

FT IRC

Sockets:

Un socket es un punto final para la comunicación en una red. Es decir, representa un extremo de la comunicación en una red(en este caso server-client).

Se utiliza para enviar y recibir datos entre aplicaciones que pueden estar en el mismo dispositivo o en diferentes dispositivos conectados a una red.

Lo que haremos en este proyecto podría resumirse como la comunicación de dos aplicaciones, una, (el **servidor**) escucha en un puerto específico, mientras que la otra aplicación (el **cliente**) se conecta a ese puerto. Una vez establecida la conexión, se pueden enviar datos entre el cliente y el servidor.

Establecer Conexiones: Los sockets permiten que un programa establezca conexiones con otros programas a través de una red.

Comunicación: Facilitan la comunicación bidireccional, lo que permite enviar y recibir datos entre aplicaciones.

Interoperabilidad: Permiten que aplicaciones escritas en diferentes lenguajes de programación y que se ejecutan en diferentes plataformas se comuniquen entre sí.

Flexibilidad: Los sockets pueden ser utilizados tanto para aplicaciones de cliente como de servidor, lo que permite una amplia gama de aplicaciones, desde navegadores web hasta aplicaciones de chat y juegos en línea.

Para crear un socket, usaremos la función socket:

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

- **Retorno:**

Si la creación del socket es exitosa, **socket ()** devuelve un descriptor de archivo (un número entero positivo) que representa el socket recién creado.

Si hay un error, devuelve -1 y establece la variable global **errno** para indicar el error específico.

- **Parametros:**

domain: Especifica el dominio de la red en el que se va a operar. Comúnmente se utilizan:

AF_INET: para direcciones IPv4.

AF_INET6: para direcciones IPv6.

AF_UNIX: para comunicación entre procesos en la misma máquina.

type: Especifica el tipo de socket que se va a crear. Los tipos más comunes son:

SOCK_STREAM: para un socket TCP (orientado a conexión).

SOCK_DGRAM: para un socket UDP (no orientado a conexión).

SOCK_RAW: para acceso a paquetes de red sin procesar.

protocol: Generalmente se establece en 0 para que el sistema elija el protocolo adecuado según el tipo de socket especificado. Si se utiliza un protocolo específico, se puede indicar aquí.

Lo siguiente es configurar las opciones específicas del socket(servidor), utilizaremos la función `setsockopt()`:

```
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

- **Parámetros:**

sockfd: Es el descriptor de archivo del socket que se va a configurar. Este descriptor se obtiene previamente al crear el socket con la función `socket()`.

level: Especifica el nivel de la opción que se va a establecer. Por ejemplo, puedes usar:

SOCKET: para opciones que son comunes a todos los sockets. Indicamos que las opciones que vamos a establecer son comunes a todos los sockets y no están restringidas a un protocolo específico como TCP o UDP

optname: Especifica el nombre de la opción que se va a configurar. Algunos ejemplos son:

SO_REUSEADDR: permite reutilizar direcciones locales.

Cuando un socket (en este caso, el socket del servidor) se cierra, entra en un estado llamado **TIME_WAIT** para asegurarse de que todos los datos se entreguen correctamente. Sin **SO_REUSEADDR**, no podríamos vincular un nuevo socket (nuevo socket del servidor) a la misma dirección hasta que este estado finalice, lo que podría causar retrasos y limitar la capacidad del servidor para aceptar nuevas conexiones.

Al habilitar **SO_REUSEADDR**, permitimos que el servidor reutilice la dirección del socket (el socket del servidor) rápidamente, mejorando la eficiencia y la disponibilidad del servicio.

&opt: Este es un puntero a la opción que estás configurando. En este caso, **&opt** es un puntero al entero **opt** que tiene el valor **1**. Este valor indica que deseas habilitar la opción **SO_REUSEADDR**. Si estuvieras deshabilitando la opción, **opt** sería **0**.

Puede ser un puntero a un entero, una estructura o cualquier tipo de dato que sea relevante para la opción específica que deseas establecer. En otros casos se requiere una estructura que marca el comportamiento del socket cuando se cierra, especificando cuánto tiempo debe esperar el socket antes de cerrarse si hay datos pendientes, o el tamaño del buffer de envío que deseas, por eso la opción **SO_REUSEADDR**, es una opción muy sencilla de implementar.

El valor **&opt** permite una gran flexibilidad al configurar sockets en tu aplicación.

Socket en modo no-bloqueante:

La función `fcntl` se utiliza para manipular los descriptores de archivo en Unix y sistemas similares.

Cuando se trabaja con sockets y se utilizan funciones como `poll()`, `fcntl` es particularmente útil para configurar la propiedad de bloqueo de un socket.

Cuando un socket está configurado como no bloqueante, no se quedará esperando en una operación de lectura o escritura. Esto es importante en un servidor que maneja múltiples clientes, ya que permite al servidor continuar procesando otras conexiones mientras espera que lleguen datos.

Por ejemplo, si un socket de cliente no tiene datos disponibles para leer, `recv()` se quedará esperando indefinidamente, impidiendo que el servidor procese otras conexiones o realice otras tareas.

Al configurar los sockets como no bloqueantes, el servidor puede continuar procesando otras conexiones.

Estructura sockaddr_in

Es una estructura en C que se utiliza para representar direcciones de sockets en la familia de protocolos de Internet (IPv4).

Se usa para almacenar la dirección IP y el número de puerto necesarios para identificar un punto final en una comunicación de red.

Puerto:

Es un número que identifica de manera lógica y única un servicio o aplicación en una máquina o dispositivo, dentro de una red.

Un puerto permite que múltiples servicios se ejecuten simultáneamente en un mismo dispositivo con una única dirección IP, diferenciando las conexiones según el tipo de servicio mediante un número único.

Y dentro de un dispositivo, como un sistema operativo, los file descriptors (FD) se utilizan para manejar operaciones de entrada y salida (I/O) a nivel del sistema, no solo en red, sino también para archivos, dispositivos, sockets, etc.

Campos de `sockaddr_in`

sin_family:

- Especifica la familia de direcciones. Para IPv4, su valor siempre será `AF_INET`.
- Tipo: `sa_family_t`

sin_port:

- Contiene el número de puerto para la conexión. Debe almacenarse en formato de red (big-endian), por lo que generalmente se usa la función `htons()` para convertir el número de puerto del formato de host al formato de red.
- Tipo: `in_port_t`

sin_addr:

- Almacena la dirección IP del socket. Este campo es de tipo struct in_addr, el cual contiene un único miembro s_addr que representa la dirección IP en formato binario (network-byte order). Para convertir una dirección IP humana (ej. "192.168.1.1") en una forma que esta estructura pueda usar, se puede utilizar inet_pton() o inet_addr().
- Tipo: struct in_addr

sin_zero[8]:

- Es un array de 8 bytes que no se utiliza, pero está presente para asegurar que la estructura sockaddr_in tenga el mismo tamaño que la estructura genérica sockaddr. Esto es para garantizar la compatibilidad cuando las funciones del sistema que manejan sockets reciben una sockaddr como argumento.

Vincular bind()

Asocia el socket del servidor con una dirección IP y un puerto específico, de manera que pueda escuchar conexiones entrantes

```
bind(this->_serverFd, (struct sockaddr *)&this->_serverAddr, sizeof(this->_serverAddr))
```

Parámetros de bind()

1. this->_serverFd:

- Este es el file descriptor (FD) del socket del servidor que se creó previamente con la función socket(). Este identificador numérico es lo que usas para interactuar con el socket.
- El socket fue creado para manejar conexiones en red (TCP o UDP) y aquí es donde lo estás asociando a una dirección IP y puerto mediante la función bind().

2. (struct sockaddr *)&this->_serverAddr:

- Este es un puntero a una estructura que contiene la dirección IP y el puerto que el servidor usará para escuchar conexiones entrantes.
- struct sockaddr_in (en este caso, la estructura a la que apunta this->_serverAddr) es la versión específica de sockaddr para IPv4.
- La estructura sockaddr_in contiene varios campos, entre ellos:
 - sin_family: la familia de direcciones, en este caso AF_INET para IPv4.
 - sin_port: el puerto del servidor (en formato de red, es decir, big-endian).
 - sin_addr: la dirección IP del servidor (puede ser cualquier IP en la máquina o una específica).
- El uso del casting (struct sockaddr *) es necesario porque la función bind() espera un puntero de tipo sockaddr, que es una estructura genérica para varias familias de direcciones (IPv4, IPv6, etc.), pero this->_serverAddr es de tipo sockaddr_in que es más específico.

3. sizeof(this->_serverAddr):

- Este es el tamaño de la estructura sockaddr_in.
- La función bind() necesita saber cuánto espacio ocupa la estructura de la dirección para realizar la asociación correctamente.
- Como this->_serverAddr es de tipo sockaddr_in, aquí estás pasando el tamaño de esa estructura con sizeof.

Socket en modo de escucha listen()

`listen(this->_serverFd, MAX_CLIENTS)`

Ponemos el socket en modo escucha. La función `listen()` convierte el socket del servidor en un socket pasivo que puede aceptar conexiones entrantes. Hasta este punto, el socket estaba simplemente configurado y vinculado a una dirección, pero no estaba esperando activamente conexiones.

Después de llamar a `listen()`, el servidor esperará que los clientes intenten conectarse. El siguiente paso sería usar `accept()` para aceptar una conexión entrante desde un cliente, pero antes debemos gestionar las múltiples conexiones de clientes.

Estructura pollfd

La estructura `struct pollfd` se utiliza junto con la función `poll()` en C/C++ para gestionar la multiplexación de entrada/salida en varios file descriptors, como sockets, de una manera eficiente.

`struct pollfd` permite monitorear múltiples file descriptors en busca de eventos (lectura, escritura, errores).

Se utiliza junto con `poll()` para implementar un modelo de E/S no bloqueante, ideal para servidores que manejan múltiples conexiones simultáneas sin bloquearse en una sola operación de E/S.

Los campos `fd`, `events` y `revents` proporcionan una forma de especificar qué file descriptor observar, qué eventos esperar y qué eventos han ocurrido.

Descripción de los campos:

1. `fd`:

- El file descriptor que se va a monitorear. Puede ser un socket, un archivo o cualquier otro descriptor de archivo.
- Por ejemplo, si estás esperando conexiones entrantes en un socket de servidor, este campo contendría el file descriptor del socket.

2. `events`:

- Especifica los eventos que te interesa monitorear en el `fd`.
- Estos eventos son una combinación de flags (máscaras de bits) que indican las acciones que te interesan, como disponibilidad para lectura o escritura.
- Algunos valores comunes que se pueden monitorear:
 - `POLLIN`: Entrada disponible (por ejemplo, datos recibidos en un socket).
 - `POLLOUT`: El socket está listo para enviar datos (escritura posible sin bloqueo).
 - `POLLERR`: Un error ocurrió en el file descriptor.
 - `POLLHUP`: Se ha colgado la conexión (el socket ha sido cerrado por el otro extremo).

3. `revents`:

- Después de llamar a `poll()`, este campo contendrá los eventos que realmente ocurrieron en el file descriptor. Es una salida de la función `poll()` y te permite saber qué tipo de eventos han ocurrido en el `fd`.
- Ejemplo: Si un socket tiene datos listos para ser leídos, este campo contendrá el flag `POLLIN`.

Poll()

Una vez hemos configurado la estructura pollfd para que este preparada para los datos recibidos events = POLLIN, se ejecutara la funcion:

```
poll(this->_pollfds.data(), this->_pollfds.size(), -1)
```

Que monitoreara el puntero al array de estructuras _pollfds.data() para ver si hay el evento que configuramos disponible.

Cómo poll() maneja la estructura pollfd:

Entrada (events):

En cada estructura pollfd, antes de llamar a poll(), defines los eventos que quieres monitorear para cada file descriptor. Esto se hace a través del campo events de la estructura pollfd. Por ejemplo, puedes poner la bandera POLLIN si quieres que poll() te avise cuando haya datos disponibles para leer en un socket.

Salida (revents):

Después de que poll() retorna, revisas el campo revents de la estructura pollfd para ver qué eventos han ocurrido. Si hay datos disponibles en un FD, se activará la bandera POLLIN en revents. Esto te permite saber qué file descriptor tiene datos listos o necesita atención.

Flujo de poll():

- Cuando llamas a poll(), la función examina los file descriptors que has pasado en el array de estructuras pollfd.
- Antes de poll():
 - Configuras los file descriptors y los eventos que quieres monitorear (por ejemplo, POLLIN para lecturas).
- Durante la ejecución de poll():
 - poll() espera hasta que ocurra un evento en alguno de los file descriptors o hasta que se alcance el tiempo límite especificado (o indefinidamente si usas -1).
- Después de poll():
 - El campo revents de cada pollfd se actualizará con los eventos que ocurrieron (por ejemplo, si hay datos listos para leer, POLLIN se establecerá en revents).
 - La aplicación puede revisar revents y actuar en consecuencia.