Laboratorio di Reti di Calcolatori

LEZ.2

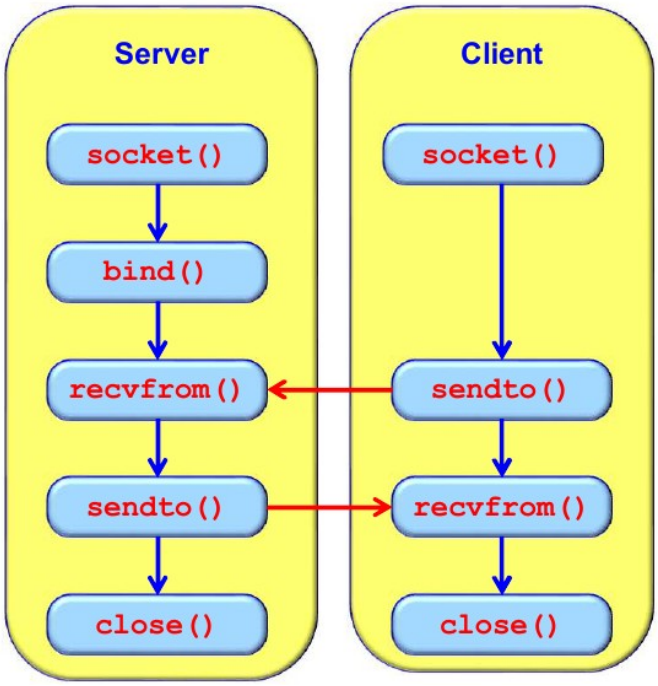
SOCKET DI BERKLEY

I socket sono definiti in modo univoco da un insieme di **5 elementi**:

- Il protocollo utilizzato
- L'indirizzo IP locale
- La porta locale
- L'indirizzo IP remoto
- La porta remota

Basta che uno solo di questi elementi sia differente per rendere diverso il collegamento.

CONNESSIONE UDP



DEFINIZIONE DI UN SOCKET IN C

```
struct sockaddr{
        u_short         sa_family;
        char            sa_data[14];
};
```

Questo socket permette di gestire varie tipologie di socket, a seconda della **famiglia** sa_family, essa può essere:

AF_UNIX, AF_NS, AF_IMPLINK, AF_INET (Ipv4), AF_INET6 (Ipv6), AF_IPX, AF_IRDA, AF_BLUETOOTH.

Linux supporta 29 tipi di famiglie.

DEFINIZIONE DI UN SOCKET AF_INET

```
struct sockaddr_in{
        short int          sin_family; //AF_INET
        unsigned           short int sin_port;
        struct             in_addr sin_addr:
        unsigned char      sin_zero[8];
};

struct in_addr{
        u_int_t s_addr;
};
```

DEFINIZIONE DI UN SOCKET AF_INET6

```
struct sockaddr_in6{
        u_int16_t       sin6_family; //AF_INET6
        u_int16_t       sin6_port;
        u_int32_t       sin6_flowinfo; //Ipv6 flow info
        struct in6_addr          sin6_addr;
        u_int32_t       sin6_scope_id; //scope id
};

struct in6_addr{
        unsigned char           s6_addr[16];
};
```

CONVERSIONE DEGLI INDIRIZZI

- **pton =** Presentation to Network
- **ntop** = Network to Presentation

Queste funzioni consentono di trasformare gli indirizzi Ipv4 o Ipv6 dal formato compatto (interno) a quello di testo leggibile e viceversa.

const char ***inet_ntop**(int af, const void *src, char *dst, socklen_t size);

int **inet_ntop**(int af, const char *src , void *dst);

esempio 1

```
struct sockaddr_in sa; // Ipv4
char ip4[16];

inet_ntop( AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);

inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr));

// INET_ADDRSTRLEN = 16
// 255.255.255.255
```

esempio 2

```
struct sockaddr_in6 sa6; //Ipv6
char ip6[46];

inet_ntop( AF_INET6, &(sa6.sin6_addr)m ip6, INET6_ADDRSTRLEN);

inet_pton( AF_INET6, “2001:d8:b3:1::3490”, &(sa6.sin6_addr));

//INET_ADDRSTRLEN6 = 46
//FFFF : FFFF : FFFF : FFFF : FFFF : FFFF : 255.255.255.255
```

ALTRE FUNZIONI DI CONVERSIONE

-unsigned short htons(unsigned short int n)

converts the unsigned short integer host short from host byte order to network byte order.

-unsigned short ntohs(unsigned short int n)

converts the unsigned short integer netshort from network byte order to host byte order.

-unsigned short htonl(unsigned short int n)

converts the unsigned integer hostlong from host byte order to network byte order.

-unsigned short ntohl(unsigned short int n)

converts the unsigned integer netlong from network byte order to host byte order.

SOCKET

#include <sys/types.h>
#include <sys/socket.h>

int **socket**(int domain, int type, int protocol);

La chiamata **socket()** restituisce un identificatore di socket. Il socket viene creato o viene definito sia dal tipo che dal protocollo utilizzato. L'utilizzo dei descrittori di socket è simile a quello dei file.

Domain:

PF_INET, PF_INET6

Type (solo per PF_INET):

SOCK_DGRAM (**UDP**), SOCK_STREAM (**TCP**)

Il protocollo dipende dalla famiglia (di seguito per AF_INET):

IPPROTO_UDP, IPPROTO_TCP, IPPROTO_ICMP, IPPROTO_RAW

spesso, al posto di PF_INET, si usa AF_INET, I due sono equivalenti, inoltre, nel campo protocolo, si può inserire il valore 0 (verrà scelto il protocollo più adatto al tipo indicato).

BIND

int bind(int socket, struct sockaddr *addr, int addrlen);

La chiamata **bind()** restituisce zero in caso di successo. La bind serve per inserire i dati locali (indirizzo e porta) nel socket.
L'effetto della bind è duplice: per il traffico in **input** server per dire al sistema a chi deve consegnare i pacchetti entranti, mentre per il traffico in **output** server per inserire il mittente nell'intestazione dei pacchetti.

Nella definizione dell'indirizzo internet, è possibile usare alcuni valori definiti tramite **macro**:

INADDR_LOOPBACK: indica l'host stesso (127.0.0.1)

INADDR_ANY: server per accettare le connessioni da qualunque indirizzo

INADDR_BROADCAST: server per mandare messaggi in broadcast

INADDR_NAME: restituisce alcune informazioni in caso di errore

SENDTO – RECVFROM

int **sendto**(int socket, void *buffer, size_t size, int flags, struct sockaddr *addr, size_t *length);

int **recvfrom**(int socket, void *buffer, size_t size, int flags, struct sockaddr *addr, size_t *length);

Queste due funzioni operano come la **send()** e la **recv()** ma sono utilizzate per le trasmissioni senza connessione. Nella sendto() deve essere specificato l'**indirizzo di destinazione**, mentre nella recvfrom() il campo indirizzo viene riempito con quello del **mittente**.

int **close**(int socket);

Permette di **chiudere** un socket aperto.

ESEMPIO UDP SENDER TO RECEIVER

Il programma più semplice che si può realizzare è composto da un sender che spedisce pacchetti e da un receiver che li riceve. La comunicazione è unidirezionale e il protocollo usato è UDP.

OSS.

Il comando **ping** su windows si trova a livello IP e non usa le porte ma il protocollo ICMP (**Internet Control Message Protocol**) e serve a controllare alcune funzioni internet. Ping richiede l'indirizzo ip tra i suoi parametri per funzionare. Se ping funziona e da risposta il problema si trova a livello superiore mentre se non funziona il livello si trova a livello inferiore.

-**ScopeLink** è un indirizzo che non viene fatto passare mai dai router, passa soltanto nelle LAN, esso è stato pensato per sostituire gli indirizzi privati.

LEZ.3

DEFINIZIONE DI PORTA

Quando parliamo di porta, il sinonimo è dovuto ad una cattiva traduzione dal termine “port”, infatti quando si parla di porte nelle reti, il sinonimo più appropriato è quello di una **casella postale**: immaginiamo che chiunque può depositare una lettera nella casella postale ma soltanto chi ha le chiavi può aprirla (non è detto che ogni casella abbia un proprietario), vederne il contenuto e deciderne cosa farne.
Diciamo che una porta è **aperta**, quando parliamo di **TCP**, ovvero quando il mittente riceve un riscontro riguardo la ricezione del messaggio, parliamo invece di porta **chiusa** quando il protocollo utilizzato è **UDP**, dunque, chi deposita il messaggio non ha modo di sapere se il messaggio è stato letto o meno. Quando inviamo un pacchetto tramite UDP, non abbiamo modo di sapere se la porta che stiamo contattando, all’indirizzo specificato è **aperta.**

OSS.

La funzione **atoi()** trasforma il valore **ascii** in input in un intero.

La **bind** chiama il sistema operativo e vincola a se stessa l’indirizzo della porta passata, questa operazione è presente solo nei server.

Per ricevere pacchetti, il ricevente, dentro un ciclo for infinito, fa una **chiamata bloccante recvfrom(),** essa riceve, il socket, il buffer in cui scrivere, quanti byte scrivere e la struttura di ricezione di indirizzo.

Nella comunicazione **One-way UDP**, noi stabiliamo per il server la porta su cui mettersi in ascolto e per il client la porta da contattare. Quando il server da in output il numero di porta che ha ricevuto, quello non è altro che la porta che il sistema operativo del client ha assegnato a quest’ultimo per effettuare la connessione verso la porta specificata, questa porta viene assegnata randomicamente tra tutte quelle di categoria tre. Questa porta scelta dal sistema operativo, per quanto riguarda i sistemi **UNIX**, viene associata al **PID** del processo richiedente, quindi, in questo programma, dato che dopo ogni comunicazione il processo termina, la porta assegnata cambierà ad ogni comunicazione.

UDP FULL DUPLEX

fgets() prende una stringa da un file, come file possiamo mandare lo standard input e lo mette in un pacchetto.

Questo programma prende in input una stringa da tastiera, la invia al server e il server si occupa di inviarla di nuovo, invertendo gli indirizzi.

LEZ. 4

CHAT UDP

Realizzeremo una chat che sfrutta il protocolo UDP ma che non implementa interfaccia grafica, per questo motivo in fatto la chat risulterà disordinata, esistono però delle librerie che permettono di gestire il terminale graficamente per rendere i messaggi più comodi da leggere.

A tale scopo, potremmo pensare di utilizzare un client UDP e un server UDP come quelli visti precedentemente, si pone per subito un problema, infatti, le chiamate **sendto** e **get** sono bloccanti e non permettono al chiamante di effettuare altre operazioni prima che esse ricevano una risposta, ovviamente questo aspetto va contro il concetto di chat, in quanto, un utente potrebbe decidere di inviare più messaggi in modo consecutivo o di inviare messaggi mentre rimane in ascolto dell'altro peer. Per ovviare a questo problema viene introdotto all'interno del codice l'uso dei processi. All'interno di unix è possibile usare la filosofia delle **fork()**: quando usiamo le fork viene generato un processo fork che non ha memoria condivisa con il padre. La fork restituisce un intero che non è altro che il **pid** del processo creato, il pid sarà uguale a 0 per il figlio e un numero diverso da 0 per il padre, a questo punto è sufficiente un **if** sul pid per separare l'esecuzione dei due processi, infatti, tutto quello che sarà eseguito nel **then** sarà opera del processo figlio e tutto quello eseguito nell'**else** sarà opera del processo padre.

Esempio:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char**argv){
        int n=0, pid;
        for(; n<5; ++n)      printf("Processo principale %d \n",n);
        pid = fork();
        if (pid == 0){
                for(; n<10; ++n){
                        sleep(1);
                        printf("Processo figlio %d %d \n", pid, n*2);
                }
                return 0;
        }
        else{
                 for(; n<10; ++n){
                        sleep(1);
                        printf("Processo padre %d %d \n", pid, n*3);
                }
        return 0;
        }
}
```

A questo punto è sufficiente avere due processi separati, uno che legge da tastiera e invia i messaggi che si blocca sulla **get** e l'altro che riceve i messaggi che si blocca sulla **recvfrom**.

Questo stesso codice può essere replicato su tutte le macchine che vogliamo fare comunicare, non esiste infatti distinzione tra client e server in quanto entrambi gli host devono assolvere sia alla funzione di invio che di ricezione.

Questo programma prenderà **tre parametri**: indirizzo di destinazione, porta di destinazione e porta del mittente.

Il meccanismo consiste nell'aprire due **socket separate**, una per la ricezione e una per l'invio. Quello che farà il socket ricevente gestito da uno dei due processi è fare la bind e mettersi in ascolto con un ciclo infinito, riproducendo a video tutto quello che riceve. L'altra socket invece, gestita dal secondo processo, creerà il ciclo infinito di invio che manderà tutto quello che viene scritto da tastiera.

CHAT UDP BROADCAST

Per effettuare una chat broadcast dovremo specificare come indirizzo di destinazione **255.255.255.255**, ovvero, l'indirizzo broadcast predefinito, inoltre, le porte devono essere uguali per tutti, quindi basterà inserire una sola porta che verrà usata sia per l'invio che per essere contattati dal destinatario,
Apparentemente, per evitare di saturare la rete, le comunicazione broadcast viene **bloccata** di default e per consentire il funzionamento vanno aggiunte le seguenti righe al codice utilizzato per la chat normale.

```
int broadcastEnable=1;
int ret = setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcastEnable,
                              sizeof(broadcastEnable));
```

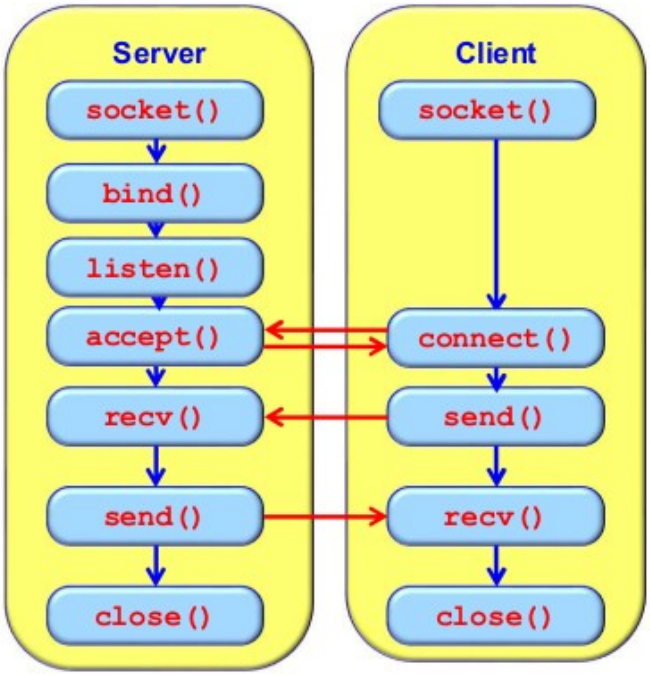
COMANDO PING

Date due macchine, A e B, quando A fa un ping all'indirizzo di B, oltre a ricevere una risposta, ping riceve anche il **tempo** di andata e risposta. Anche questo comando usa due processi, uno per la send e uno per la revc. Per usare questo comando, ho bisogno di scandire degli intervalli regolari (che non possiamo ottenere grazie alla sleep) e per ottenerli è necessario un **timer,**
Il comando mi fornisce anche il tempo di trasmissione, infatti esso è presente nel payload del pacchetto inviato.
Ovviamente per creare una comunicazione ping avrò bisogno di un programma che invia il ping e uno che lo riceve, avrò dunque un programma **ping deamon** che risponderà al programma **ping.**
Il demone non è altro che un programma di ricezione UDP come quelli presentati precedentemente mentre il programma di spedizione.

Per calcolare il tempo viene utilizzata una struttura che prende il nome di **timespec.**
Quello che fa il mittente è prendere il tempo di sistema ed inviarlo, una volta arrivato, il demone, prenderà il tempo ricevuto e il tempo di sistema in quell'istante e tornerà al mittente la **differenza.**

CONNESSIONI TCP

La prima differenza che salta all'occhio è che le primitive da sendto() e recvfrom() diventano send() e recv(), questo perché banalmente, quando client e server comunicano, non hanno bisogno di identificarsi in quanto grazie al three way handshake la connessione è già stata stabilita.



int listen(int socket, int backlog);

La chiamata **listen()** abilita il socket a **ricevere connessioni** rendendolo quindi un server socket. Il parametro **n** indica quante richieste in sospeso devono essere accodate. Intendiamo richieste di inizio connessione,e non connessione già stabilite.

int accept(int socket, struct sockaddr *addr, socklen_t *addrlen);

La chiamata **accept()** è **bloccante**. Non appena arriva una richiesta di connessione, crea un nuovo socket e ne restituisce il descrittore. Il vecchio socket rimane aperto e non connesso. L'indirizzo restituito è quello di chi ha effettuato la **connect().**

int connect(int socket, struct sockaddr *addr, int addrlen);

La chiamata **connect()** inizializza una connessione con un socket remoto. L'indirizzo che viene passato è relativo ad un **host** remoto. La **connect() è** bloccante, finché non vengono negoziati i parametri della trasmissione (il protocollo è il TCP). La funzione viene richiamata da un client che vuol connettersi ad un server (il cui socket deve già essere aperto).

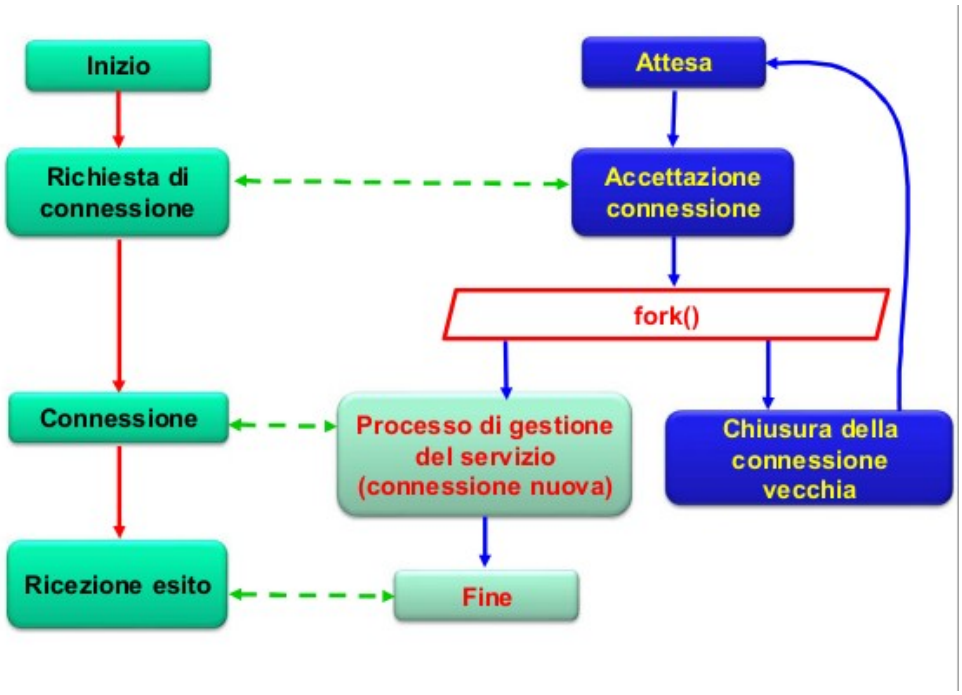
int send(int socket, void *buffer, size_t size int flags);

int recv(int socket, void *buffer, size_t size, int flags);

Queste due funzioni operano come la **write()** e la **read()** per i file normali. Sono utilizzate per trasmissioni con connessione. L'ordine di **send()** e **recv()** non è importante e può essere casuale.

GESTIONE DEI SERVER MULTIPLI

TCP permette la gestione di **server multipl**i, infatti a differenza delle connessioni UDP che vengono soddisfatte in sequenza, TCP deve stabilire una **connessione stabile** e continua, quindi, usa la porta principale come se fosse un **centralino**, ovvero, prende la chiamata e la dirotta su un'altra porta libera e si rimette in ascolto in modo tale da essere contattata da un altro client. Questa operazione avviene grazie ad una **fork()** che viene chiamata ogni volta che il server riceve una richiesta nella porta su cui è in ascolto: essendo il processo generato una copia del padre, viene copiata anche la socket, in questo modo, il figlio continua la connessione mentre il padre la chiude e si rimette in ascolto.



A differenza di UDP, dopo l'invio di un messaggio, il canale **rimane aperto**, se una seconda macchina client dovesse contattare lo stesso server verrebbe creato un altro canale grazie al processo di gestione fork() sul socket.

Per ogni client connesso, il server genererà un processo identico (fork) che gestirà tutte le singole richieste, dunque ad ogni client sarà associato un processo del server a differenza di UDP dove un unico processo gestisce tutte le connessioni.