



Reti Laboratorio – Appunti Dominick Ferraro

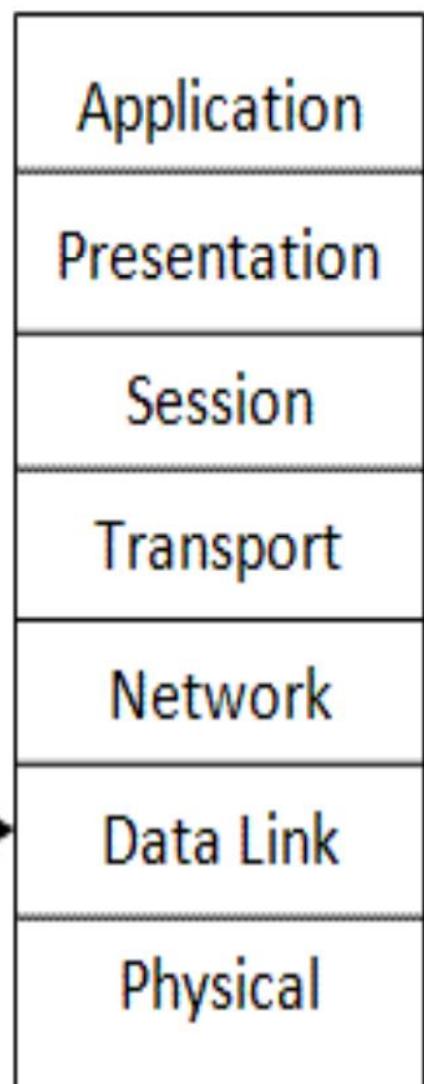
Anno 2021-2022

Applicazioni di rete: sono processi in esecuzione su macchine differenti, operano in modo indipendente scambiando info attraverso una rete, non utilizzano spazi di indirizzamento condivisi all'intero del SO o dei processi ma usano un sotto sistema di comunicazione, cioè quella parte del SO che consente ai processi di accedere alla rete.

I processi non hanno nulla in condivisione, devono accedere al sottosistema di rete del SO per scambiarsi informazioni.

Affinchè ci sia uno scambio di info, è necessario che i processi “parlino” la stessa lingua, che aderiscono tutti a uno stesso insieme di regole detto PROTOCOLLO.

Il protocollo più usato è quello ISO/OSI che implementa 7 livelli con delle regole, l'info parte dal livello applicazione, attraversa tutti i livelli, attraversa la rete e arriva dall'altra parte risalendo al contrario, dall'altro lato avremo esattamente gli stessi livelli, ciò significa che applicazione 1 e 2 parlano la stessa lingua. Se tutti i livelli sono andati a buon fine allora il programma 2 arriverà all'applicazione.

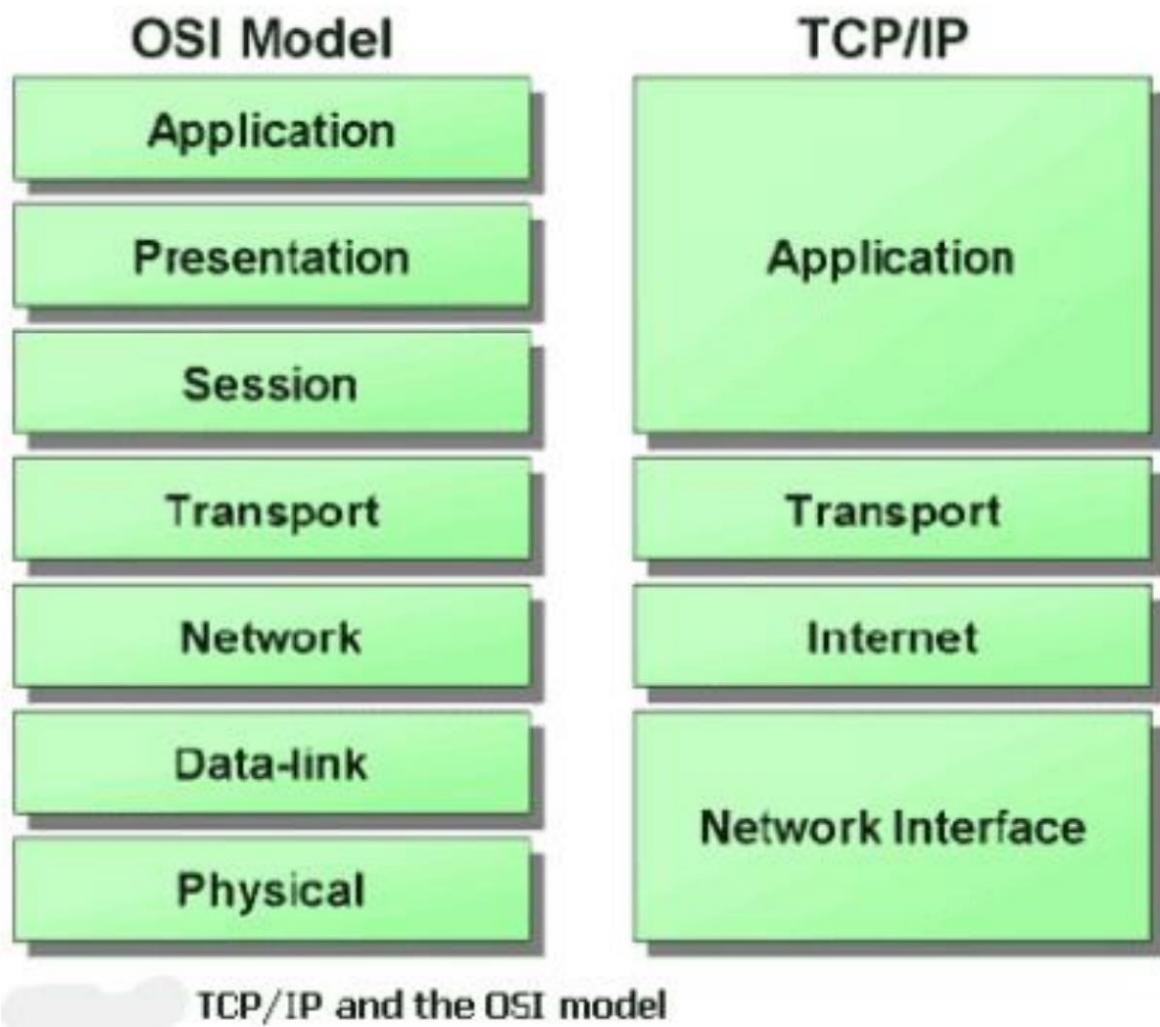


Application-1

Application-2

Passa per
la rete

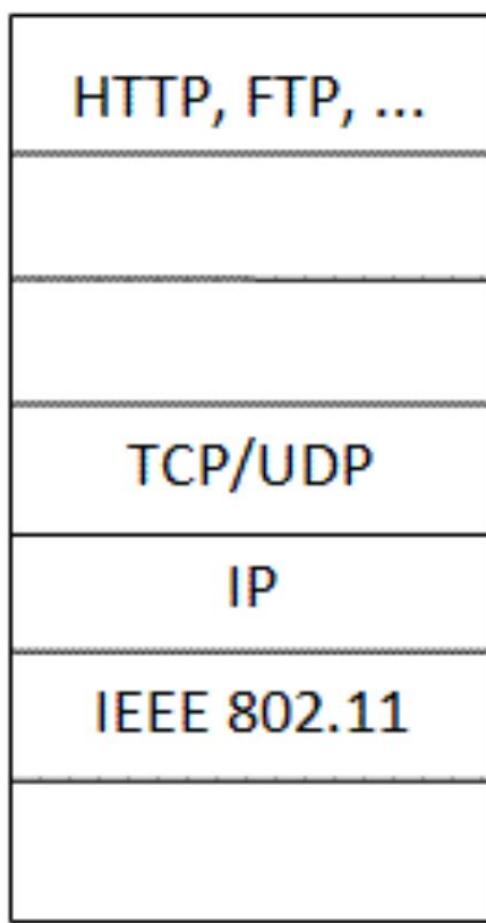
Il modello che viene implementato però è quello TCP/IP



TCP/IP recepisce tutti i livelli ISO/OSI ma li accorpa in campi diversi. Tutte le regole previste dal modello per sessione, applicazione e presentazione devono essere implementate nel processo applicazione. I livelli del TCP/IP sono quindi 4.

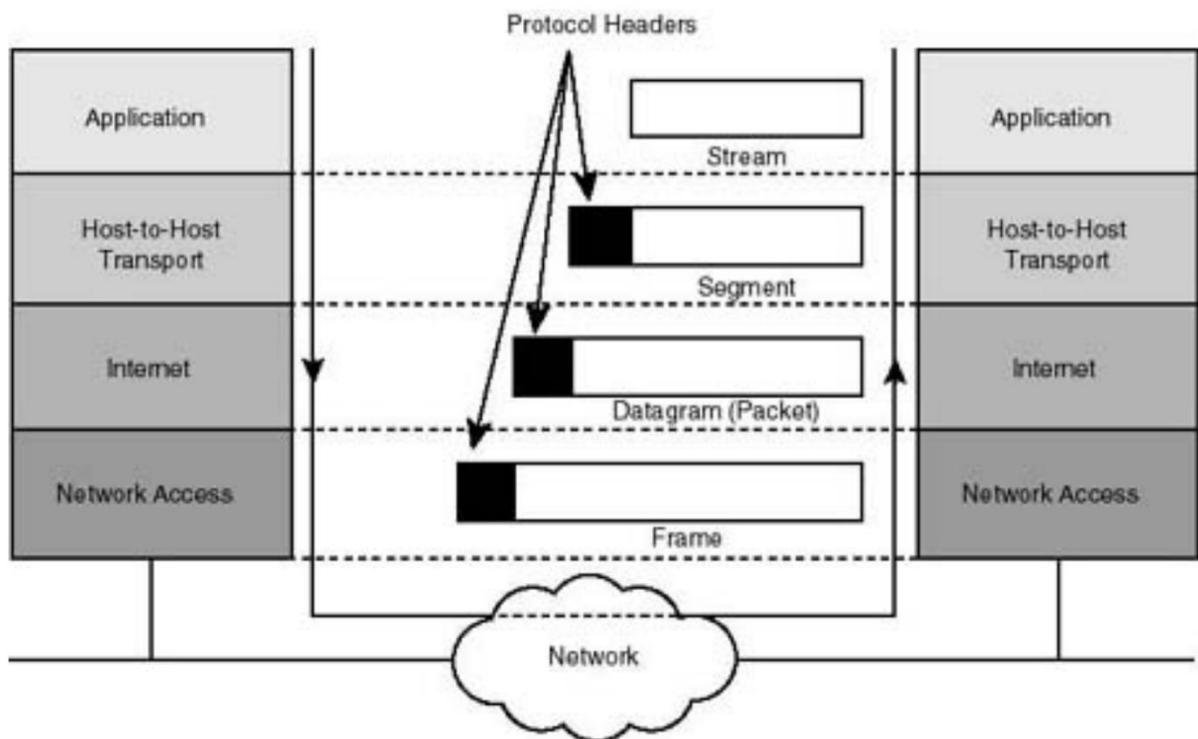


ISO/OSI 7-layer
Networking Model



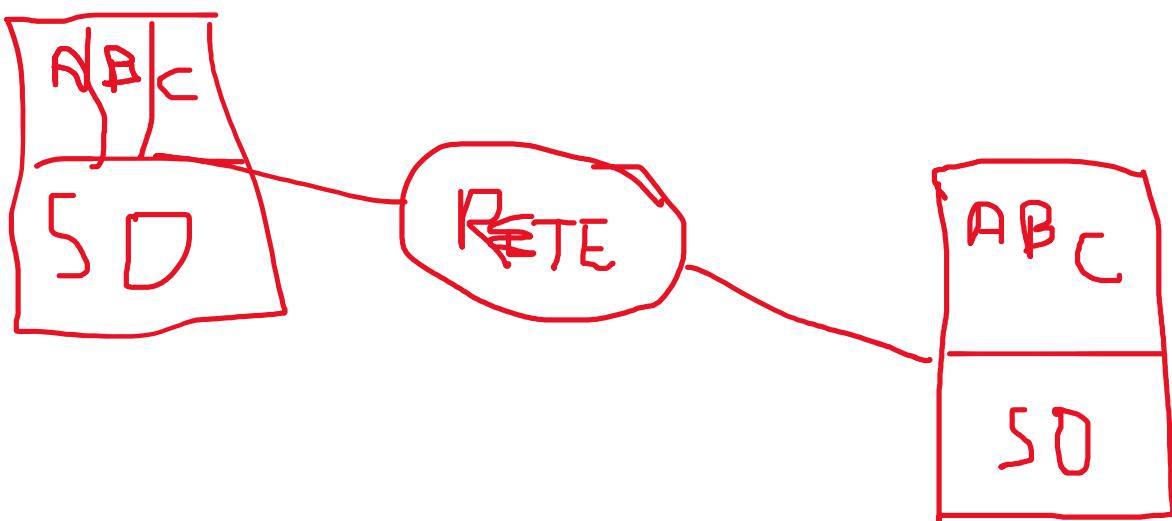
TCP/IP-based
Network

TCP/IP headers



Ogni volta che si passa di livello si aggiungono informazioni aggiuntive di ogni livello per stabilire se la sequenza di byte segue correttamente quello del livello corrente. Quando si arriva dall'altro lato si verifica se le regole e le info arrivate col messaggio sono corrette. Se è corretta si sale di livello e così via.

Supponiamo di avere più processi (host) con diverse applicazioni connesse a una stessa rete. Come comunica host1 con host2, sapendo che le applicazioni sono differenti? (Il processo A di h1 potrebbe essere un browser mentre il processo A di h2 potrebbe essere Whatsapp). Tutti i dati passano per il SO e confluiscono nella rete.



Identificazione Del Server:

- Ogni qual volta due endpoint vogliono comunicare devono identificarsi univocamente.
- Tale identificazione avviene attraverso due livelli di indirizzamento:

- Il primo identifica l'host su cui è in esecuzione il processo
 - Il secondo determina il processo con cui si vuole comunicare.
-
-

INDIRIZZO IP

Un Indirizzo IP è un numero che identifica univocamente un dispositivo collegato ad una rete informatica che utilizza lo standard IP (Internet Protocol).

Un esempio di indirizzo IPv4 è 192.168.1.1

Gli indirizzi IP sono costituiti da 32 bit (4 byte).

Questa rappresentazione limita lo spazio a circa 4,5 miliardi di indirizzi univoci possibili. Questo spazio è insufficiente e si è passato a IPv6.

IPv6: Indirizzi lunghi 128 bit, invece di 32.

L'indirizzo viene suddiviso in 8 blocchi di 16 bit ciascuno.

Nel “quadrato nero” di informazione aggiuntiva abbiamo tutto queste info:

4-bit	8-bit	16-bit	32-bit			
Ver.	Header Length	Type of Service	Total Length			
Identification		Flags		Offset		
Time To Live	Protocol		Checksum			
Source Address						
Destination Address						
Options and Padding						

Ci interessa il source address (mittente) e il destination address (destinatario).
Quindi se host1 manda un messaggio a h2 nel sorgente mette il suo IP mentre come destinazione metterà quello di h2. Quando h2 risponde a h1, come IP sorgente mette il proprio e come destinazione quello di h1.

L'indirizzo IP è associato alle macchine, non ai programmi.

Quindi dopo aver identificato l'host devo trovare il processo che deve ricevere il messaggio. Di

questo si occupa il livello di “trasporto”. Come identifica i vari processi?

Attraverso il concetto di “porta”. La porta è un intero a 16 bit senza segna che viene associato a ogni processo che richiede l’accesso a una rete.

- Da **0 a 1023**: **porte riservate** (ai processi di root)
- Da **1024 a 49151**: **porte registrate**
- Da **49152 a 65535**: **porte effimere** (per i client, ai quali non interessa scegliere una porta specifica)

Questo codice nel SO identifica univocamente un processo. Finchè è in uso, nessun altro processo può avere la stessa porta.

Lez 2

Gli indirizzi IP devono essere visti come delle stringhe = “192.168.1.1”.

Ogni host che si trova sulla rete ha un indirizzo IP. Un host potrebbe avere più interfacce di rete, dove ognuna avrà un suo indirizzo IP. Ad esempio possiamo avere su una macchina una porta ethernet con un determinato IP e un’altra interfaccia con la scheda Wi-Fi che avrà un altro indirizzo.

Per quello che abbiamo detto finora si immagina che ogni host sia identificato da un solo indirizzo IP.

Se siamo sulla stessa rete, poiché un indirizzo IP deve identificare univocamente una macchina, ogni host sulla stessa rete avrà un indirizzo IP **univoco differente**.

La porta è un intero a 16 bit e posso avere un numero di porta compreso tra 0 e 65535.

I protocolli di trasporto che analizziamo sono TCP e UDP che hanno 2^{16} porte ciascuna.

Tramite la porta le applicazioni possono identificarsi a vicenda.

Se h1 con porta 1 vuole inviare un messaggio a h2 con porta 5 succede questo:

- A livello application viene creato dal sistema un header col messaggio + le porte 1 e 5 e trasferite a livello inferiore
- A livello trasporto c'è un altro header con il messaggio + i due indirizzi IP (sorgente e destinazione)
- Tutto ciò viene trasportato poi a livello fisico

È necessario che l'host da cui parte il primo messaggio deve conoscere la porta e l'IP su cui si trova il destinatario della comunicazione.

Soltamente significa che c'è uno sbilanciamento di ruolo. Si dice che chi fa partire la richiesta di un servizio si chiama *client* (chi usufruisce del servizio) che deve conoscere necessariamente info del *server* che fornisce il servizio.

Supponiamo che un utente voglia accedere a una pagina web dell'ateneo. Deve conoscere l'IP (o nome del sito) del sito. Il browser assume che la pagina richiesta verrà fornita da un processo in esecuzione sulla porta 80. La nostra richiesta sarà: 192.168.0.2:80 (voglio l'IP sulla porta 80).

Nel processo client-server quindi il client deve già conoscere l'IP e la porta del server.

Lo stesso vale per l'app Teams: l'app sa che dovrà collegarsi al server Microsoft su una porta specifica.

Tutti i comandi (crea riunione ecc...) sono regole di livello applicazione, cioè avvengono dopo che è stato instaurato il canale tra client e server.

TCP E UDP

I due principali protocolli relativi a livello di trasporto sono TCP e UDP.

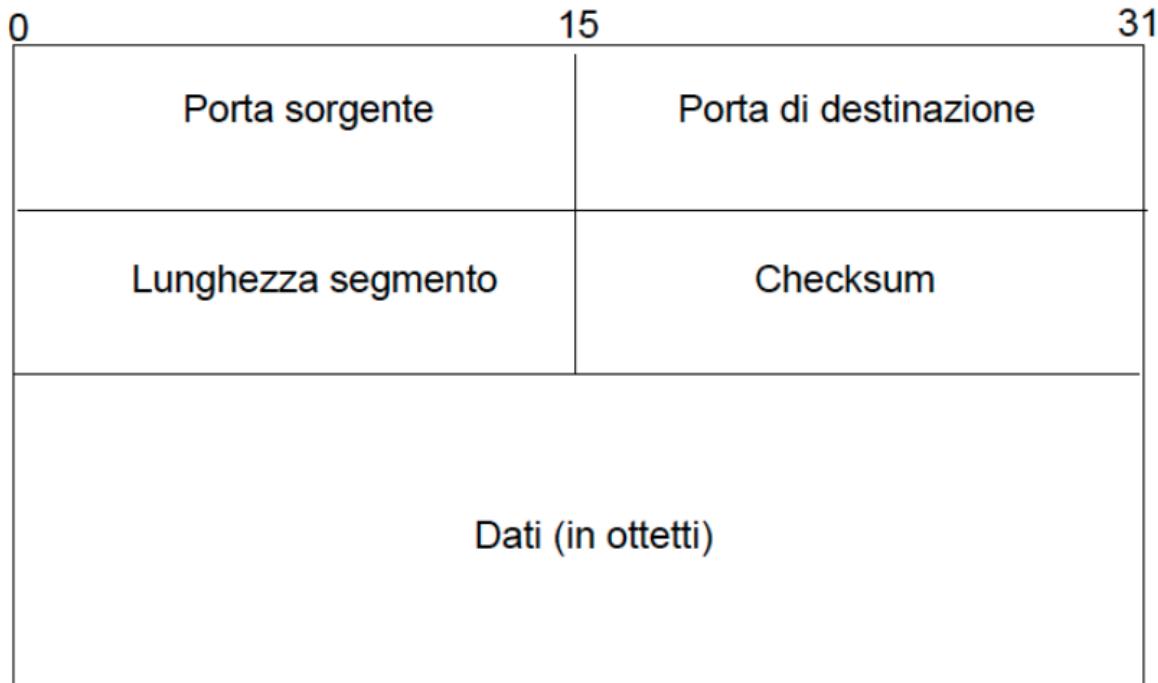
- In generale, il TCP è un servizio con connessione affidabile. Dati due host A e B, tra questi a livello Application viene creato un flusso bidirezionale continuo di dati.
(A →<– B).

- In UDP i messaggi vengono ricevuti e inviati uno alla volta.
 - Il TCP garantisce attraverso delle regole del protocollo che tutto ciò che si scrive nel canale, finchè è attivo, verrà portato dall'altra parte, esattamente nell'ordine di inserimento.
 - UDP non garantisce ciò. Ogni messaggio è indipendente dagli altri. Non c'è nessuna regola che garantisce il fatto che un messaggio venga recapitato a destinazione, potrebbe partire da A e non arrivare mai a B.
-

UDP

Rappresenta un'interfaccia ad IP per le applicazioni. Realizza un meccanismo di multiplexing e demultiplexing per l'invio e la ricezione dei datagram ai processi grazie al concetto di porta.

Datagramma UDP



TCP

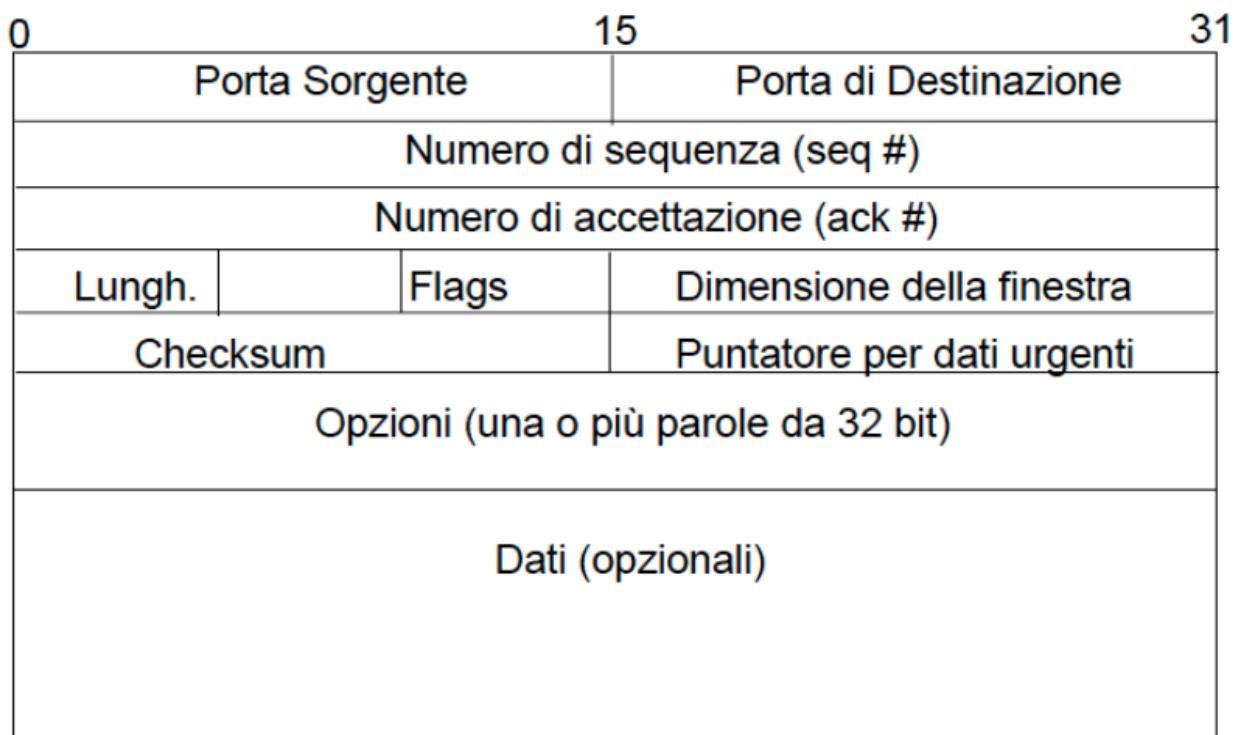
Garantisce diverse proprietà:

1. Trasferimenti dati a flusso (full duplex)
2. Affidabilità
3. Controllo del flusso (se A va in fibra e B sta con modem lentissimo, il resto della rete si perde. Quindi A deve rinviare ciò che si perde continuamente quello che perde. L'idea è appunto di controllare il flusso abbassando la velocità di trasmissione uniformandosi al più lento per risparmiare)

4. Trasferimenti full duplex

Per garantire l'affidabilità il TCP introduce l'ACK (acknowledge): per ogni byte inviato il destinatario deve confermare la ricezione del byte. A invia a B, quando B riceve il byte, B risponde con un ACK della ricezione di quel byte specifico.

Segmento TCP



Il campo flags è un campo di 8 bit dove 1 è attivato e 0 è off. Un flag attivato vuol dire che il

pacchetto analizzato è particolare. I flag definiscono un sotto tipo del TCP.

THREE WAY HANDSHAKE

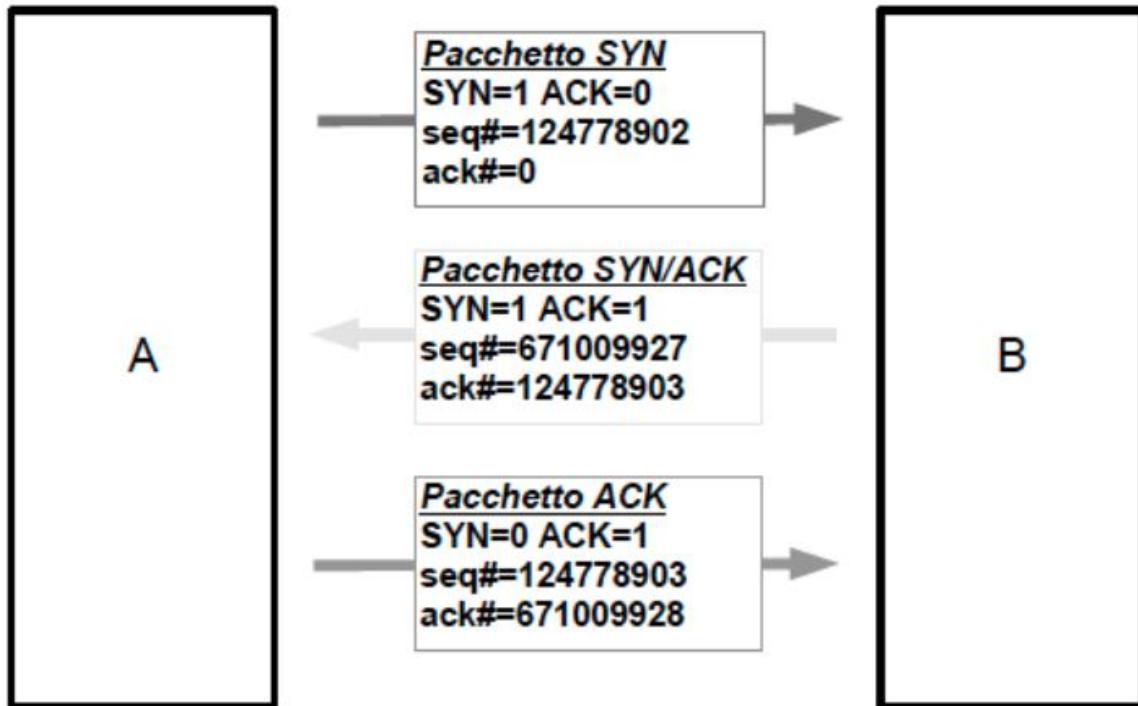
È una regola del protocollo TCP.

Quando un processo A sull'host C (**Client**) desidera aprire una connessione TCP con un processo B sull'host S (**Server**):

- C genera un numero di sequenza iniziale N e invia a S un segmento TCP in cui il flag SYN (SYNCRONIZE) è impostato su 1, il flag ACK a 0 e i campi seq# e ack# sono impostati rispettivamente a N e 0.
- S genera un numero di sequenza iniziale M e risponde con un segmento in cui i flag SYN e ACK sono 1, mentre seq#=M e ack#=N+1.
- C invia ad S un segmento in cui SYN=0, ACK=1, seq#=N+1 e ack#=M+1.

Dopo queste fasi la connessione è stabilita e i processi A e B possono scambiare dati.

Three way handshake



MODELLO CLIENT-SERVER

Nel modello client-server si distinguono due entità:

- I programmi che forniscono un servizio, server
- I programmi di uso, detti client che effettuano richieste

Un server di norma deve essere in grado di rispondere a più di un client.

Distinguiamo due classi di server:

- Concorrenti
- Iterativi

Seguono questo modello tutti i servizi fondamentali di internet:

- http, ftp, telnet, ssh, ecc...
-

IDENTIFICARE UNA COMUNICAZIONE

- TCP e UDP usano 4 info per identificare una comunicazione:
 - Indirizzo IP del server
 - Numero di porta del servizio lato server
 - Indirizzo IP del client
 - Numero di porta del servizio lato client

Lez 3

Configurazione delle interfacce di rete: *nix

```
root@bt:~# ifconfig -a
eth0      Link encap:Ethernet HWaddr 00:0c:29:bb:2f:08
          inet addr:10.5.1.13 Bcast:10.5.1.255 Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:febb:2f08/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:396842 errors:1 dropped:0 overruns:0 frame:0
          TX packets:27280 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:259527380 (259.5 MB) TX bytes:1782935 (1.7 MB)
          Interrupt:19 Base address:0x2000

lo       Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:16436 Metric:1
          RX packets:71771 errors:0 dropped:0 overruns:0 frame:0
          TX packets:71771 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:9693762 (9.6 MB) TX bytes:9693762 (9.6 MB)

root@bt:~# ■
```

ifconfig <interfaccia> mostra le informazioni relative ad un' interfaccia specifica

Quando parliamo di interfaccia parliamo di schede di rete (Wi-Fi, Ethernet).
Un host può avere più interfacce di rete e ognuna va configurata.

Il comando usato in ambiente Unix è *ifconfig*: fa vedere tutte le interfacce attualmente configurate sulla macchina.

Nell'esempio in slide abbiamo l'interfaccia *eth0* con info relative ad essa. È un'interfaccia Ethernet.

- *HWaddr* è l'indirizzo HW (MAC Address della scheda) che identifica univocamente ogni interfaccia di rete, una parte identifica il produttore e un'altra la singola scheda univoca.
- *Inet Addr* è l'indirizzo di rete IPv4.
- *BCast* è l'indirizzo di BroadCast che è un IP
- *Mask* è la maschera di rete che usa la sintassi degli indirizzi IPv4 e serve a capire se l'host con cui vogliamo comunicare si trova sulla stessa rete nostra o su un'altra.
- *Inet6 addr:* è l'IPv6

Abbiamo poi **un'altra** interfaccia *lo* che è virtuale. Si chiama *loopback* : crea un loop a rientrare: i pacchetti escono da questa interfaccia e rientrano nella stessa, non c'è una rete fisica ma è virtuale (SW) creata dal SO per poter testare in locale (senza avere una rete fisica) delle applicazioni che richiedono un utilizzo della rete. La “*lo*” ha un

indirizzo particolare: 127.0.0.1 che ci permette di avere un client e un server con la stessa macchina con quell'indirizzo. Manca l'HWAddr perché è virtuale.

CONFIGURARE INTERFACCIA

Se voglio configurare un'altra interfaccia uso sempre il comando *ifconfig* però dobbiamo fornire i parametri per configurarla.

- Assegnare indirizzo IP, netmask e indirizzo di broadcast

ifconfig <interface> <address> netmask <mask> broadcast <broadcast-address>

- **<interface>** è l' interfaccia da configurare (es. eth0, eth1, ...)
- **<address>** è l' indirizzo IP da assegnare (es. 192.168.8.27)
- **<mask>** è la netmask da associare all' IP (es. 255.255.255.0)
- **<broadcast-address>** è l' indirizzo di broadcast della rete (es. 192.168.8.255)

- Attivare un' interfaccia



ifconfig <interface> up

- Disattivare un' interfaccia

ifconfig <interface> down

Solitamente le interfacce Ethernet prendono il nome di *eth:numero*.

Es: *ifconfig ETH1 192.168.0.1 netmask 255.255.255.0 broadcast 192.168.0.255*

- Per attivare il comando di rete o aggiungo *up* alla fine (*ifconfig ETH1 192.168.0.1 netmask 255.255.255.0 broadcast 192.168.0.255 up*) oppure scrivo “*ifconfig ETH up*”.
-

NETMASK

Internet è un insieme di sottoreti più piccole. Immaginiamo di avere 3 reti che sono collegate tra loro, ognuna di esse ha al suo interno un certo numero di host. Immaginiamo che l'host A della rete 1 voglia scambiare messaggi con l'host B della rete 3. Se il destinatario è un host che si trova sulla stessa rete allora la comunicazione è diretta, viceversa, prima dobbiamo trovare la “strada” per trovare la rete e poi recapitare il messaggio con IP e porta.

Quindi prima di inviare un pacchetto dobbiamo capire se il destinatario è sulla stessa rete o in un'altra rete.

Abbiamo l'IP 192.168.0.1 della nostra macchina. L'indirizzo di rete è 255.255.255.0.

L'IP è composto di due parti: una è la parte network, cioè la parte che definisce la rete a cui appartiene e l'altra è la parte host che lo identifica.

Immaginiamo di voler inviare un pacchetto a 192.168.0.2.

Si fa un AND BIT-a-BIT (L'operazione bit a bit AND, indicata con &, esegue un confronto tra due variabili dando come risultato una terza variabile che presenta un 1 in quelle posizioni in cui entrambe le variabili di partenza presentano 1 e uno 0 in tutte le altre.)

tra la sequenza di bit dell'indirizzo e della netmask: facendo l'AND bit a bit con i tre campi che sono 255 si estrae 192.168.0 cioè la parte network della nostra macchina, il terzo ottetto è 0, se lo metto in AND con qualunque numero fa 0 e avremo 192.168.0.0 → questo è l'identificativo della rete.

Facciamo un AND bit a bit con la rete destinataria (192.168.0.2) e vediamo che siamo sulla stessa rete.

Vediamo ora un altro caso dove il destinatario è 192.168.1.2 e lo metto in AND con 255.255.255.0 e avremo 192.168.1.0: ho scoperto che il destinatario non si trova sulla stessa rete!

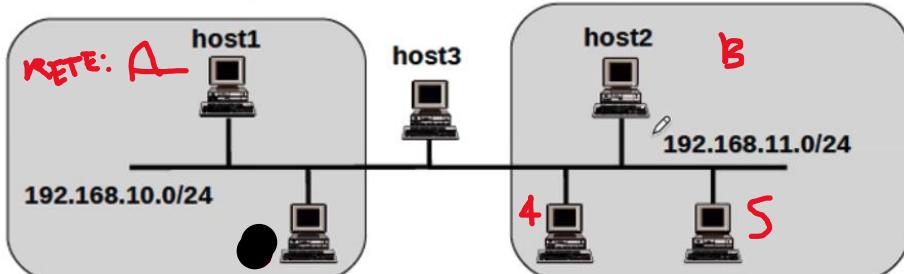
Quindi se non siamo sulla stessa rete si invia il pacchetto a un host particolare della rete chiamato *Gateway* che avrà un indirizzo IP, solitamente è il primo indirizzo libero sulla rete es: 192.168.0.1.

Ogni volta che un host si rende conto che il destinatario del pacchetto non è sulla rete lo invia al *gateway* che sa come gestirlo.

Su un'interfaccia fisica possiamo configurare più interfacce virtuali.

Si impostano gli alias come se fossero interfacce fittizie.

Eth0 può avere gli alias eth0:0, eth0:1, eth0:2 ...



Vogliamo creare una rete con h1 e h3 e una rete con h2, h3, h5.

Se h3 vogliamo farlo comunicare con entrambi le reti A e B devo avere un'interfaccia che è collegata su una e sull'altra: questo con una rete fisica è impossibile: o immagino di avere due interfacce fisiche oppure Unix permette di configurare l'interfaccia fisica su una rete con una virtuale.

```
[root@host3 root]# ifconfig eth0 192.168.10.3 netmask 255.255.255.0  
broadcast 192.168.10.255  
  
[root@host3 root]# ifconfig eth0:0 192.168.11.3 netmask 255.255.255.0  
broadcast 192.168.11.255
```

Host 3 ha quindi due interfacce collegate con A e B e sarà in grado di scambiare pacchetti con tutti gli host delle reti.

Host3 prende il nome di **bridge**: fanno da ponte tra due reti diverse e possono dialogare con entrambe.

La cosa interessante è che il cavo fisico è uno solo, anche se le reti sono due.

Le interfacce di rete si riconoscono o per il loro nome o per il loro MAC Address che è uguale.

Il comando *ip* è la nuova versione del comando *ifconfig*

```
ifconfig  
ip addr show  
ip link show
```

```
ifconfig eth0 192.168.0.77 netmask 255.255.255.0 broadcast 192.168.0.255
```

```
ip addr add 192.168.0.77/24 broadcast 192.168.0.255 dev eth0
```

```
ifconfig eth0:1 10.0.0.1/8
```

```
ip addr add 10.0.0.1/8 dev eth0 label eth0:1
```

CONFIGURAZIONE INTERFACCE WIFI

- Configura i parametri per le interfacce Wi-Fi (IEEE 802.11)
- iwconfig <interface> [options]
- <interface> è l'interfaccia da configurare (es. eth1, ath0, wlan0, ...)
- **essid X**: sceglie X come SSID della rete wireless a cui connettersi
- **mode M**: imposta la modalità di funzionamento
M = Ad-Hoc, Managed, Monitor, Repeater, Master, Auto
- **channel C**: imposta il canale (o la frequenza) su cui operare
- **ap A**: specifica l'indirizzo MAC dell'access point a cui agganciarsi

INTERNET CONTROL MESSAGE PROTOCOL

PROTOCOLLO ICMP

Tramite il comando ping pc1 invia pacchetti a pc2 che per conferma invia altri pacchetti a pc1.

```
pc1:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=2.65 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.357 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.380 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.349 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.348 ms

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4078ms
rtt min/avg/max/mdev = 0.348/0.818/2.656/0.919 ms
pc1:~# █
```

- pc1 and pc2 can reach each other

Tramite TCPDUMP vediamo a video i pacchetti.
Su pc1 facciamo ping mentre su pc2 facciamo
tcpdump <> nome_rete >

pc2

```
pc2:~# tcpdump -i eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96
bytes
19:27:17.899782 arp who-has 10.0.0.2 tell 10.0.0.1
19:27:18.002578 arp reply 10.0.0.2 is-at fe:fd:0a:00:00:02
19:27:18.004384 IP 10.0.0.1 > 10.0.0.2: icmp 64: echo request seq 1
19:27:18.005806 IP 10.0.0.2 > 10.0.0.1: icmp 64: echo reply seq 1
19:27:18.920463 IP 10.0.0.1 > 10.0.0.2: icmp 64: echo request seq 2
19:27:18.920605 IP 10.0.0.2 > 10.0.0.1: icmp 64: echo reply seq 2

6 packets captured
6 packets received by filter
0 packets dropped by kernel
```

ICMP → INTERNET CONTROL MESSAGE

PROTOCOL

È un protocollo del livello network per la segnalazione di eventi relativi allo stato della rete.

Quando un host o un router devono informare la sorgente di un datagram su eventi relativi al trasferimento del datagram, utilizzano ICMP.

ICMP usa IP come se fosse un protocollo di livello di trasporto: un messaggio ICMP viene incapsulato in un datagram IP. Ovviamente il pacchetto IP non cambia il suo ruolo.

Un messaggio ICMP è individuato attraverso un codice numerico, composto da un tipo e da un eventuale sottotipo.

TABELLA CODICI ICMP

Tipo	Nome	Utilizzo
0	Echo reply	Risposta da un host attivo sulla rete
3	Destination unreachable	Diagnostica sui problemi inerenti la trasmissione di un datagram; prevede diversi sottotipi
4	Source quench	Segnala che un router intermedio non ha spazio per mettere in coda il datagram
5	Redirect	Emesso da un router intermedio, segnala il router cui inviare i datagram seguenti
8	Echo request	Verifica della presenza di un host sulla rete
11	Time exceeded	Segnala che il campo TTL del datagram è esaurito senza che sia stata raggiunta la destinazione
12	Parameter problem	Indica che si è verificato un problema nell'elaborazione dell'header IP
13	Timestamp request	Utilizzato per il debugging e la misura di prestazioni), richiede un timestamp
14	Timestamp reply	Il messaggio di risposta ad una richiesta di timestamp
...

INTERROGAZIONI ICMP

Comandi come ping e tracert usano il protocollo ICMP per determinare rispettivamente:

- Se un host è sulla rete
- Qual è l'instradamento seguito per un host

Ping usa ***echo request*** ed ***echo reply***, calcolando il tempo di risposta.

Tracerout invia datagrammi IP con bassi valori TTL (*time to live*), in modo che essi siano scartati dai router lungo la destinazione, ricevendone il messaggio time exceeded.

Ogni volta che un pacchetto passa da un host al successivo (hop) il TTL viene decrementato di 1. Quando il TTL arriva a 0, il pacchetto viene scartato e viene generato un ICMP di tipo time exceeded perché vogliamo comunicare al mittente che la destinazione non è stata raggiunta.

L'ICMP time exceeded va a finire in un pacchetto IP che al suo interno avrà IP sorgente e IP destinazione. L'IP sorgente sarà quello che ha messo a 0 il TTL mentre IP destinazione sarà il primo pacchetto. Quando il sorgente riceve il pacchetto ricava due info: il pacchetto non è

arrivato a destinazione e ha l'IP dell'ultimo salto che ha fatto il pacchetto.

ROUTE

Il comando *route* modifica la tabella di routing.
Imposta l'indirizzo del gateway

Es.:

route add default gw 10.10.10.1

- E' possibile verificare la tabella di routing con il comando

netstat -rn oppure route

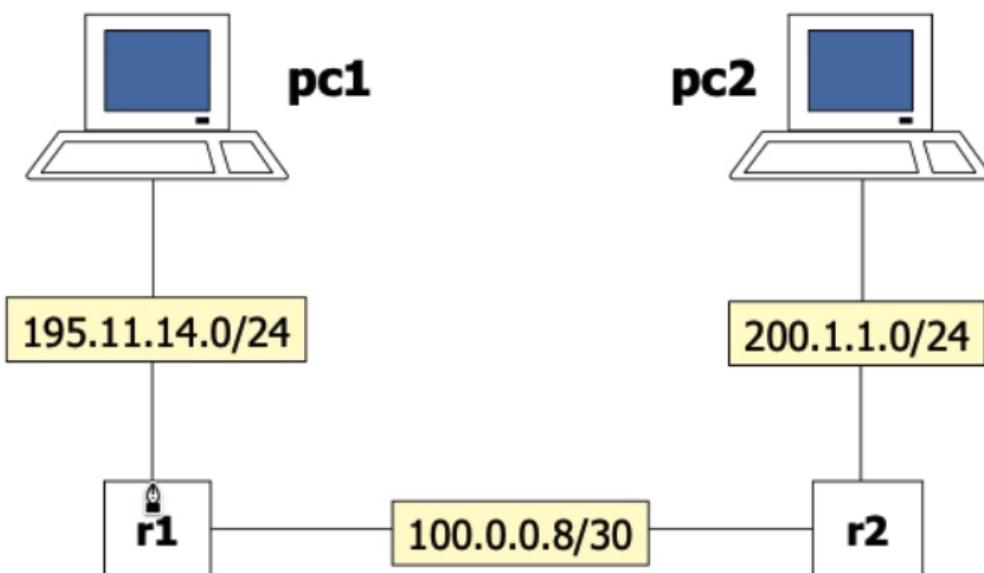
Lez 4

STATIC ROUTING

L'instradamento dei pacchetti verso la destinazione può essere fatto sia in maniera statica che dinamica.

Per capire cosa succede vediamo staticamente dove il routing è determinato a priori.

Ci sono 3 reti pc1 con r1, pc2 con r2 e r1 con r2. Capiamo che sono 3 reti diverse perché se mettiamo in AND le maschere di rete vediamo che sono diverse.



/24 è la maschera di rete.

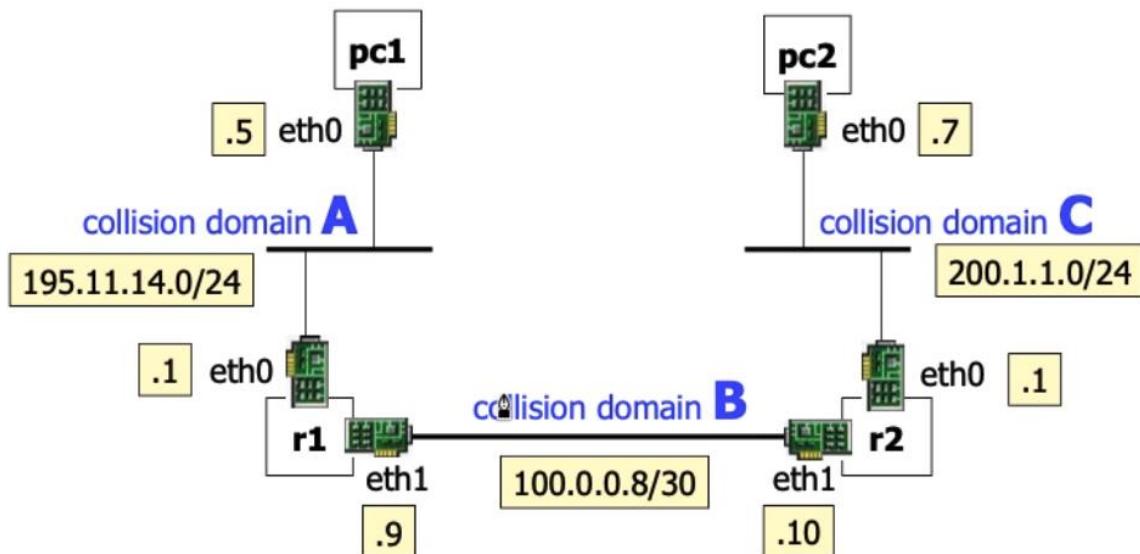
Per quanto riguarda pc1 e pc2 viene usato il comando *ifconfig*.

```
pc1.startup  
ifconfig eth0 195.11.14.5 netmask 255.255.255.0 broadcast 195.11.14.255 up  
#route add default gw 195.11.14.1 dev eth0  
  
pc2.startup  
ifconfig eth0 200.1.1.7 netmask 255.255.255.0 broadcast 200.1.1.255 up  
#route add default gw 200.1.1.1 dev eth0
```

the routing table entries will be added manually

Stessa cosa lo si fa per r1 e r2. Poiché hanno ognuno 2 interfacce dovrò lanciare 2 volte il comando *ifconfig* per ogni rete.

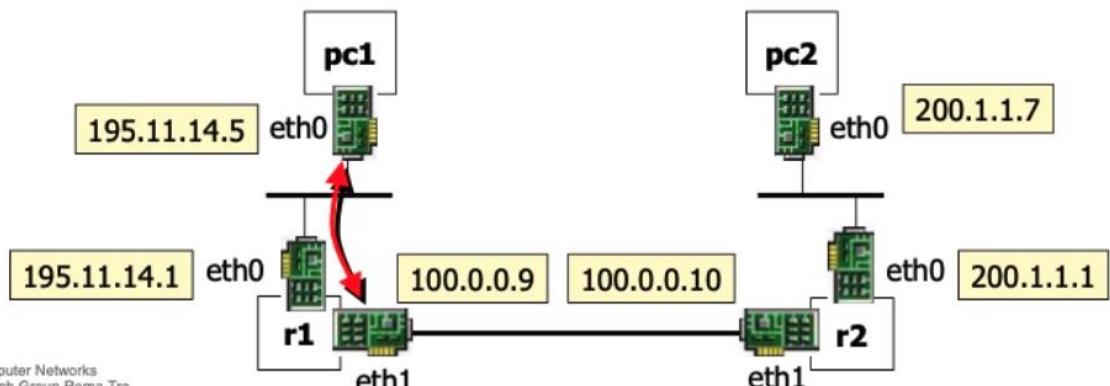
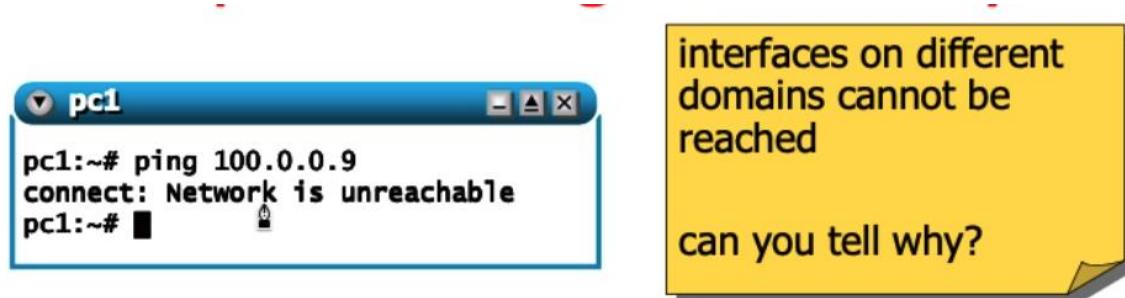
Dopo aver lanciato le macchine virtuale e i 6 comandi *ifconfig*, pc1 può scambiare pacchetti con la eth0 di r1. Pc2 vede la eth0 di r2 e r1 vede r2 (e viceversa) tramite la eth1. Queste interfacce possono comunicare solo localmente.



Il nostro obiettivo è far comunicare pc1 con pc2. Bisogna fare *ping* tra pc1 e pc2 e vedere se c'è connessione.

Con la configurazione di partenza facciamo un ping da pc1 a eth0 di r1. Ottengo risposte perché sono sulla stessa rete.

Dopo ping 100.0.0.9 cioè la eth1 di r1. La risposta del ping è negativa: la rete di destinazione non è raggiungibile poiché non avendo configurato un gateway di default poiché facendo l'AND bit a bit tra le maschere si accorge che siamo su reti differenti e devo inviare i pacchetti al gateway che non è configurato.



Se usiamo il comando *route* vediamo che non c'è un gateway configurato.

Lo configuriamo usando il seguente comando:

The image shows two terminal windows, each with a blue title bar labeled 'pc1' and 'pc2' respectively. Both windows contain terminal session logs. The 'pc1' window shows the command 'route add default gw 195.11.14.1' followed by the output of the 'route' command. The 'pc2' window shows the command 'route add default gw 200.1.1.1' followed by the output of the 'route' command. Both outputs show the kernel IP routing table with two entries: one for the local network (195.11.14.0/24 or 200.1.1.0/24) and one for the default gateway (195.11.14.1 or 200.1.1.1). The tables include columns for Destination, Gateway, Genmask, Flags, Metric, Ref, Use, and Iface.

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
195.11.14.0	*	255.255.255.0	U	0	0	0	eth0
default	195.11.14.1	0.0.0.0	UG	0	0	0	eth0

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
200.1.1.0	*	255.255.255.0	U	0	0	0	eth0
default	200.1.1.1	0.0.0.0	UG	0	0	0	eth0

Così diciamo che se i pack sono fuori dalla rete locali bisogna mandarli al gateway 195.11.14.1.

Lo stesso lo si fa per pc2 (200.1.1.1).

Il fatto che ci sia scritto *default* è dovuto alla possibilità di poter impostare all'interno di una stessa rete diversi gateway. Posso avere più gateway per destinazioni, se nessuna di queste è quella che sto cercando allora uso quello di *default*.

Abbiamo impostato i gateway per le due reti locali.

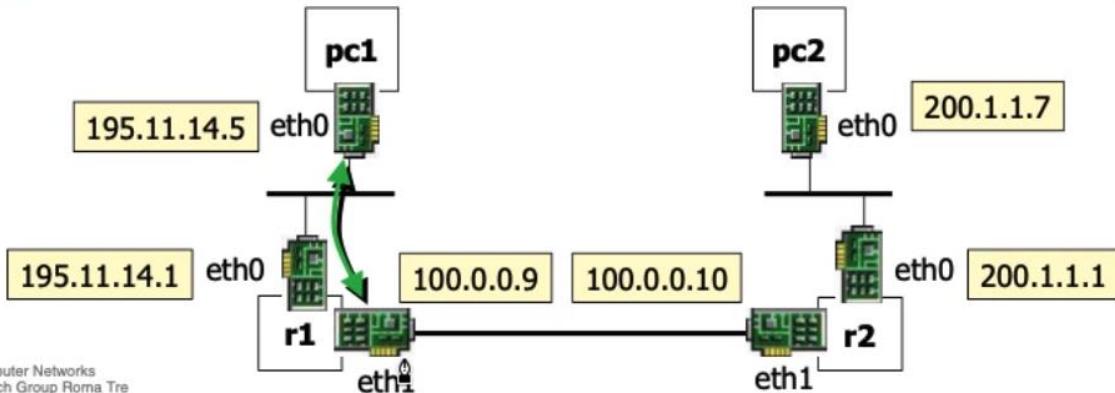
Proviamo a pingare e va tutto bene.

```

pc1:~# ping 100.0.0.9
PING 100.0.0.9 (100.0.0.9) 56(84) bytes of data.
64 bytes from 100.0.0.9: icmp_seq=1 ttl=64 time=0.451 ms
64 bytes from 100.0.0.9: icmp_seq=2 ttl=64 time=0.299 ms
64 bytes from 100.0.0.9: icmp_seq=3 ttl=64 time=0.320 ms
--- 100.0.0.9 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.299/0.356/0.451/0.070 ms
pc1:~# ■

```

the "backbone interface" of r1 is reachable



Il passo successivo è pingare 100.0.0.10
Cosa succede?

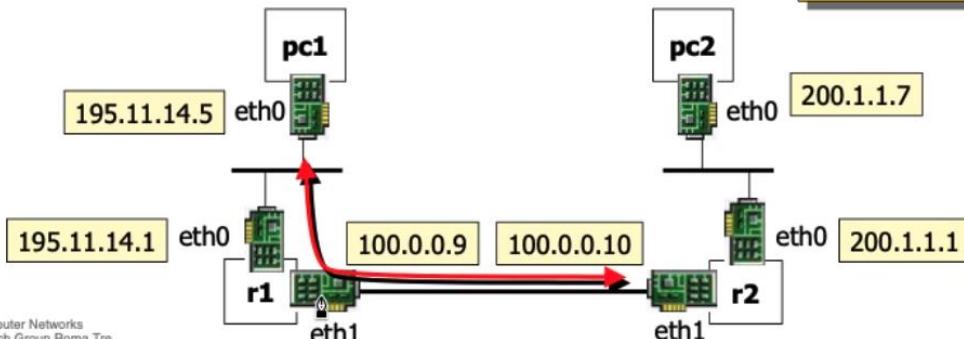
In questo caso, faccio il ping e non ottengo risposta.

```

pc1:~# ping 100.0.0.10
PING 100.0.0.10 (100.0.0.10) 56(84) bytes of data.
--- 100.0.0.10 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6105ms
pc1:~# ■

```

interfaces on r2 seem unreachable!
can you tell why?



I pacchetti sono sulla stessa rete ma non ci sono risposte perché? R2 dovrebbe inviare *l'echo reply* a 195.11.14.5: vede che la rete è diversa e ci accorgiamo che non abbiamo configurato il gateway per r2. Come facciamo? Usiamo *tcpdump* per vedere tutti i pacchetti che passano. Effettivamente gli *echo request* arrivano. Manca è il gateway di ritorno.

```
r2:~# tcpdump -i eth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
16:06:58.977851 arp who-has 100.0.0.10 tell 100.0.0.9
16:06:59.088906 arp reply 100.0.0.10 is-at fe:fd:64:00:00:0a
16:06:59.089990 IP 195.11.14.5 > 100.0.0.10: icmp 64: echo request seq 1
16:06:59.989368 IP 195.11.14.5 > 100.0.0.10: icmp 64: echo request seq 2
16:07:01.001888 IP 195.11.14.5 > 100.0.0.10: icmp 64: echo request seq 3

5 packets captured
5 packets received by filter
0 packets dropped by kernel
r2:~#
```

echo requests are arriving!

Usiamo una sintassi di route particolare: specifichiamo qual è la network di destinazione (195.11.14.0) <<se i pacchetti sono destinati alla rete 195.11.14.0 si usa il gateway 100.0.0.9>>.

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
100.0.0.8	*	255.255.255.252	U	0	0	0	eth1
200.1.1.0	*	255.255.255.0	U	0	0	0	eth0
195.11.14.0	100.0.0.9	255.255.255.0	UG	0	0	0	eth1

A questo punto possiamo pingare 200.1.1.7 (pc2) ottengo le risposte, e viceversa.

Una volta che ho configurato i gateway le strade sono definite e avviene la connessione tra i due pc.

IL LIVELLO APPLICAZIONE

I protocolli di livello applicazionisi collocano al di sopra dello stack TCP/IP.

Usano UDP o TCP come meccanismo di trasporto.

I protocolli che impiegano il livello di trasporto usano la nozione di porta per l'individuazione di un servizio.

Alcuni servizi usano una coppia di porte per la realizzazione della modalità duplex.

CLIENT SERVER

Un modello per implementare queste applicazioni è il modello CLIENT-SERVER.

Gli utenti interagiscono col client. Il server riceve una richiesta, esegue il sercizio richeisrtoe invia i risultati al client.

La comunicazione è effettuata mediante API.

PEER TO PEER

Il modello P2P è un paradigma di progettazione per le applicazioni distribuite dove le entità sono sullo stesso livello.

Ogni peer che partecipa allo stesso tempo fornisce un servizio e usa servizi offerti da altri peer (entità).

Nei sistemi P2P gli user accedono alle risorse in seguito a una fase di ricerca di un nodo già connesso.

SERVIZI DI RETE

I servizi di rete vengono attivati all'avvio del SO.

I servizi di reti sono configurati in due modi:

- Autonoma, detti ***standalone***
- Gestiti da un processo supervisore che è un processo di sistema che va a gestire i server.

I servizi standalone sono:

- Programmi avviati al boot del sistema
- Sono sempre in esecuzione
- Si occupano di ascoltare su una porta di rete
- Provvedono da soli al controllo degli accessi al servizio.

I servizi gestiti dal supervisore sono avviati dal supervisore in caso di richiesta del servizio. Il supervisore si occupa di ascoltare su tutte le porte dei servizi che controlla. Il vantaggio di questo approccio è che occupa risorse solo quando vengono richieste.

Es: NFS è standalone, FTP è gestito.

DNS

La gestione degli IP in forma numerica è inaccettabile dal lato utente.

Per tale ragione agli IP sono associati dei nomi.

La trasformazione di un indirizzo in un nome può avvenire in due modi:

- Tramite un elenco indirizzo-nome
- Usando il servizio DNS

L'uso del DNS impone l'uso della convenzione dei nomi di dominio.

Per fare in modo che gli user non devono ricordare gli IP ma i nomi (google.com, facebook.com,...) usiamo i DNS: in input gli diamo un indirizzo alfanumerico e lui restituisce l'indirizzo IP tradotto.

I DNS vanno visti in 3 punti di vista:

- Organizzazione dei nomi sulla rete
- L'info corrispondente al nome e all'IP deve essere memorizzata da qualche parte nella rete.
- Query

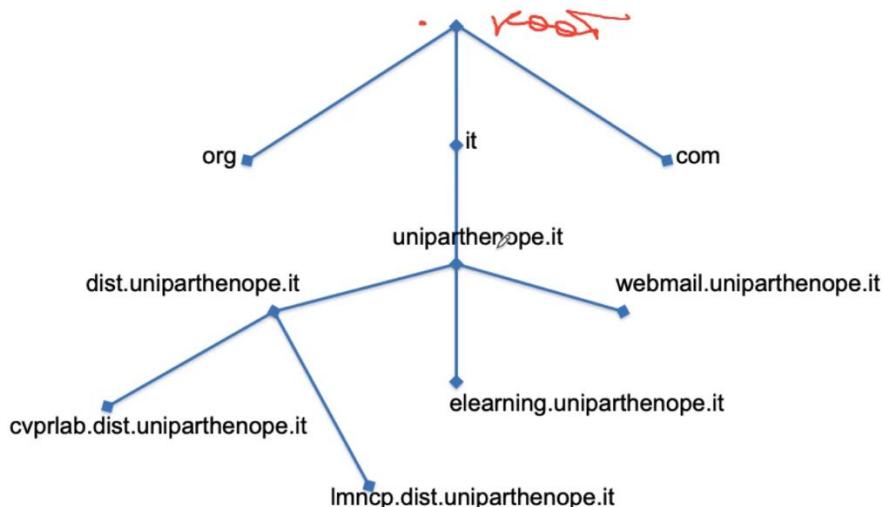
I nomi vengono organizzati in una struttura dati ad albero dove la radice che è rappresentata da un punto (.).

Abbiamo poi i domini di primo livello: org, it, com...

Sotto i domini di primi livello abbiamo quelli di secondo livello (gazzetta.it, unina.it, mammt.it).

Abbiamo quelli di 3 livello(elearning.uni.it) e così via...

Ogni volta che passiamo da un livello a un sottolivello inseriamo un punto.



Partendo dalla foglia saliamo fino alla radice per avere un nome completo DNS.

Le foglie dell'albero sono host della rete (sono macchine fisiche) e quindi dobbiamo avere un IP specifico.

Questi nomi vengono assegnati da autorità di registrazione.

L'ente capo è ICANN che gestisce i DNS a livello globale.

L'ICANN gestisce la radice dell'albero (il punto): gestire la radice significa avere il nodo più importante della struttura dati.

INTERROGAZIONE WHOIS

I database WHOIS sono interrogabili via WEB dai siti delle società che effettuano le registrazioni dell'ICANN.

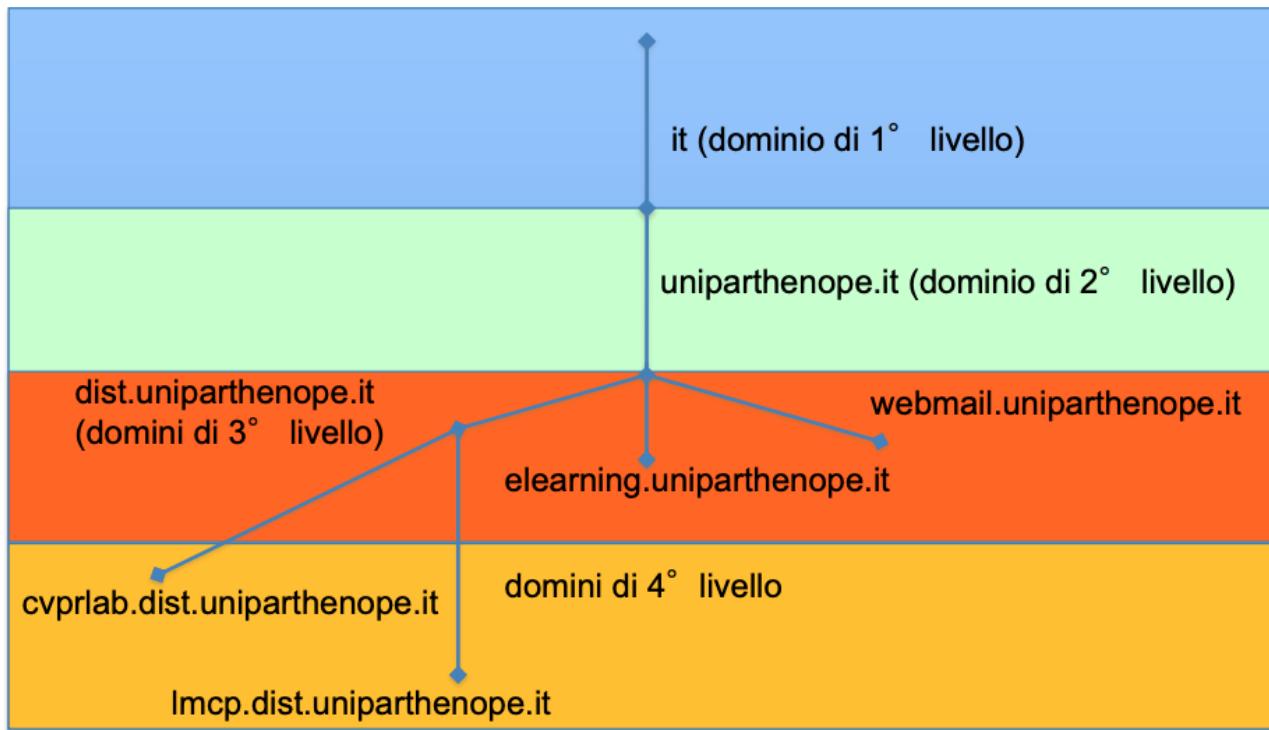
```
ale@mbp ~# whois uniparthenope.it

*****
* Please note that the following result could be a subgroup of      *
* the data contained in the database.                                *
*                                                               *
* Additional information can be visualized at:                      *
* http://www.nic.it/cgi-bin/Whois/whois.cgi                          *
*****
```

...

```
Nameservers
naval.uniparthenope.it
parthenope.uniparthenope.it
ns1.garr.net
```

```
ale@mbp ~#
```



Ogni zona organizza le informazioni di sua competenza in record di risorsa.

RECORD DI RISORSA

Nome	TTL	Classe	Tipo	Dati
------	-----	--------	------	------

Nome: nome del dominio.

TTL: time to live.

Classe: individua il tipo di protocollo.

Tipo: info associata al nome di dominio

Dati: info associato al nome del dominio (indirizzo IP).

Questi record vengono conservati in delle zone ma vengono memorizzati fisicamente su host. Per ogni zona (colori) abbiamo 3 server:

- Primario: ha la copia ufficiale all'interno della zona.
- Secondario: conserva una copia ufficiale ma non quella di riferimento perché a intervalli regolari si fa una copia di quello primario.
- Caching-only (non autoritativo): la risposta che ottengo non è certa. È una risposta cachata in qualche momento passato che non potrebbe essere aggiornato.

Se voglio una risposta ufficiale nome-IP devo richiederla a un serevr autoritativo (che sia primario o secondario). Altri server in giro per la rete potrebbero dare comunque la risposta ma eventualmente errata o obsoleta.

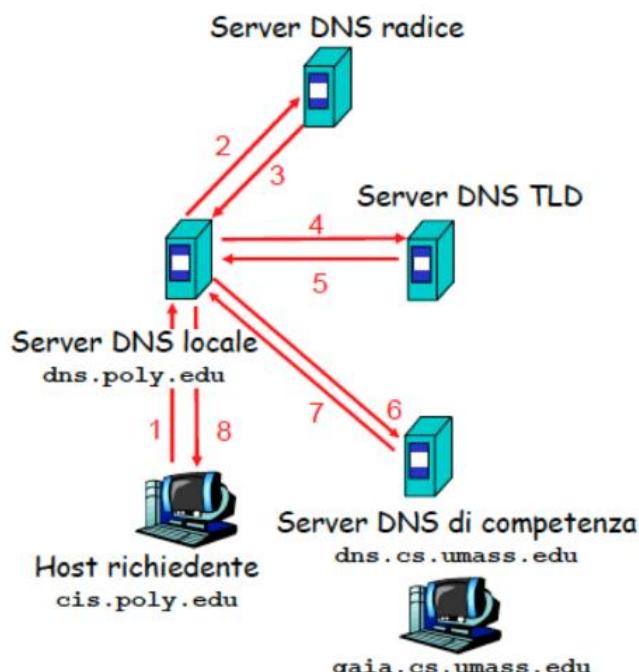
RISOLUZIONE DIRETTA: QUERY ITERATIVA

L'host richiedente fa una query all'host locale che fa la richiesta alla radice che gli dà la risposta (qual è l'indirizzo IP), il server locale fa la query a .com che restituisce la risposta, faccio la query e mi collego.
È iterativa perché il server di zona fa tutto il lavoro di traduzione.

Risoluzione diretta: query iterativa

Esempio:

- L'host cis.poly.edu vuole l'indirizzo IP di gaia.cs.umass.edu
- Il server locale contatta un sever radice, o un server TLD
- Il server contattato risponde con il nome del server da contattare:
"Io non conosco questo nome, ma puoi chiederlo a quest'altro server"
e così via...

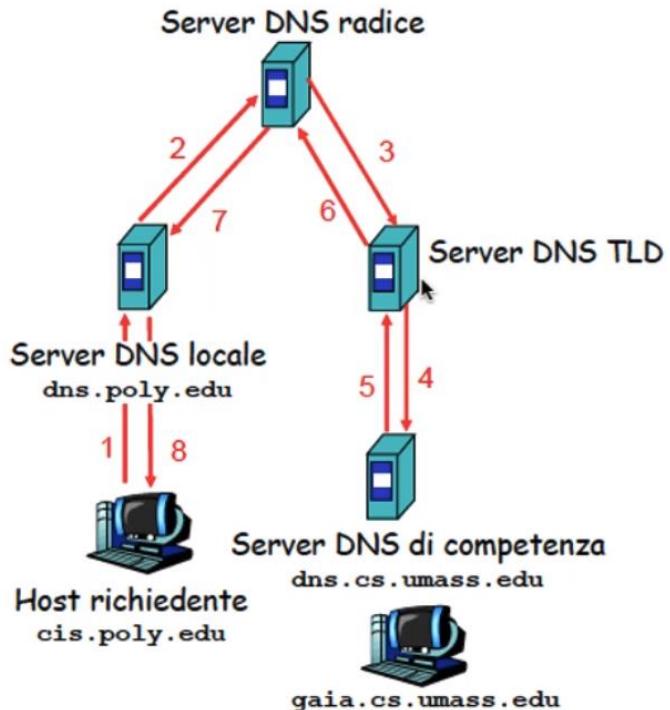


QUERY RICORSIVA

Il server chiede alla radice che prosegue la traduzione, chiede a .com, il quale chiede al dominio, il quale restituisce l'indirizzo IP.

Risoluzione diretta: query ricorsiva

- Il server locale affida al server contattato il compito di tradurre il nome
- Ogni server che non è in grado di rispondere si rivolge esso stesso ad un altro server ed attende la risposta
- La risposta torna al server locale seguendo lo stesso percorso



INTERROGAZIONE DIRETTA

Esiste un tool *nslookup* che ci permette di fare le query.

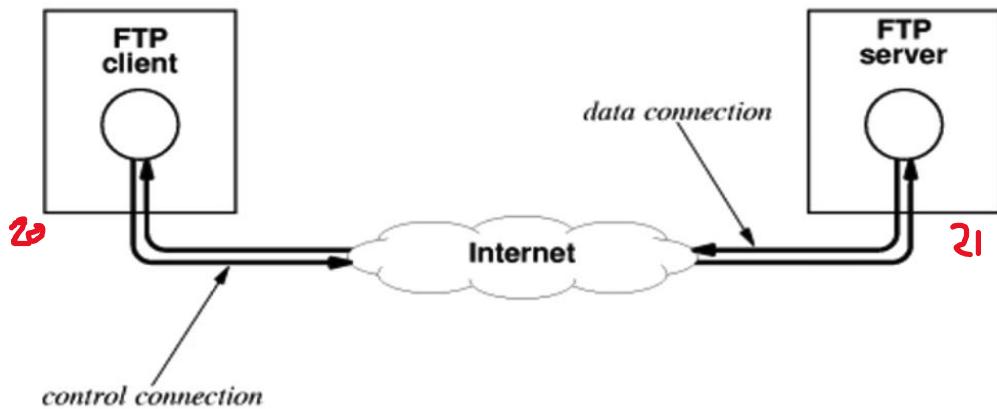
```
[dominickferraro@MacBook-Pro-di-Dominick ~ % nslookup -type=ns google.com
Server:      192.168.1.1
Address:     192.168.1.1#53]
```

FILE TRANSFER PROTOCOL FTP

Il servizio FTP è un protocollo che si usa per trasferire file tra computer collegati con una rete.

FTP usa due tipi di connessioni:

- Connessione di controllo sulla porta 21
- Connessione dati sulla porta 20



LO CHIEDE ALL'ESAME:

FTP usa due protocolli diversi:

- PI → attraverso cui il client invia i comandi e riceve le risposte dal server
- DTP → attraverso il quale il client e il serevr si scambiano dati.

Questo protocollo si chiama *Out Of Band (OOB)*.

Lez 5

Andremo ad implementare un'architettura client-server a livello applicazione.

I livelli che ci interessano al di sotto sono il livello di trasporto e di rete.

A livello di rete usiamo il protocollo IP, a livello di trasporto usiamo il protocollo TCP.

Creeremo una connessione tra client e server.

L'host che ospita il server avrà il suo indirizzo IP e dovrà avere assegnata una porta.

Lato client per effettuare una richiesta di connessione e avviare il 3 way handshake dovrò avere l'IP del server e la porta. Il client girerà su un host che ha un proprio IP, la porta è assegnata dal SO nel momento in cui il client effettua la richiesta di connessione.

BERKELEY SOCKETS

I socket di berkeley sono un'API che definisce una libreria C per comunicazioni inter-processo

anche su rete. Sono lo standard per la realizzazione di applicazione di rete.

Il socket è rappresentato da un descrittore che è un numero intero, che è usato alla stessa stregua dei file.

SOCKET

I socket sono uno dei principali meccanismi di comunicazione

STRUTTURA DI UN CLIENT

- Crea il socket
- Si connette ad un server
- ...
- Chiude il socket
- `socket(...)`
- `connect(...)`
- ...
- `close(...)`

Per prima cosa deve creare la socket rappresentata da un numero intero.

Dopo aver creato la socket richiedo al SO la connessione al server, per effettuare la connessione devo fornire IP e porta del *server*. Se la connect va a buon fine allora è stato creato il canale di connessione.

Una volta che il client ha finito chiude il descrittore, cioè chiude la connessione. Nel momento in cui il client chiude il descrittore, il server viene notificato di questa chiusura.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
    int sockfd, n;
    char recvline[1025];
    struct sockaddr_in servaddr;
    if (argc != 2) {
        fprintf(stderr,"usage: %s <IPaddress>\n", argv[0]);
        exit (1);
    }
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr,"socket error\n");
        exit (1);
    }
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(13);
    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0) {
        fprintf(stderr,"inet_pton error for %s\n", argv[1]);
        exit (1);
    }
```

Verifichiamo se sono stati passati 2 argomenti. Sockfd è il valore di ritorno e crea un descrittore.

Devo chiedere ora al SO di connettersi al server, per fare ciò servono l'indirizzo IP e la porta.

Per specificare queste info si usa una struttura *struct sockaddr_in* e al suo interno definiamo i 3 campi:

- *Sin_family*: famiglia dei protocolli internet.
- *Sin_port*: porta su cui è in ascolto il server
- Definiamo l'indirizzo IP a cui bisogna connettere questo socket che viene a passato come primo argomento della linea di comando e viene memorizzato nel campo *sin_addr*

Dopo queste 3 istruzioni ho memorizzato la famiglia, IP e porta del server.

Dopo aver inserito queste info faccio la *connect* chiedendo al SO di collegare *sockfd* al socket del server e di collegare il canale che viene creato al descrittore passato come argomento. Se la *connect* va a buon fine si è creato un canale di comunicazione e il lato client per accedervi deve usare *sockfd*. È come se stessi chiedendo al SO di collegare il mio *sockfd* al canale.

Quando va a buon fine, leggendo e scrivendo su sockfd sto leggendo e scrivendo sul canale TCP del server.

Dopo la creazione del canale posso leggere dal descrittore. Implicitamente stiamo definendo una regola: il client appena connesso legge. Nell'esempio il client legge tramite la read da socketfd al più 1024 byte che verranno memorizzati in *recvline*.

La *exit* implicitamente chiude tutti i descrittori.

```
if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {  
    fprintf(stderr,"connect error\n");  
    exit(1);  
}  
while ( (n = read(sockfd, recvline, 1024)) > 0) {  
    recvline[n] = 0;  
    if (fputs(recvline, stdout) == EOF) {  
        fprintf(stderr,"fputs error\n");  
        exit(1);  
    }  
}  
if (n < 0) {  
    fprintf(stderr,"read error\n");  
    exit(1);  
}  
exit(0);  
}
```

Socket

- `int socket(int famiglia, int tipo, int protocollo);`
- Famiglia Scopo man
- PF_UNIX,PF_LOCAL Local communication unix(7)
- PF_INET IPv4 Internet protocols ip(7)
- PF_INET6 IPv6 Internet protocols
- PF_IPX IPX - Novell protocols
- PF_NETLINK Kernel user interface device netlink(7)
- PF_X25 ITU-T X.25 / ISO-8208 protocol x25(7)
- PF_AX25 Amateur radio AX.25 protocol
- PF_ATMPVC Access to raw ATM PVCs
- PF_APPLETALK Appletalk ddp(7)
- PF_PACKET Low level packet interface packet(7)

Definiti in <sys/socket.h>

Socket - tipo

- `int socket(int famiglia, int tipo, int protocollo);`
- **SOCK_STREAM** canale bidirezionale, sequenziale affidabile che opera su connessione. I dati vengono ricevuti e trasmessi come un flusso continuo
- **SOCK_DGRAM** usato per trasmettere pacchetti di dati di lunghezza massima prefissata (datagram), indirizzati singolarmente senza connessione in maniera non affidabile.
- **SOCK_SEQPACKET** canale bidirezionale, sequenziale e affidabile che opera su connessione. I dati vengono trasmessi per pacchetti di dimensione massima fissata e vanno letti integralmente
- **SOCK_RAW** canale di basso livello per accedere ai protocolli di rete e alle varie interfacce. Solitamente non utilizzato dalle applicazioni
- **SOCK_RDM** canale di trasmissione di dati affidabile in cui non è garantito l'ordine di arrivo dei pacchetti.
- **SOCK_PACKET** Obsoleto

Creata la socket, andiamo a riempire la struttura *struct sockaddr_in* con i suoi 3 campi (*sin_family*, *sin_port*, *sin_addr*).

!!!!!! LO CHIEDE ALL'ESAME!!!!!!

Abbiamo usato due funzioni:

- *Inet_pton*: prende l'IP e lo trasforma in un IP a 32 bit e lo trasforma in big endian e lo memorizza nel campo *sin_addr*.
- *Htons*: *host to network short* → trasforma l'argomento passato da formato host a formato network. Esistono due grandi famiglie di processori: quelli con codifica big endian (byte più significativa a sinistra) e quelli little (contrario di big). Si differenziano nell'ordine in cui vengono memorizzati i byte.
Per convenzione il dato è in big endian, se è little allora verrà trasformato.

!!!!

!!!!!!

!!!!!!

!!!!!!

CONNECT

Connette il socket *sockfd* all'indirizzo *serv_addr*. Il secondo argomento è l'indirizzo della struttura di cui viene fatto il cast a *struct sockaddr* che è il

tipo generico che funziona per tutti i tipi di protocolli.

Dal terzo argomento la funzione capisce quale argomento utilizzare.

- `connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr))`

Restituisce 0 se la connessione è a buon fine, -1 se errore.

STAMPA DEL MESSAGGIO

Si fa una lettura e scrivo sullo stdout.

- ```
while ((n = read(sockfd, recvline, 1024)) > 0)
{
 (recvline[n] = 0;
 if (fputs(recvline, stdout) == EOF) {
 fprintf(stderr,"fputs error\n");
 exit(1);
 }
}
```

---

## SERVER

- Creare il socket.
- Chiedere al SO di assegnarci un indirizzo (parliamo della coppia IP, porta).
- Si mette in ascolto

- Accetta una nuova connessione
- Chiude la socket

- Crea il socket
  - Gli assegna un indirizzo
  - Si mette in ascolto
  - Accetta una nuova connessione
  - ...
  - Chiude il socket
- socket(...)
  - bind(...)
  - listen(...)
  - accept(...)
  - ...
  - close(...)

Per prima cosa quindi creiamo la socket.

```
if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
 perror("socket");
 exit(1);
}
```

### Creazione della socket

Stessa procedura del client.

Secondo passo: dobbiamo valorizzare i campi della struttura *struct sockaddr\_in* come col client.

Nella famiglia ci mettiamo AF\_INET, nel campo *sin\_port* andiamo a specificare la porta che stiamo chiedendo al SO per accettare il servizio (il server una volta partito resterà su quella porta

in attesa di client che si connettano). L'ultimo campo da riempire è il campo *sin\_addr* dove nel lato client abbiamo messo l'IP del server, mentre lato server andiamo a inserire o un IP specifico che è l'interfaccia sulla quale accetto le connessioni o tutti quanti: se voglio usare tutti gli IP locali sulla macchina usiamo la macro *INADDR\_ANY*: se sulla macchina ho 3 interfacce, il client potrebbe collegarsi a una delle 3, qualunque essa sia, sulla porta "n" risponde il server.

```
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(13);
if (bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
 perror("bind");
 exit(1);
}
```

Una volta che abbiamo definito l'indirizzo chiediamo l'assegnazione dell'indirizzo(IP, porta che viene collegata a quel descrittore di socket) con una *bind*: *chiediamo al SO di collegare listenfd alla porta 13 e agli indirizzi IP a livello di rete.*

```
if (bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
 perror("bind");
 exit(1);
}
```

**Assegnazione Indirizzo**

Se la bind è andata a buon fine, il descrittore è collegato alla porta, il prossimo step è mettere in ascolto la porta in modo che i client possono collegarsi con la syscall *listen*: gli passiamo il descrittore del socket e la lunghezza della coda di ascolto, che definisce la lunghezza di una coda: quanti client possono mettersi in attesa? 1024 in questo caso.

```
if (listen(listenfd, 1024) < 0) {
 perror("listen");
 exit(1);
}
```

Messa in ascolto

1024 è il numero massimo di client che può servire il server? NO perché ovviamente la coda si aggiorna dato che si libera e riempie sempre.

Ora il server ha aperto la porta ed entra in un ciclo infinito *for(;;)* e il server lì.

Dobbiamo accettare una richiesta di connessione con *accept*: è una chiamata bloccante(il processo si blocca sulla chiamata finchè non arriva una richiesta di connessione da parte del client).

Quando la *accept* si sblocca restituisce un altro descrittore: *connfd*. Su *listenfd* si accettano le

richieste di connessioni mentre su connfd  
gestisco il servizio.

```
if ((connfd = accept(listenfd, (struct sockaddr *) NULL, NULL)) < 0) {
 perror("accept");
 exit(1);
```

**Accettazione nuova connessione**

---

## Lez 6

Ricordiamo che lato client

- Creiamo il descrittore del socket con la chiamata socket a cui passiamo AF\_INET, SOCK\_STREAM , 0 (sottotipo del livello di trasporto).
- Specifichiamo l'ip e la porta del server a cui vuole connettersi valorizzando i campi della struttura *sockaddrin*: *sin\_family:famiglia* (*AF\_INET*), *sin\_addr* e *sin\_port* (*porta su cui è in ascolto il server che memorizziamo tramite la funzione htons*)).
- Fatto ciò, passiamo la struttura alla connect() che ci darà il descrittore connesso col server.

- Il client effettua una richiesta con una `write()` e si mette in attesa di una risposta del server.
- Quando termina chiude la connessione e termina l'uso del servizio.

## LATO SERVER:

- Identica creazione della socket
- Valorizziamo la struttura : famiglia, porta che il server richiede al SO e nel campo `sin_addr` mettiamo `INADDR_ANY`.
- Passo l'indirizzo della struttura e il `sockfd` alla `bind` che collega la porta sull'IP.
- Se la `bind` va a buon fine, richiamo `listen()` dove passo `sockfd` e la lunghezza della coda di attesa del server.
- Inizia il ciclo infinito (;;) dove il server accetta (`accept()`) una richiesta di connessione. Il socket effettivamente connesso è quello restituito dalla `accept()`.

- Leggo le richieste, le elaboro e le restituisco.
- Il server dopo aver letto l'EOF viene chiuso il descrittore.

## WRITE

- `ssize_t write(int fd, const void *buf, size_t count);`

Si usa per scrivere su una socket.

La write può essere interrotta: potrebbe scrivere un numero inferiore di byte richiesti e generare errori.

Devo avere un metodo per cui se si interrompe la write devo rilanciarla per finire esattamente i byte richiesti.

Dobbiamo essere in grado di comunicare il numero esatto di byte che l'altra parte attende di ricevere. Non possono esserci byte mancanti poiché il protocollo non può essere portato a termine.

Per ovviare a questo problema di interruzione delle syscall, introduciamo la *fullwrite()*;

## FULLWRITE

È una write che garantisce, in caso di interrupt, di inviare esattamente i byte richiesti a meno che non si verifichi un errore irreversibile.

La fullwrite() ha la stessa firma della write(), inizializza una variabile nleft = count.

Inizia un ciclo che viene ripetuto finché non terminano tutti i byte che devo scrivere.

Nleft è un contatore che viene decrementato ogni volta che viene scritto un byte, quando sarà zero posso uscire.

Nel ciclo effettuo la write(), se si verifica un'interruzione (la write non è stata portata a termine), ritento la scrittura, se invece la write è fallita perché non è stata in grado di scrivere, c'è stato un errore non gestibile.

Una volta che la write ha effettuato una scrittura, sottraggo i byte scritti a left e li uso per spiazzare il puntatore a buff.

```

#include <unistd.h>
ssize_t FullWrite(int fd, const void *buf, size_t count)
{
 size_t nleft;
 ssize_t nwritten;
 nleft = count;
 while (nleft > 0) { /* repeat until no left */
 if ((nwritten = write(fd, buf, nleft)) < 0) {
 if (errno == EINTR) { /* if interrupted by system call */
 continue; /* repeat the loop */
 } else {
 exit(nwritten); /* otherwise exit with error */
 }
 }
 nleft -= nwritten; /* set left to write */
 buf += nwritten; /* set pointer */
 }
 return (nleft);
}

```

Finchè ci sono byte da scrivere, tento la scrittura.

Se *nwritten* <0 ci sono due casi:

1. Interruzione
2. Errore write

Se *nwritten* è <0 controllo la variabile *errno*  
*(variabile di ogni processo che contiene il codice di uscita dell'ultima syscall del processo)* e se è *EINTR* vuol dire che la write è stata interrotta:  
non esco ma ritorno al while e tento di nuovo la scrittura.

Se invece  $nwritten < 0$  e  $errno != EINTR$  esco perché c'è stato un errore non gestibile. Quando esco dal while, sottraggo il numero di byte scritti a  $nleft$  e spiazzo il puntatore a buff di  $nwritten$  byte. Alla fine restituisco  $nleft$  (che è zero) cioè ho finito i byte rimanenti.

---

## READ

La stessa logica vale per la read che potrebbe leggere meno byte e implementiamo *FullRead*

Il codice è simile.

Bisogna controllare che  $nread == 0$  e siamo nel EOF, l'unica cosa in più.

$nleft$  viene aggiornato sottraendo il numero di byte effettivamente letti e si spiazza il puntatore. Alla fine ritorniamo  $nleft$  (che è zero).

```

#include <unistd.h>
ssize_t FullRead(int fd, void *buf, size_t count)
{
 size_t nleft;
 ssize_t nread;
 nleft = count;
 while (nleft > 0) { /* repeat until no left */
 if ((nread = read(fd, buf, nleft)) < 0) {
 if (errno == EINTR) { /* if interrupted by system call */
 continue; /* repeat the loop */
 } else {
 exit(nread); /* otherwise exit */
 }
 } else if (nread == 0) { /* EOF */
 break; /* break loop here */
 }
 nleft -= nread; /* set left to read */
 buf += nread; /* set pointer */
 }
 buf=0;
 return (nleft);
}

```

## FULLREAD E FULLWRITE LA CHIEDE ALL'ESAME.

---

### SERVER CONCORRENTI

Gestiscono più connessioni contemporaneamente.

Usano una seconda istanza di sé stessi per gestire le connessioni client.

Usano la *fork()* (*chiede all'esame*) per generare un processo figlio.

I processi server padre e figli sono eseguiti “contemporaneamente”.

Il padre accetta connessioni e delega i figli a gestire le richieste dei client.

Abbiamo listenfd dove il server è bloccato, quando arriva la richiesta di connessione, la accept() restituisce la connsd. Fatto ciò, il server fa la fork() che copia esattamente il server. Così facendo avrò i descrittori listen e connfd del padre e figlio. Il padre è interessato solo alla listenfd e quindi chiude la connsd e ritorna a bloccarsi sulla accept(). Il figlio, interessato alle richieste del client, chiude la listenfd().

---

## SERVER CONCORRENTI

```
if(logging)
{
 inet_ntop(AF_INET,&client.sin_addr,buffer,sizeof(buffer));
 printf("Request from host %s, port %d\n",buffer, ntohs(client.sin_port));
}
```

*Inet\_ntop* è l'inversa della *inet\_pton*: prende un puntatore alla struttura *sin\_addr* (l'IP in formato network) e lo trasforma in dotted decimal. Il campo *sin\_addr* lo prende da un'altra struttura *client* (*che è il codice del client connesso*) →

Nella *accept* gli passo listenfd, (struct sockaddr\*)&client, sizeof(client));

```

while(1)
{
 len = sizeof(client);
 conn_fd=accept(listefd,(struct sockaddr*)&client,sizeof(client));
 ... /*COMPLETE HERE*/
 /* fork to handle connection */
 if((pid=fork())<0)
 {
 perror (" fork error ");
 exit (-1);
 }
}

```

Nei campi della struttura client avrò l'IP del client appena connesso e nel campo sin\_port la porta da cui si sta connettendo il client. Per trasformare la porta da network a formato host uso *ntohs*.

*!!! DOMANDA D'ESAME:*

*è necessario per il server conoscere le info relative all'ip e alla porta del client?*

1. No poiché prima passavamo ad accept null e null
2. La accept restituisce un descrittore di socket connesso quindi il server legge e scrive su di esso e ho la garanzia che quei dati vengano da quel client e saranno diretti ad esso. Non è necessario avere queste info perché il server ha bisogno solo del descrittore connesso. Queste info sono un

plus che il server potrebbe usare per fornire un servizio diverso, lanciare una routine diversa se il client viene da una particolare rete o porta, si potrebbe effettuare un filtraggio dalla connessioni: quelle che vengono da un det IP vengono rifiutate o smistate ecc...

CHIEDE ANCHE LA FORK()!

!!!! ----- !!!!!!! ----- !!!!!!!

---

TERMINAZIONE DI UN PROCESSO FIGLIO

Quando un processo figlio termina:

- Viene inviato il segnale *SIGCHLD* al padre.
- Il processo diventa “zombie”.

Gli zombie sono processi che hanno terminato l'esecuzione ma restano presenti nella tabella dei processi.

In genere possono essere identificati dall'output del comando *ps* per la presenza di una Z nella colonna di stato.

## SEGNALI

Per gestire i segnali esiste la sycall *signal()* e possiamo definire il comportamento di un processo a fronte della ricezione di un segnale. Ci sono un insieme fissato di segnali a cui corrispondono delle azioni di default. Non ci sono punti in cui il processo si mette in attesa del segnale, può arrivare sempre.(asincrono)

Possiamo modificare però l'azione di un segnale con una procedura specifica detta *handler*.

### HANDLER

```
void funzione(int num_segnale) {
 printf("%d", num_segnale);
}
```

Quando l'handler termina, l'esecuzione del processo riprende dal punto in cui era stato interrotto.

## CATTURARE UN SEGNALE

- `signal(SIGINT, handit)`
- imposta la funzione handit come handler del segnale SIGINT
- E' anche possibile ignorare un segnale
  - `signal(SIGINT, SIG_IGN)`
- oppure ritornare alla reazione di default
  - `signal(SIGINT, SIG_DFL)`

*Handit* è la funzione mandata in esecuzione quando il processo riceve *SIGINT*.

Es: il padre crea un figlio quando accetta una connessione, il figlio crea la stringa con data e ora ed esegue la exit. In questo caso il SO invia un SIGCHILD al padre.

---

## IGNORARE TERMINAZIONE DEI FIGLI

In linux, se ignoro la terminazione dei figli non diventano zombie. Non è conforme allo standard POSIX.

---

## OPZIONI DEL SOCKET

Per risolvere il problema della *bind()* su una porta già occupata: *address already in use*.

- Perché la *bind()* dice address e non *port already in use*? La compilazione IP porta è già stata occupata, c'è un altro processo che la sta gestendo.

Ci sono casi dove vogliamo che più processi possano gestire la stessa combinazione IP-porta. In questo caso si deve trattare dello stesso codice che viene lanciato più volte. Questo controllo è demandato al programmatore e non al SO.

Per effettuare questa tecnica (avere più processi che sono collegati alla stessa coppia IP porta) è possibile modificare le opzioni del socket.

Dopo aver creato il socket ho delle funzioni che permettono di agire sul comportamento della socket.

Queste funzioni sono *getsockopt* e *setsockopt*.

- `#include <sys/socket.h>`
- `int getsockopt(int sd,int level, int optname, void* optval, socklen_t optlen);`
- `int setsockopt(int sd,int level, int optname, const void* optval, socklen_t* optlen)`

Possiamo chiedere di modificare alcune di queste opzioni.

Queste due funzioni prendono il descrittore su cui vogliamo modificare le opzioni, il livello a cui vogliamo lavorare: possiamo richiedere di lavorare a più livelli rispetto alla pila ISO OSI, optname è l'opzione che voglio modificare per ogni livello e optval è il valore che voglio assegnare a quell'opzione, optlen è la lunghezza del campo.

*Getsockopt* recupera il valore di una opzione: in sd c'è il descrittore, level è come prima, lo stesso optname, optval conterrà il valore di quell'opzione.

---

## LIVELLO DELLE OPZIONI

Il livello più alto è *SOL\_SOCKET* (*usiamo questo*) cioè modifichiamo le opzioni a livello applicazione.

Possiamo lavorare a livello IPv4, IPv6, ICMP e livello di trasporto.

Ogni livello ha varie opzioni.

---

## OPZIONI DI LIVELLO SOCKET

- SO\_BROADCAST permette il broadcast
- SO\_DEBUG abilita le informazioni di debug
- SO\_DONTROUT Esalta il lookup nella tavola di routing
- SO\_ERROR legge l'errore corrente
- SO\_KEEPALIVE controlla che la connessione sia attiva
- SO\_LINGER controlla la chiusura della connessione
- SO\_RCVBUF grandezza del buffer in ricezione
- SO\_SNDBUF grandezza buffer in spedizione
- SO\_RCVLOWAT soglia per il buffer in ricezione
- SO SNDLOWAT soglia per il buffer in spedizione
- SO\_RCVTIMEO timeout per la ricezione
- SO\_SNDTIMEO timeout per la spedizione
- **SO\_REUSEADDR permette riutilizzo indirizzi locali**
- SO\_REUSEPORT permette riutilizzo porte locali
- SO\_TYPE il tipo di socket
- SO\_USELOOPBACK per i socket di routing (copia i pacchetti)

- *SO\_REUSEADDR*: permette il riuso della coppia IP-porta

- Es.: dopo `socket()` e prima di `bind()`

```
int enable = 1;
```

```
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int))
```

Con questo esempio stiamo dicendo che possiamo usare la stessa configurazione senza attendere che il processo principale vada in timeout.

Affinchè questa tecnica funzioni, tutti i processi che richiedono la stessa ip port devono lanciare la *setsockopt* per confermare il fatto che vogliono “condividere” la porta.

---

## Lez 7

Vediamo come interagire col DNS attraverso i client-server.

Il DNS è il servizio distribuito che associa a una stringa alfanumerica un indirizzo IP perché nel momento in cui andiamo a richiamare la *connect()*, per collegarci al server dobbiamo fornire IP-porta, la rete di vari livelli necessita del concetto di indirizzo IP.

Prima di effettuare una connessione col serevr, se abbiamo solo il nome dobbiamo ricavarci l'IP, servizio offerto dal DNS.

---

### FQDN

Il *fully qualified domain name*, composto dal nome locale dell'host e dal dominio di appartenenza: ***cvprlab.dist.parthenope.it***

L'associazione tra nomi simbolici e indirizzi viene effettuata dal DNS.

L'albero dei domini contiene tutti i possibili nodi canonici sulla rete internet. Conservare l'info che

associa ad ogni host il suo IP in unico punto è impossibile dati gli infiniti host.

L'albero viene partizionato in zone di competenza: si creano porzioni di alberi e la loro intersezione è vuota: solo in quella zona ho la corrispondenza host ip per tutti i nodi che si trovano lì dentro.

All'interno di ogni zona di competenza abbiamo 3 server.

## DNS

Questi server mantengono una tabella di record di risorsa che hanno un tipo e il dato associato.

- Per effettuare conversione da IP a host usiamo i record di tipo A che associano FQDN a indirizzi IPv4
- AAAA associa FQDN a IPv6.
- PTR associa un indirizzo IP a un hostname.
- MX specificano chi agisce da mail exchnger per un dominio.
- CNAME forniscono alias per un host.

I record PTR non vengono quasi più usati: nelle zone di competenza i 3 server (primario, secondario e cache) non recuperano la traduzione inversa PTR.

---

## SERVER DNS

Dalla shell è possibile usare i comandi *nslookup*, *host* e *dig*.

- *Il comando host effettua query al DNS.*

## RESOLVER

Il linguaggio C fornisce un insieme di routine detto *resolver* che consentono di effettuare query al DNS delle app.

Tutti i client hanno la possibilità di accedere al resolver ma prima di connettersi al server deve avere l'IP. Ogni client fa una richiesta al modulo che si occupa di effettuare la risoluzione dei nomi.

Il resolver fa una query per analizzare i file statici: sono file dove è possibile inserire in

formato testuale un'associazione indirizzo IP nome canonico. È come se inserissi un record di tipo A in un file. Quando il client fa una richiesta, il resolver verifica la presenza di una corrispondenza all'interno del file statico. Immaginiamo l'uso del file etc/hosts e scivo google.com con l'IP di google: nel momento in cui il client (browser) ha bisogno dell'IP di google, il resolver trova la corrispondenza nel file e non fa la query al DNS ma restituisce direttamente al client l'indirizzo che ha segnato. Questo potremmo farlo con tutti gli indirizzi che vogliamo.

Qual è il vantaggio?

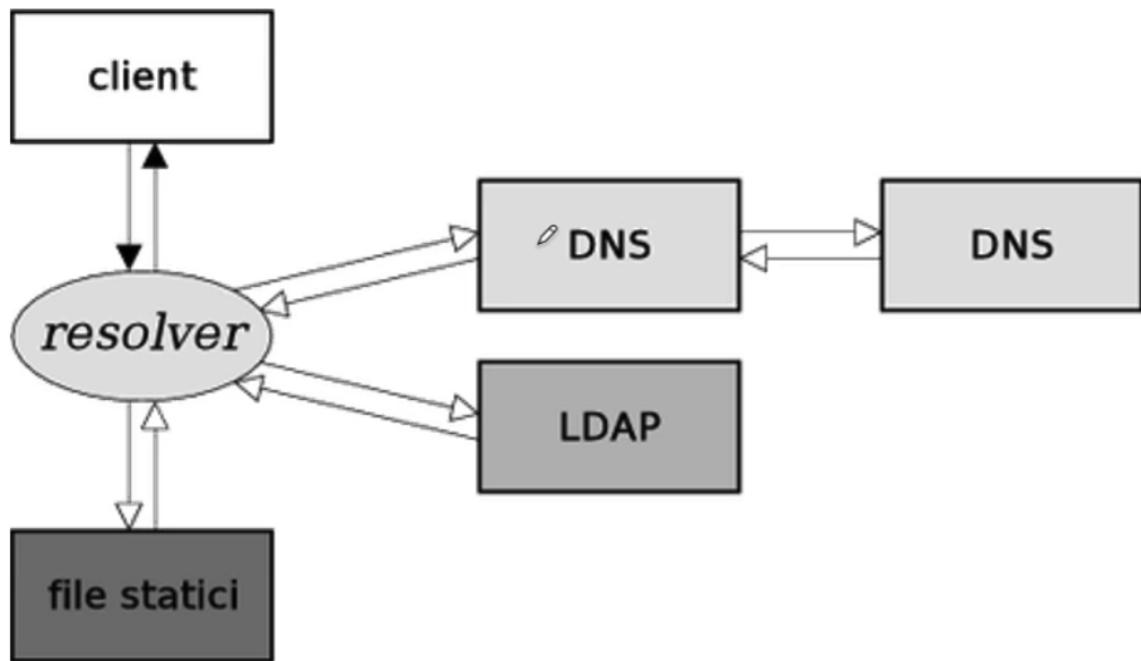
- La velocità
- Meno traffico
- Meno risorse in rete

Svantaggio:

- Occupare memoria
- I file sono statici: gli IP potrebbero cambiare e il resolver restituirà sempre lo stesso IP scritto in passato. È come se avessimo una cache che non può essere aggiornata. Al

crescere del numero di host un approccio del genere è disastroso.

## Struttura del resolver



Tuttavia un approccio del genere ha una ragione di esistere:

se abbiamo da fare un sito “www.esempio.it”, se non abbiamo registrato il dominio, come otteniamo l’ip di esempio.it? nel file etc/hosts scrivo semplicemente che esempio.it si trova all’indirizzo di loopback (127.0.0.1). a questo punto le richieste andranno al web server esempio.it. Quando registreremo il sito,

dovremo spostare il sito senza cambiare nulla perché tutti i link e riferimenti sono stati fatti a esempio.it stesso.

Il secondo passo di resolver è conoscere l'IP che gestirà le query. Questi vanno configurati nel file *etc/resolve.conf*

Il servizio di resolving restituisce sempre una risposta:

- Se non trova la corrispondenza va a chiedere al DNS
  - Se non trova corrispondenza al DNS lo manda alla zona superiore
  - Se non esiste, il DNS dà sempre una risposta, NXDOMAIN, *non existent domain*. Questo vuol dire che quel nome non è stato registrato all'interno dell'albero dei domini.
- 

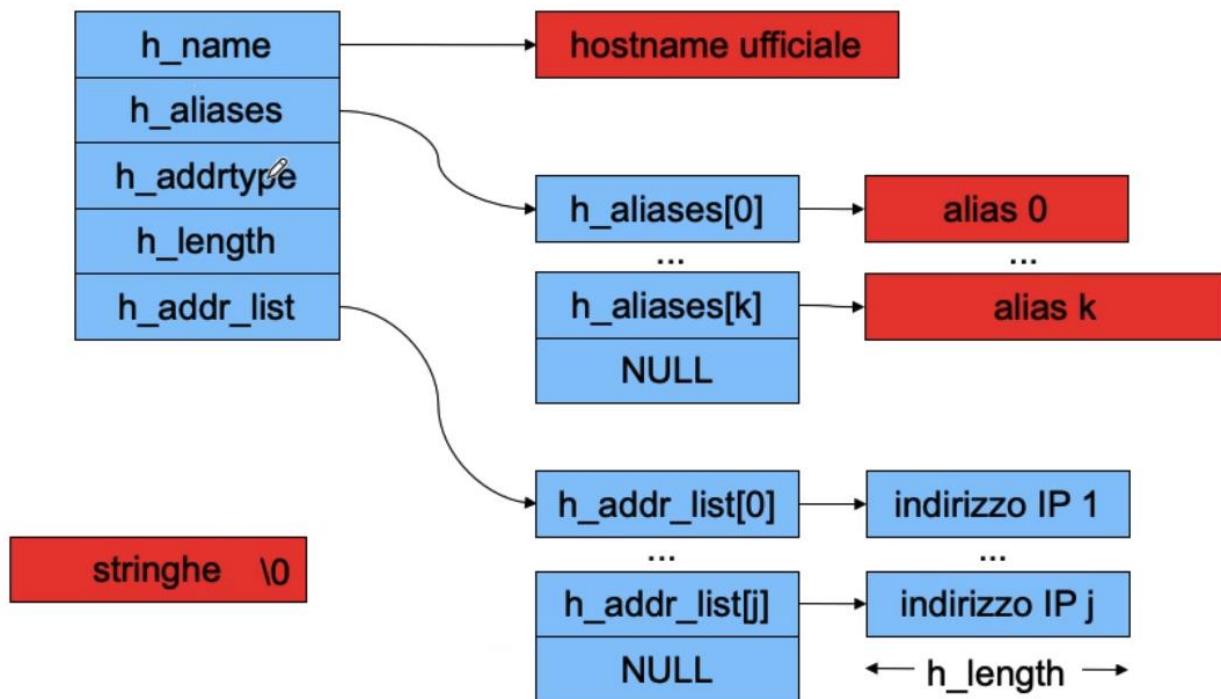
## GETHOSTBYNAME

- **#include <netdb.h>**
- **struct hostent \*gethostbyname (const char \*nome);**

Prende come argomento un array di caratteri che è il nome canonico di cui vogliamo l'IP e ci restituisce un puntatore a una struttura *hostent*: è la struttura che contiene le info relative al nome dell'host passato come argomento.

- Contiene il nome canonico dell'host
- Doppio puntatore a caratteri in cui sono memorizzati tutti gli alias (nomi alternativi per quel nome canonico richiesto)
- La famiglia degli indirizzi
- La lunghezza dell'indirizzo
- Puntatore di puntatore a carattere che è la lista di indirizzi IP dell'host. Sono IP in formato network.
- struct hostent {
- char \*h\_name; /\* nome canonico dell'host \*/  
*stringa zero-terminata*
- char \*\*h\_aliases; /\* lista di alias \*/  
*array di string il cui ultimo elemento è null*
- int h\_addrtype; /\* famiglia dell'indirizzo \*/  
*famiglia degli indirizzi*
- int h\_length; /\* lunghezza dell'indirizzo \*/  
*lunghezza degli indirizzi*
- char \*\*h\_addr\_list; /\* lista di indirizzi \*/  
*lista di indirizzi IP dell'host*
- }

In rosso abbiamo le stringhe.



## GETHOSTBYNAME2

Ci permette di lavorare sia con IPv4 o IPv6 impostando come argomento l'indirizzo delle famiglie

## INET\_NTOP

Converte l'indirizzo memorizzato in formato network in notazione dotted decimal.

## GETHOSTBYADDR

Per effettuare la conversione da IP a nome host usiamo questa funzione.

Prende come argomento un puntatore a char ma questo puntatore è un puntatore a una struttura *in\_addr* che contiene l'indirizzo in formato network.

Significa che dobbiamo prendere la struttura *in\_addr* e fare il cast al puntatore a caratteri: questo serve a trattare quel dato come un array di byte, indipendente dalla sua forma. Se faccio il cast a carattere la funzione la vede come una sequenza di byte.

Socklen e int family definiscono il tipo di dato.

---

## HERROR

- **#include <netdb.h>**
- **void perror(const char \*s);**
- **Sostituisce perror in caso di errore delle funzioni del resolver**

Prende una stringa di caratteri e la stampa prima dell'errore associato alla risoluzione. È analoga alla perror e riguarda solo gli errori di risoluzione.

---

Commento al codice degli esercizi

Dichiariamo due strutture hostent data e rdata.

L'idea è del codice è prendere il nome per fare la risoluzione diretta e prendo la struttura hostent. Da questa prendo l'ip e faccio la risoluzione inversa.

Su argv[1] richiamo gethostbyname() con data. se data== null la risoluzione è fallita e con error stampo ed esco.

Altrimenti stampo il nome canonico.

Vado poi a stampare gli alias.

*H\_aliases* è un puntatore di puntatore a caratteri.

Memorizzo in alias *h\_aliases* .

Finche *alias!= null* cioè finchè ci sono stringhe, stampo il contenuto della stringa e avanzo *alias++*.

*Stampo addrtype con AF\_inet o AF\_inet6.*

Faccio poi la stessa cosa con *h\_addr\_list*: in alias metto l'indirizzo base dell'array, finchè !=null,

effettuo la ntop per convertire l'indirizzo da formato network a IP come stringa.

## //RISOLUZIONE INVERSA

Per prima cosa memorizzo *addr* in *raddr* con la funzione *inet\_aton*.

A questo punto uso *gethostbyaddr* che è un array di byte grazie al cast di caratteri.

Se *rdata* == NULL errore di risoluzione. Altrimenti *alias* = *rdata* → *h\_aliases* e stampo l'array di *alias*. Dopo stampo il nome canonico (*h\_name*).

---

**DOMANDA ESAME:** *gesthostbyname* e *gethostbyaddr* restituiscono una struttura *hostent* e vuole sapere le info che contiene (ip,alias, indirizzi canonici...).

## Lez 8 ESERCITAZIONE

La *fullRead* è stata introdotta per evitare che un flusso di dati non si perda e venga completamente inviato, nello stesso ordine di invio.

Se il client invia una richiesta di 3 byte, il server per interpretare il protocollo deve poter leggere esattamente l'intera richiesta.

La *read* poteva essere interrotta e non fallire mentre la *fullRead* rimane bloccata nel ciclo finchè non legge tutti i byte.

Perché client o server rimangono bloccati sulla *fullRead* quando ci sono più richieste? La condizione di uscita è che *nleft=0* cioè non ci sono più byte da leggere.

Il *peer* che legge, come fa a sapere quanti byte deve leggere?

Guardando il codice che abbiamo usato fino ad ora mettevamo 1024 che era arbitrario perché il client dopo aver ricevuto qualcosa chiude la connessione e il server è bloccato sulla *fullRead*, gli arriva un “/0”, si sblocca la *read* e si blocca

sulla *accept* del *client* successivo o nel caso del server multiprocesso uccide il figlio.

Dobbiamo fare in modo che tutto ciò che viene scritto dal client deve essere letto ma il server deve sapere quanto deve leggere.

Vedremo 2 possibilità.

---

## Client/Server Stringhe

Chi scrive, prima di scrivere i byte, invia il numero di byte che andrà a scrivere nella successiva scrittura.

Il client deve fare una richiesta: calcola la dimensione richiesta. Il primo invio lo fa con la dimensione della richiesta, dopodichè invia la richiesta vera e propria. Questo escamotage risolve il problema perché quando invia la DIM della richiesta sta inviano un numero intero.

Con la Read posso inviare un numero intero?

- Posso usare quell'intero come array di byte facendo un cast. Come ultimo parametro della read gli passiamo il *sizeof* dell'array. Così comunico dall'altra parte quanti byte manderò alla prossima lettura. Il prossimo passo è quindi fare una write di TOT byte.

- Dall'altra parte, chi riceve la richiesta, farà una read di un intero, leggendo preventivamente quanti byte legge nella successiva richiesta. Infine riceve la vera richiesta.

### Client

n=strlen(s)

Fullwrite(socket,&n,sizeof(int)); Fullread(socket,&n,sizeof(int));

Fullwrite(socket,s,n); Fullread(socket,s,n);



### Server

-

Il client genera una richiesta e ne calcola la lunghezza.

Il server dall'altra parte leggerà l'intero.

Il server sa che nella successiva scrittura, il client scriverà N byte e fa una fullread di *n* byte.

**QUINDI: La fullread si bloccava poiché gli passavamo 1024 ma magari la write mandava solo 3 byte...**

Questo è il primo modo di risolvere questo problema.

---

Un altro modo invece lo introduciamo con un possibile problema.

Vediamo come è possibile definire semplicemente tutti i passi che ci portano dal problema all'implementazione.

L'esempio che vediamo è un gioco.

## Client/Server battaglia navale

- Scrivere un client ed un server che implementino il gioco della battaglia navale.

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 |

In 1 c'è la nave e 0 no. Client e server si alternano negli attacchi dando le coordinate di righe e colonna. Se l'attacco va a buon fine, chi subisce l'attacco risponderà con un'ACK oppure con "acqua".

La prima cosa da fare è definire l'architettura. Un giocatore sarà il client e l'altro il server.

Architettura significa individuare le entità che partecipano al programma e capire come sono collegate tra loro, chi funge da client e server.

- Schema architettura:



Sarebbe il primo paragrafo della relazione!!

Dopo aver definito l'architettura si devono definire le regole a livello applicazione.

Dobbiamo definire le regole del gioco.

- Protocollo applicazione:
- Il client ed il server si sincronizzano
- Il client gioca per primo
- Il client ed il server si alternano negli attacchi.
- Per eseguire un attacco si inviano le coordinate
  - riga e colonna
- Chi subisce l'attacco verifica
  - l'esito dell'attacco (copito/acqua)
  - l'esito della partita (finite/non finita)

Siamo a livello applicazione: nella documentazione non posso scrivere uso *read*, *bind* ecc.. quello è al livello architetturale.

Dopo aver definito le regole vado a specificarle ulteriormente:

- **Protocollo applicazione:**
  - valore di sincronizzazione arbitrario
  - riga/colonna: 1...n ( $n = \text{dimensione della matrice}$ )
  - esito attacco: 1=colpito 0=acqua
  - esito partita: 0=finita 1=non finita
- **Pacchetto applicazione:**



Chi attacca invia riga e colonna. Chi riceve 1 significa che è andata a buon fine e se riceve poi 0 significa che ci sono ancora altre navi.  
Chi riceve non sa quanti byte deve ricevere e la soluzione è creare un pacchetto applicazione: ad esempio abbiamo inventato un pacchetto

contentente riga, colonna, esito attacco ed esito partita.

In ogni parte inserisco il campo corrispondente con la dimensione.

Questo significa che definirò un ***pacchetto***.

Un pacchetto, a livello di programmazione, è una ***struct { int riga; int colonna; ...}.***

Quando uno dei due peer esegue l'attacco, verranno valorizzati solo i campi riga e colonna. Chi subisce l'attacco verifica e riempie i campi esito attacco e partita.

Client e server si scambiano pacchetti di livello applicazione.

In questo caso sto dicendo che anche se non usati, tutti i campi fanno parte del pacchetto.

Potremmo immaginare di definire un pacchetto di richiesta con righe e colonne e un pacchetto di risposta con 2 interi che sono esito attacco ed esito partita. Potremmo pensare che l'attacco sia solo 2 interi, mentre la risposta contiene le coordinate dell'attacco, l'esito dell'attacco e della partita. Questo pacchetto lo possiamo fare complesso quanto lo si vuole, non c'è limite, con TCP al massimo viene spacchettato.

Il vantaggio di questo approccio è che client e server sanno dove prendere le info e sanno esattamente la dimensione dei dati che si stanno scambiando: il **client** invierà pacchetto richiesta e il **server** risponderà con pacchetto risposta.

A questo punto, il client e il server non dovranno fare la doppia ***fullRead*** e ***doppiaFullwrite*** perché conoscono la DIM del pacchetto.

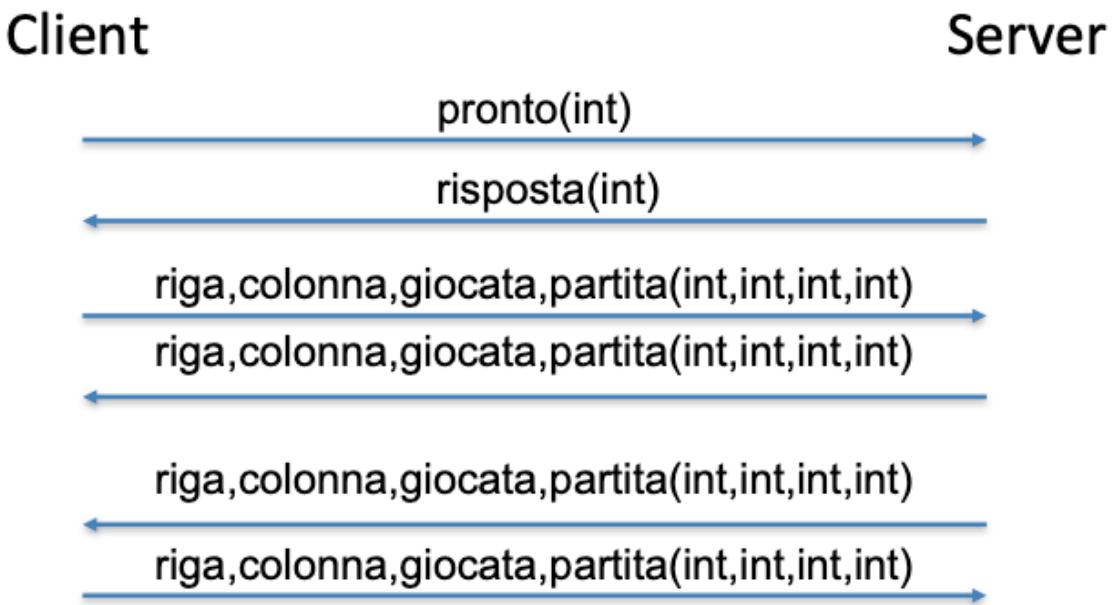
Per implementare le regole decido di definire uno o più pacchetti di livello applicazione. In fasi diverse posso usare l'uno l'altro pacchetto.

L'importante è che client invia pack di tipo ***n*** e server risponde con pack di tipo ***n***.

C'è però un caso particolare: ci sono il campo ***payload*** che può contenere una quantità variabile di byte. Quella quantità di byte potrebbe essere abbastanza grande. Se decidiamo di avere un pacchetto del tipo intero-intero-array di byte, pur avendo definito il pacchetto, devo fare un invio preventivo della DIM di esso perché potrei metterci infiniti byte e il ricevente non sa quanti sono stati messi.

Il passo successivo dopo aver definito il pacchetto applicazione, importantissimo per

documentare il protocollo applicazione è creare un *sequence diagram* in cui si definisce passo passo chi scrive, cosa scrive e chi legge cosa deve leggere.



Abbiamo una linea temporale: il client invia un intero (per sincronizzarsi) appena ha finito di definire il suo campo di gioco e attende la risposta del server che verrà inviato quando ha posizionato le sue navi.

A questo punto client e server hanno il campo di gioco pronto. Il client attacca prima e invia riga, colonna, giocata e partita.

Il server attende e quando riceve risponde con riga colonna giocata e partita.

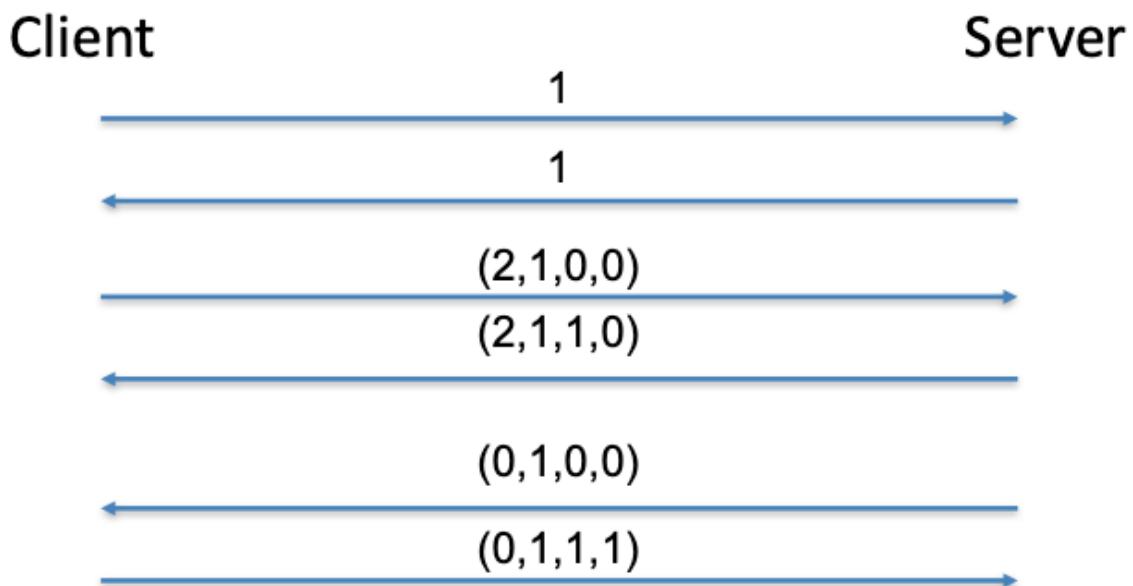
In questo schema sto definendo il client cosa invia e il server cosa deve leggere e come risponde.

## **DOMANDA ESAME GLI SCHEMI INOLTRE LI VUOLE NELLA RELAZIONE.**

Questo deve essere fatto per ogni regola definita nel protocollo applicazione.

Questo schema serve per dire a una persona che vuole implementarsi il server. A prescindere da chi ha implementato client e server, se rispetto le regole la comunicazione va a buon fine.

Alla fine possiamo mostrare un'istanza del protocollo mettendo valori reali:



## Ricapitolando:

- Definire le entità e come sono collegate
- Definiamo il protocollo applicazione: regole che permettono al client e al server di scambiare richiesta e risposta
- Specificare i valori per implementare quelle regole.
- Se definiamo un pacchetto applicazione va dimostrato graficamente, qual è la struttura e la DIM dei vari campi.
- Mostriamo la struttura e l'implementazione del progetto e del pacchetto.

## Lez 9

Oggi riprendiamo l'esercizio del conta stringhe.

Abbiamo visto già un problema associato all'esercizio (*Full Read/Write*) e abbiamo visto nella lezione precedente come ovviare.

Un altro problema che sorge quando realizziamo l'esercizio è che può essere necessario dover monitorare più descrittori contemporaneamente. Il contastringhe doveva monitorare 2 descrittori:

- Lo **stdin** per l'input dell'user
- Quello del socket

I nostri client, soprattutto quelli interattivi, hanno necessità di monitorare almeno 2 descrittori. **Perché è necessario monitorare questi descrittori?**<sup>1</sup>

- Stdin perché il nostro client deve essere responsive verso l'user.
- Dall'altra parte dobbiamo essere collegati e responsive verso il server. Se il server ci invia un messaggio e cade la connessione, il client

---

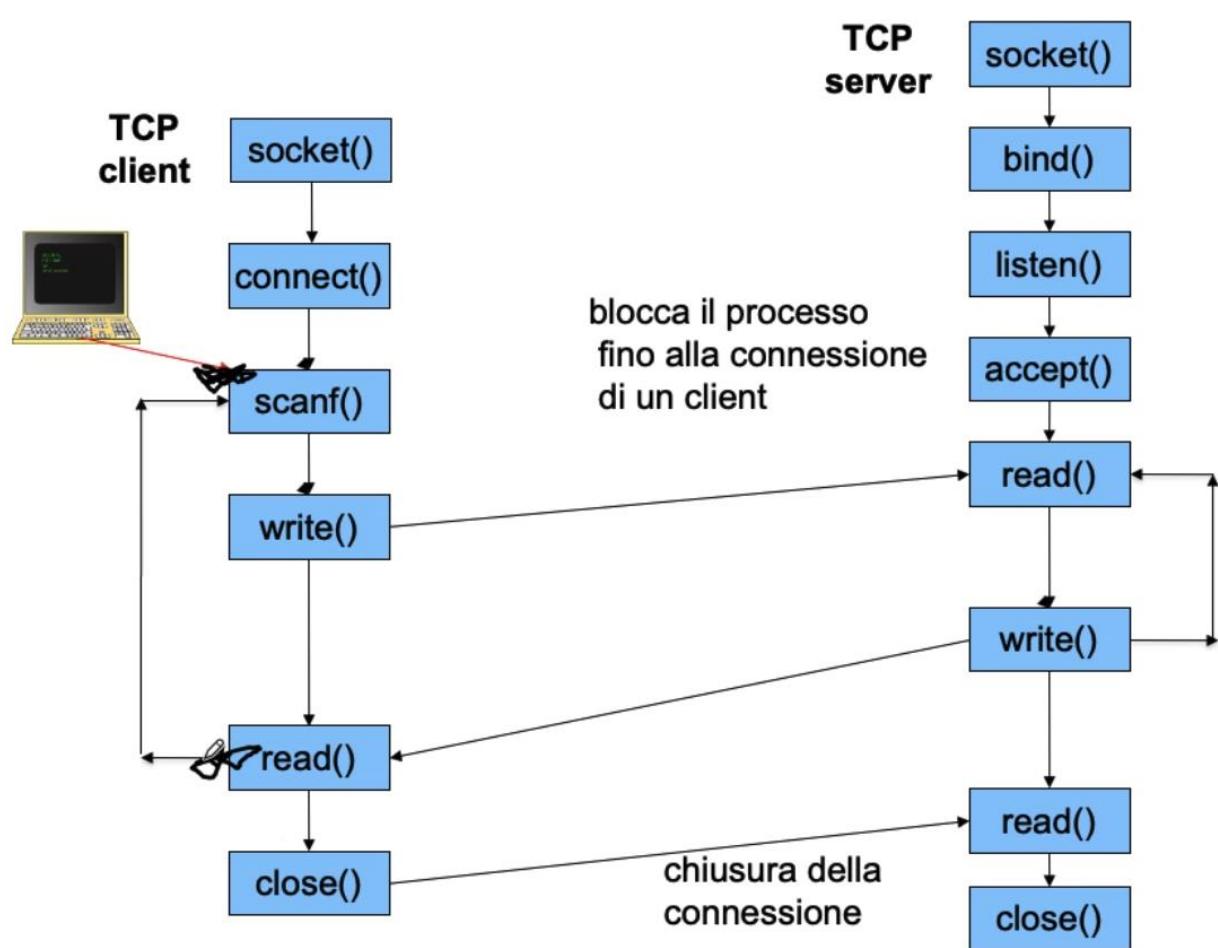
<sup>1</sup> Esame

deve rilevare l'evento e comportarsi di conseguenza.

Il problema è che un processo deve essere in grado di poter monitorare 2 descrittori contemporaneamente, 2 per ora.

---

Se prendiamo client e server connessi, il problema è che il nostro processo client deve bloccarsi contemporaneamente sulla *scanf* dello *stdin* e bloccarsi sulla *read* connessa col server.



Sappiamo che la ***read*** è ***bloccante***. Se il processo è bloccante sulla scanf, non potrà bloccarsi sulla read contemporaneamente.

Quindi non potrà mai sapere se il server ha inviato qualcosa al client. Se la connessione cade non se ne può accorgere.

Potrebbe bloccarsi sulla read per conoscere la connessione ma finché non succede qualcosa sul quel canale non pu leggere l'input.

Oggi vogliamo vedere quali sono gli approcci che consentono di monitorare gli eventi che si verificano su 2 descrittori.

---

I modelli che vedremo sono:

- Bloccante
  - Non bloccante
  - I/O multiplex
  - I/O attivato da segnali
  - I/O asincrono
- 

## FASI DELL'INPUT

Si distinguono 2 fasi per le operazioni di input:

- 1- Attesa per disponibilità dei dati

## 2- Copia dei dati dalla memoria del kernel al processo.

Quando un processo user chiede una **ssyscall**, l'esecuzione del servizio avviene in ***kernel\_space***. Ci deve essere un momento in cui l'esecuzione passa **da *user\_space* a *kernel\_space***, come se fosse un cambio di priorità.

In *user space* lavoriamo coi permessi *user* mentre in *kernel* con permessi *monitor*.

Quando faccio una *read* cosa gli passo come argomenti: descrittore, buffer, size buffer; il buffer che gli passiamo è allocato in *user space*, il descrittore è un collegamento con un dispositivo che è gestito in *kernel\_space*. Cosa chiedo con la *read* al *SO*? Recupera i dati collegati al dispositivo fd, copia i dati in una struttura in ***kernel\_space*** e una volta disponibili copiali nel buffer che si trova in ***user\_space***.

La prima fase quindi è l'attesa che i dati siano pronti e disponibili in *kernel\_space*. La seconda fase: dopo che sono pronti, il SO li copia nel buffer e la *read* ritorna, il valore di ritorno è il numero di byte inseriti in *buffer*.

---

## I/O BLOCCANTE

Il modello di **default** è quello bloccante. Quando mi blocco su una read il processo va in **waiting**: va in attesa di diventare **ready**.

Faccio la read in *userspace* e resto bloccato finchè nonsono disponibili i dati in *kernelspace*, poi il kernel li copia in userspace nel buffer e ritorna.

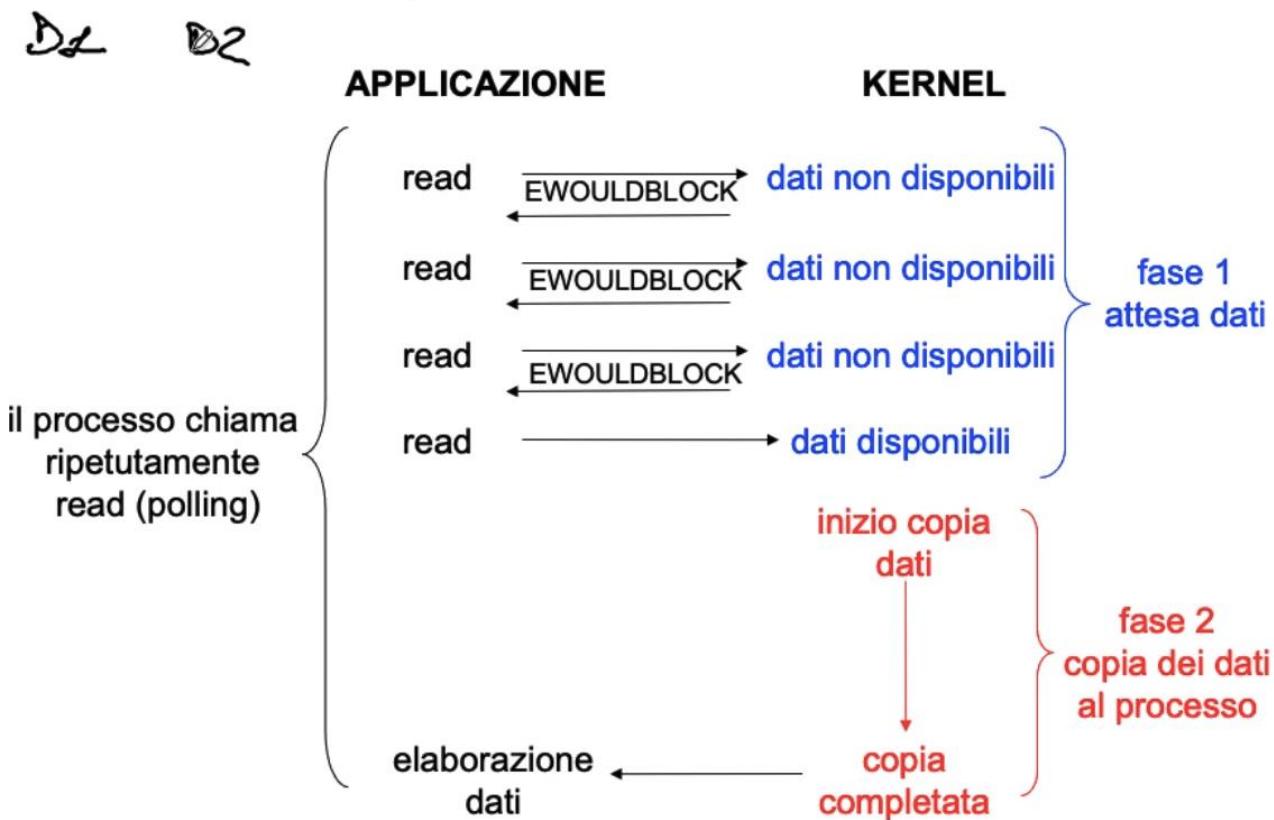
**Altra criticità** dell' I/O bloccante è che può bloccarsi o sull'uno o sull'altro descrittore: posso impostare il descrittore in modalità non bloccante.

## I/O NON BLOCCANTE

La mia read ritorna subito, se non ci sono dati disponibili ritorna **EWOULDBLOCK** e inizia a ciclale all'infinito (**polling**), quando il valore di ritorno è diverso da **EWOULDBLOCK** significa che i dati sono stati copiati nel buffer. Il **polling genera busy waiting**. Il processo viene schedulato e usa risorse inutilmente finchè non escono dei dati. Questo approccio può essere usato su dei sistemi dedicati che devono solamente leggere più descrittori senza bloccarsi.

Il vantaggio è che la risposta è immediata appena ci sono dati.

## I/O non bloccante



L'altra possibilità è I/O MULTIPLEX

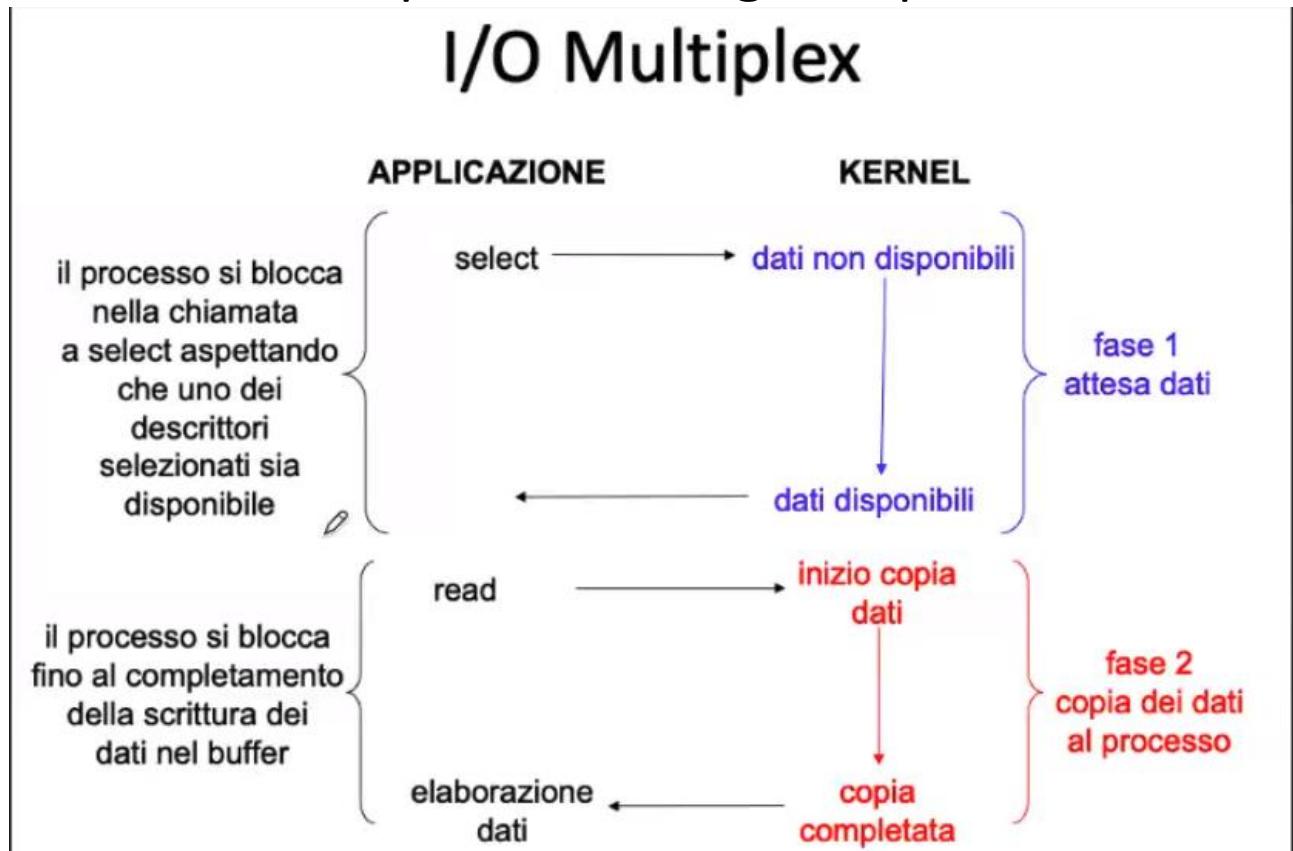
Consiste nella possibilità di registrare attraverso una *ssyscall* (*select*) più descrittori contemporaneamente.

Delego al SO di monitorare i descrittori. La *select* è bloccante. Il mio processo si blocca su una *ssyscall* ma la *ssyscall* monitora più descrittori. Nel momento in cui ci sono dati pronti sui descrittori la *select* ritorna.

Il processo esegue la select delegando al SO quali vuole monitorare. Il SO monitora e aspetta dati su tutti i descrittori, quando su uno o più sono pronti dei dati la select ritorna e di fatto la select gestisce solo la prima fase: l'attesa che i dati siano disponibili in *kernel space*.

Quando ritorna la select significa che i dati sono pronti in *kernel space* se faccio una read non devo attendere la prima fase ma solo la copia da *kernel* a *user* e perché sono già disponibili.

## I/O Multiplex



## I/O CONTROLLATO DA SEGNALI

Quando ci sono dati pronti, il processo riceve *SIG\_IO*, registro un handler per il segnale e quando arrivano id ati, faccio la read e ho risolto in modo asincrona la gestione dell'attesa.

Imposto handler, il processo prosegue l'esecuzione, quando i dati sono disponibili il processo riceve il segnale, va in esecuzione l'handler e viene fatta la read.

Poiché ho ricevuto *SIG\_IO* quando i dati sono pronti in kernel space, fare la read non sarà bloccanti e si prosegue con l'esecuzione.

---

## I/O ASINCRONO

***aio\_read()*** prende una struttura come argomento con i parametri necessari per effettuare la scrittura:

- La chiamata *aio\_read()* consente di
  - leggere *aiocbp->aio\_nbytes*
  - dal desrittore *aiocbp->aio\_fildes*
  - a partire dall' offset *aiocbp->aio\_offset*
  - nel buffer *aiocbp->aio\_buf*
- La chiamata ritorna dopo che la richiesta è stata accodata nella coda del descrittore

Do al SO tutte queste info e il processo può

continuare nell'esecuzione. È simile a I/O con segnali.

Richiamo *aio\_read()*, attesa dei dati disponibili, copia in user space, emissione segnale specificato in *aio\_read* ed elaborazione dei dati.

Il risultato può essere letto dalla funzione *aio\_return()*. All'interno del buffer *aio\_buf* troverò i dati copiati da kernel a user space.

Volendo posso fare anche polling andando a ciclare su *aio\_error()*. *Aio\_error()* finchè restituisce **EINPROGRESS** vuol dire che non ci sono dati pronti (equivale a **EWOULDBLOCK** dell'I/O non bloccante).

Quindi I/O asincrono è come i segnali ma al contrario entrambe le fasi sono gestite dal SO.

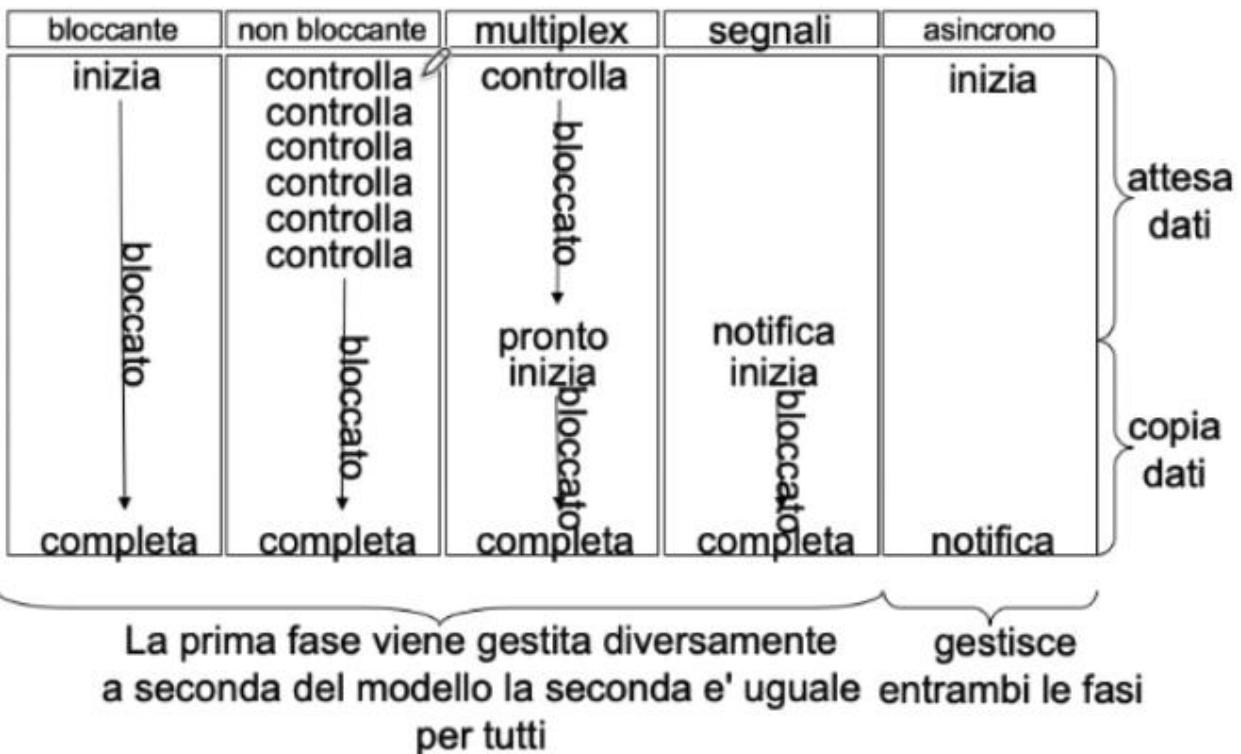
---

## THREAD ED I/O

Un altro modo è usare i thread. Ogni thread ha un descrittore da monitorare e rimane bloccato sul suo descrittore.

---

# RIEPILOGO



## SELECT

È una syscall. Chiedo al SO di monitorare un certo numero di descrittori. I descrittori posso monitorarli per la lettura (quando un insieme di descrittori è pronto per essere letto, i dati sono pronti in kernel space e posso fare una read per portarli in user space) oppure per scrittura.

*Ricordiamo che la write è bloccante [i dati devono passare da user a kernel space e una volta fatto devono essere scritti sul descrittore e finire in un buffer in kernelspace. La write si*

*blocca quando il buffer è pieno e la prossima scrittura deve attendere che si svuoti]*

(la select si sblocca quando ci sarà spazio nel buffer kernel space e la write non sarà bloccante).

Altra possibilità è il monitoraggio di un'eccezione (non un errore). Le eccezioni sono situazioni particolari per i descrittori di socket si tratta di dati **OOB**(out of band) cioè ci sono protocolli, come il **TCP**, dove l'invio e la ricezione dei pacchetti avviene fuori banda. Nel momento in cui ci sono questi dati pronti, se ho registrato un descrittore nella select per l'eccezione la select si sblocca.

Altra possibilità è quella di mettere un timeout alla select: registro un certo numero di descrittori da monitorare e la select si blocca, se imposto un timeout, scaduto, la select si sblocca e ritorna. Il mio protocollo potrebbe prevedere che posso aspettare max 10 secondi...

---

# file descriptor

- Il file descriptor identifica il file nelle operazioni di lettura e scrittura
- Per convenzione Unix associa:
  - 0 standard input (STDIN\_FILENO)
  - 1 standard output (STDOUT\_FILENO)
  - 2 standard error (STDERR\_FILENO)
- Per conoscere il file descriptor associato ad un file stream (FILE \*) si utilizza la funzione `fileno`

A fileno passo il puntatore al file stream e restituisce il descrittore associato al flusso.

La select restituisce un intero, -1 se errore, 0 se è scaduto il timeout, altrimenti restituisce il numero di descrittori pronti tra quelli che ho chiesto di monitorare.

```
int select(int maxfdp1, fd_set *readset, fd_set
*writeset, fd_set *exceptset, const struct
timeval *timeout);
```

## TIMEOUT

Posso passare il puntatore alla struttura timeval che permette di definire un tempo in termini di secondi e microsecondi. Imposto un tempo e lo passo come puntatore. La select resta bloccata al più per quel tempo, se non ci sono dati e passa il tempo la select restituisce 0.

## GLI INSIEME DI DESCRITTORI

I parametri 2,3,4 della select specificano gli insieme di descrittori da monitorare:

- Readset: pronti per la lettura
- Writeset: pronti per la scrittura
- Exceptionset: dati urgenti OOB

## ESEMPIO

Immaginiamo che il mio processo voglia monitorare il descrittore 3, 4, 5, 12. Il processo ha aperto quindi i descrittori. Immaginiamo che su 3,4 voglio effettuare delle letture. Questi due descrittori li monitoro per sapere se ci sono dati pronti da leggere. 5 e 12 li monitoro in scrittura (il processo ha dati pronti per essere scritti e vuole certezza che quando chiamo la write non si

blocchi). Ho un unico processo che deve leggere da 2 descrittori e scrivere sugli altri 2 senza bloccarsi.

Cosa faccio?

Dico alla select di monitorare in lettura 3 e 4 e in scrittura 5 e 12.

Il processo chiama la select() e passa le info di cui sopra.

Immaginiamo che arrivino dei dati sul descrittore 3, significa che il buffer in kernel space contiene dei dati pronti e immaginiamo che il buffer collegato a 12 in scrittura è vuoto e quindi posso scriverci su. Su 5 ho il buff pieno e su 4 non ci sono dati e non ho nulla da leggere.

Quanti descrittori sono pronti rispetto a quelli chiesti? 2: uno in lettura e uno in scrittura.

La select restituisce il numero di descrittori pronti, quindi 2.

Se avessi messo un timeout di 10 e nessuno fosse stato pronto avrebbe restituito 0.

L'ultimo parametro è un puntatore alla struttura timeva, se gli passo null, la select è bloccante, se gli passo zero la select ritorna subito.

La select quidni restituisce 2: due dei 4 descrittori quindi sono pronti, ora devo andare ad analizzare tutti i descrittori che avevo chiesto di monitorare verificando se quel descrittore è pronto o no: se è pronto significa che ci sono dati pronti per essere letti, se in lettura e quindi su 3 posso fare una read che di fatto non sarà bloccante (attendo tempo di copia da kernel a user space solo), su 4 non mi blocco poiché non ci sono dati pronti (ce lo dice il kernel), su 5 non faccio write perché non pronto mentre su 12 che è pronto posso fare la write senza bloccarmi.

Da questo schema cosa ricaviamo? In input devo fornire alla select i descrittori, in output la select restituisce il numero di descrittori e quali sono pronti. Solo con queste 2 info posso vedere quali di quelli che avevo chiesto di monitorare sono pronti per essere letti o scritti.

FINE ESEMPIO.

---

Vediamo come passare alla select i descrittori da monitorare.

Per specificare questi descrittori uso

*Readset, writeset ed exceptionset, sono di tipo puntatore a fd\_set.*

I descrittori sono numeri interi, invece di memorizzare il numero intero a 16 o 32 bit, l'`fd_set` crea degli array di bit in cui vado a impostare a 1 la posizione corrispondente al numero del descrittore che voglio monitorare. Se voglio monitorare il descrittore zero, il bit 0 va a 1. Se voglio monitorare il descrittore 3 il bit 3 va a 1. Esempio:

- Per specificare l'insieme  $\{0,3,5\}$

Ovviamente non lavoriamo con singoli bit, ci sono macro che ci permettono di impostare i bit corrispondenti:

- FD\_ZERO mette tutti i bit a zero
  - FD\_SET mette il bit corrispondente a fd a 1.
  - FD\_CLR azzerà il bit corrispondente a fd.
  - FD\_ISSET mi permette di sapere se il bit in posizione fd è 0 o 1. Questa macro serve per

vedere se dopo che la select si è bloccata,  
vedi registrazione... (31.30 rec teams).

- Esempi d'uso delle macro per manipolare le variabile di tipo fd\_set

`fd_set readset;`

`FD_ZERO( &readset );` inizializza a 0 tutti i bit

`FD_SET ( 1, &readset ); /* 1 appartiene all'insieme */`

`FD_SET ( 4, &readset ); /* 4 appartiene all'insieme */`

`FD_ISSET ( 4, &readset ); /* verifica che 4 appartiene all'insieme e restituisce un valore non nullo */`

`FD_ISSET ( 3, &readset ); /* verifica che 3 appartiene all'insieme e restituisce zero */`

- Fd\_set nell'esempio quindi imposta a 1 il bit 1.
- Fd\_set(4 &read\_set) imposta a 1 il bit in posizione 4. Sto dicendo monitora in lettura il descrittore 4.
- Fd\_isset(4,&readset): verifica se 4 appartiene all'insieme, in questo caso torna 1
- Se faccio fd\_isset su 3 restituisce 0 perché non appartiene all'insieme dell'esempio.

2,3,4 argomento della select sono argomenti di *input/output*.

L'ultimo argomento è *maxfdp1* che è il numero massimo di descrittori da monitorare.

Prendendo l'esempio di prima, il descrittore più grande da monitorare era 12; *maxfdp1* quindi dovrà monitorare 13 descrittori (partono da zero). Questo primo parametro serve per ridurre le risorse che usa la select. Se come *maxfdp1* mettessi 20, so che dal descrittore 12 al 19 non interessa quello che succede.

---

## DESCRITTORI PRONTI IN LETTURA

Sono pronti quando c'è almeno un byte da leggere in kernel space.

Un descrittore su cui il server è in attesa di richiesta di connessione deve essere monitorato in lettura, quando arriva una richiesta di connessione, il server si sblocca. Anche *listenfd*, descrittore di ascolto per le connessioni, va monitorato in lettura

## PRONTI IN SCRITTURA

Hanno spazio in kernelspace per scrivere, i descrittori su cui posso fare una connect che va a buon fine e inoltre, si sbocca in scrittura quando il descrittore è stato chiuso e quando faccio una write su un descrittore chiuso ricevo una SIGPIPE

---

---

## Commento al codice

- While
- Fdzero mette a zero tutti i bit
- Imposto a 1 lo stdin\_fileno (lo voglio monitorare)
- Lo stesso lo faccio per il bit in posizione sockfd.
- Vado a vedere quale dei due descrittori è maggiore: se sockfd>stdi\_fileno allora la variabile maxd=sockfd+1 altrimenti stdin\_fileno+1

- Blocco il processo finchè non ci sono dati pronti su stdin o sul socket o su entrambi
- Se la select > 0 errore
- Poiché l'ultimo argomento è null non può restituire zero quindi rimane bloccata finchè non ci sono dati
- Con FDISSET verifico se ci sono dati su sockfd oppure se ci sono stdi\_fileno e nel caso li leggo.

Abbiamo risolto il problema dell'attesa su più descrittori.

# Schema di un'applicazione che utilizza l'I/O Multiplex

```
while (1) {
 FD_ZERO (&set);
 FD_SET (STDIN_FILENO,&set);
 FD_SET (sockfd,&set);
 if (sockfd > STDIN_FILENO)
 maxd = sockfd + 1;
 else
 maxd = STDIN_FILENO + 1;
 if(select(maxd, &set, NULL, NULL, NULL) < 0)
 exit(1);
 if(FD_ISSET(sockfd, &set)) {
 ... /*leggi da sockfd */
 }
 if(FD_ISSET(STDIN_FILENO, &set)) {
 ... /*leggi da standard input */
 }
}
```

T



---

Sulla piattaforma c'è il codice. Lato server non cambia nulla

Lato client:

creo socket, valorizzo struttura, faccio la connect, chiamo ClientEcho, gli passo stdin, e il socket che ho appena connesso.

Vado a calcolare qual è il massimo tra il descrittore associato a filein e il socket appena connesso, sarà maxfd.

Entro nel while infinito, all'inizio del ciclo dove monitoro tramite la select devono azzerare i bit con FD\_ZERO, dopodichè imposto lo standard input con FD\_SET e faccio la select(), mettendo maxfd come primo argomento.

Quando la select ritorna, il valore di ritorno non lo prendo perché non serve su 2, faccio i due if, per recuperare lo stato del descrittore uso FD\_ISSET su fd\_set. Se lo stdin è pronto in lettura faccio la fgets o scanf e la scrivo con la fullwrite sul descrittore.

Con un altro if verifico se ci sono dati da leggere sul descrittore usando fdset facendo poi una read sul descrittore, se > 0 errore, se = 0 è stato chiuso oppure il server ha mandato dei dati e li scrivo sullo stdout.

A questo schema che c'è sulla piattaforma possiamo aggiugere:

- Al posto della read possiamo usare fullread come abbiamo visto la scorsa volta.
- Dopodichè un'altra modifica (importante per progetto quando si usa I/O multiplex) se un descrittore è pronto per essere letto poi la FULLWrite si lascia non monitorata: la

fullwrite potrebbe bloccarsi e il processo si bloccherebbe. Provare a immaginare come modificare il codice in modo che solo se ci sono dati che possono essere scritti, cioè come monitorare il socket in scrittura.

Riga 78: se il descrittore del socket non è pronto in scrittura potrei perdere dati in lettura.

## Lez 10

Abbiamo visto come usare *select* per monitorare più descrittori contemporaneamente.

**La select restituisce quanti descrittori sono pronti.**<sup>2</sup> Il primo argomento della select è il massimo descrittore +1 che serve a limitare il numero di monitoraggi contemporanei della select, 2,3,4 argomento sono di tipo *fd\_set*. Sono passati come puntatori perché sono parametri di *Input Output*. L'ultimo parametro è una struttura *timeval* e se è settata a NULL significa che è bloccante: finchè non si verifica un evento sui descrittori, la select è bloccante, se metto 0 sto facendo polling sui descrittori da monitorare. La select restituisce 0 se non ci sono descrittori pronti o diverso da zero se ci sono.

- Perché viene passato come puntatore l'ultimo parametro (struttura *timeval*)?
  - Perché all'interno della struttura, in output ci troverò il tempo rimanente fino al massimo. Se lo registro per 10 sec e dopo 5 sec si verifica un evento, nella struttura

---

<sup>2</sup> select

timeval avrà 5 sec rimanenti. Se la select ritorna a zero vuol dire che il tempo è scaduto e non si è verificato nessun evento.

---

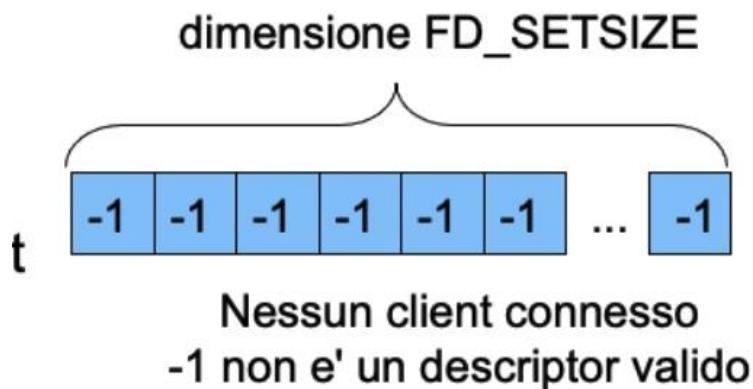
Oggi vedremo 2 argomenti: il primo argomento è capire come possiamo sfruttare la *select* per implementare un server iterativo, cioè un unico processo che gestisce sia i client connesso sia il descrittore su cui andiamo ad accettare le richieste di connessione. Il server concorrente aveva un processo padre il cui compito era monitorare *listenfd* (*quello dove si accettavano connessioni*). *Ogni volta che si connetteva un client, si faceva una fork che gestiva la comunicazione col client e il padre torna a bloccarsi sulla accept.*

In questo schema invece realizziamo un server iterativo dove un solo processo fa entrambe le cose autonomamente.

Per fare questo usiamo la `select()`.

Oltre all'*fd\_set* usiamo un array di interi di appoggio dove segniamo quali descrittori sono aperti.

Inizialmente non ho client connessi. Il mio array di appoggio conterrà un valore etichetta -1 che sta a significare che nessuna posizione dell'array ha un descrittore valido.



Abbiamo poi il descrittore ***Fd\_set*** con i bit tutti a 0.

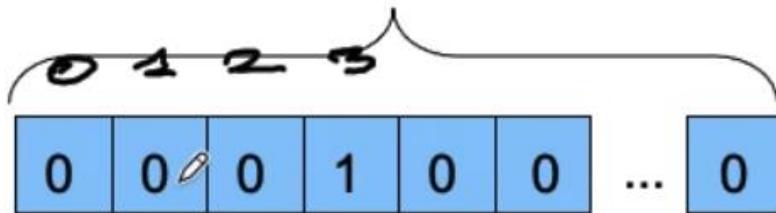
La prima cosa che faccio nel *server*, creo il *socket* di ascolto, faccio la *socket*, valorizzo le strutture, richiamo la *bind*, la *listen* e con la *accept* mi blocco su *listenfd*.

Immaginiamo che 0,1,2 sono ancora aperti e il primo descritore libero è il 3.

Il nostro server registra in posizione 3 con la ***fd\_isset*** un 1, la select si blocca e monitora solo questo descrittore. Questo descrittore va

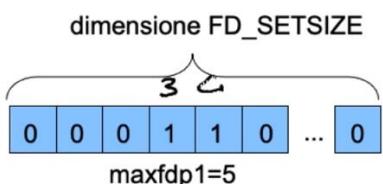
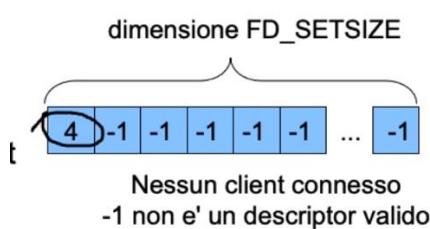
registrato in lettura poiché ci sono richieste di connessione verso di lui.

### dimensione FD\_SETSIZE



l'unico descrittore da monitorare e' quello in ascolto ovvero il 3  
maxfdp1=4

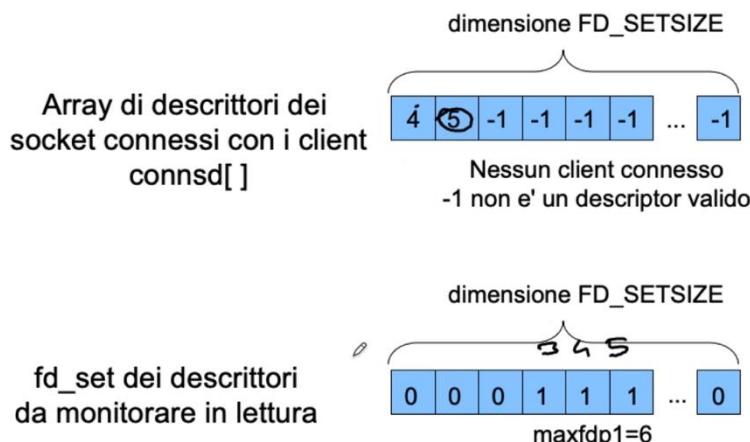
Nel momento in cui arriva il primo client, la accept restituisce un nuovo descrittore che è quello connesso col client. Questo descrittore, 4, viene registrato sia nell'array di appoggio che nella ***fd\_set***.



Quando passo fd\_set alla select →  
select(5,&fd,NULL,NULL,NULL);

avendolo passato come secondo argomento, la select si sbloccherà quando o il client sta mandando dei dati..

immaginiamo che arrivi un nuovo client, quindi la accept ci restituisce un nuovo descrittore, 5. Registro nella stessa fd set anche il 5 e nell'array di appoggio oltre a 4 aggiungo il 5.



Mi blocco sulla `select`, e gli eventi che possono generare uno sblocco sono:

- Il 3 fa nuova richiesta
- Il 4 sta inviando dati
- Il 5 sta inviando dati

Immaginiamo che il client 4 chiuda la connessione. Cosa devo fare? Il bit corrispondente nell'`fd_set` va a zero e il 4 che avevo messo nell'array lo devo cancellare. Il

nostro server sarà sempre in attesa poiché è in un loop dove accetta e gestisce le richieste dei client connessi.

- Quindi a cosa serve l'array di appoggio?
  1. Se non salvo il descrittore non posso interagire col client. Se ho  $n$  client connessi devo avere tutti e  $n$  salvati da qualche parte.
  2. Abbiamo detto che  $\&fd$  è un paramentro di input output. Immaginiamo che in  $fd$  registro 3,4,5 mettendo 1,1,1 in input, significa che li voglio monitorare. La select si sblocca e in output il 3 lo trovo a 1 e 4,5 li trovo a 0, cosa significa? Che il 3 è pronto e quindi n=1. Dopo aver accettato la nuova richiesta non devo fare più nulla, perché la select aveva solo 1 descrittore attivo. In quanto server devo monitorare 4,5 e il nuovo arrivato. All'inizio del ciclo devo aver scritto quali sono i descrittori aperti per poterli reimpostare nell' $fd$ . All'inizio di ogni ciclo mette a 1 tutti quelli da monitorare, fa

*la select che modifica l'fd\_set e quando  
rinizia il ciclo deve reimpostare i descrittori.*

Come array di appoggio, per tenere traccia dei descrittori pronti, potrei usare un *fd\_master* dove mi segno tutti quelli da monitorare e all'inizio di ogni ciclo faccio una copia e la mando alla *select()*.

---

## STRUTTURA DEL CODICE SERVER ITERATIVO

- Maxfd è posto a listenfd cioè il descrittore di ascolto.
- In posizione maxfd dell'array fdopen metto 1 che significa che è aperto. È più efficiente che scorrere tutto l'array.
- Faccio il ciclo while infinito
- Nel primo for faccio la scansione di fd\_open, se l'iesimo descrittore è aperto (diverso da zero) allora lo imposto nell'fdset.
- Finito del for all'interno del fset ho tutti i descrittori da monitorare.

- Mi blocco sulla select che restituisce il numero di descrittori pronti in lettura.
- Se il descrittore di ascolto è pronto in lettura allora decremento  $n$  perché ho trovato il primo pronto e vado a fare la accept
- Imposto a 1 il descrittore corrispondente e vedo se il nuovo descrittore fd è maggiore edl vecchio massimo. Questo perché il processo, man mano che i descrittori si aprono e si chiudono, ptrebbe restituire un descrittore minore rispetto al precedente, quindi se il descrittore del nuovo client è il nuovo massimo lo registro come *max\_fd*.
- Imposto una variabile *i* a *listenfd* che è il primo descrittore che ho registrato.
- Inizio un ciclo su *n* che è il numero di descrittori pronti.
- Se ho 1 solo descristtore pronto decremento *n* e non entro nel while. Se invece oltre a quello ho altri descrittori pronti significa che qualche client connesso ha inviato una richiesta, quidni devo iniziare un ciclo per vedere quale dei descrittori registrati è effettivamente pronto.

- Incremento i, indice per sovrizzare l'array di fd.
  - Se  $fd\_open[i]=0$  significa che l'isimo descrittore non è aperto, allora passo all'indice successivo
  - Altrimenti vedo con  $fd\_isset$  se è impostato a 1 cioè c'è qualcosa da leggere. A quel punto ho trovato un altro descrittore e decremento n e vado a leggere la richiesta.
  - Se  $nread < 0$  errore
  - Se  $nread == 0$  significa che il client ha chiuso la connessione. Se ha chiuso vuol dire che il descrittore in posizione i non deve essere più monitorato quindi lo imposto a zero.
  - Vado poi a vedere se  $i == max\_fd$  vuol dire che l'isimo descrittore era il massimo quindi ora devo trovare il secondo massimo
  - Decremento i finchè  $fd\_open[i]=0$ : significa che parto dal descrittore i e finchè ho 0 decremento, appena trovo un descrittore a 1, quello sarà il descrittore maggiore che devo monitorare e lo metto in  $max\_fd$ .
  - Se tutto ciò non succede allora invio una risposta con  $fullWrite$
-

## Nuovo argomento

### RETI P2P

Le reti p2p si differenziano rispetto all'architettura client/server per il ruolo che hanno all'interno dell'architettura.

In client/server le architetture sono sbilanciate perché uno chiede e l'altro serve. In particolare notiamo lo sbilanciamento quando il server cade: il servizio non potrà essere più utilizzato.

L'alternativa è usare un modello p2p dove tutte le entità hanno la stessa responsabilità.

Ogni nodo deve funzionare sia da client che da server.

Ogni nodo fornisce servizio agli altri nodi e richiede servizi agli altri nodi. In un'architettura del genere, se uno dei nodi dovesse cadere, il servizio potrà essere gestito dagli altri nodi rimanenti.

Ad esempio: il nodo A chiede servizio sia a B che C, se B cade, il nodo A chiede a C. A sua volta deve però fornire il servizio. L'idea delle reti p2p

è quella di sfruttare la ridondanza delle risorse per offrire una maggiore qualità del servizio. Questo ovviamente ha un costo.

---

## CARATTERISTICHE DELLE RETI P2P

Quindi ogni nodo funziona sia come client che come server: il processo dovrà avere al suo interno sia la parte server che client. Questa applicazione consente di sfruttare al massimo le risorse che si trovano ai margini della rete cioè gli host degli user poco utilizzate.

Una tipica applicazione p2p è *torrent* cioè scambio di file. Qual è la risorsa che ognuno di noi che partecipa a questa rete mette a disposizione della rete torrent? Spazio sul disco. L'host, come nodo terminale della rete, ha risorse libere che mette a disposizione degli altri nodi. La caratteristica di questi nodi ai margini della rete è che la loro connettività è intermittente: quando chiudo la macchina la mia risorsa non è più disponibile. L'infrastruttura non è controllabile e le componenti hw non sono affidabili.

---

Un altro problema da risolvere riguarda la connettività dei nodi:

ho più nodi che si trovano al margine della rete internet. Dall'altra parte ho un'architettura client/server dove il client si rivolge al server. Come abbiamo detto, il client per ottenere il servizio dal server deve conoscere IP/PORTA del server, il server invece non ne ha bisogno.

Nell'architettura p2p, ogni nodo è sia client che server. Come raggiungo un nodo? Sempre con IP-Porta.

Per accedere all'info che si trova distribuita su tutti i nodi, dovrei conoscere ip e porta di tutti i nodi che fanno parte della rete p2p, che potrebbero essere molti.

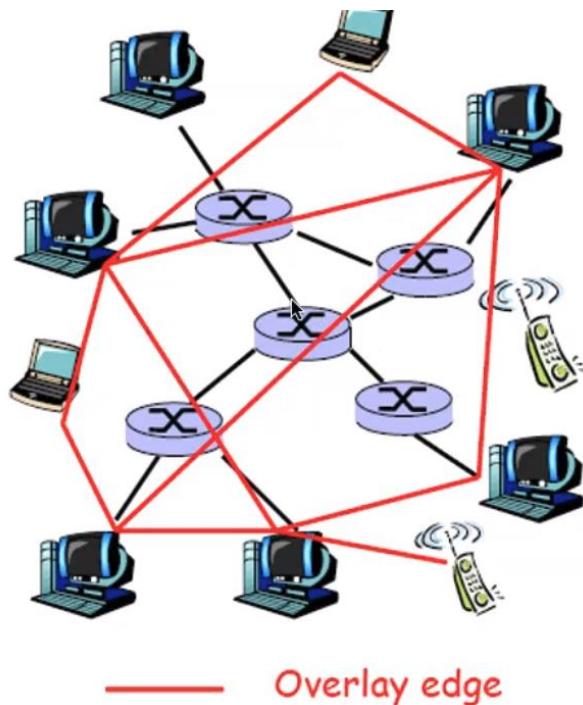
- Ho quindi un primo problema fondamentale, come faccio ad entrare all'interno della rete p2p?
- Secondo problema: immaginiamo di conoscere ip/porta degli altri nodi, a quali e quanti nodi mi collego? Se n è dell'ordine di 1 milione di nodi, è impossibile connettersi a tutti... è chiaro che questi nodi, tra di loro,

devono creare una rete che è data dalle connessioni che esistono tra ogni coppia di nodi ma che non può realizzare un grafo completamente connesso. Significa che a livello applicazione devo realizzare una rete di connessione tra i nodi che consenta in qualche modo ad ogni nodo di poter accedere alle info presenti su tutti gli altri nodi. Devo creare quindi una rete a livello applicazione che definisce quali nodi possono richiedere servizio a quali altri nodi. Questa rete è chiamata *overlay network*.

Diciamo che in qualche modo sono entrato nella rete p2p, ho definito l'overlay network, ora ho bisogno di cercare i servizi all'interno della rete, non tutti i nodi hanno tutti i file all'interno della rete (non avrebbe senso). Sorge quindi il terzo problema: come si trova l'info all'interno di queste reti? (potremmo dividere la rete in zone: il dns ogni volta che faccio una query di ricerca in una zona, automaticamente sto escludendo l'altra e registro il campo, nel caso del DNS sto cercando l'IP di un nome a dominio).

## OVERLAY NETWORK

Dobbiamo quindi per prima cosa definire un'*overlay network*. È una connessione tra peer che definiscono quale peer è connesso con altri peer. Questa connessione avviene a livello applicazione, sto sfruttando la pila ISO/OSI per realizzare una rete di connessione a livello applicazione.



Le adiacenze sono definite fisicamente dalla connessione di 2 dispositivi: c'è un mezzo fisico che collega dispositivi. Quella che vediamo in rosso è l'overlay network cioè una rete di connessione dove vengono definite nuove adiacenze che non sono determinate da una

connessione fisica ad esempio il nodo in alto a destra è collegato a quello in basso a sinistra, fisicamente non sono adiacenti ma nell'overlay network lo sono.

Si chiama overlay perché si poggia al di sopra dell'infrastruttura fisica.

---

Il passo successivo è come entrare all'interno della rete.

Ho 3 tipi di reti p2p:

- 1- Ibride e decentralizzate p2p
- 2- Puramente decentralizzata p2p
- 3- Parzialmente centralizzata p2p

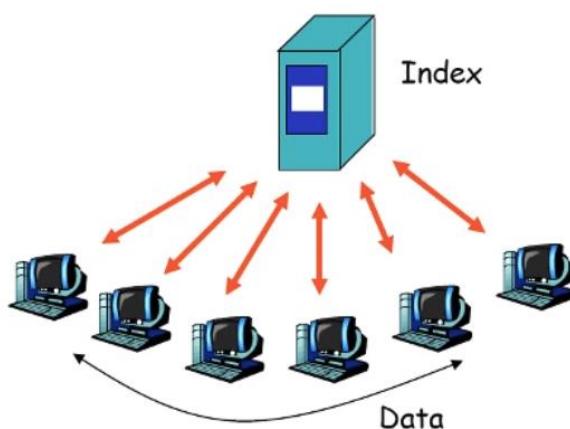
Il terzo problema che abbiamo visto è quella della struttura: come viene memorizzata l'info nella rete p2p e come deve essere ricercata l'info all'interno dell'overlaynetwork, il servizio potra essere richiesto solo ai nodi adiacenti della rete overlay.

---

## IBRIDA DECENTRALIZZATA

Ho un server che tiene traccia di tutti i peer che fanno parte dell'overlay.

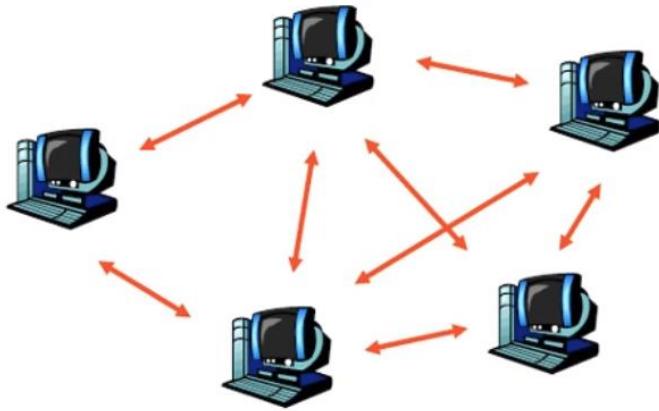
È ibrida perché c'è una parte server dove il peer si rivolge per sapere i peer connessi e la parte decentralizzata costituita dai peer con cui mi devo connettere per creare l'overlay network.



- Vantaggio: il server centrale facilita l'interazione tra peer.
  - Svantaggio: il server centrale è un *single point of failure*
- 

## PURAMENTE DECENTRALIZZATA

Come entro a far parte dell'overlay ? non c'è il server, ci sono solo peer. Almeno un peer che fa parte della rete lo devo conoscere (ip e porta).



C'è un maggiore overhead di comunicazione perché ogni peer è collegato a più peer: se uno cade, i nodi che conoscevano solo quel peer non hanno nessuna info più.

L'unico vantaggio è che tutti i nodi sono uguali.

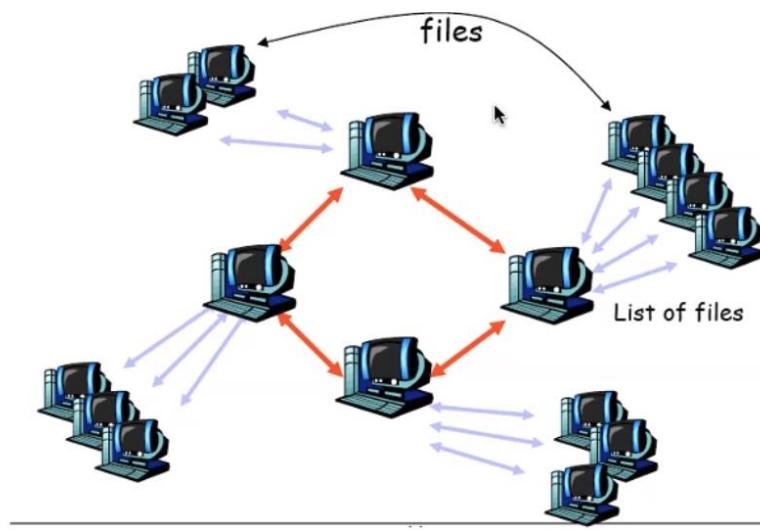
---

## PARZIALMENTE CENTRALIZZATE P2P

Invece di avere un unico server, ho un comitato di server. Questo approccio lo abbiamo visto nel DNS: nella radice non c'è una sola macchina ma un comitato. Ho molte macchine collegate tra loro cos' che se una cade, le altre possono sopperire alla mancanza.

In questo caso ho un certo numero di macchine al centro edella rete che forniscono servizi alle macchine ad esse collegate, se una dovesse

cadere, i client connessi potrebbero cercare servizio da altri server.



L'idea è quella di distribuire la responsabilità tra più server. Questo vuol dire aumentare ridondanza e quindi la QoS.

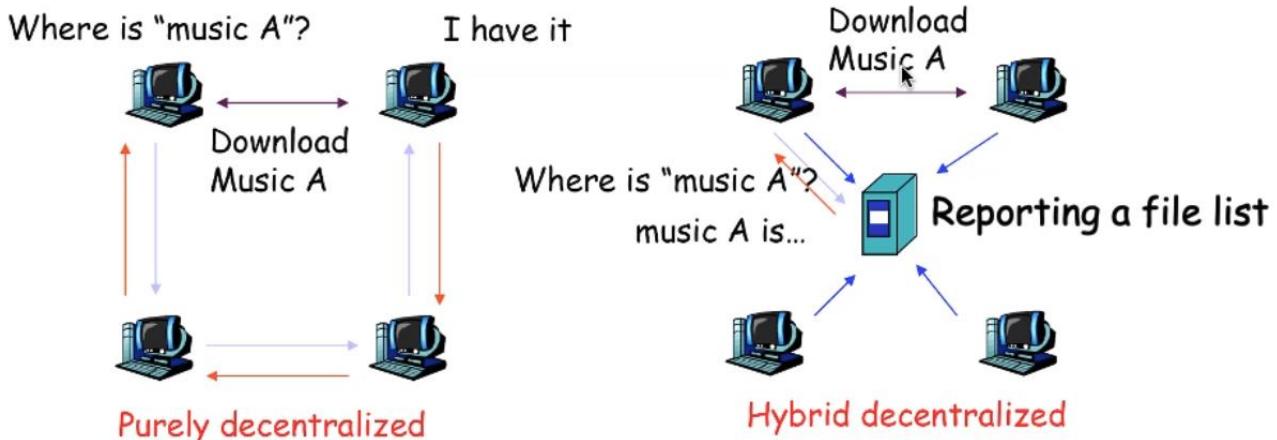
## UNSTRUCTURED P2P

Vuol dire che il peer che vuole accedere a una info lo deve chiedere a tutti.

Se ho un'architettura decentralizzata non posso fare altro che inviare richiesta a tutta la rete, se qualcuno risponde avrò il servizio.

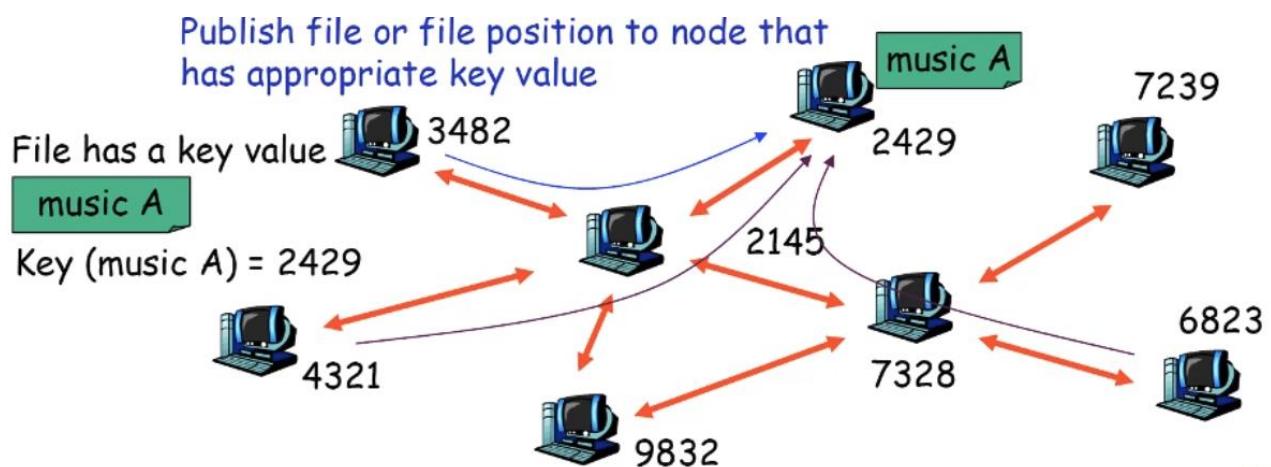
Se ho una rete ibrida decentralizzata, faccio la richiesta al server. Vuol dire che il server deve

mantenere il mapping nodo-file per tutti i file disponibili (vantaggio 1 richiesta, svantaggio se cade il server è finita).



## RETE P2P STRUTTURATA

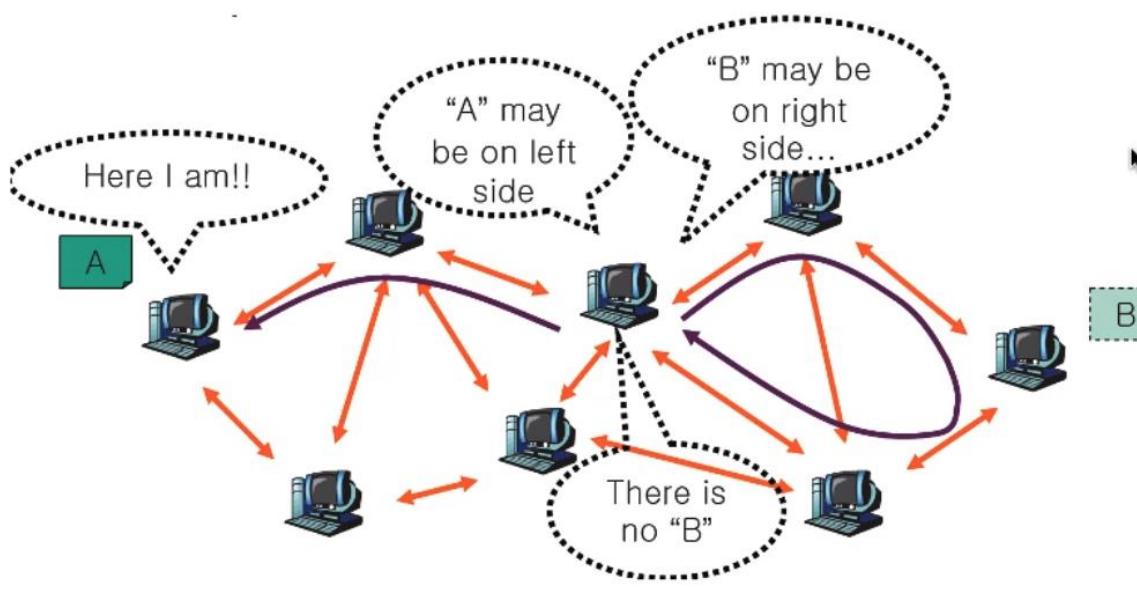
Esiste una funzione che associa la risorsa ad un nodo. La possiamo immaginare come una funzione di *hash*: chiedo un dato e mi restituisce la posizione.



## LOOSELY STRUCTURED P2P

Sul modello del DNS, faccio un'richiesta e il peer che risponde non mi dà una risposta ma mi dice in quale parte della rete si trova.

Le risposte sono vaghe ma restengono sempre più il numero di nodi a cui fare la richiesta fino ad arrivare al nodo che contiene la risorsa.



## APPLICAZIONI P2P

### File sharing

L'idea è che ogni host della rete p2p mette a disposizione una sua risorsa cioè spazio sul disco. Quello che si vuole realizzare è una content distribution network distribuita in cui ogni nodo può sia scaricare contenuti che inviarli.

Una rete bittorrent funziona quando ci sono tanti peer che partecipano alla rete in down/up loading.

L'idea è ottimizzare al massimo le risorse facendo in modo che tutti i peer possano usare una parte della banda per scaricare e una parte per fare upload. Immaginiamo i peer organizzati in un albero binario dove avrà  $n/2$  foglie e  $n/2$  nodi interni all'albero che ricevono i dati dai loro genitori e li propagano ai loro figli. Le foglie, non avendo figli, ricevono solamente.

In questa situazione sfavorevole, dobbiamo costruire un altro albero dove i ruoli sono invertiti:  $n/2$  foglie fanno upload e  $n/2$  interni diventano foglie: ci troviamo nella situazione dove ogni nodo usa metà della banda per scaricare e l'altra per fare upload. Questa è la situazione ideale dello *swarm*: *insieme dei peer che stanno accedendo alla stessa risorsa*.

BITTorent → ci sono 3 problemi da risolvere:

1. Come un peer trova contenuto
2. Come replicare il contenuto tra peer
3. Come incoraggiare i peer a condividere.

## TROVARE CONTENUTI IN BITTORENT

BT crea un file torrent che contiene un tracker cioè l'IP porta di un server che contiene le info di tutti i peer che hanno una parte o tutto il file che stiamo cercando. Inoltre contiene un elenco di *chunk* ovvero le parti in cui è diviso il contenuto.

All'interno del file torrent ho quindi queste info. Il tracker mantiene le info *dello swarm*: *tutti i peer che contengono il file con i quali devo fare up/download*.

Secondo questo schema BT implementa un'architettura ibrida.

---

## REPLICARE IL CONTENUTO

Quando uno swarm è appena formato, alcuni peer (*seeder*) devono necessariamente avere tutti i chunk.

Per evitare che tutti i peer scarichino i chunk nello stesso ordine, i peer si scambiano le liste di chunk e scelgono di scaricare prima i chunk più difficili da trovare. Questo comportamento porta in breve tempo ad un'ampia disponibilità dei chunk.

## CONDIVIDERE I CONTENUTI

Il protocollo deve favorire chi fa upload.

Chi fa solo download viene chiamato *leecher* e ha il servizio peggiore perché ostacola il funzionamento della p2p.

Ogni peer seleziona in modo casuale gli altri peer, scegliendo quelli che offrono migliori prestazioni. Più peer contribuisce al download di un contributo più potrà aspettarsi chunk in cambio.

## Lez 11 → Ultima lezione

### UDP

In una comunicazione dati Datagram il canale:

- Non è affidabile
- È condiviso
- Non preserva l'ordine delle informazioni.

Le applicazioni che usano UDP sono:

- DNS
- NFS → network file system. È un filesystem distribuito. Immaginiamo di poter montare un disco. Quando facciamo partire un SO *unix-like*, il filesystem non è collegato al modulo del kernel che lo deve gestire. Quindi il SO appena parte è tutto caricato in memoria. Per usare i file su disco, il modulo deve essere collegato alla periferica fisica. Questa operazione di collegare al modulo del *fs* un dispositivo viene detto *montaggio* perché c'è un comando *mount* che permette di collegare e rendere visibile i dischi che colleghiamo. Quando inseriamo una chiavetta è il montaggio del *fs* della chiavetta

sul SO. Il NFS permette di effettuare questa operazioni di montaggio (rendere disponibile un fs al kernel locale) che si trova su un'altra macchina collegata alla rete. Ad esempio potrei montare una parte di fs che è fisicamente memorizzata sul disco di un'altra macchina. Devi avere il permesso dall'amministratore del modulo. Il NFS si basa su un protocollo specifico. NFS usa nel livello applicazione, diversi layer che realizzano il livello *sessione*, *presentazione* e *infine applicazione*.

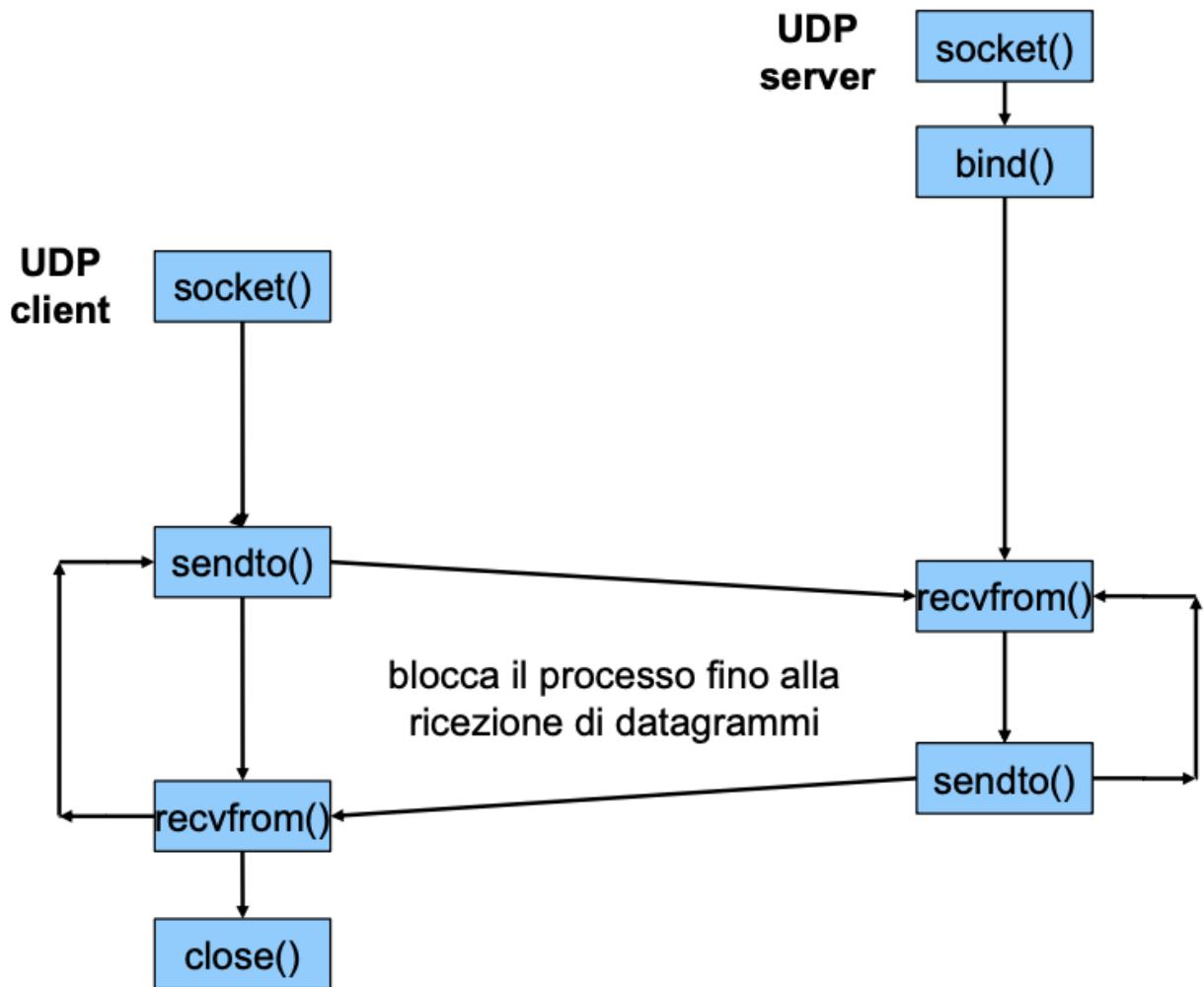
---

- SNMP.

---

Poiché non abbiamo il concetto di connessione a livello UDP, lato client si perde la *connect()*. Lato server perdiamo la *listen()* e *accept()* perché il canale è sempre aperto: i client creano datagrammi e li inviano a quella porta. Il server andrà direttamente a leggere datagrammi dal socket collegato alla IP/porta e invierà una risposta.

Vediamo ora lo schema client/server di uno schema UDP.



Non abbiamo la necessità di fare una `connect()`. Dovremo creare il pacchetto UDP, mettere i dati e inviarlo al server.

Col server creiamo il socket, eseguiamo la `bind` e ci mettiamo in attesa di ricevere datagrammi.

Al posto della `read` e della `write` si usano **`sendto()`** al posto della `write` e la **`recvfrom()`** al posto della `read`. Vediamo poi in cosa si differenziano.

## Perché abbiamo necessità di introdurre queste altre funzioni?

Immaginiamo di aver creato la socket lato client, la socket TCP faceva la connect che prendeva come argomento il descrittore, la struttura dati (famiglia porta e ip del server) e la lunghezza mentre la write prende il descrittore, un buffer da inviare e la dimensione del buffer.

Se perdo la *connect()* quali info non ho più collegate al descrittore fd? L'IP e la porta.

Dopo aver creato il descrittore con socket, la write non posso farla perché al descrittore mancano 2 info fondamentali, cioè IP e porta destinatari. Nella write devo avere la possibilità di specificare questi dati a chi devono essere inviati, per questo motivo introduciamo la *sendto()* dove oltre a specificare i dati da inviare, specifico anche IP e porta dell'altra parte che dovrà ricevere il datagramma.

Stesso discorso per chi riceve: non facendo più la *accept()* dove avevo la struttura *sockaddr* e la sua dimensione, devo passare questi argomenti alla *recvfrom()* che oltre a leggere deve recuperare le info relative al client che sta

inviamo il datagramma. Le info che perdiamo con connect e accept le spostiamo nella *sendto()* e *recvfrom()*.

---

## SOCKET

L'unica cosa che cambia è il secondo argomento. Per leggere e scrivere, al posto di read e write uso *recvfrom* e *sendto*.

---

## RECVFROM

Prende come *arg* descrittore, buffer e dimensione, abbiamo poi dei flag per impostare la modalità bloccante o non, una struttura *sockaddr* e una dimensione *socklen\_t* che sono la parte mancante che avremmo facendo la accept(). Nel momento in cui faccio la recvfrom che è bloccante sto leggendo dal descrittore *sd* un dato buffer di dimensione massima *len* che è arrivato dalla *\*from* specificati nella struttura.

- `ssize_t recvfrom(int sd, void *buf, size_t len,  
int flags, struct sockaddr *from, socklen_t  
*fromlen);`

Devo sapere queste info perché non essendoci la connessioni non so da chi è arrivato il datagramma.

La *recvfrom* possiamo vederla come una *accept* seguita da una *read*.

---

## SENDTO

I primi 3 argomenti sono quelli della *write* e a seguire ho un puntatore alla struttura *sockaddr* con la sua dimensione.

Se sto inviando un datagramma, poiché non c'è connessione devo specificare a chi sto inviando questi dati nella struttura *sockaddr*.

Ricordiamo che a livello IP avrò IP sorgente e IP destinazione: IP sorgente è inserito dal SO mentre destinazione dalla struttura *sockaddr*. A livello di trasporto avrà porta sorgente (inserita dal SO) mentre porta destinazione dalla struttura *sockaddr*.

Mentre in *TCP* i pacchetti sono riempiti dal SO, in *UDP* dobbiamo specificare info perché ogni datagramma deve portare queste info per conto suo.

Allo stesso modo la *sendto* è vista come una *connect* seguita da una *write*.

---

# Struttura di client e server UDP

- Server

- `Socket(...);`
- `Bind(...);`
- `for ( ; ; ) {`
- `Recvfrom(...);`
- `Sendto(...);`
- `}`

- Client

- `Socket(...)`
- `...`
- `Sendto(...);`
- `Recvfrom(...);`
- `...`

Il server crea la *socket*, valorizza la struttura: nel campo *sin\_port* metterò la porta su cui voglio ricevere i datagrammi, nel campo *sin\_addr* va messo il *htonl(INADDR\_ANY* → localmente posso avere più *IP*. Quando chiedo a livello *TCP* o *UDP* di riservarmi una porta, se metto *INADDR\_ANY* vuol dire che la collego a tutti gli IP collegati alla mia macchina è come fare una *bind* su tutti gli IP locali.). faccio una bind, poi con un for infinito leggo un datagramma e invio un datagramma di risposta.

Lato client: dopo creato il socket UDP, invio con la sendto il datagramma.

Nel protocollo TCP veniva assegnata la porta al client durante la *connect()*, se la porta non è pronta viene assegnata una porta *effimera*.

In UDP la porta viene assegnata con la *sendto()*. Se dopo la recvfrom faccio di nuovo la sendto, non verrà assegnata una nuova porta al descrittore perché la riutilizzerà. Questa porta non è assegnata in maniera esclusiva: ci potrebbero essere più client o server.

Lato server in UDP invece la porta è assegnata nella *bind()*, esattamente come avviene in TCP.

**ESAME:** la **bind** assegna l'**indirizzo** al descrittore, l'indirizzo è la coppia IP PORTA.<sup>3</sup>

---

Mentre TCP a ogni connessione assegna un descrittore dedicato per quella connessione, con UDP non abbiamo un nuovo descrittore ogni volta che si connette un client ma tutti i client

---

<sup>3</sup> esame

usano un canale condiviso. Sull'unico canale arrivano tutti i datagrammi dei client che inviano info al server. Grazie alla *recvfrom* ogni volta il server estrae dalla *struttura IP* e porta per capire chi sta inviando i datagrammi perché poi dovrà inviare con la *sendto()* una risposta specifica.

---

## INAFFIDABILITÀ'

Un client invia la sua richiesta con una *sendto()* e si blocca sulla *recvfrom()* in attesa di una risposta che potrebbe non arrivare mai per 2 motivi:

- La richiesta non è arrivata al server.
- La risposta non è arrivata al client.

In entrambi i casi il client **resta bloccato** sulla *recvfrom()*.

**IMPORTANTE:** Nel momento in cui usiamo un protocollo UDP dobbiamo implementare un timer per far proseguire il client, inviando un'altra richiesta o facendo altro.<sup>4</sup>

Una possibilità è appunto impostare un timeout.

---

<sup>4</sup> esame

```
struct timeval tv;
setsockopt(rcv_sock, SOL_SOCKET, SO_RCVTIMEO,&tv,sizeof(tv))
```

Definiamo un tempo massimo di attesa.

Potremmo fare *una select* oppure fare *polling* rendendo la *recvfrom* **non bloccante**.

---

## VERIFICA DEL MITTENTE

Essendo UDP **non connesso**, potrei non sapere chi mi sta inviando un datagramma.

Immaginiamo il client che invia un datagramma al server e il server che risponde al client: il client ha una *porta aperta* e chiunque potrebbe inviargli un datagramma; poiché il client vuole dati specifici di un server, nella *sendto()* va a definire la coppia IP/Porta che è del server che conosce. Il server risponderà con la sua *sendto()* inviando i dati alla coppia IP/porta che ha letto dalla *recvfrom()*. Il client nella *recvfrom* come 5 e 6\* argomento abbiamo la struttura *sockaddrin*: prendo IP/Porta, se sono uguali a quelli a cui avevo inviati il datagramma, accetto la risposta. A meno di attacchi particolari, il client riceverà i datagrammi del server specifico.

Il server però potrebbe avere più *interfacce*: il server nella bind potrebbe aver passato una struttura con la macro ***INADDR\_ANY*** e potrebbe usare entrambe le interfacce. Potrebbe succedere che il client invii una richiesta su *interfaccia1* che arriva al server, il server produce il datagramma e usa però *interfaccia2*. Quando va a fare la verifica nella *recvfrom()* il client non si trova e scarta.

Come si risolve questa situazione?

Ci sono 2 soluzioni: una a livello client e l'altra a livello server.

- Il client invia un datagramma, riceve da un IP specifico, fa una verifica sul DNS con la struttura *hostent* e verifica se c'è IP2, nel caso accetta.
  - Lato server invece si fanno più bind per ogni interfaccia che vuole usare: avremo *bind1* e *bind2 (Seguendo l'esempio) e ho 2 descrittori*, se ricevo dal descrittore *bind1* rispondo con lo stesso.
-

## ASSENZA DI SERVER

Un altro problema è l'invio di datagrammi a porte che non sono aperte.

Il problema è che poiché non c'è l'idea di connessione, quando si verifica un errore e poiché il canale UDP è condiviso, l'errore generato (ICMP → serve per generare e ricevere pacchetti di controllo sullo stato della rete), è ***port unreachable***: *il pacchetto è arrivato a destinazione ma la porta non c'è*. Il problema è che UDP realizza un canale condiviso e la porta da cui è partito il datagramma potrebbe essere usato da più client, il SO non può portare a livello applicazione questo errore perché potrebbe esserci un nuovo processo che non ha nulla a che fare col datagramma che ha generato quell'errore, appunto perché UDP è condiviso.

Si introduce quindi una **modalità UDP CONNESSO che non ha nulla a che fare con la connessione TCP**. Non c'è nulla di simile al 3-w-h  
Nel SO del client si segna in una tabella che il processo P1 usa la porta 1. Il processo P1 farà una *connect()* ma come primo argomento passa

un descrittore di socket UDP: questa connect() non instaura una connessione col server ma serve al SO del client di segnarsi in una tabella che P1 vuole usare in maniera esclusiva quella porta, usando sempre UDP.

Questo implica che il client P1, usando quel descrittore, potrà inviare datagrammi solo a quel server specifico, così che se su quel canale viene generato un errore di tipo *ICMP* potrà portare l'errore al client.

Non userò più *sendto()* e *recvfrom()* ma tornerò ad usare *read()* e *write()*.

### Lato client:

- creo la socket (AF\_INET, SOCK\_DGRAM, 0) → creo descrittore UDP
- valorizzo la struttura *sinfamily*, *sinport*, *sinaddr*
- effettuo la connect(fd, (sockaddr\*)&server, sizeof)
- dopo la connect invio il datagramma con la *write(fd, buff, len)*
- la risposta la ottengo con la *read(fd, buff, len)*

Questo perché il SO con la `connect()` si è segnato la porta che abbiamo valorizzato.

Al posto di `read` e `write` posso usare *sendto* e *recvfrom* però metto tutto a NULL per quanto riguarda 5 e 6° argomento.

---

## PERFORMANCE

Se il protocollo prevede più invii quindi si può creare un UDP connesso per inviare più datagrammi.

Esempio FTP: il server dovrà inviare più datagrammi poiché è molto grande. Per ragioni di efficienza si crea un UDP connesso per inviare e ricevere solo da quel client specifico.

---

## DISCONNESSIONE

Per disconnettere un socket UDP si specifica ***AF\_UNSPEC*** nel campo *sin\_family*: *la famiglia non è specificata e ha l'effetto di disconnettere il descrittore.*

La *connect* potrebbe restituire l'errore  
**EAFNOSUPPORT** ma si ottiene lo stesso la  
disconnessione.