaurata la distinzione tra client e server non esiste più anche se normalmente il primo invio di dati viene fatto dal client con la primitiva **send()** la cui corrispondente **recv()** deve essere attivata dall'altro estremo.

Nota: In Unix/Linux si possono usare anche le primitive ***read()/write()***

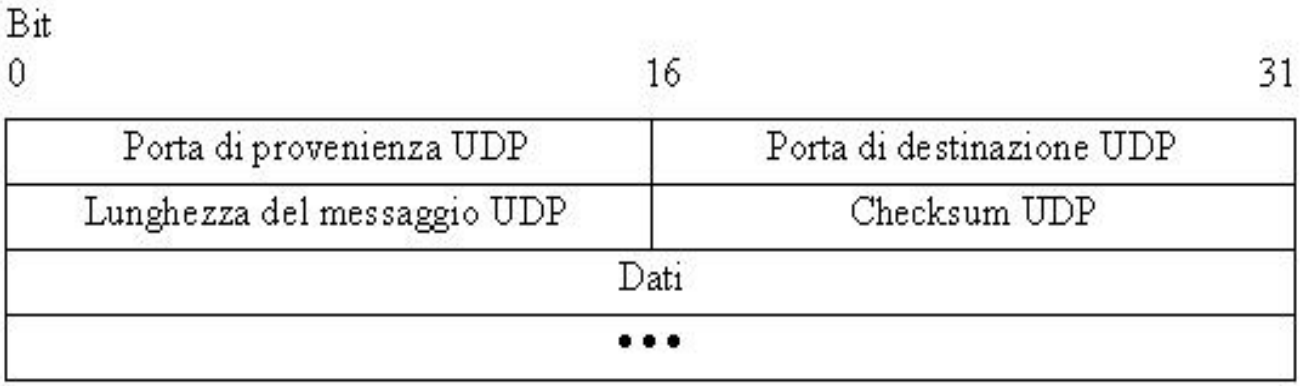
Programmazione Servizi Connection-Oriented





Il protocollo UDP

- Descritto in RFC768, offre alle applicazioni un modo per inviare datagrammi senza dover stabilire una connessione.
- L'unica differenza importante rispetto a IP è l'aggiunta delle porte di origine e destinazione necessarie per il demultiplexing.
- L'intestazione UDP contiene inoltre la lunghezza del segmento (header+dati) e il checksum facoltativo che è la somma delle sequenze di 16 bit in complemento a 1.
- UDP Viene utilizzato per
  - l'implementazione di protocolli applicativi che richiedono lo scambio di brevi messaggi (esempi: **DHCP, DNS, TFTP**)
  - la costruzione (a livello applicativo) di servizi di trasporto più astratti denominati "protocolli Middleware". (esempi: **RPC** e **RTP)**.
  - Comunicazioni multicast o broadcast



Il protocollo TCP

- Descritto in RFC 793 per fornire un **flusso di Byte** end-to-end affidabile a partire da un servizio di rete inaffidabile (IP).
- Le connessioni TCP sono **full-duplex e Unicast** (no multicast, no broadcast)
- TCP riceve flussi di byte dai processi locali, li spezza in segmenti e li spedisce in datagrammi IP separati.
- L'applicazione che spedisce (send()) consegna i dati in un buffer di spedizione. I byte possono essere raggruppati (o frazionati) in segmenti da consegnare al livello rete.
- Segmenti di max 64KB, ma quasi sempre MSS=1460 byte che, con le aggiunte degli header TCP e IP, arriva a 1500 che è l'MTU di Ethernet
- Il flag "PUSH" può essere usato per l'invio non ritardato.
- Il livello TCP di destinazione scrive i segmenti nel buffer di destinazione e consegna all'applicazione (recv()) tutti i byte riscontrati (ricevuti in ordine), ricostruendo il flusso originale.

Servizio orientato alla connessione

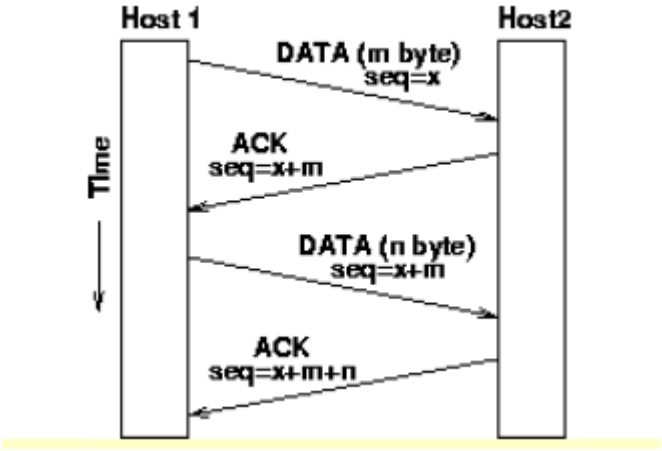
Le problematiche per questo tipo di servizio (analogamente al livello Data_Link) sono le seguenti:

- **Come si attiva una connessione**
- **Come si chiude una connessione**
- **Come si controlla l'ordinamento dei dati**
- **Come si controlla il flusso**
- **Come si gestiscono gli eventuali errori o perdite di pacchetti**

Corretta consegna e ordinamento dei segmenti

Il **riscontro** (conferma dell'avvenuta ricezione di un segmento di dati) o ACKnowledgement , abbinato al **numero di sequenza** attribuito ad ogni pacchetto dati (esempio tftp) o a ogni byte del flusso (esempio tcp), rappresentano un strumento molto utilizzato per la verifica della **corretta consegna e dei pacchetti e del relativo ordinamento.** Il protocollo consiste nei seguenti passi:

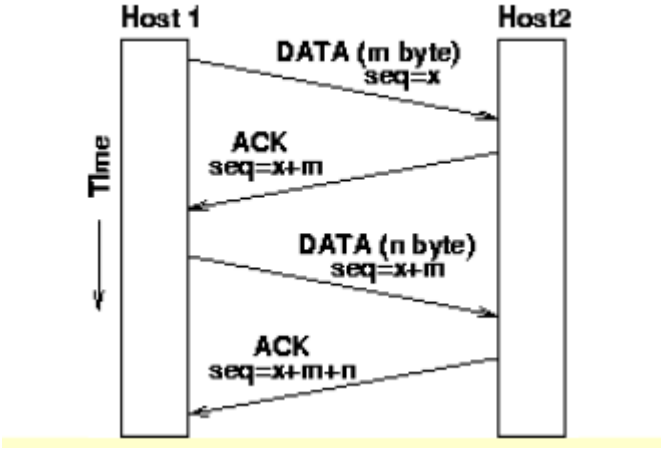
- Il mittente invia un **segmento di dati** assieme ad un numero progressivo
- Il destinatario invia un pacchetto con un Flag (**ACK**) attivo e il numero del byte (o del segmento) ricevuto correttamente
- Il mittente attiva un **Retransmission Time Out (RTO)** per ogni segmento inviato.



Corretta consegna e ordinamento dei segmenti in TCP

r l'ordinamento dei dati TCP usa la numerazione dei Byte:

- Il numero di sequenza è il numero del primo byte dei dati contenuti nel segmento
- Il numero di riscontro, che accompagna l'ACK, è il numero del prossimo byte che il destinatario si aspetta di ricevere
- Il numero iniziale della sequenza non è 0, ma è determinata in modo da evitare che in seguito alla reinizializzazione di una connessione si faccia confusione tra vecchi e nuovi pacchetti
- Il mittente gestisce **un unico timer** per la ritrasmissione (**Retransmission Time Out - RTO),** basato sul RTT e associato al più vecchio segmento non riscontrato. Quando arriva una notifica intermedia, si riavvia il timer sul più vecchio segmento non riscontrato.
- Se non riceve ACK di un segmento ricomincia a spedire dall'ultimo byte riscontrato (GoBack-N) a meno che non sia concordato il Selective ACK (TCP con SACK).



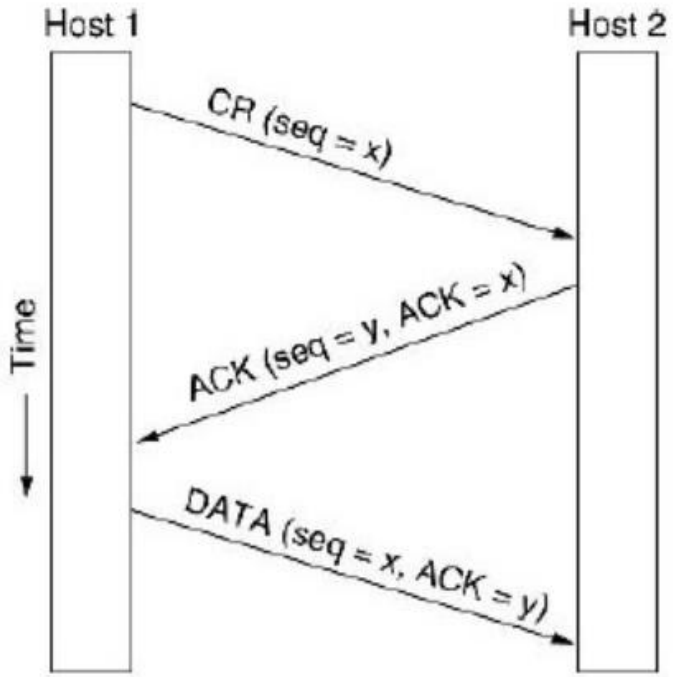
Attivazione della connessione

Il dialogo tra client e server per l'attivazione di una connessione deve tenere conto dell'inaffidabilità della rete sottostante.

Il problema maggiore è dato dai possibili duplicati ritardati che non devono essere confusi con nuove connessioni.

La soluzione proposta da **Tomlinson** (1975) è un meccanismo di HandShaking a 3 vie:

1) Il client invia un segmento di Connection Request (CR) con un valore iniziale di sequenza
2) Il server risponde con un ACK che riscontra il valore di sequenza proposto dal Client e propone un valore iniziale di sequenza per il senso inverso (da server a client).
3) Il client invia un terzo segmento con ACK e il riscontro della sequenza del server (che eventualmente può anche trasportare i primi dati)



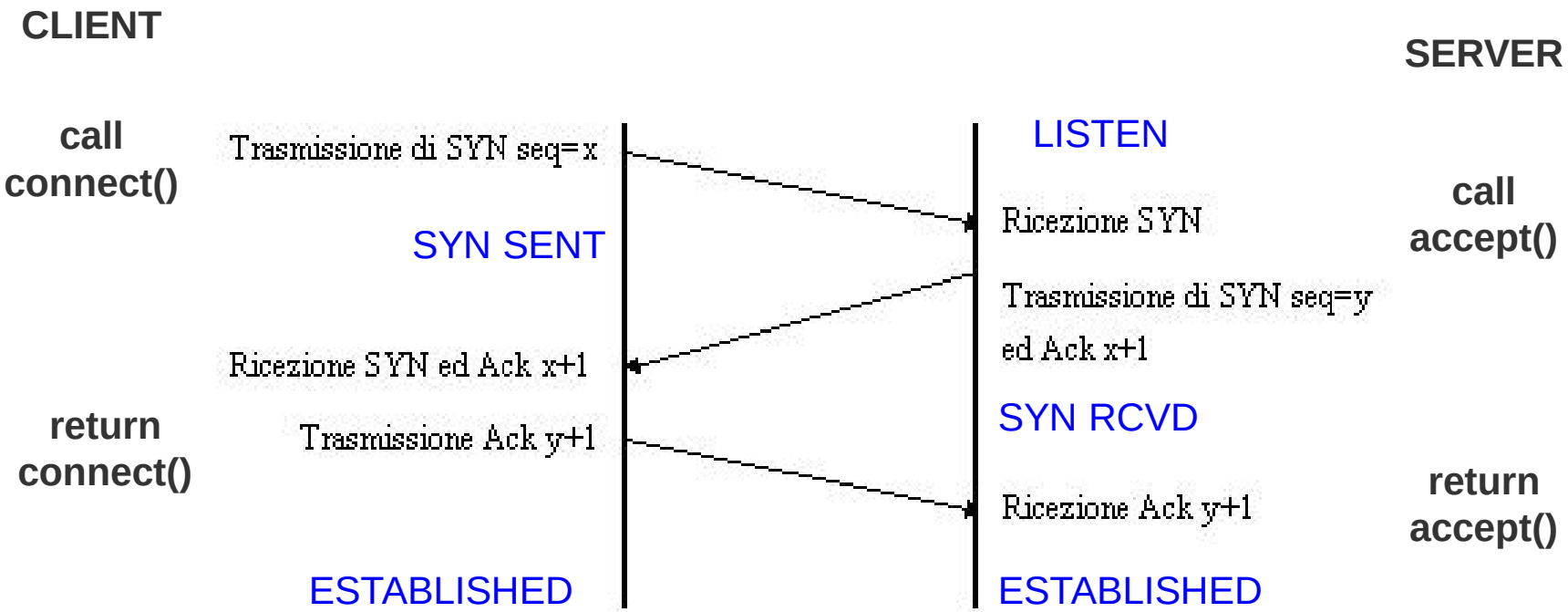
Apertura di una connessione TCP

La soluzione in uso in TCP è derivata dall'algoritmo di Tomlinson:

1) La CONNECT sul **client** invia un segmento con SYN=1, ACK=0, seq=x (random)

2) Se il **server** è in ascolto (LISTEN) e accetta la connessione, risponde con un segmento in cui ACK=1, SYN=1, ACKseq=x+1 (il destinatario riscontra il byte numero x e dichiara che x+1 è il prossimo byte che si aspetta di ricevere) e seq=y (random)

3) Il **client** termina l'apertura riscontrando la sequenza del server: ACK=1, ACKseq=y+1



Chiusura della connessione TCP

Si utilizza un handshake a 2 vie per ogni direzione.

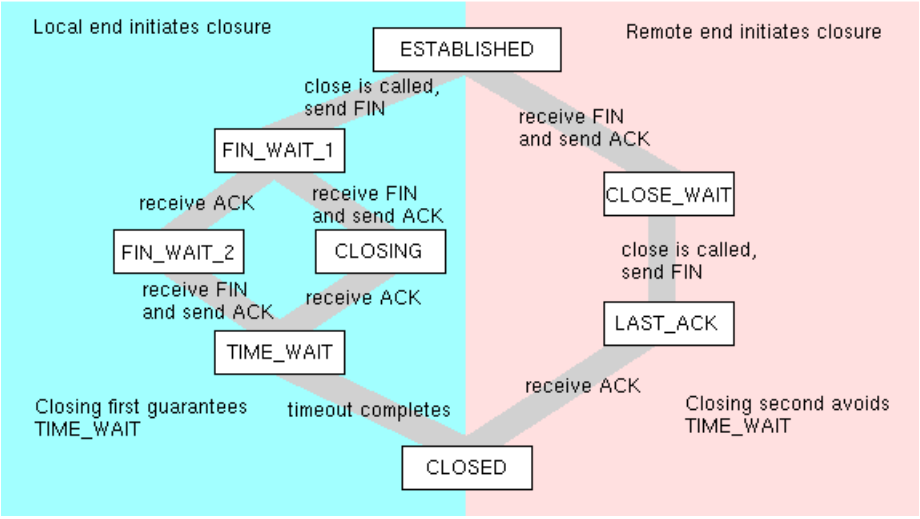
Generalmente viene fatto con 3 segmenti, inviando il secondo FIN assieme all'ACK

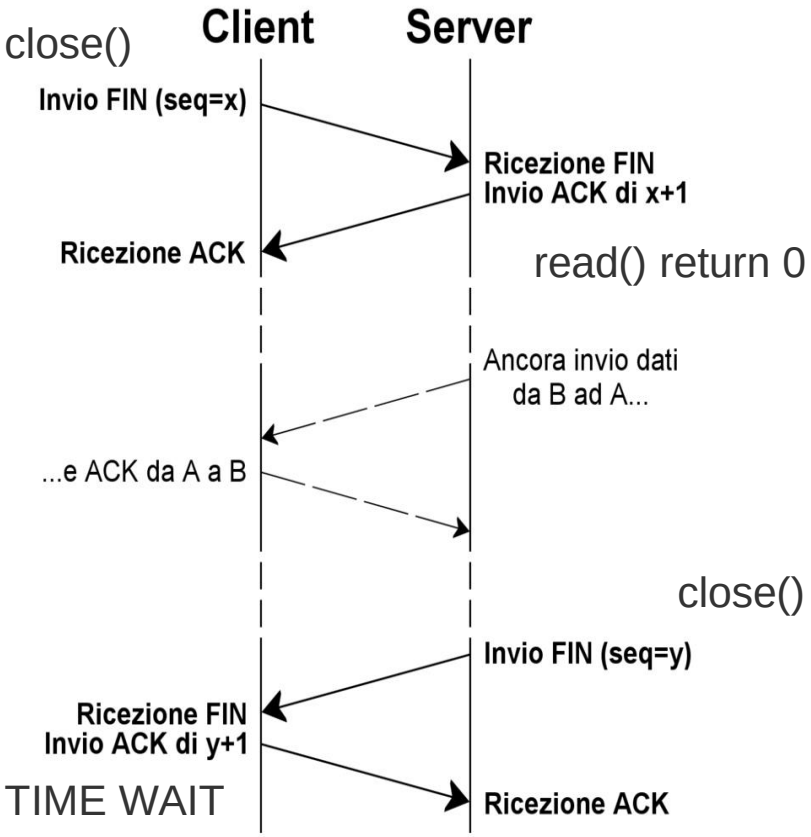
La primitiva close() determina l'invio del FIN, marca come chiuso il canale e ritorna immediatamente. Il canale non è più utilizzabile con read() o write().

Se una risposta FIN non arriva entro 2 RTT il mittente FIN rilascia la connessione.

TIME WAIT attende per 2 MSL (Maximum Segment Lifetime) l'arrivo di eventuali pacchetti ancora in rete.

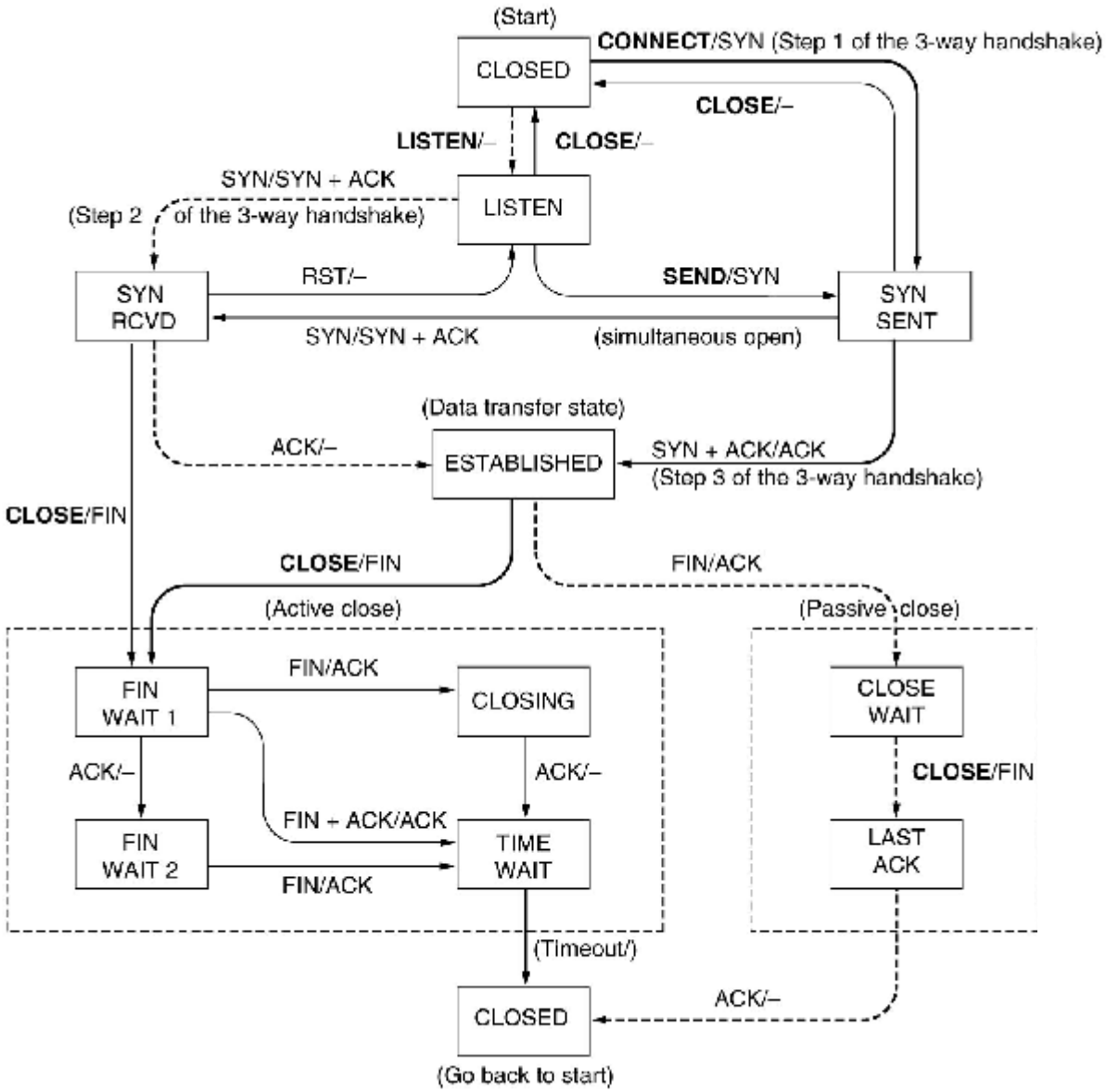
MSL è una stima del tempo di vita di un segmento. In Linux è tipicamente 30 s.





Connessione tra A e B chiusa

Gli stati della connessione TCP



Linee continue grosse: client
Linee tratteggiate: server
Linee sottili: eventi inusuali

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 fo
SYN SENT	The application has started to open a cor
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 simultaneo
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Buffering

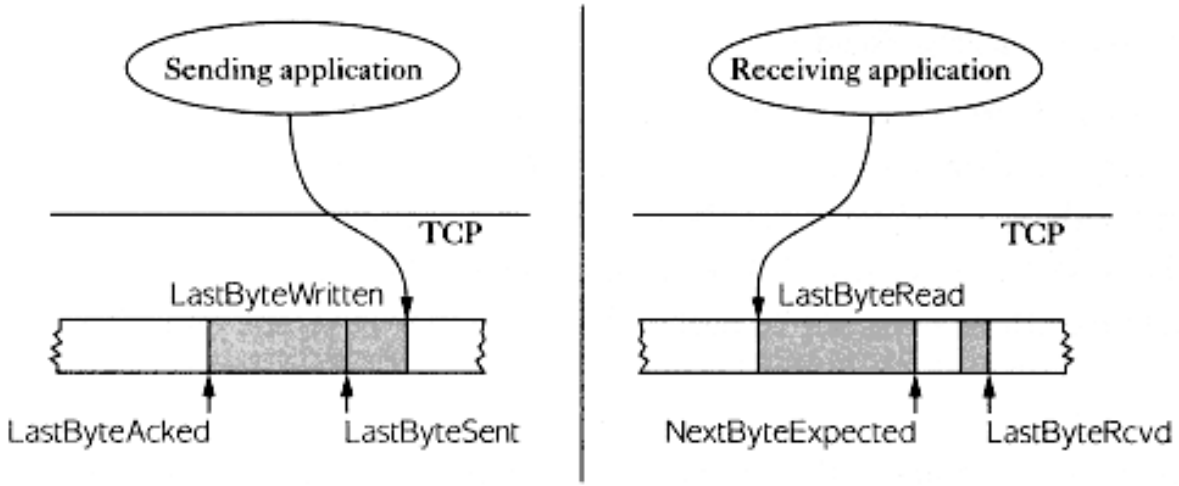
Per ogni connessione TCP è necessario un buffer (coda circolare) di trasmissione e un buffer di ricezione poiché i segmenti potrebbero andare perduti / fuori ordine e perché i processi di scrittura e lettura potrebbero lavorare a diverse velocità

Il **buffer di trasmissione** contiene:

- dati spediti ma non ancora riscontrati (tra LastByteSent e LastByteAcked)
- dati ancora da spedire (dopo LastByteSent)
- Spazio libero

Il **buffer di ricezione** contiene:

- dati ricevuti e riscontati non ancora letti dall'applicazione
- dati ricevuti non ancora riscontrati (tipicamente dati ricevuti fuori ordine)
- spazio libero



I buffer in Linux

In ambiente Linux possiamo controllare i buffer nel seguente modo:

```
sysctl -a | grep tcp
net.ipv4.tcp_rmem= 4096 87380 174760 (buffer ric.  min-init-max)
net.ipv4.tcp_wmem= 4096 16384 131072 (buffer sped. min-init-max)
```

La dimensione dei buffer può essere modificata con setsockopt().

SO_RCVBUF SO_SNDBUF Imposta dimensioni del Buffer di Ricezione/Trasm

```
//Esempio di raddoppio del buffer del ricevente
int rcvBufSize;
int sockOptSize;
sockOptSize = sizeof(rcvBuffersize);                //preleva la dimensione di origine del buffer
if(getsockopt(m_socket, SOL_SOCKET, SO_RCVBUF, &rcvBufSize, &sockOptSize) < 0)
/*gestione errore*/
printf("initial receive buffer size: %d\n", rcvBufSize);
RcvBufSize *=2;                                     //raddoppia la dimensione del buffer
if(setsockopt(m_socket, SOL_SOCKET, SO_RCVBUF, &rcvBufSize, sizeof(rcvBufSize)) < 0)
 /*gestione errore*/
```

Socket Non Blocking

La **send() è per default bloccante**; si blocca quando il buffer in trasmissione è pieno e ritorna quando si libera spazio nel buffer di trasmissione. Se lo spazio nel buffer è insufficiente per i dati da spedire viene effettuata una scrittura di una porzione di dati minore o uguale alla dimensione del buffer libero, e la send() restituisce il numero di byte scritti.

Se **il buffer è pieno** e il socket è impostato come **non bloccante** non ci sarà nessun blocco ma ritornerà un -1 settando la variabile di errore **EWOULDBLOCK .**

La **recv() è per default bloccante**; si blocca quando il buffer in ricezione è vuoto e ritorna quando ci sono dati nel buffer. Il numero di byte letti può essere inferiore al numero di byte richiesto.

Ritorna 0 quando non ci sono dati nel buffer e l'altro peer ha chiuso la connessione.

Se il **buffer è vuoto** e il socket è impostato come **non bloccante** non ci sarà nessun blocco ma ritornerà un -1 settando la variabile di errore **EWOULDBLOCK**

Per mettere il socket in modalità non bloccante:

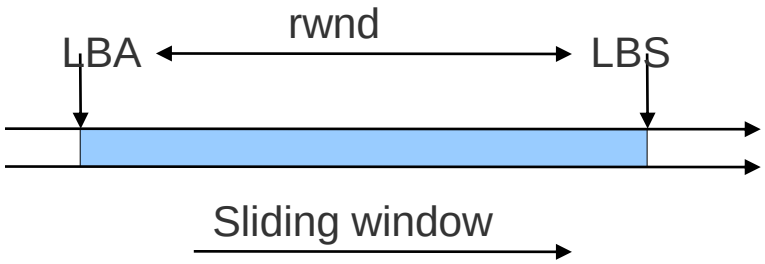
```
int flags, sockfd;
sockfd = socket(....);
if ( (flags=fcntl(sockfd,F_GETFL,0)) <0 ) exit(1);
flags |= O_NONBLOCK;
if ( fcntl(sockfd,F_SETFL,flags) <0 )   exit(1);
```

Controllo di Flusso : Sliding Window

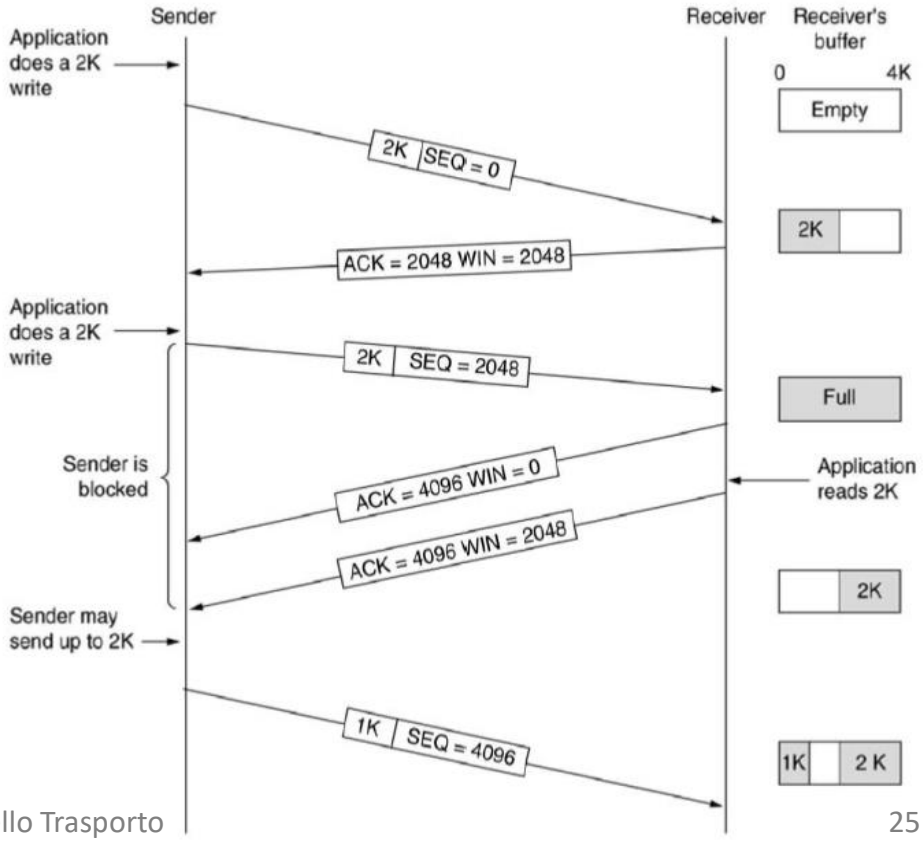
Per il controllo del flusso e l'ottimizzazione del throughput della rete in TCP si utilizza Il meccanismo denominato Sliding Window (finestra scorrevole):

- Il **Ricevente** annuncia al trasmettitore la **Receiver Window Size (rwnd)**, che generalmente corrisponde al numero di byte liberi sul buffer di ricezione e indica quanti byte possono essere inviati a partire dall'ultimo riscontrato.
- Il trasmettitore può inviare anche più dati senza riscontro, purché il numero di byte non riscontrati non ecceda rwnd:

LastByteSent - LastByteAcked < rwnd



Il ricevitore può bloccare la trasmissione (ad eccezione degli Urgent Data) inviando rwnd=0 (Stop-and-wait)



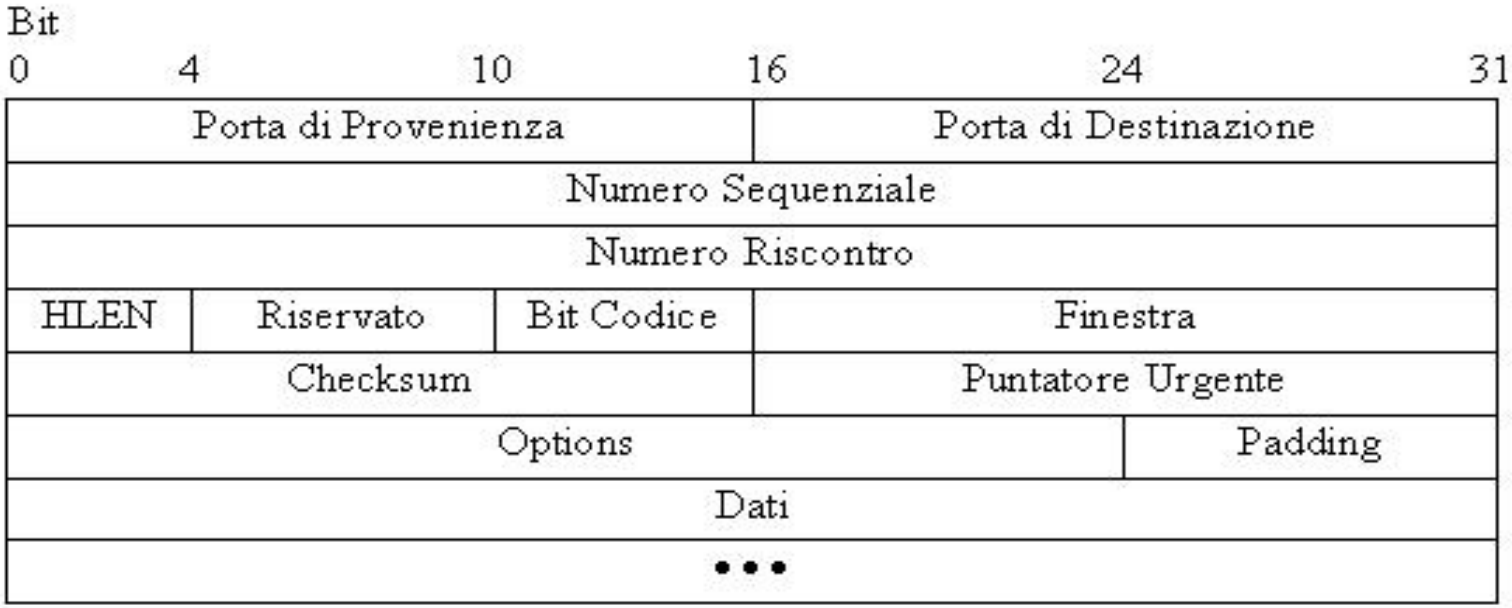
La Trama TCP

Porta di provenienza e destinazione identificano gli estremi della connessione.

Il **Numero Sequenziale** è il contatore del flusso di byte spediti. Indica il numero del **primo byte** di dati contenuto nel segmento.

Il **Numero di Riscontro** è il contatore del numero di byte ricevuti. Indica il numero del **prossimo byte che il destinatario si aspetta di ricevere**.

HLEN (parole di 32 bit dell'Header) è necessario perché il campo opzioni è variabile.



La Trama TCP : I bit di codice

Dopo HLEN ci sono 4 bit riservati per sviluppi futuri, poi abbiamo 8 bit di codice.

Se attivi (posti a 1) significano:

- **CWR e ECE** vengono utilizzati quando è attivo ECN (gestione della congestione - RFC 3168)
  -- **ECE (ECN-Echo):** usato per mandare ad un host l'indicazione di rallentare
  -- **CWR** e' generato dall'host per indicare che ha ridotto la finestra di congestione

- **URG**: si deve considerare il campo "**puntatore urgente**"
- **ACK** : si deve considerare il "**numero di riscontro**"
- **PSH** : il ricevente non deve bufferizzare, ma renderli subito disponibili all'applicazione
- **RST**: reset della connessione a causa di qualche tipo di problema.
- **SYN**: utilizzato nella fase di attivazione di una connessione
- **FIN** : utilizzato nella fase di rilascio di una connessione

La Trama TCP: finestra, checksum e urgente

Finestra (16 bit): è la dimensione della Sliding Window, ovvero il numero di byte che il destinatario è in grado di ricevere a partire dall'ultimo byte riscontrato.

La dimensione massima sarebbe di 64KB , ma può essere aumentata attraverso la Scala della Finestra (vedi opzioni).

Il **CheckSum** (16 bit): somma in complemento a 1 delle sequenze di 16 bit del segmento TCP (header e dati) e la "pseudo-intestazione"

La **pseudo-intestazione** include ulteriori informazioni importanti di IP e TCP (IP source, IP dest, 0x00, 0x06, TCP Segment length), violando però l'indipendenza dei protocolli perché include dati del Layer IP.

Puntatore URGENTE (16 bit): Puntatore a dato urgente, ha significato solo se il flag URG è impostato a 1 ed indica lo scostamento in byte dell'ultimo dato urgente. Tipicamente sono messaggi di controllo; usato raramente.

La Trama TCP: opzioni

Ciascun segmento SYN contiene in genere delle **Opzioni** per il protocollo TCP che servono per comunicare all'altro capo una serie di parametri utili a regolare la connessione.

Normalmente vengono usate le seguenti opzioni:

- **MSS:** massima dimensione del segmento che il destinatario può accettare.
  (E' possibile determinare questo valore con l'opzione del Socket TCP_MAXSEG)

- **Scala della finestra:** La finestra è normalmente di 16 bit (max 64K).
  Con il crescere In alcuni casi (alta velocità o alta latenza) questa finestra è insufficiente.
  In questo caso la Scala determina numero di shift a sinistra (2x) della finestra.

- **SACK**: Selective ACKnowledgement

- **timestamp**: (**TSval** , timestamp value) è un marcatore temporale spedito dal mettente e rimbalzato poi dal destinatario (**TSecr** , timestamp echo reply), per il calcolo del RTT (RTT = current time – TSecr).

Se i byte delle opzioni non sono multipli di 4 (parole di 32 bit) viene aggiunto un padding opportuno.

Determinazione del MSS ottimale (RFC 1191)

La frammentazione introduce un **overhead sull'attività dei router.**

Frammenti troppo piccoli determinano un **overhead sul traffico di rete.**

L'MSS ottimale è l'MTU minimo tra tutti gli MTU incontrati nel tragitto, ma questo dato non è noto quando si inizia una trasmissione e potrebbe cambiare nel tempo.

L'algoritmo generalmente utilizzato è descritto nell'RFC1191:

- Il mittente determina la MSS (Maximum Segment Size) generalmente con la MTU dell'interfaccia locale (meno 20 byte dell'header TCP e 20 dell'header TCP) e lo comunica all'altro host attraverso le **Opzioni dell'header TCP**
- Quindi invia il primo Segmento con il bit DF (Don't Fragment --- Non Frammentare) settato a uno. Se durante il cammino si incontra un router con MTU inferiore questo invierà al mittente un pacchetto ICMP di errore che verrà utilizzato per correggere l'MSS.

La Trama TCP: Opzione SACK

Normalmente TCP funziona con **GoBackN**: Se il ricevente ottiene un segmento errato e N segmenti validi, il riscontro si ferma all'ultimo valido prima dell'errore. Questo manda in TimeOut il mittente che rimanda tutti i Segmenti a partire da quello errato.

Per migliorare le prestazioni evitando la ritrasmissione di segmenti validi è stata proposta la tecnica **SACK (Selective ACK,** RFC 2108 e 2883):
il ricevitore indica al trasmettitore quali segmenti sono arrivati correttamente in modo che possa determinare quali segmenti devono essere rispediti.

Funziona con **2 opzioni dell'header TCP:**

– SACK Permitted : Viene incluso in un segmento SYN per indicare la capacità di gestire la tecnica SACK.

– SACK: viene utilizzato dal ricevente per comunicare le informazioni SACK (i segmenti ricevuti correttamente).

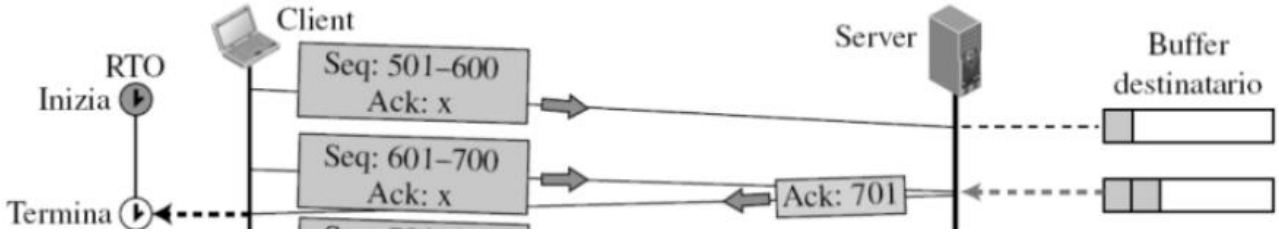
Ad esempio: blocco1 (primo-ultimo), blocco2 (primo-ultimo) , . …

Ottimizzazioni TCP: ACK delayed, ACK cumulativo

Quando il destinatario riceve un segmento in ordine può attendere fino a 500ms l'arrivo del prossimo segmento (**delayed acknowledgement, RFC 1122**)

Se durante l'attesa arriva un'altro segmento in ordine risponde con un singolo **ACK cumulativo** che riscontra l'ultimo byte della sequenza.

Questa tecnica è utilizzata in alcune implementazioni di TCP.

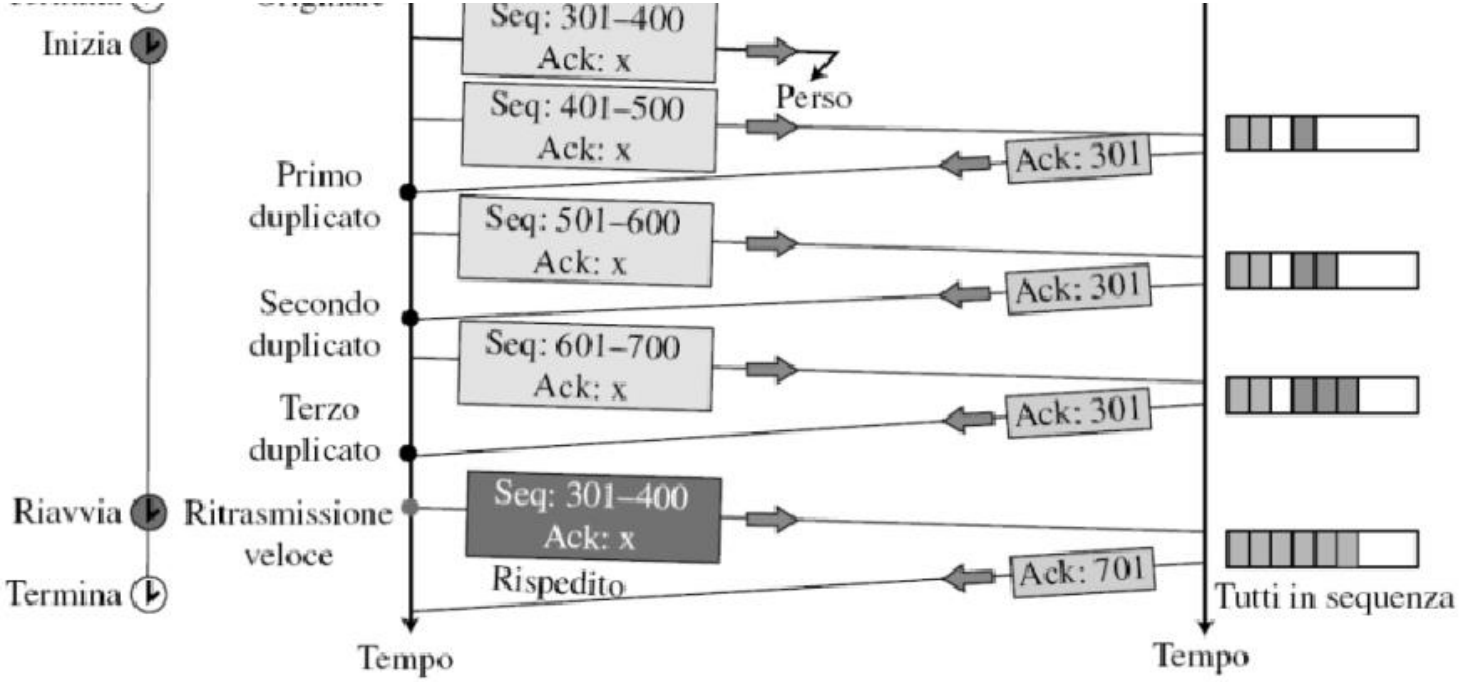


Ottimizzazioni TCP: fast retrasmission

Se il destinatario riceve un segmento fuori ordine (successivo ad altri segmenti non ricevuti) invia un ACK in cui viene riscontrato l'ultimo segmento ricevuto in ordine.

Quando il ricevente riceve 3 ACK che riscontrano lo stesso numero capisce che un segmento è andato perduto e lo ritrasmette, senza attendere lo scadere del timer (Fast Retransmission).

Il destinatario risponde con un singolo ACK che riscontra anche i successivi segmenti ordinati (ACK cumulativo).



Ottimizzazioni TCP: Algoritmo di Nagle e soluzione di Clark

Le prestazioni possono degradare in alcuni casi particolari quali:

- **il trasmettitore genera dati lentamente;** ad esempio quando si edita un file per ogni tasto premuto girano 4 pacchetti IP per un totale di 162 byte.
- **il ricevitore consuma dati lentamente;** ad esempio il destinatario pubblica finestre di pochi byte, perché l'applicazione legge pochi byte per volta, il mittente è costretto a spezzare il flusso in tanti segmenti (problema della finestra futile)

Per attenuare il problema **lato mittente** si usa l'**algoritmo di Nagle:**

- se il mittente ha **pochi byte da spedire** (a causa dell'applicazione o della finestra del destinatario) e **ci sono dati non riscontrati** → aspetta ACK, anche se la finestra scorrevole consentirebbe l'invio di altri dati.
- se il mittente ha **molti byte da spedire** oppure i **segmenti piccoli sono riscontrati** → spedisci subito

Questo algoritmo può essere disabilitato con l'opzione **TCP_NODELAY**

Esempio in C: setsockopt (sock, SOL_TCP, TCP_NODELAY, …);

Per attenuare il problema **lato ricevente** si usa la soluzione **di Clark:**

- Se il ricevente pubblica finestre troppo piccole l'algoritmo forza il ricevitore ad attendere che la finestra raggiunga un valore minimo prima di comunicarlo al mittente.

Il Retransmission TimeOut (RTO) di TCP

Serve per decidere quando un pacchetto deve considerarsi perduto.

Deve essere almeno pari a RTT, ma deve aggiornarsi dinamicamente e deve gestire situazioni di congestione (backoff).

Algoritmo di Jacobson (1988):

Se l'ACK torna indietro prima dello scadere dell'RTO l'algoritmo calcola il valore del RTT (Round Trip Time) e aggiorna le variabili:

RTTMedio = α RTTMedio + (1-α) RTT

DevMedia = α DevMedia + (1-α) abs(RTT - RTTMedio)

(α e' il peso che si vuole dare ai precedenti valori medi, valore tipico 0.9)

RTO = RTTMedio + 4 x DevMedia

Reti congestionate: Algoritmo di Backoff (di Karn)

Se l'RTO scade significa che la rete è congestionata. In questo caso l'algoritmo di Karn prevede di non aggiornare L'RTTMedio e raddoppiare l' RTO fino a quando i segmenti non arrivano a destinazione al primo tentativo

RTO (i+1) = 2 *RTO(i) (backoff esponenziale binario)

I timer di TCP in Linux

Oltre a RTO, che è il più importante, TCP gestisce altri Timer:

- **Timer di Persistenza**: viene attivato quando la finestra viene chiusa (rwnd=0). Se il pacchetto che riapre va perduto, allo scadere del timer il mittente invia una “window probe” che sollecita la rispedizione della finestra. Se la finestra è ancora 0 il timer viene reimpostato.
- **Timed Wait**: Tempo di attesa dopo un FIN. Prima di rilasciare la connessione viene attivato questo timer per gestire eventuali pacchetti circolanti dopo la chiusura. Generalmente corrisponde a doppio del tempo di vita massimo di un pacchetto.
- **Timer di Keepalive:** parte quando la linea è inattiva. Se arriva a zero TCP invia un ACK; se non riceve risposta la connessione viene considerata interrotta.

```
Esempi con Linux:
sysctl -a | grep tcp | grep time
net.ipv4.tcp_fin_timeout = 60          # Time wait: 1 minuto
net.ipv4.tcp_keepalive_time = 7200     # keepalive: 2 ore

echo 900> /proc/sys/net/ipv4/tcp_keepalive_time # keepalive a 15 min.
```