

Reti di Telecomunicazioni



Transport Layer



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



Queste slides possono essere copiate e distribuite gratuitamente soltanto con il consenso dell'autore e a condizione che nella copia venga specificata la proprietà intellettuale delle stesse e che copia e distribuzione non siano effettuate a fini di lucro.



Transport layer



Introduzione

Layer: Modello OSI e TCP/IP

Physics Layer

Data Link Layer

MAC sublayer

Network Layer

Transport Layer

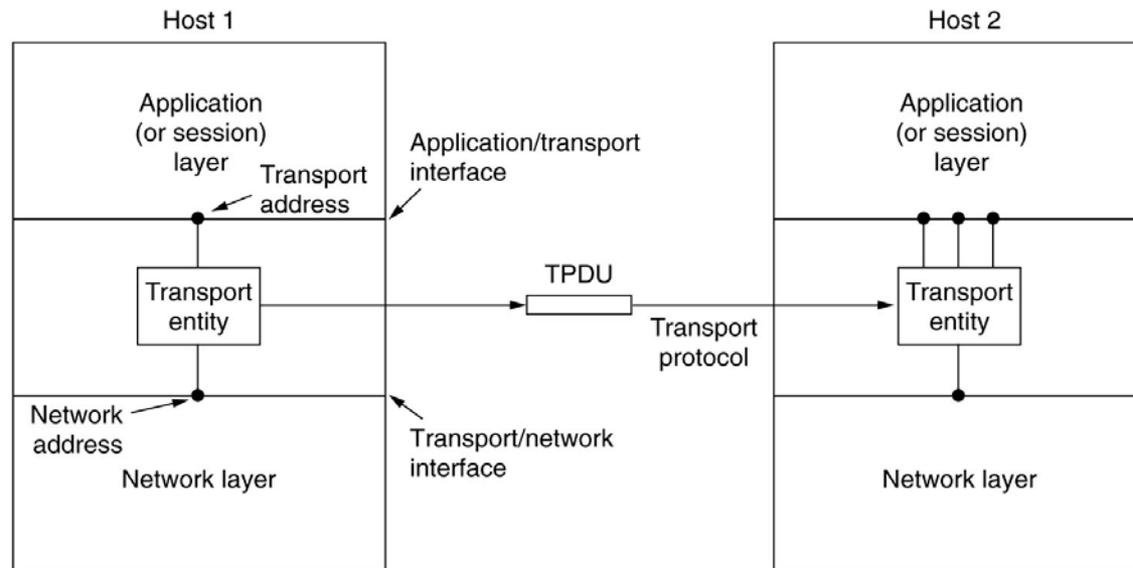
Application Layer



Transport Layer

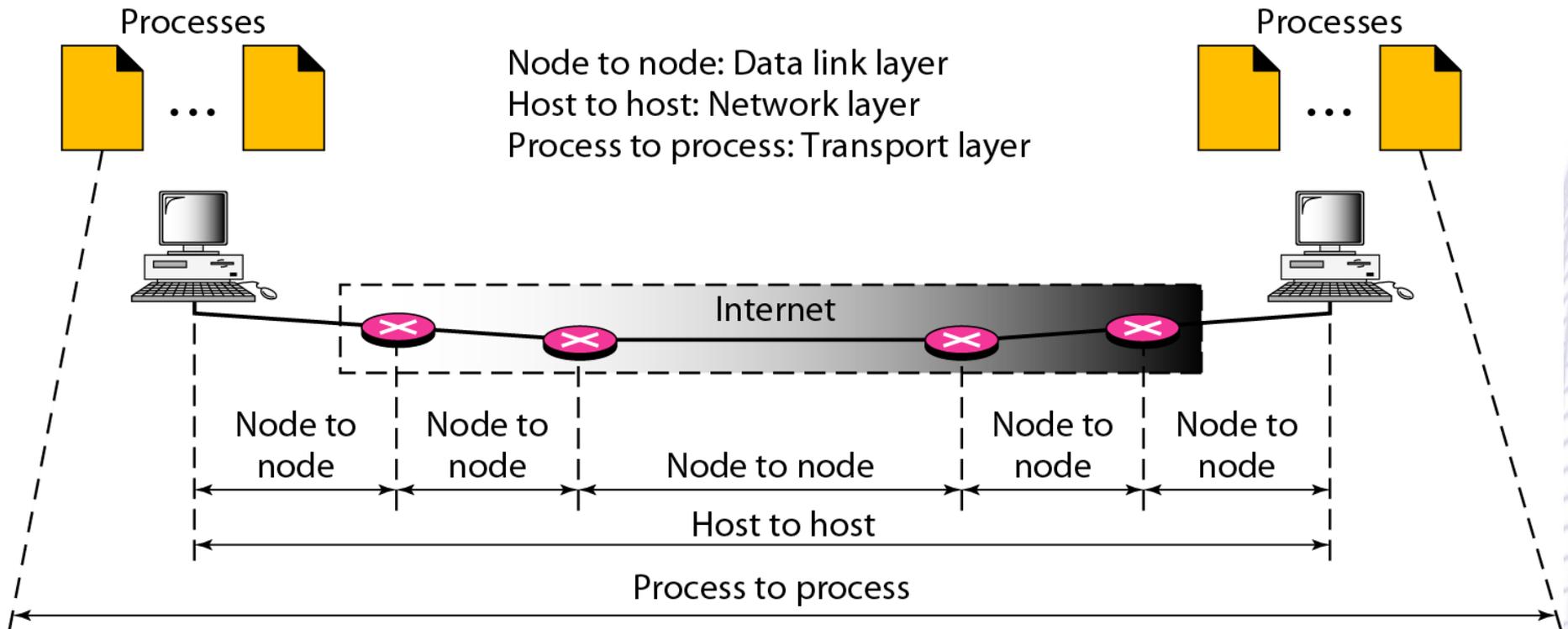


- Deve fornire un servizio di trasporto efficiente ed affidabile ai suoi clienti, normalmente processi del livello applicativo
- Si serve dei servizi forniti dal livello di network
- Solitamente questo software è nel **kernel** del sistema operativo o in processo separato dello spazio **user** in una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





Trasporto





Due tipi di servizio



- Come a livello network, anche a livello trasporto di sono due tipi di servizio:
- **Connection Oriented** e **Connectionless**
- Il servizio di trasporto connection oriented somiglia molto a quello equivalente a livello di rete OSI
 - connessione in tre fasi:setup, trasferimento dati, disconnessione
 - Indirizzamento, flow control
- Anche il livello Connectionless esegue le stesse funzioni del livello connectionless a L3 OSI



Perché due layer?



- Se sono così simili perché abbiamo bisogno di due layer separati?
- Il livello di trasporto gira sulla macchina dell'utente, mentre il livello network gira nei router che sono gestiti dagli operatori della rete
- Cosa succede se una rete offre servizi non adeguati? Es 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
- In pratica permette di avere un servizio più affidabile della rete sottostante



Perché due layer?



- Quindi a questo livello mi occupo 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 Users



- Generalmente si fa una distinzione tra i layer sopra e sotto il livello 4
- I tre layer inferiori si possono vedere come Transport Service Provider
- I restanti layer superiori come Transport Service User



Primitives



- Un livello di trasporto come abbiamo detto deve essere reliable anche se la rete sottostante non lo è
- Prendiamo 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 **primitives** di trasporto sono usate dai normali programmatori mentre pochi utenti si scrivono le proprie funzioni di trasporto e quindi non devono usare le primitives di network.
 - Le **primitives** di trasporto devono quindi essere facili da usare



Esempio



- Esempio di primitives di trasporto, ridotte all'osso
- Prendiamo un server con diversi 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 trasport layer per l'entità di trasporto del server

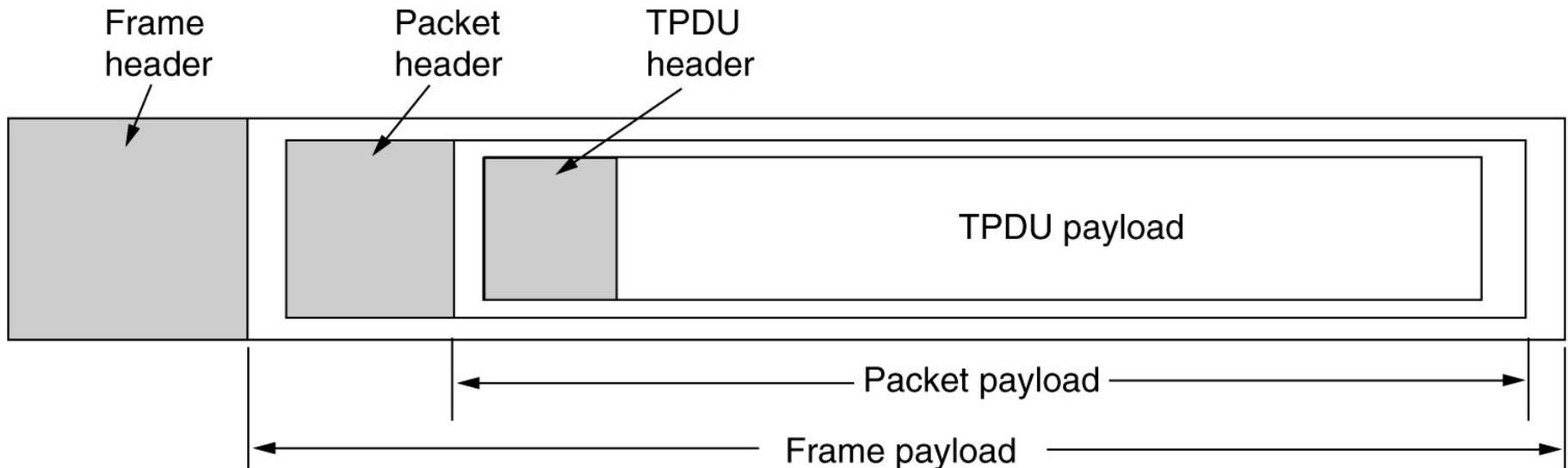
Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection



TPDU



- Chiamiamo TPDU, Transport Protocol Data Unit il pacchetto di livello 4 per distinguerlo dalla PDU del livello 3 (network) che si solito si chiama pacchetto
- Invece a livello 2 abbiamo i frame





Connessione ok



- Quando il client chiama la **CONNECT**, una TPDU di **CONNECTION REQUEST** arriva al server.
- L'entità di trasporto 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 è stabilita



Data Transfer



- Ora client e server si possono mandare dati, usando le primitives **SEND** e **RECEIVE**
- Es uno dei due fa una chiamata blocking 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.
- Semplice e funziona bene solo se entrambe le parti sono d'accordo sui turni



Complicazione



- A questo livello 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:**

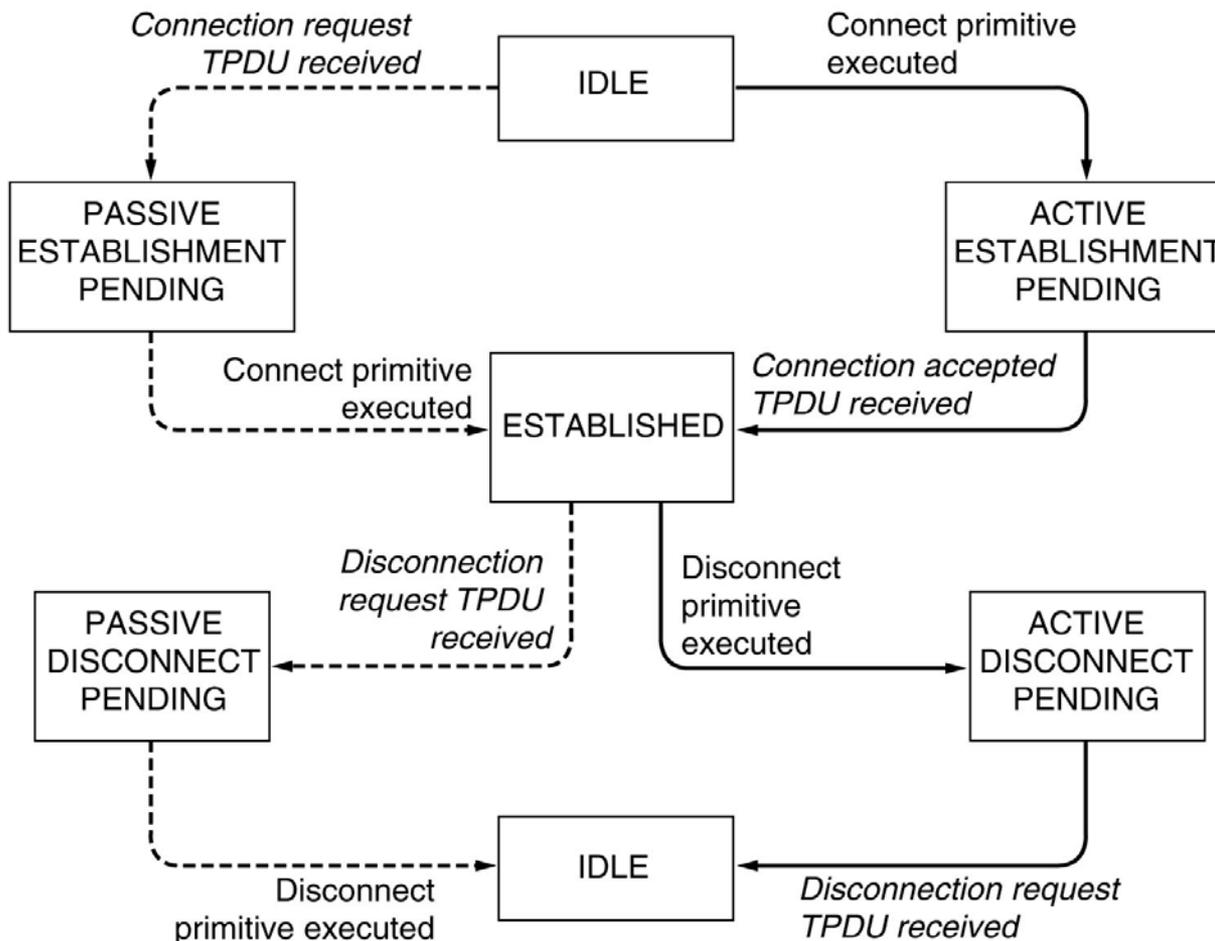
- Uno dei due manda una **DISCONNECT TPDU** all'altra e considera la connessione chiusa
- All'arrivo della TPDU la connessione è chiusa anche per l'altro

- **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



- Insieme “reale” di primitive, molto usate per programmazione in ambiente Internet, sono quelle usate per TCP nel Berkeley Unix
- Molto simili a quelle “basic” già viste nel primo modello ma qui abbiamo più dettagli e più flessibilità
- Lasciamo perdere la TPDU che vedremo più avanti quando studieremo TCP



Primitives dei socket 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

- Le prime quattro sono chiamate dal server nell'ordine indicato



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



BIND



- Un socket appena creato non ha un indirizzo di rete. Questo viene assegnato dalla **BIND**
- Solo a questo punto il client può connettersi
- Il SOCKET non si crea direttamente l'indirizzo perché alcune applicazioni ci tengono al loro indirizzo e se lo vogliono scegliere, mentre ad altre non interessa



LISTEN



- La **LISTEN** alloca dello spazio per accodare chiamate entranti, nel caso diversi client vogliano connettersi allo stesso istante
- Al contrario del primo modello semplice, nel modello SOCKET 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



Client side



- 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



Esempio d'uso



- Proviamo a costruirci un file server
- Il server sta sulla porta 12345 e sta sempre ad aspettare una connessione in cui il server riceve da client il nome di un file che poi manda al client
- Il client contatta il server specificando il nome di un file e lo manda in standard output
- Uso: ***client*** `fileserver /path/to/filename > file`



Client side

Include di sistema

```
/* This page contains a client program that can request a file from the server program
 * on the next page. The server responds by sending the whole file.
 */
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

Porta da usare e block transfer size

```
#define SERVER_PORT 12345
#define BUF_SIZE 4096
```

```
/* arbitrary, but client & server must agree */
/* block transfer size */
```

Controllo argomenti

```
int main(int argc, char **argv)
{
```

```
    int c, s, bytes;
    char buf[BUF_SIZE];
    struct hostent *h;
    struct sockaddr_in channel;
    /* buffer for incoming file */
    /* info about server */
    /* holds IP address */
```

Conversione nome indirizzo

```
    if (argc != 3) fatal("Usage: client server-name file-name");
    h = gethostbyname(argv[1]);
    if (!h) fatal("gethostbyname failed");
    /* look up host's IP address */
```

Creo il socket

```
    s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (s < 0) fatal("socket");
    memset(&channel, 0, sizeof(channel));
    channel.sin_family = AF_INET;
    memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
    channel.sin_port = htons(SERVER_PORT);
```

Cerco di connettermi al server

```
    c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
    if (c < 0) fatal("connect failed");
```

```
    /* Connection is now established. Send file name including 0 byte at end. */
    write(s, argv[2], strlen(argv[2])+1);
```

Loop

```
    /* Go get the file and write it to standard output. */
    while (1) {
```

leggo un bufer dal socket

```
        bytes = read(s, buf, BUF_SIZE);
        if (bytes <= 0) exit(0);
        write(1, buf, bytes);
        /* read from socket */
        /* check for end of file */
        /* write to standard output */
```

fino a end of file

```
    }
```

Scrivilo su standard output

```
fatal(char *string)
{
    printf("%s\n", string);
    exit(1);
}
```

Stampa di errore



Server side

```

#include <sys/types.h> /* This is the server code */
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345 /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096 /* block transfer size */
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{
    int s, b, l, fd, sa, bytes, on = 1;
    char buf[BUF_SIZE]; /* buffer for outgoing file */
    struct sockaddr_in channel; /* holds IP address */

    /* Build address structure to bind to socket. */
    memset(&channel, 0, sizeof(channel)); /* zero channel */
    channel.sin_family = AF_INET;
    channel.sin_addr.s_addr = htonl(INADDR_ANY);
    channel.sin_port = htons(SERVER_PORT);

    /* Passive open. Wait for connection. */
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); /* create socket */
    if (s < 0) fatal("socket failed");
    setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));

    b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
    if (b < 0) fatal("bind failed");

    l = listen(s, QUEUE_SIZE); /* specify queue size */
    if (l < 0) fatal("listen failed");

    /* Socket is now set up and bound. Wait for connection and process it. */
    while (1) {
        sa = accept(s, 0, 0); /* block for connection request */
        if (sa < 0) fatal("accept failed");

        read(sa, buf, BUF_SIZE); /* read file name from socket */

        /* Get and return the file. */
        fd = open(buf, O_RDONLY); /* open the file to be sent back */
        if (fd < 0) fatal("open failed");

        while (1) {
            bytes = read(fd, buf, BUF_SIZE); /* read from file */
            if (bytes <= 0) break; /* check for end of file */
            write(sa, buf, bytes); /* write bytes to socket */
        }

        close(fd); /* close file */
        close(sa); /* close connection */
    }
}

```

Parametri del socket

Creo il socket

Opzione del socket per riusare l'indirizzo

BIND

Disposto ad accettare connessioni

Loop infinito

accetto connessione

leggo nome file

loop sul file

leggo buff size

Miche



compilazione



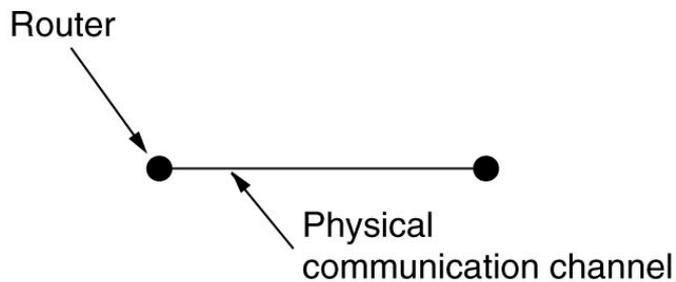
- Questi programmi si compilano in questo modo
- `cc -o client client.c -lsocket -lnsl`
- `cc -o server server.c -lsocket -lnsl`
- La procedura ***fatal*** stampa un messaggio di errore ed esce. Viene usata dal client e dal server ma dal momento che i due programmi vanno compilati separatamente e runnano in macchine differenti, i due programmi non possono dividerne il codice



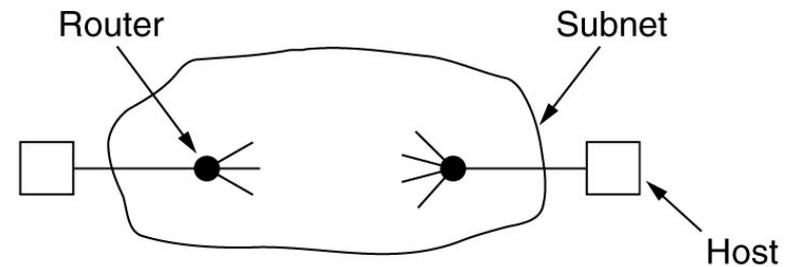
Trasporto vs data link



- A prima vista un protocollo di trasporto assomiglia ad un protocollo datalink
 - Entrambi devono gestire controllo degli errori, sequenze, flow control tra le altre cose
 - Tuttavia mentre a livello datalink due router comunicano direttamente sul canale fisico, **a livello di trasporto al posto del canale fisico c'è una intera subnet**



(a)



(b)



Differenze



- Layer Datalink

- Non devo indirizzare l'altro estremo del link. Ogni linea ha un unico router all'altro lato
- Facile stabilire la connessione. All'altro lato c'è sempre qualcuno che risponde (se non è crashato, ma allora ci sarebbe comunque poco da fare)
- Il link non ha capacità di storage
- Differenza di quantità. Se ho tante linee posso riservare un numero fisso di buffer per ognuna per averne sempre uno disponibile

- Layer Trasporto

- Devo esplicitamente indirizzare la mia destinazione
- Vedremo che ci sono molti problemi nello stabilire una connessione
- La rete si può tenere i pacchetti, perderli, fargli fare il giro del mondo e farli spuntare nel momento peggiore dopo 30 secondi, quando magari mi hanno già mandato un sostituto
- Troppe possibili connessioni per avere un buffer per ciascuna



Indirizzamento



- Indirizzi di trasporto

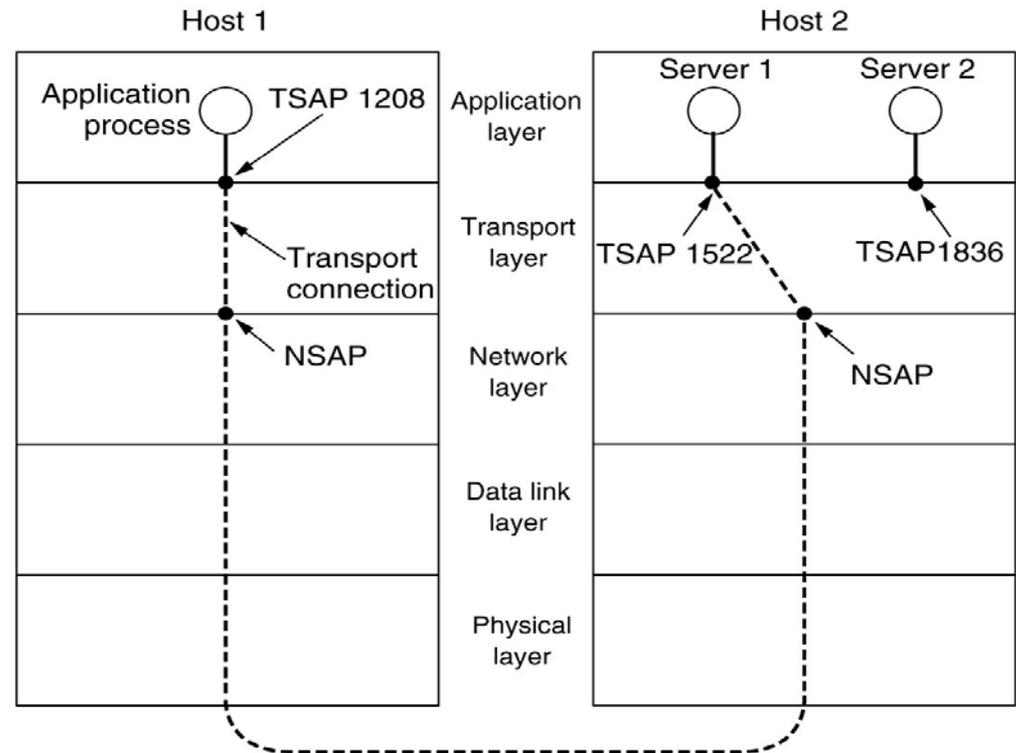
- Quando un processo di un'applicazione vuole stabilire una connessione con un processo di un'applicazione remota deve sapere a quale connettersi
- Lo stesso nel caso connectionless, a chi mando il datagram?
- Si definiscono degli indirizzi di trasporto che sono chiamati **port** (porte o (?) porti)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



TSAP e NSAP



1. Un processo “time of day” su host2 si attacca al TSAP 1522 in attesa di una call (tipo una LISTEN)
2. Un processo su host1 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





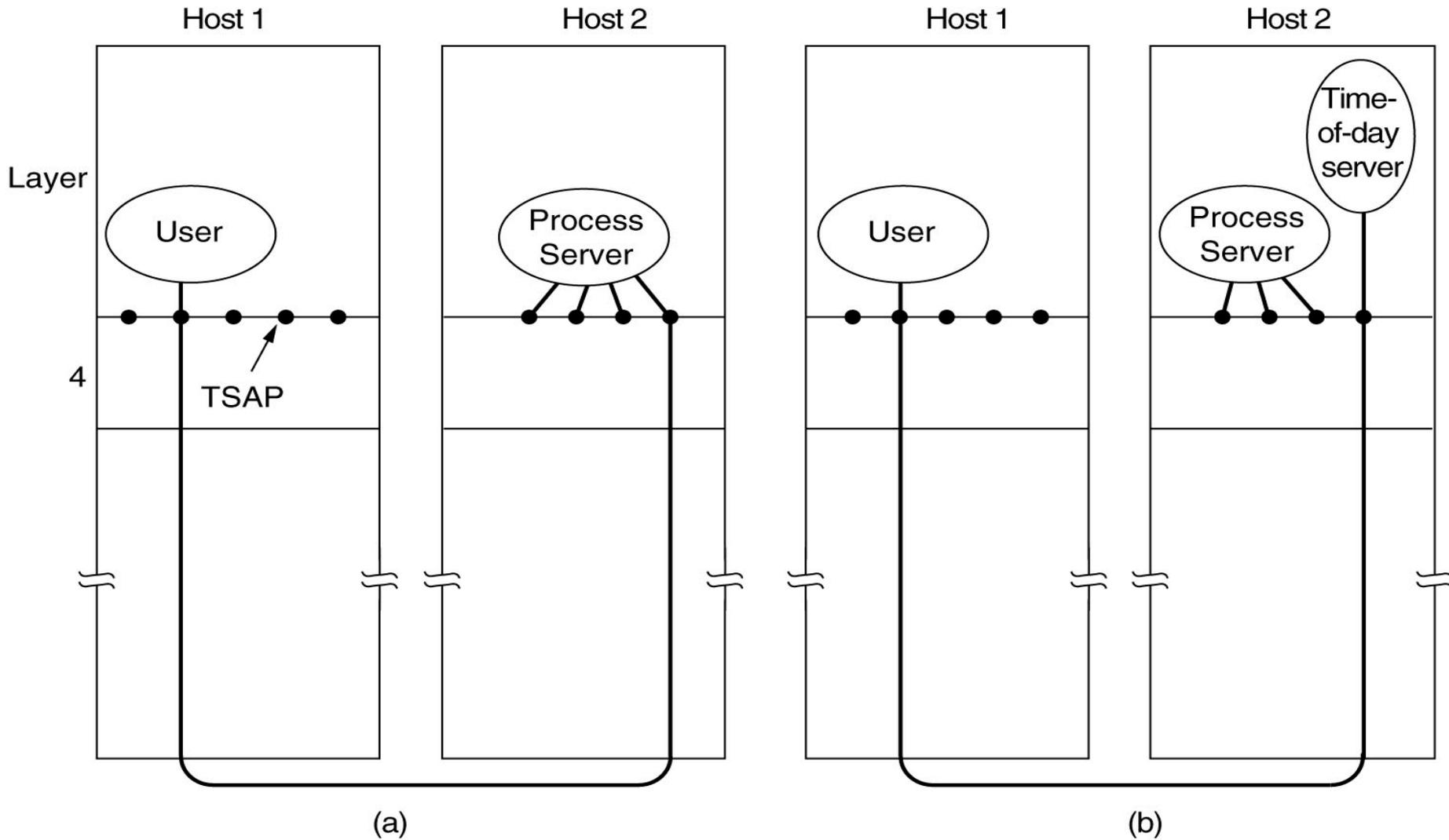
Come trovo la porta?



- 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 know services
 - 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

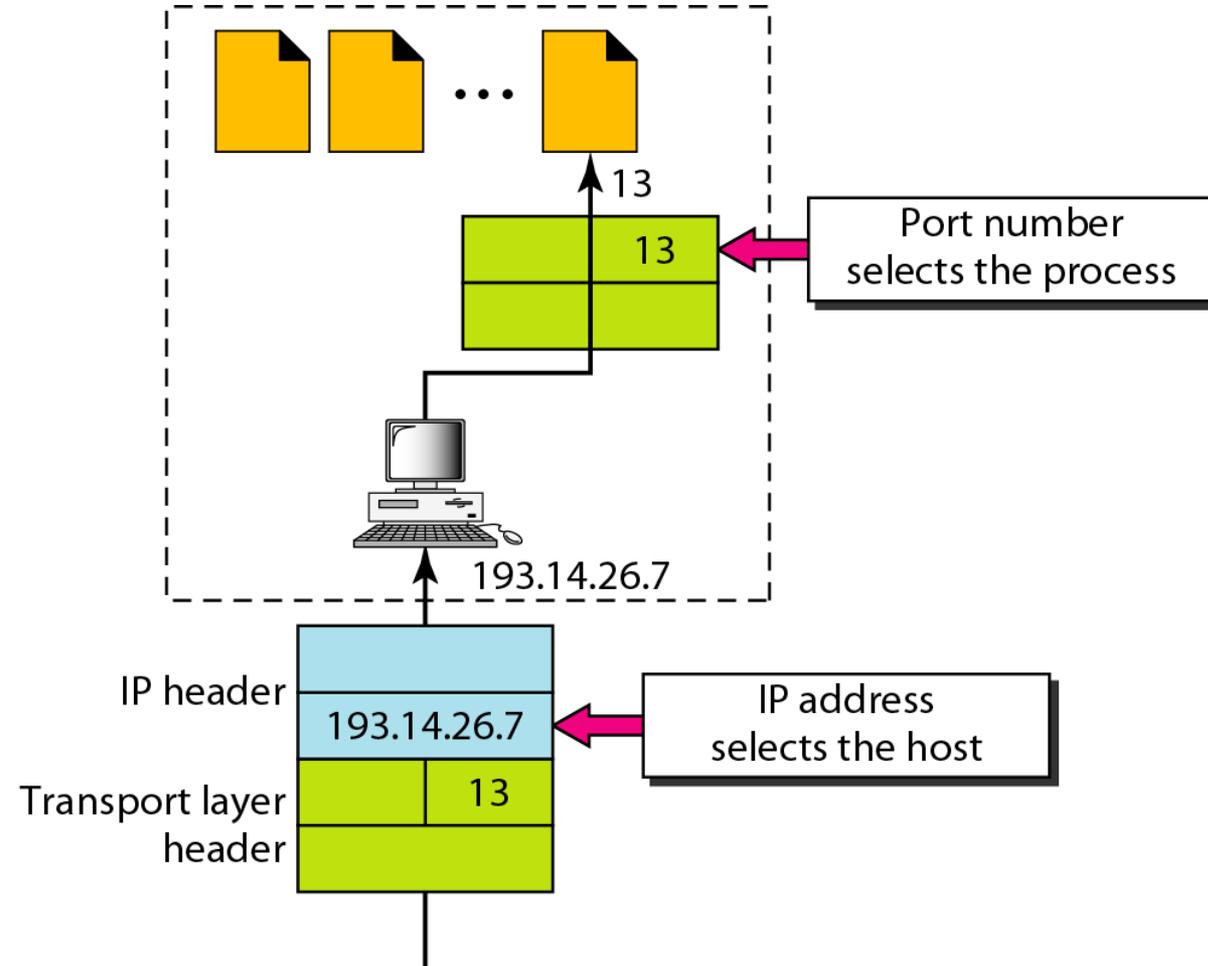


Process Server





Time of day TCP/IP





Setup della 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
 - Per evitare questo dovrei ogni volta buttare via l'indirizzo usato ma non funzionerebbe mai il nostro primo esempio del "time of day"



Evitare le duplicazioni



- Allora potrei numerare le 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 subnet. 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 ed ogni hop lo decremento. Quando arrivo a zero butto 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



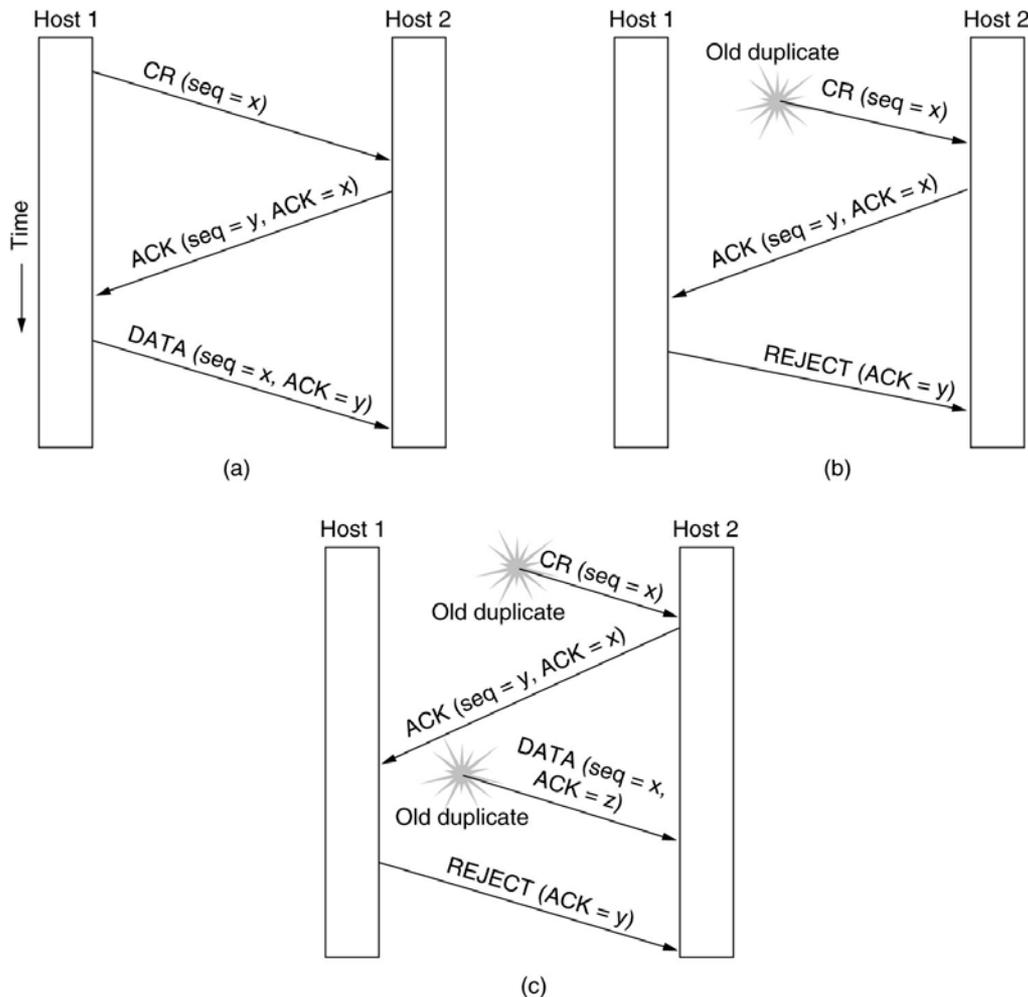
Three-way handshake



- 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 in figura (a).
 - 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



3 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



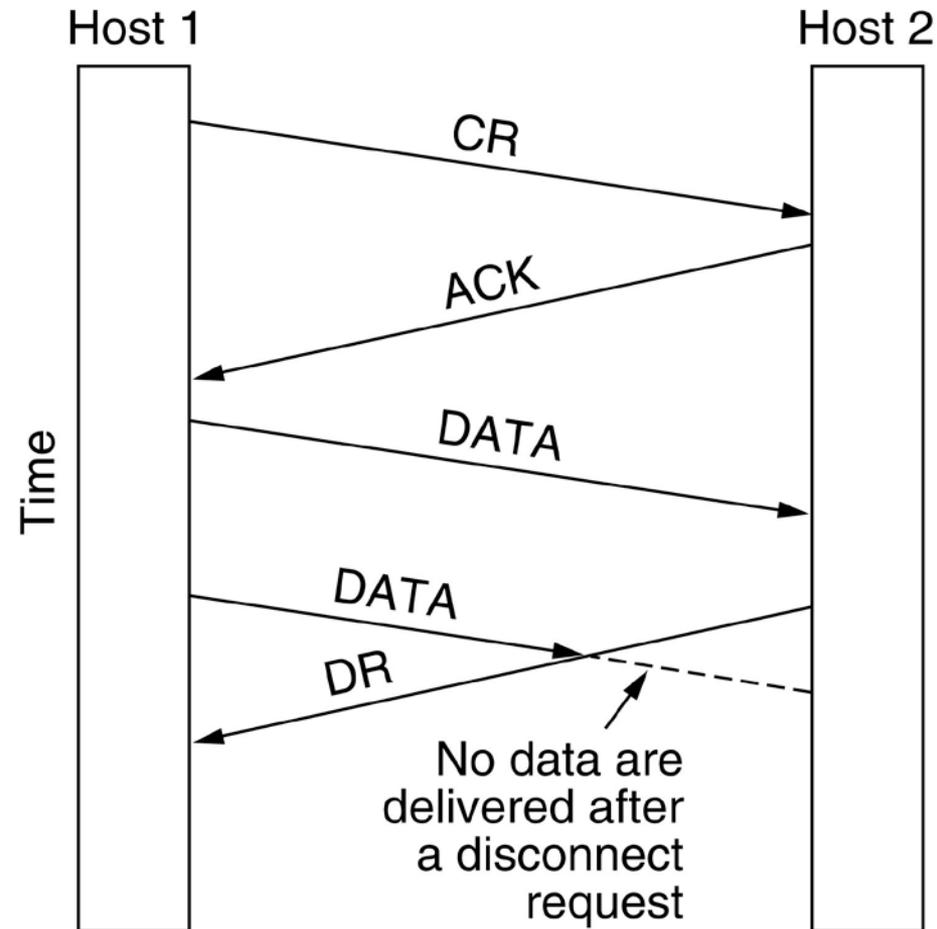
- Chiudere una connessione è più facile che aprirla ma ci sono comunque dei tranelli
- Chiusura asimmetrica: come al telefono, uno dei due attacca e la connessione è rotta
- Chiusura simmetrica: la connessione viene considerata come due connessioni unidirezionali separate ognuna delle quali deve essere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



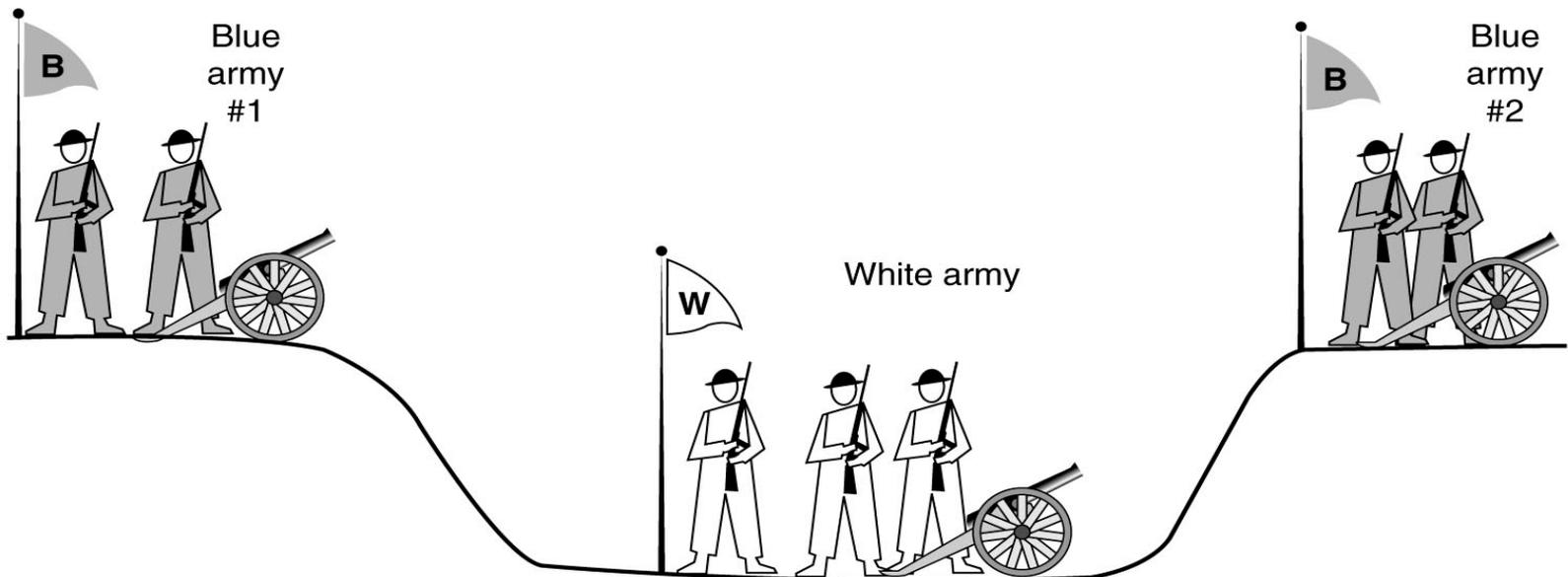
- La connessione viene chiusa in modo indipendente per cui un host rimane in ascolta anche dopo aver mandato la DR TPDU
- Quindi un host dice “ho finito, tu hai finito?” e l’altro risponde “ho finito anche io” allora la connessione si può chiudere
- Ma questo semplice protocollo non sempre funziona.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



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?

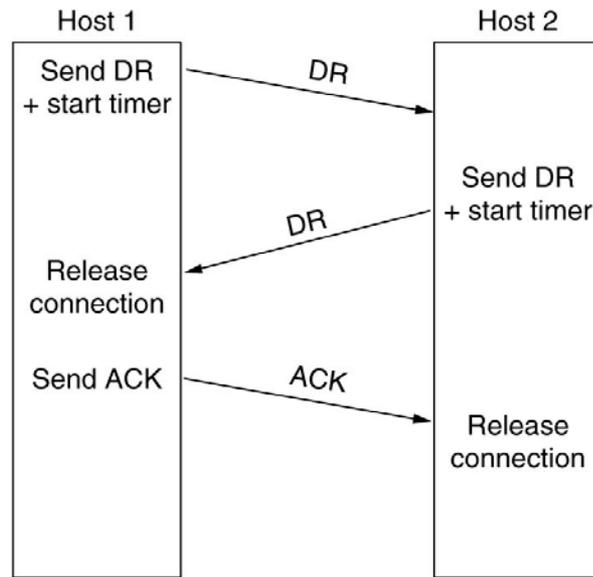


- Ogni ACK che viene mandato da uno dei due eserciti ha bisogno di un ACK per avere la certezza che sia arrivato e dare il via all'attacco
- Sostituendo “**disconnect**” ad “**attacco**” abbiamo lo stesso problema. Se nessuno dei due si disconnette fino a quando non è convinto che l'altro si sia pronto 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

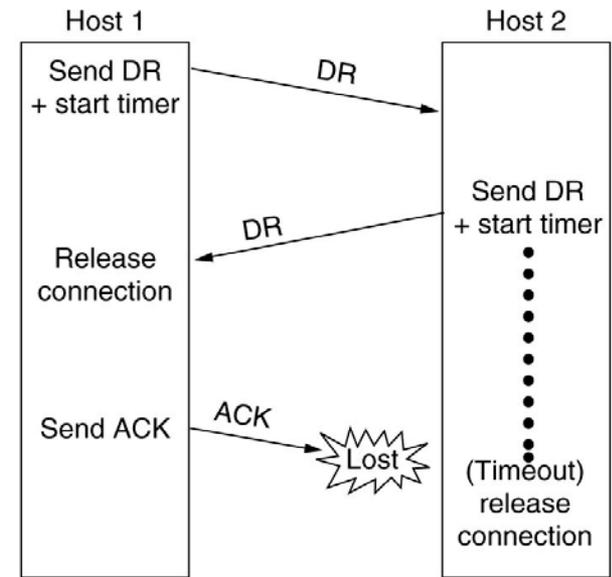


Casi

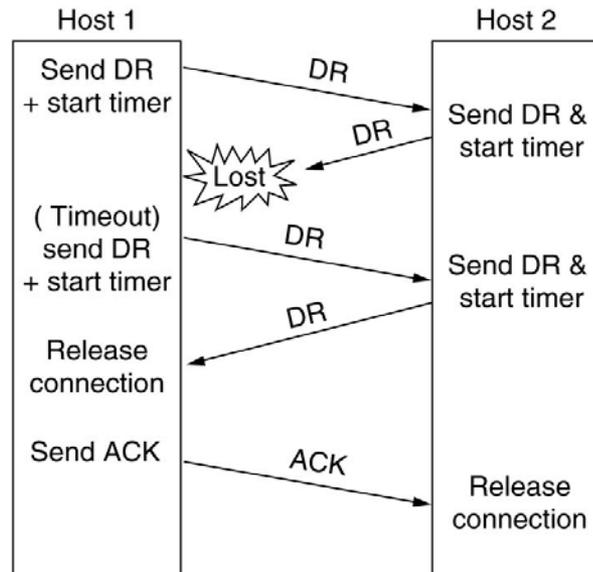
- Questo copre la maggior parte dei casi
- In teoria fallisce se il DR iniziale e tutte le N ritrasmissioni vengono perse
- Alle fine il mittente rinuncia mentre l'altro lato non vede nulla e rimane completamente attivo
- In pratica una connessione mezza aperta



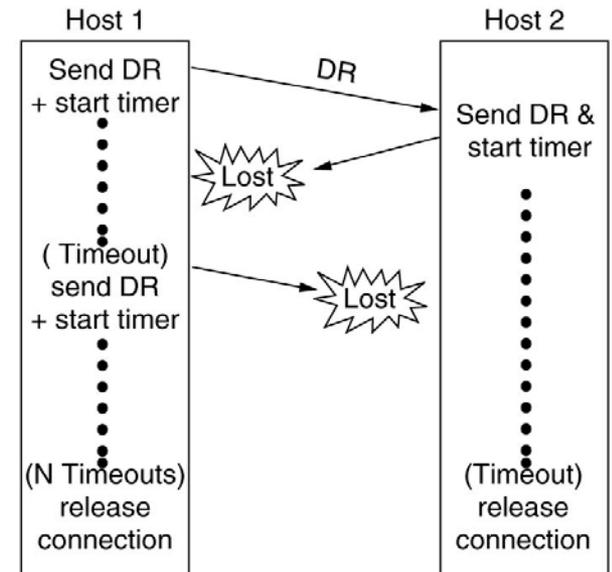
(a)



(b)



(c)



(d)



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



Flow control



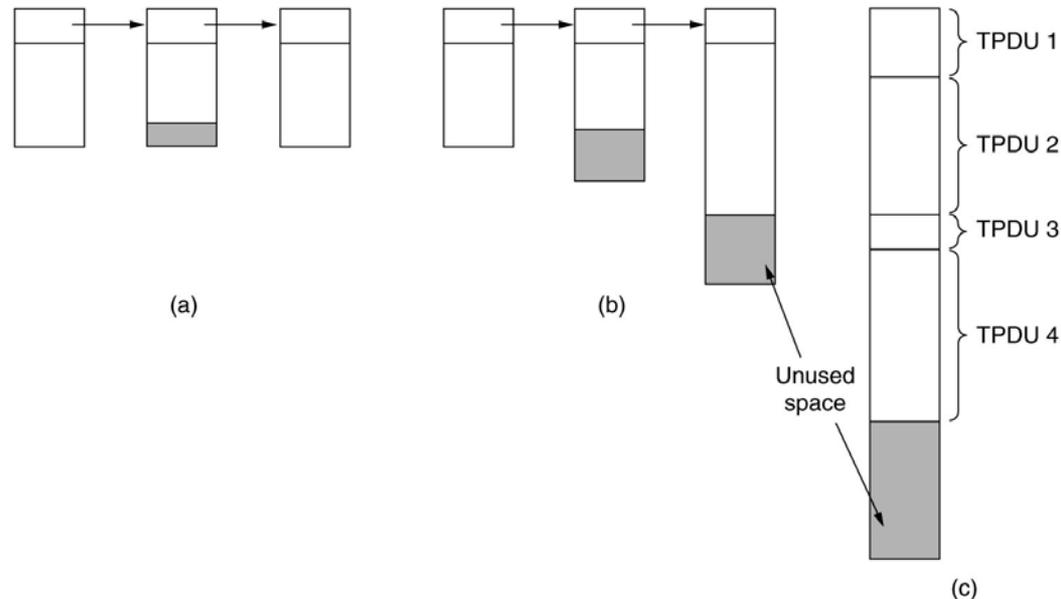
- Il problema del flow control è simile a quanto visto a livello data link
 - Per cui entrambi possono usare algoritmi tipo sliding windows o altri schemi per ogni connessione
- La differenza principale è che un router ha un numero limitato di linee
 - per cui può dedicare un numero di buffer pari alla lunghezza della finestra per il numero di link
 - L'host invece ha un numero enorme di possibili connessioni per cui conviene dedicare un pool di buffer condiviso da tutte le connessioni



Tipi di buffer



- Buffer di dimensioni fissa vanno bene se tutte le TPDU hanno dimensioni uguali.
- Altrimenti devo usare buffer di dimensioni variabili che sono però più difficile da gestire
- O anche un grande buffer circolare





Source o Destination?



- Conviene che sia il mittente o il ricevente a bufferizzare?
 - Se il link a livello network non è affidabile meglio bufferizzare tutte le TPDU che mando fino a quando non vengono ACKed
 - Se invece ho un livello network molto affidabile e il sender sa che ci sarà sempre spazio di buffer disponibile posso non bufferizzare alla sorgente e farlo solo a destinazione



Quale è meglio?



- Link lenti con traffico burst
 - come un terminale interattivo: conviene non dedicare alcun buffer ma allocarli dinamicamente ad entrambi i lati. Il mittente tiene una copia delle TPDU fino a quando non viene ACKed
- File transfer su link veloci
 - Meglio se il ricevente dedica una intera finestra per andare subito alla massima velocità



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.
 - Questa situazione si chiama upward multiplexing
 - Fig(a) quattro diverse connessioni di trasporto usano la stessa connessione di rete (indirizzo IP)



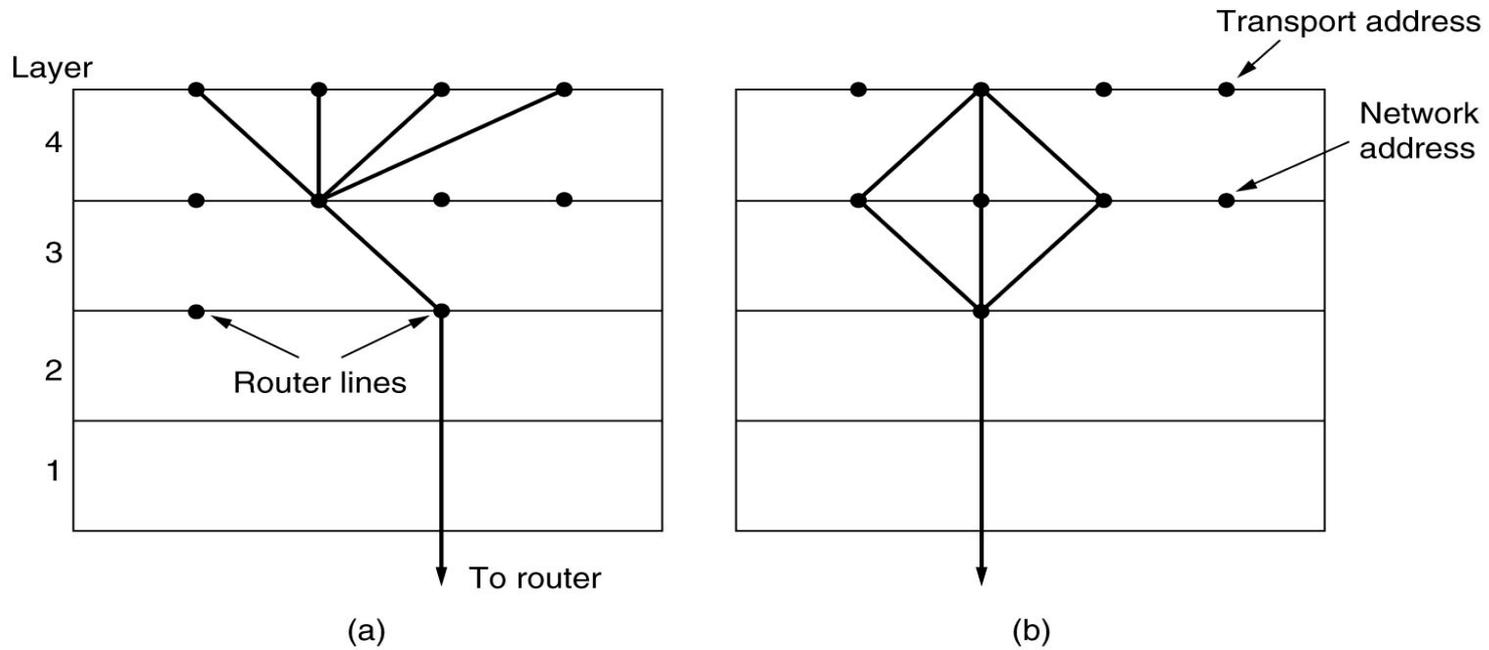
Downward multiplexing



- 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 aumento 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) upward multiplexing (b) downward multiplexing