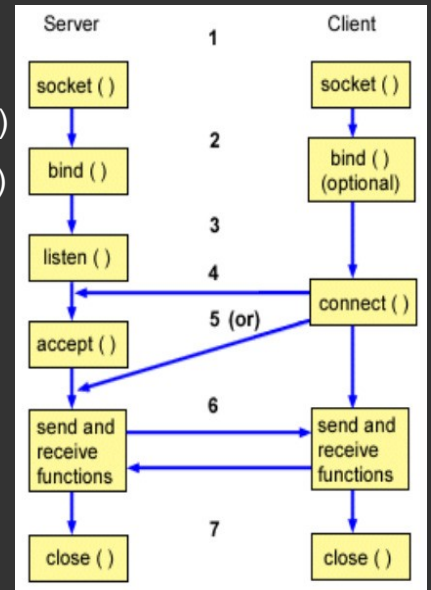


FT_IRC C++

A la hora de conectar con un servidor, el flujo es el siguiente:

1. **socket()** - Creamos la conexión (**compramos el teléfono**)
2. **bind()** - Asignamos el puerto (**damos número al teléfono**)
3. **listen()** - Nos ponemos a la escucha.
4. **accept()** - Contestamos la llamada del cliente
5. **send/recv** - Comunicamos (**hablamos por teléfono**)
6. **close()** - Cortamos la comunicación.



socket()

`#include <sys/socket.h>` (funcion de C usada en C++)

Crea un socket (punto de comunicación) para enviar/recibir datos por la red. **Es como comprar un teléfono. No está conectado y no tiene número pero está listo para ser usado. Mas bien es como crear una centralita para luego conectar cada una de las líneas que entran como hacían las telefonistas.**

`int socket(int domain, int type, int protocol);` //sintaxis

`int server_fd = socket(AF_INET, SOCK_STREAM, 0);` //en ft_irc. Traduccion: Dame un socket IPv4, tipo TCP, y el protocolo automático (en este caso TCP).

Devuelve : el File Descriptor como `int >= 0` si funciona. `-1` si hay error.

DOMAIN.

`AF_INET` : IPv4 (192,168,1,1)

`AF_INET 6` : IPv6 (2001:0db8::1)

`AF_UNIX` : comunicación local. Mismo Ordenador

TYPE.

`SOCK_STREAM` : TCP (protocolo de envio seguro. HTTP, chat, ssh, email)

`SOCK_DGRAM` : UDP (protocolo para juegos en tiempo real, streaming, DNS)

SOCK_RAW : Para acceso directo a IP sin TCP/UDP (ping, traceroute, hacking)

PROTOCOL.

0 : Automático (elige según el **TYPE**)

IPPROTO_TCP : TCP

IPPROTO_UDP : TCP

Aunque parezca chorrada lo del protocolo por que pilla de Type o especifica, hay más protocolos como:

SOCK_STREAM = TCP, SCTP (IPPROTO_SCTP), Bluetooth

SOCK_DGRAM = UDP, ICMP.

ERRORES: (almacenada en la variable **errno** `#include <errno.h>`, **perror()** o **strerror()**)

EACCES : Sin permisos (puertos < 1024 requieren root) usar >1024

EMFILE : Demasiados FDs abiertos. Cerrar sockets viejos

ENFILE : Sistema sin recursos. Reiniciar o reducir carga

EPROTONOSUPPORT : Protocolo no soportado. Verificar parámetros

bind()

`#include <sys/socket.h>` (No hay version C++. Solo C)

Asigna una dirección IP y un puerto al Socket abierto. **Es como asignar un número de teléfono al teléfono que compramos.**

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen); //sintaxis
```

sockfd : El File Descriptor del socket abierto por **socket()**

addr : Puntero a la estructura con IP:puerto

addrlen : Tamaño de la estructura **addr**

Una vez obtenido el FD del socket, antes de asignarle la IP/puerto con **bind()** tenemos que crear una estructura **sockaddr_in** (`#include <netinet/in.h>`):

```
struct sockaddr_in{
    sa_family_t    sin_family; // AF_INET(IPv4)
    in_port_t      sin_port; // puerto (en network byte order)
    struct in_addr  sin_addr; // Direccion IP};
```

y la rellenamos:

```
struct sockaddr_in server_addr; //server_addr es una variable creada nuestra y vacia.
```

```
server_addr.sin_family = AF_INET; //IPv4
```

```
server_addr.sin_port = htons(6667); //Puerto 6667. (uint16_t (0-65535))
```

```
server_addr.sin_addr.s_addr = INADDR_ANY; // Todas las IPs locales. Escucha en todas las interfaces (0.0.0.0)
```

```
int result = bind(server_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)); //se castea a(struct sockaddr*) siempre.
```

.sin_family Puede ser:

AF_INET	: IPv4
AF_INET6	: IPv6
AF_UNIX	: Sockets Locales (mismo ordenador)
AF_BLUETOOTH	: Bluetooth
AF_PACKET	: Acceso a bajo nivel a red.

.sin_port Es el puerto, y es un numero uint16_t (0 a 65535) donde:

0	: El sistema asigna un puerto automáticamente
1-1023	: Requieren Root ya que son puertos reservados
1024 - 49151	: Puertos registrados por aplicaciones conocidas. 6667 por IRC
49152 - 65535	: Puertos dinámicos / Privados

.sin_addr.s_addr Es una estructura con un único parámetro. Se mantiene así por compatibilidad con antiguas versiones.

INADDR_ANY	: Todas las interfaces locales (wifi, Ethernet, localhost...)
INADDR_LOOPBACK	: Localhost (127.0.0.1) = htonl(0x7F000001)
inet_addr("192.168.1.3")	: IP específica.
inet_pton(AF_INET, "192.168.1.3", &addr, sin_addr)	: Conversion Moderna.

Devolverá (result) **0** si tiene éxito o **-1** si da error (que se consulta con **errno**)

ERRORES:

- EADDRINUSE** : El puerto ya está en uso
- EACCES** : Puerto <1024 sin permisos de root
- EINVAL** : Socket ya enlazado.

Notas:

1. Por que se solicita una dirección de memoria de **struct sockaddr**, y nosotros generamos un **sockaddr_in**?

Son dos estructuras diferentes. **sockaddr** es más genérica y no tiene parámetros que sí tiene **_in**. Por ejemplo los puertos, sí los tiene **_in** y no **sockaddr**.

Entonces ¿como sabe que es un **sockaddr_in** cuando lo casteamos a **sockaddr** en **bind()**? Porque mira a **sin_family** (que está también en **sockaddr** y en este caso es **AF_INET**) y sabe que es un **sockaddr_in**

1. **htons** y **htonl** son funciones de network para enviar datos. Diferentes CPUs almacenan los números de forma diferente:

Big-endian (0x1234) se almacena el byte más significativo primero 12 – 34

Little-endian(0x1234) Byte menos significativo primero 34-12

Para traducir esto a cada CPU se usan estas funciones

htons() : Host to Network short (16 bits)

htonl() : Host to Network Long (32 bits)

ntohs() : Network to Host Short. Aquí no enviamos, sino recibimos

ntohl() : Network to Host Long. Aquí no enviamos, sino recibimos

Si nuestra CPU es ya Big-endian, como es el estándar cordado no hace falta estas funciones, pero está bien en caso de ser necesario, para traducírselo a una CPU que sea Little-endian.

listen()

#include <sys/socket.h> (No hay version C++. Solo C)

Marca el socket como pasivo (espera conexiones entrantes). **Es como activar el contestador del teléfono para recibir llamadas.**

```
int listen(int sockfd, int backlog); //sintaxis
```

sockfd : File descriptor del **socket()**

backlog : Número máximo de conexiones en cola de espera. Ej: **5** (Hasta 5 conexiones esperando)

SOMAXCONN (máximo del sistema (128 – 4096 según OS))

Devuelve:

0 : Éxito

-1 : Error (consultar **errno**)

```
int result = listen(server_fd, 10); //espera hasta 10 conexiones
```

```
if (result == -1){  
    perror("listen"); return 1;}  
}
```

ERRORES:

EADDRINUSE : El puerto ya está en uso

EBADF : FD inválido

ENOTSOCK : FD no es un socket.

connect() cliente.

```
#include <sys/socket.h> (No hay version C++. Solo C).
```

Conecta un socket cliente a un servidor remoto. **No hace falta para el IRC ya que no piden cliente, podemos usar irssi, weechat, hexchat...**

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen); //sintaxis
```

sockfd : File descriptor del **socket()**

addr : Dirección del servidor (IP + puerto)

addrlen : Tamaño del addr

Devuelve **0** éxito(conectado), **-1** error

```

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

// Crear socket
int client_fd = socket(AF_INET, SOCK_STREAM, 0);

// Configurar servidor
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(6667); // Puerto IRC
inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr); // IP servidor

// Conectar
if (connect(client_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) {
    perror("connect");
    return 1;}

std::cout << "Conectado al servidor IRC\n";

// Enviar/recibir...
send(client_fd, "NICK usuario\r\n", 14, 0);

```

accept()

`#include <sys/socket.h>` (No hay version C++. Solo C)

Acepta una conexión entrante y crea un nuevo socket para comunicarse con ese cliente. Es como contestar el teléfono, pero ese nuevo socket abierto es como si el teléfono que teníamos fuera una centralita, y al hacer `accept()` se genera una conexión directa con él (conecta los cables la telefonista)

```
int accept(int sockfd, struct sockaddr *addr, socklen_t addrlen); //sintaxis
```

sockfd : FD del socket servidor (el de `listen()`)

addr : Salida, info del cliente (IP, puerto)

addrlen : Entrada/Salida. Tamaño de addr.

Devuelve un nuevo FD (≥ 0) para comunicarse con ese cliente. -1 si hay un error.
Ejemplo para ft_irc:

```
struct sockaddr_in    client_addr;  
socklen_t            client_len = sizeof(client_addr);  
int    client_fd = accept(server_fd, (struct sockaddr*)&client_addr, &client_len);  
if (client_fd == -1){perror("accept"); return 1;}
```

En este ejemplo `server_fd` sigue escuchando nuevas conexiones, y `client_fd` comunicamos con ese cliente específico.

IMPORTANTE:

`accept()` bloquea hasta que llega una conexión (a menos que se use non-blocking)

Cada cliente tiene su propio FD y el `server_fd` no cambia. Sigue escuchando.

El `listen()` se hacía fuera de un bucle, el `accept()` dentro de un bucle `while (server_running)` Pero esto bloquearía escuchar al resto de clientes. Para ello se hace un `fork`, un `std::thread` O un `poll/epoll` (`poll(fds, nfds, -1)`)

INFO DEL CLIENTE:

```
char *client_ip = inet_ntoa(client_addr.sin_addr);  
int    client_port = ntohs(client_addr.sin_port);  
std::cout << "cliente: " << client_ip << ", " << client_port << std::endl;
```

send/recv()

`#include <sys/socket.h>` (No hay version C++. Solo C)

Para Enviar y Recibir Datos.

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags); //sintaxis  
ssize_t recv(int sockfd, void *buf, size_t len, int flags); //sintaxis
```

sockfd : FD del socket (servidor o cliente en caso de `send()`)
buf : Buffer con datos a enviar o donde Guardar los datos recibidos
len : Número de bytes a enviar o Tamaño máximo del buffer en `recv()`

flags : Opciones (normalmente 0)

Devuelve: **send()** → Número de bytes enviados (≥ 0) ó -1 si hay error.

recv() → Número de bytes recibidos (≥ 0) ó -1 si hay error.

Ejemplo **send()**:

```
std::string msg = "Bienvenido a IRC\r\n";
ssize_t sent = send(client_fd, msg.c_str(), msg.size(), 0);
if (sent == -1){perror ("send");}
```

Ejemplo **recv()**:

```
char buffer[512];
ssize_t bytes = recv(client_fd, buffer, sizeof(buffer) - 1 , 0);
if (bytes == -1){perror("recv");}
else if (bytes == 0) {
    std::cout << "cliente desconectado\n";
    close(client_fd);}
else{
    buffer[bytes] = '\0'; //terminador de cstring
    std::cout << "Recibido: " << buffer << std::endl;}
```

FLAGS:

0 : Comportamiento normal. Espera al cliente para recibir. No bloquea. Por eso usar **MSG_DONTWAIT** o **poll/epoll()**

MSG_DONTWAIT : No bloquear (devuelve inmediatamente). Alternativa a **poll()**

MSG_PEEK : Leer sin eliminar datos del buffer. Cuando se llama de nuevo a **recv()**, se eliminan los datos del buffer si está con flag **0**, pero aquí no se eliminan.

MSG_WAITALL : Esperar hasta recibir **len** bytes completos. Solo para **recv()**. **send()** no tiene un flag como este y por eso mirar **IMPORTANTE!!**. Pero mejor no usarlo mucho y usar más delimitadores como **\r\n** en IRC y leer línea a línea)

IMPORTANTE:

Cuando send() NO puede enviar todo, ya sea por Buffer TCP lleno, congestión de la red, Socket no bloqueante o interrupción de señal del sistema (EINTR)

```
ssize_t send_all(int fd, const char *data, size_t len){
    size_t total_sent = 0;
    while (total_sent < len){
        ssize_t sent = send(fd, data + total_sent, len - total_sent, 0);
        if (sent == -1){
            if (errno == EINTR) continue; //reintentar si fue interrupción
            return -1; //error real
        }
        total_sent += sent;
    }
    return (total_sent);
}
```

```
std::string msg = "Mensaje extra largo...."; //10.000 bytes
if (send_all(client_fd, msg.c_str(), msg.size()) == -1){perror ("send_all");}
```

close()

#include <unistd.h> (No hay version C++. Solo C)

Cierra un File Descriptor, ya sea socket, archivo, pipe....

```
int close(int fd); //fd es el file descriptor a cerrar
```

Devuelve 0 como éxito o -1 si hay error.

Esto libera recursos, y envia FIN al TCP.

poll()

#include <poll.h> (No hay version C++. Solo C). O(n)

Monitorea múltiples file descriptors a la vez para ver cuáles están listos para leer, escribir o tienen error. **Es como vigilar varios teléfonos a la vez, sin quedarse bloqueado.**

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout); //sintaxis
```

fds : Array de FDs a monitorear.

```
struct pollfd{
    int    fd;           // File descriptor a monitorear
    short  events;       // Eventos que QUIERES detectar (INPUT)
    short  revents;      // Eventos que OCURRIERON (OUTPUT)}
events / revents:
POLLIN    : Hay datos para leer
POLLOUT   : Listo para escribir
POLLERR   : Error en el socket
POLLHUP   : Cliente cerró la conexión
POLLNVAL  : FD inválido
```

nfds : Número de elementos en fds

timeout : Milisegundos a esperar (-1 = infinito)

Devuelve: Número de FDs listos (>0), 0 timeout o -1 error.

Ejemplo:

```
#include <poll.h>
#include <vector>

std::vector<struct pollfd> fds;
// Agregar servidor
struct pollfd server_poll;
server_poll.fd = server_fd;
server_poll.events = POLLIN; // Solo leer (nuevas conexiones)
fds.push_back(server_poll);
// Bucle principal
while (true) {
    int ready = poll(fds.data(), fds.size(), -1); // Espera infinita
    if (ready == -1) {
        perror("poll");
        break;
    }
    // Revisar cada FD
    for (size_t i = 0; i < fds.size(); i++) {
        if (fds[i].revents & POLLIN) { // Hay datos para leer
            if (fds[i].fd == server_fd) {
                // Nueva conexión
            }
        }
    }
}
```

```

    int client_fd = accept(server_fd, NULL, NULL);
    struct pollfd client_poll;
    client_poll.fd = client_fd;
    client_poll.events = POLLIN;
    fds.push_back(client_poll);
    std::cout << "Nuevo cliente: " << client_fd << std::endl;
} else {
    // Cliente existente envió datos
    char buffer[512];
    ssize_t bytes = recv(fds[i].fd, buffer, 512, 0);
    if (bytes <= 0) {
        // Cliente desconectado
        std::cout << "Cliente " << fds[i].fd << " desconectado\n";
        close(fds[i].fd);
        fds.erase(fds.begin() + i);
        i--; // Ajustar índice
    } else {
        buffer[bytes] = '\0';
        std::cout << "Recibido: " << buffer << std::endl;
        // Procesar comando IRC...
    } } } }

```

epoll()

`#include <sys/epoll.h>` (No hay version C++. Solo C). O(1)

Lo mismo que `poll()` pero más eficiente para muchos más clientes y solo funciona en Linux. `poll()` recorría todos los FDs cada vez y `epoll()` solo notifica FDs activos.

- `int epoll_create1(int flags);` //crear instancia epoll. `epoll_create(int size)` está obsoleto.

flags : Opciones de creación

0 : Sin opciones especiales

EPOLL_CLOEXEC : Cierra automáticamente el FD en `exec()`

Devuelve FD de `epoll(>=0)` o `-1` si error

- `int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);` //Añadir/modificar FD

epfd : FD de epoll, que viene de `epoll_create1()`

op : Operación a realizar

EPOLL_CTL_ADD : Agregar FD a monitoreo

EPOLL_CTL_MOD : Modificar eventos de FD existente

EPOLL_CTL_DEL : Eliminar FD del monitoreo

fd : FD del socket a agregar/modificar/eliminar
event : Configuración de eventos (NULL para DEL)

```
struct epoll_event {
    uint32_t      events;    // Eventos a monitorear
    epoll_data_t  data;    // Datos del usuario};
typedef union epoll_data {
    void          *ptr;
    int           fd;
    uint32_t      u32;
    uint64_t      u64;
} epoll_data_t;
```

EVENTOS (events)

EPOLLIN : Hay datos para leer
EPOLLOUT : Listo para escribir
EPOLLERR : Error en el FD
EPOLLHUP : Cliente cerró la conexión
EPOLLET : Edge-triggered (solo notifica cambios)
EPOLLONESHOT : Notifica solo una vez.

Devuelve **0** como éxito **-1** error

• **int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);**
//Esperar Eventos

epfd : FD de epoll
events : Array donde se guardan los eventos listos.
maxevents : Tamaño máximo del array
timeout : Milisegundos a esperar (-1 para infinito)

Devuelve número de FDs listos (>0), **0** timeout, **-1** error

Ejemplo:

```
#include <sys/epoll.h>
```

```
int epoll_fd = epoll_create1(0);
```

```
// Agregar servidor
```

```
struct epoll_event ev;
```

```
ev.events = EPOLLIN; //monitorea lectura
```

```
ev.data.fd = server_fd; //guardar FD
```

```
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, server_fd, &ev);
```

```
// Esperar eventos
```

```

struct epoll_event events[10];
while (true) {
    int ready = epoll_wait(epoll_fd, events, 10, -1); //espera infinita

    for (int i = 0; i < ready; i++) {
        if (events[i].data.fd == server_fd) {
            // Nueva conexión...
        } else {
            // Cliente envió datos...
        }
    }
}
}

```

fcntl()

`#include <fcntl.h>` (No hay version C++. Solo C).

Manipula opciones de un file descriptor (flags, permisos, modo bloqueante, etc...) y para IRC lo que hará es entrar en modo bloqueante. **epoll()** dice cuando hay datos, pero **accept()** / **recv()** pueden bloquearse si no hay. El modo no bloqueante devuelve **EAGAIN** / **EWOULDBLOCK** inmediato.

Caso real:

1. el Cliente intenta conectar
2. **epoll_wait()** detecta actividad en el **server_fd**
3. Antes de hacer **accept()**, el cliente cancela
4. **accept()** se bloquea esperando otra conexión. Con no bloqueante se evita eso. Ver ejemplo al final de esta sección.

```
int fcntl(int fd, int cmd, ... /* arg */); //sintaxis.
```

fd : File descriptor a modificar

cmd : Comando / operación a realizar

F_GETFL : Obtener flags actuales del FD

F_SETFL : Establecer flags del FD

F_GETFD : Obtener descriptor flags

F_SETFD : Establecer descriptor flags

FLAGS:

O_NONBLOCK : Modo no bloqueante
O_APPEND : Escribir siempre al final
O_ASYNC : I/O asíncrono

arg : Argumento opcional según el **cmd**

Devuelve número que depende del comando, **-1** error

EJEMPLO CON MODO NO BLOQUEANTE:

```
#include <fcntl.h>

// 1. Obtener flags actuales
int flags = fcntl(client_fd, F_GETFL, 0);
if (flags == -1) {
    perror("fcntl F_GETFL");
    return -1;}

// 2. Añadir O_NONBLOCK
flags |= O_NONBLOCK;

// 3. Aplicar flags
if (fcntl(client_fd, F_SETFL, flags) == -1) {
    perror("fcntl F_SETFL");
    return -1;}

std::cout << "Socket en modo no bloqueante\n";
```

FUNCION HELPER (para evitar código repetitivo):

Sin helper:

```
// Cliente 1
int flags = fcntl(client_fd1, F_GETFL, 0);
flags |= O_NONBLOCK;
fcntl(client_fd1, F_SETFL, flags);

// Cliente 2
flags = fcntl(client_fd2, F_GETFL, 0);
flags |= O_NONBLOCK;
fcntl(client_fd2, F_SETFL, flags);

// Servidor
flags = fcntl(server_fd, F_GETFL, 0);
```

```
flags |= O_NONBLOCK;
fcntl(server_fd, F_SETFL, flags);
```

Con helper:

```
int set_nonblocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) return -1;
    return fcntl(fd, F_SETFL, flags | O_NONBLOCK);}
```

// Uso:

```
set_nonblocking(server_fd);
set_nonblocking(client_fd);
```

EJEMPLO CON POLL()

// Poner servidor no bloqueante

```
set_nonblocking(server_fd);
```

// Poll detecta nueva conexión

```
if (fds[0].revents & POLLIN) {
    int client_fd = accept(server_fd, NULL, NULL); // No se bloquea
```

```
    if (client_fd == -1) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            // Normal, no hay conexión realmente
        } else {
            perror("accept");
        }
    }
}
```

```
} else {
    set_nonblocking(client_fd); // Cliente también no bloqueante
    // Agregar a poll... } }
```

Explicación más detallada de por que hacer no bloqueante:

Después de `epoll_wait()`, las llamadas `accept()` / `recv()` pueden bloquearse

// epoll dice: "server_fd tiene actividad"

```
if (events[i].data.fd == server_fd) {
    int client_fd = accept(server_fd, NULL, NULL);
```

// ↑ Puede BLOQUEARSE si:

// - Hubo race condition

// - Cliente cerró conexión antes de accept() }

Solución (modo no bloqueante. Obligatorio para IRC):

```
set_nonblocking(server_fd);
```

```
int client_fd = accept(server_fd, NULL, NULL);
```

```

if (client_fd == -1 && errno == EAGAIN) {
    // No pasa nada, no hay conexión ahora}

... y lo mismo con recv()

// epoll dice: "client_fd tiene datos"
char buffer[512];
ssize_t bytes = recv(client_fd, buffer, 512, 0);
// ↑ Podría bloquearse si datos desaparecieron (race condition)

```

getaddrinfo()

#include <sys/types.h> <sys/socket.h> <netdb.h> (No hay version C++. Solo C).

Resuelve hostname o IP a una dirección de socket utilizable. Reemplaza a las obsoletas gethostbyname() e inet_addr()

```

int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints,
                struct addrinfo **res); //sintaxis

```

node : Hostname ("localhost") o IP ("127.0.0.1") o NULL

service : Puerto ("6667") o nombre servicio ("http")

hints : Criterios de búsqueda (opcional, puede ser Null)

res : Salida: Lista enlazada de resultados

Devuelve 0 como éxito y cualquier otro valor como código de error (usar gai_strerror())

Estructura addrinfo:

```

struct addrinfo {
    int ai_flags;           // Flags (AI_PASSIVE, etc.)
    int ai_family;         // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype;       // SOCK_STREAM, SOCK_DGRAM
    int ai_protocol;       // IPPROTO_TCP, IPPROTO_UDP, 0
    socklen_t ai_addrlen;   // Tamaño de ai_addr
    struct sockaddr *ai_addr; // Dirección del socket
    char *ai_canonname;     // Nombre canónico
    struct addrinfo *ai_next; // Siguiente resultado (lista enlazada)};

```

ai_flags : Banderas

- AI_PASSIVE** : Para servidores (bind a 0.0.0.0)
- AI_CANONNAME** : Devuelve nombre canónico
- AI_NUMERICHOST** : node es IP numérica, no hostname

Ejemplo Servidor (bind)

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

struct addrinfo hints, *res;

// 1. Configurar hints
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;           // IPv4
hints.ai_socktype = SOCK_STREAM;     // TCP
hints.ai_flags = AI_PASSIVE;         // Para servidor

// 2. Resolver
int status = getaddrinfo(NULL, "6667", &hints, &res); //aquí se hace malloc de una lista de
//estructuras y la dirección del primer nodo apuntará estará reflejado por lo que apunta res. O sea que hay
//que liberarlo al final con freeaddrinfo(res).
if (status != 0) {
    std::cerr << "getaddrinfo: " << gai_strerror(status) << std::endl;
    return 1;}

// 3. Crear socket con resultado
int server_fd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// 4. Bind
bind(server_fd, res->ai_addr, res->ai_addrlen);

// 5. IMPORTANTE: Liberar memoria
freeaddrinfo(res);
```

freeaddrinfo()

Libera la memoria reservada por **getaddrinfo()**. Ejemplo justo arriba.

inet_ntoa() / inet_ntop()

Convierten una dirección IP binaria (network format) a string legible. **struct in_addr** → "127.0.0.1" **inet_ntoa()** es obsoleta. Ahora se usa **inet_ntop()**

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

af : AF_INET (IPv4) o AF_INET6 (IPv6)

src : Dirección binaria que se guarda en struct in_addr como uint32_t (0x7F000001)

dst : Buffer donde escribir el string

size : Tamaño del buffer

INET_ADDRSTRLEN : IPv4

INET6_ADDRSTRLEN : IPv6

Devuelve dst como éxito y NULL en error

Ejemplo:

```
struct sockaddr_in client;  
socklen_t len = sizeof(client);
```

```
accept(fd, (struct sockaddr *)&client, &len);
```

```
// IP binaria:
```

```
client.sin_addr
```

```
char ip[INET_ADDRSTRLEN];  
inet_ntop(AF_INET, &client.sin_addr, ip, INET_ADDRSTRLEN);
```

getsockname()

Obtiene la dirección IP y puerto local asociados a un socket ya creado. Sirve para saber en qué IP está bindeado el socket, y qué puerto usa realmente. Se usa después de bind() y después de connect() para obtener la IP local y el puerto asignado automáticamente.

```
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen); //sintaxis
```

sockfd : FD de un socket válido

addr : Salida: dirección local

addrlen : Entrada: tamaño / Salida: tamaño real.

Devuelve 0 éxito y -1 en error

```
int fd;
struct sockaddr_in addr;
socklen_t len;
char ip[INET_ADDRSTRLEN];

fd = socket(AF_INET, SOCK_STREAM, 0);

memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(0); // puerto automático

bind(fd, (struct sockaddr *)&addr, sizeof(addr));

len = sizeof(addr);
getsockname(fd, (struct sockaddr *)&addr, &len);

inet_ntop(AF_INET, &addr.sin_addr, ip, INET_ADDRSTRLEN);

printf("IP local: %s\n", ip);
printf("Puerto local: %d\n", ntohs(addr.sin_port));
```