Guía de Sockets

... o cómo conectar dos procesos sin morir en el intento

Objetivo

El presente documento tiene como fin explicar conceptos básicos de redes para luego poder adentrarse en qué son, cómo funcionan y cómo implementar sockets para crear un modelo cliente-servidor funcional para el trabajo práctico cuatrimestral

¿Qué es un cliente-servidor?

Antes de poder adentrarnos a hablar de sockets y cómo conectar dos procesos en C, primero hay que definir y tener bien en claro un par de conceptos de redes que se necesitan para poder arrancar.

Casi todo el mundo de la informática está tomado por la arquitectura de cliente-servidor. En ella definimos dos partes que son, como el nombre indica, el cliente y el servidor.

El propósito del cliente es iniciar una conexión contra el servidor para realizar solicitudes, y este último es el que este va a entender y resolver. Cabe mencionar que si el servidor no está andando, el cliente no puede realizar ningún tipo de solicitud, y por lo tanto no puede operar.

Por otro lado, es responsabilidad del servidor aceptar las conexiones entrantes de los muchos clientes, es decir, ser capaz de atender varios clientes a la vez, y mantener disponibilidad de sus servicios una vez se haya terminado la solicitud de un cliente; por ejemplo, imagínense cómo sería el mundo si Google atendiera a un usuario a la vez cada solicitud que se hiciera una búsqueda, o si se cayera su servidor cada vez que un cliente cierra la pestaña.

A su vez, los servidores pueden ser clientes de otros servidores. Esta jerarquía puede ser indefinida.

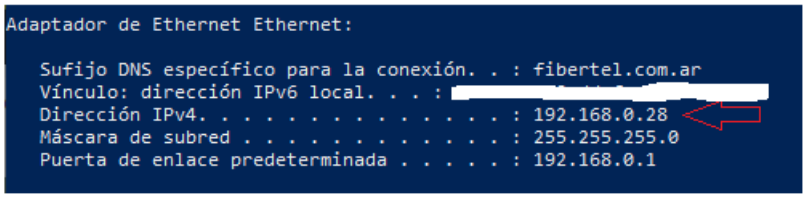
Algunas preguntas clave para saber dentro de una arquitectura quién es cliente y quién es servidor:

- ¿Quién inicia la conexión con quién?
- ¿Qué proceso necesita atender a N de otro?
- ¿Qué proceso puede continuar funcionando con normalidad si se cae otro?

IPs y puertos. ¿Dónde atraco el barco?

Cuando tenemos dos computadoras o más dentro de una misma red (sea esta doméstica, empresarial o pública) dentro de esa red cada uno de los dispositivos tiene una IP, un ID único para cada dispositivo dentro de la red. Las IPs tienen la particularidad de ser cuatro números, separados por un punto, donde cada uno va de 0 a 255 (por ejemplo, 192.168.0.16). Si ahora mismo abrimos un cmd, powershell o cualquier otra terminal y escriben ipconfig para Windows o ifconfig para Linux y MacOSx , vamos a ver una pantalla que, entre otras cosas, nos dirá algo así como 192.168.algo.algo. Esa es la IP que la

computadora tiene asignada dentro de nuestra red doméstica. Los celulares, otras computadoras u otros dispositivos que se conecten a Internet tienen sus propias IPs asignadas, todas distintas entre sí.



Luego, por otro lado, tenemos los puertos. Estos son unidades lógicas dentro del sistema operativo para que una misma computadora, y por lo tanto la misma IP, pueda tener varios procesos corriendo que necesiten operar dentro de la red. ¿Por qué necesitaríamos que nuestras computadoras sean capaces de tener más de un proceso de red corriendo? Tomen por ejemplo sus PCs en estos momentos, con un navegador abierto, y en el background aplicaciones como Discord, Spotify, Steam, actualizaciones del SO y muchísimas más corriendo en simultáneo. Cada una de estas ocupando un puerto distinto para conectarse a sus respectivos servidores, en sus respectivas IPs y puertos.

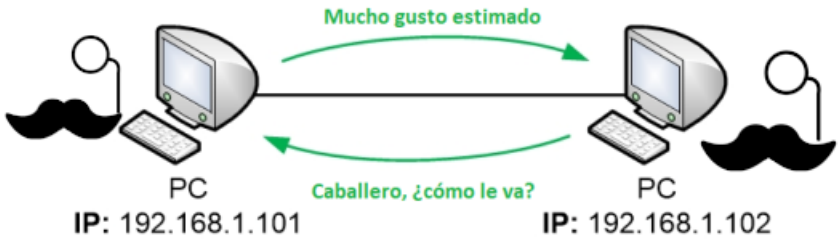
Cada sistema operativo tiene 65535 puertos (2^16), y tiene los primeros 1000 (por lo menos) reservados para sus tareas y otras cosas que no son relevantes para la materia. Lo importante es que cuando desarrollemos la comunicación entre nuestros procesos, debemos tomar puertos que estén libres porque, como ya dijimos, dos procesos no pueden ocupar el mismo puerto al mismo tiempo.

En síntesis, podemos pensar en la red como una calle donde la IP es análoga a la altura donde está el edificio al que queremos llegar, y el puerto es análogo al timbre/departamento que queremos llamar.

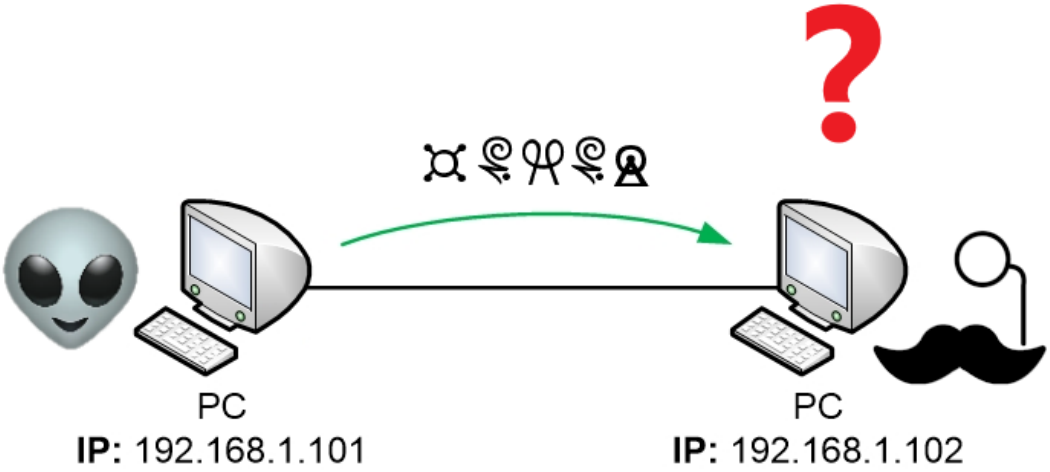
Protocolo de comunicación

Podemos definir un protocolo como una serie de pasos a seguir ante determinado evento. Por lo tanto, un protocolo de comunicación no es más que una definición y estandarización de cómo se van a estar comunicando dos procesos distintos en la red. Podríamos hacer una analogía de decir que para que dos procesos se puedan comunicar, necesitan estar hablando en el mismo idioma (protocolo).

Cuando dos procesos van a conectarse, lo primero que se realiza es una operación llamada handshake. No es más que una presentación por parte del cliente para hacerle saber al servidor que cumplen el mismo protocolo, y por lo tanto pueden entablar una comunicación.



Por otro lado, también es responsabilidad del servidor detectar cuando se le quiere conectar un cliente de otro protocolo para rechazar esa conexión

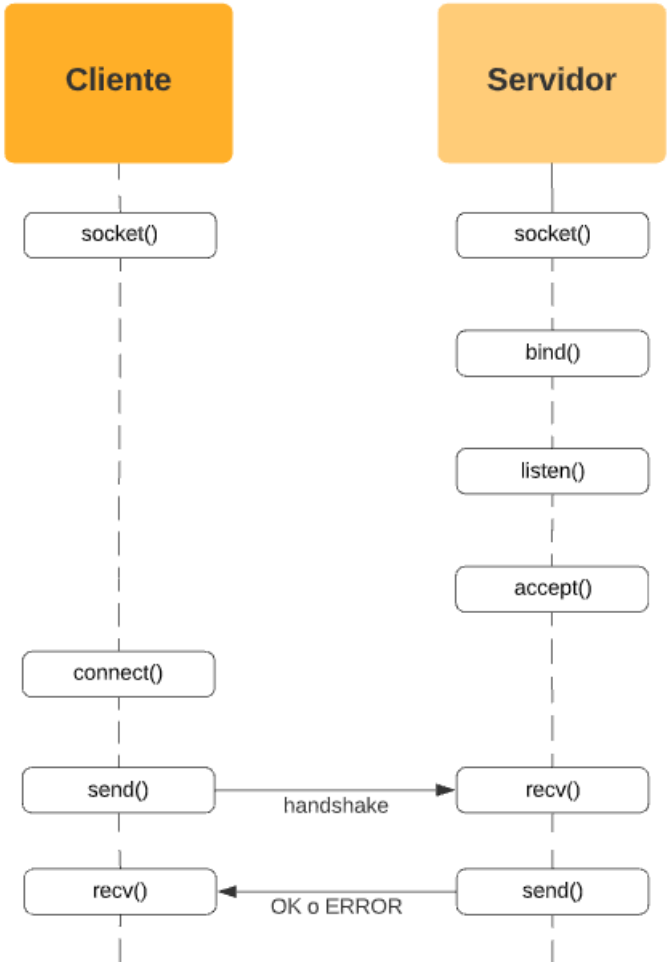


Sockets y Soquetes

Hablamos largo y tendido sobre las conexiones entre procesos, cómo se hacen y sobre los clientes y servidores que necesitan comunicarse mediante esas conexiones. Bueno, un socket es la representación que el sistema operativo le da a esa conexión. Cuando un cliente quiere iniciar una conexión contra su servidor, lo que hace es solicitarle al sistema operativo que cree un socket, hacer que ese socket ocupe un puerto libre, y se conecte con el socket del servidor que está a la espera de nuevos clientes, ocupando también un puerto del servidor.

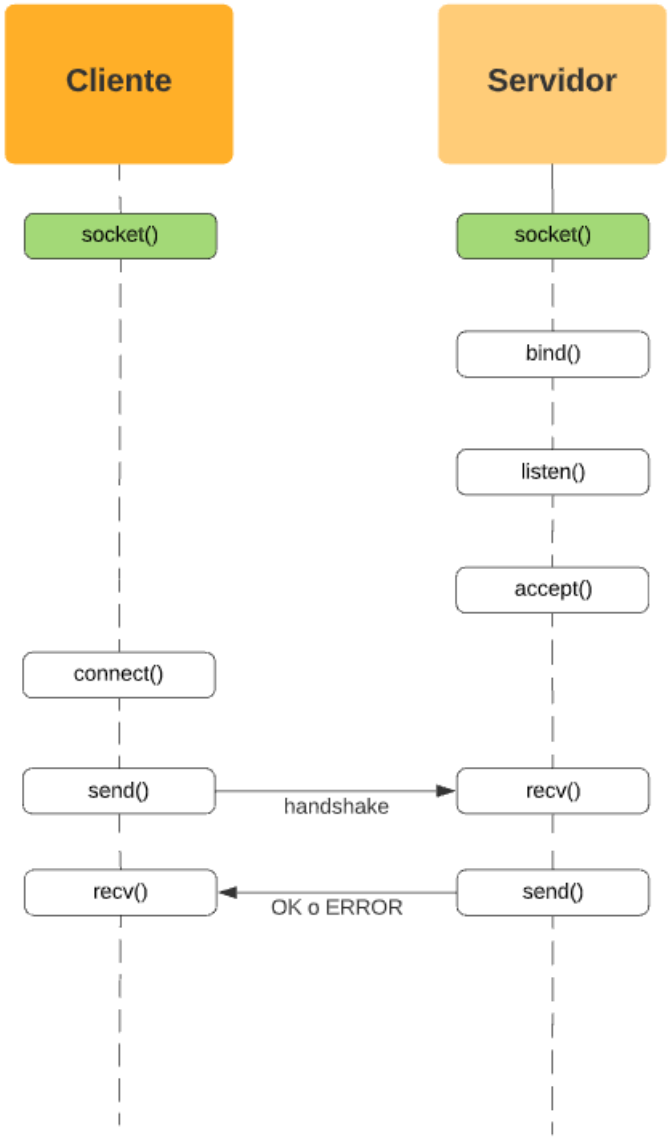
Algunos se estarán preguntando; si interviene el sistema operativo, ¿hay llamadas al sistema? SÍ

Las llamadas al sistema que el cliente tiene que realizar para conectarse a su servidor son distintas a las que el servidor tiene que hacer para prepararse para recibir conexiones entrantes de los clientes. Por suerte, vamos a explicar el flujo y orden de cada una de ellas.



> **WARNING**
>
> El diagrama indica la secuencia en la que deberían ser llamadas las funciones. Que éstas estén al mismo nivel, no significa que deban suceder exactamente al mismo instante.

socket()



La primera syscall que hay que utilizar para iniciar una conexión por socket entre dos procesos es socket. Esta lo que hace es generar lo que se llama un file descriptor, que son básicamente los IDs que Linux utiliza para representar cualquier cosa del sistema (archivos, bloques de memoria, teclados, impresoras, monitores, discos rígidos, etc). Estos file descriptors son representados en los programas C por un entero, lo cual no quiere decir que todo entero sea un file descriptor.

Para poder crear un socket cliente con un servidor corriendo en la misma máquina, podemos hacerlo de la siguiente manera:

c

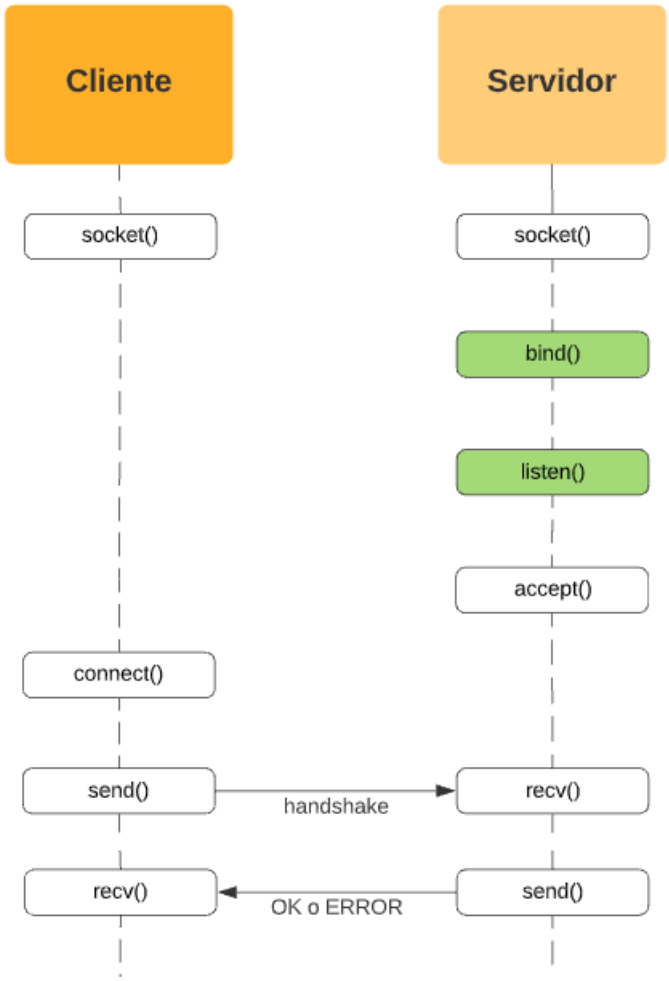
```c
struct addrinfo hints;
struct addrinfo *server_info;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("127.0.0.1", "4444", &hints, &server_info);

int socket = socket(server_info->ai_family,
                    server_info->ai_socktype,
                    server_info->ai_protocol);

```

```
14 freeaddrinfo(servinfo);
```

Para el caso particular de `socket()` no vamos a entrar en muchos detalles de qué es cada parámetro de la función porque son conceptos que se escapan del alcance de la materia (y se verán en detalles en una correlativa de ésta). A fines didácticos, quedemos en que esta configuración nos crea un socket capaz de hacer todo lo que vamos a necesitar durante el cuatrimestre.

`getaddrinfo()` es una llamada al sistema que devuelve información de red sobre la IP y puerto que le pasemos, en este caso del servidor. Escapa un poco del objetivo de este documento su funcionamiento, pero lo único que necesitamos saber es que inyecta los datos necesarios para la creación de sockets en la variable `server_info` que le estamos pasando. Liberamos la memoria que ocupó con `freeaddrinfo()` .

bind() y listen()



Ya dijimos que los clientes para poder comunicarse con sus servidores, deben hacerlo a través de la IP del servidor físico en el que estén corriendo y el puerto que estén ocupando en dicho servidor a la espera de nuevas conexiones por parte de los clientes. Las llamadas al sistema que realizan esa preparación por parte del proceso servidor son `bind()` y `listen()` .

`bind()` lo que hace es tomar el socket que creamos con anterioridad y pegarlo con pegamento industrial al puerto que le digamos.

Por otro lado, `listen()` toma ese mismo socket y lo marca en el sistema como un socket cuya única responsabilidad es notificar cuando un nuevo cliente esté intentando conectarse.

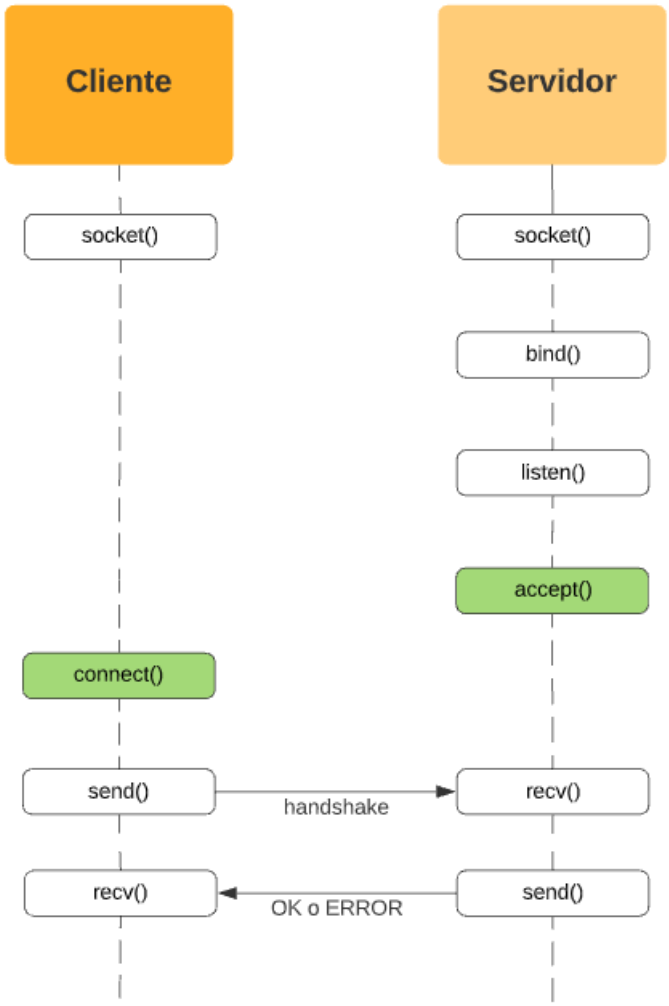
Una vez realizados ambos pasos, nuestro servidor está listo para recibir a los clientes.

```c
int socket_servidor;

struct addrinfo hints, *servinfo;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

getaddrinfo(NULL, "4444", &hints, &servinfo);

socket_servidor = socket(servinfo->ai_family,
                         servinfo->ai_socktype,
                         servinfo->ai_protocol);

bind(socket_servidor, servinfo->ai_addr, servinfo->ai_addrlen)

listen(socket_servidor, SOMAXCONN);

freeaddrinfo(servinfo);

```

`bind()` está recibiendo el puerto que debe ocupar a partir de los datos que le suministramos al getaddrinfo con anterioridad. En este caso estamos diciendo que obtenga información de red sobre la IP "127.0.0.1", osea la misma máquina en la que está corriendo en lugar de otra, en el puerto 4444, arbitrario elegido para este ejemplo. Le decimos que obtenga información sobre la computadora local porque es en la computadora local donde estamos tratando de levantar el servidor para que los clientes en otras computadoras se puedan contectar.

Luego, `listen()` recibe como segundo parámetro la cantidad de conexiones vivas que puede mantener. `SOMAXCONN` como indica el nombre, es la cantidad máxima que admite el sistema operativo.

accept() y connect()



Una vez el socket del servidor se marcó en modo de escucha, estamos preparados para empezar a recibir las conexiones de nuestros clientes.

Para hacer esto, el servidor utiliza la llamada al sistema `accept()` , la cual es bloqueante. Esto significa que el proceso servidor se quedará bloqueado en accept hasta que le llegue un cliente.

Cuando el cliente intente conectarse al servidor, lo hará mediante la llamada al sistema connect(). Si el servidor no está en accept, connect fallará y devolverá un error.

Servidor:

```c
int socket_cliente = accept(socket_servidor, NULL, NULL);
```

Cliente:

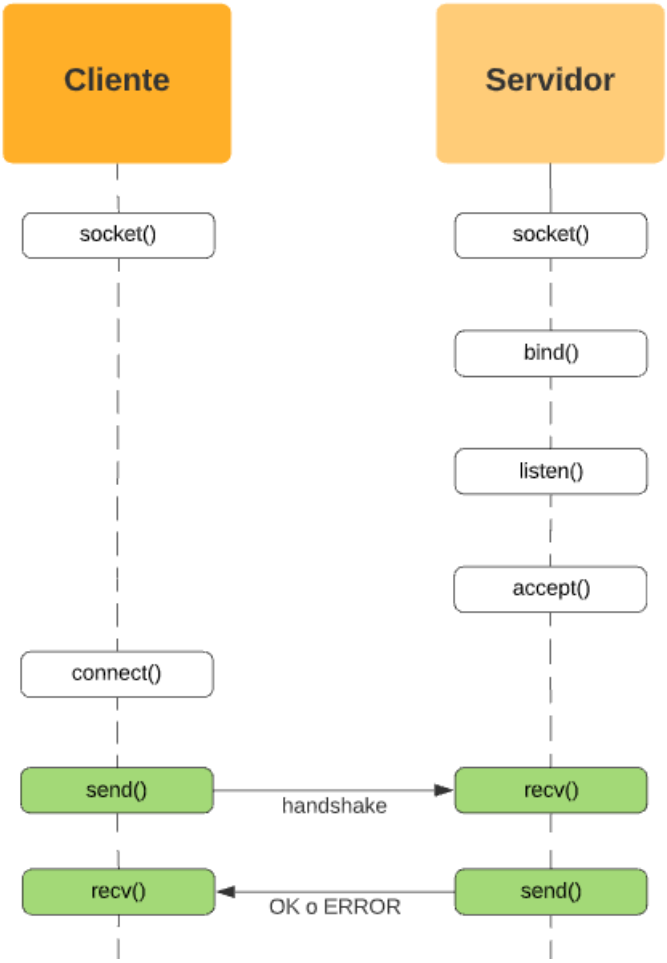
```c
//continuando sobre cuando creamos el socket del cliente

connect(socket_cliente, server_info->ai_addr, server_info->ai_addrlen);

```

Y así de simple, nuestros procesos ya están conectados.

Una vez que el cliente fue aceptado, accept retorna un nuevo socket (file descriptor) que representa la conexión BIDIRECCIONAL entre el servidor y el cliente. Esto quiere decir que la comunicación entre el cliente y el servidor se hace entre el socket que realizó connect hace el servidor, y el socket que fue

devuelto por el accept. El socket que está marcado como listen no participa de esta comunicación. Su única función es escuchar la llegada de nuevos clientes mediante accept.

send() y recv()



Una vez establecida la conexión entre el cliente y el servidor, ya estamos casi listos para comenzar a enviar mensajes libremente entre ambos. El último paso que debemos cumplir es el proceso de handshake, que como ya dijimos antes, su único propósito es enviar un paquete que le informe al servidor cuál es el protocolo con el que está intentando iniciar una conversación para que el servidor le conteste si es capaz de entender ese protocolo (OK) o le informe que no en caso contrario (ERROR).

Entonces, a modo de ejemplo, tomemos un servidor super minimalista donde su handshake es que el cliente le envíe un `uint32_t` (entero no signado de 32 bits) cuyo valor sea 1. Digamos también por simpleza que cuando el servidor responde 0 es OK y -1 es ERROR.

Cliente:

```c
uint32_t handshake = 1;
uint32_t result;

send(socket_cliente, &handshake, sizeof(uint32_t), NULL);
recv(socket_cliente, &result, sizeof(uint32_t), MSG_WAITALL);

//aca va alguna lógica para manejar el OK o el ERROR del servidor

```

Servidor:

```c
uint32_t handshake;
uint32_t resultOk = 0;
uint32_t resultError = -1;

recv(socket_cliente, &handshake, sizeof(uint32_t), MSG_WAITALL);
if(handshake == 1)
   send(socket_cliente, &resultOk, sizeof(uint32_t), NULL);
else
   send(socket_cliente, &resultError, sizeof(uint32_t), NULL);
```

`recv` en este caso es bloqueante debido a que le estamos pasando el flag `MSG_WAITALL`, que se encarga de esperar a que llegue por socket la cantidad de bytes que le decimos en el tercer parámetro. A causa de esto, el cliente espera la respuesta del handshake antes de continuar.

Por otro lado, si el cliente en lugar de enviar un 1, estuviera enviando otro entero, o un `char` con el valor "1", el servidor le devolvería error, porque éste entiende que está solicitando comunicarse mediante otro protocolo cuyo handshake sea ese.

Una vez pasado el proceso de handshake, ya el cliente se encuentra en vía libre para poder enviarle otros mensajes al servidor para que éste le conteste con los resultados de sus solicitudes. Tanto `send` como `recv` se encargan de mover bytes de datos a través de la red, y eso es lo que representa el segundo parámetro de ambas funciones, y ese es el motivo por el que reciben una posición de memoria.

Todo muy lindo para los datos "simples" (int, char, char*/string, float, etc), pero si la operación a solicitarle al servidor requiere más de un parámetro estamos en un problema con el ejemplo visto. Para poder enviar datos más complejos a través de nuestros sockets, necesitamos enviar la información **serializada**.

> **WARNING**
>
> Para extender el concepto de **serialización**, tenemos disponible la **guía**, que incluye un modelo de paquete propuesto para el envío mediante sockets.

Algunos quizá estén pensando "¿por qué no puedo simplemente realizar múltiples sends para una sola operación/mensaje?" Porque al estar trabajando con paquetes de red no puedo garantizar el orden de llegada. Si necesito enviar un único mensaje simple cuyo contenido sea un entero o un string esto no es un problema ya que los datos primitivos están serializados en sí mismos, pero al complejizarse las solicitudes debido a que el servidor necesita más datos para poder procesarlos, serializar no es una recomendación, es una necesidad

close()

Por último, e igual de importante que todo lo demás, los sockets una vez que no los usemos más deben ser cerrados con `close(socket)`. Esto normalmente se realiza del lado del cliente ya que, como dijimos antes, el servidor debe dar disponibilidad constante para todas las solicitudes de todos los clientes que tenga en el tiempo que este viva. Cuando el cliente se desconecta, `recv` nos retorna el

valor 0 para que podamos manejar esa desconexión y cerrar el socket que (accept)amos con anterioridad.

Multiplexando ando

> **WARNING**
>
> Se recomienda fuertemente esperar a ver el concepto de "hilo" en la teoría de la materia y ver el **video de hilos en C** antes de comenzar este apartado.

Ya charlamos antes sobre que una de las condiciones para que un proceso servidor sea considerado servidor es necesario que sea capaz de atender a múltiples clientes de manera concurrente. El concepto de poder atender dos o más conexiones al mismo tiempo se conoce como multiplexación.

Pensemos en todo lo que hicimos hasta ahora. Creamos sockets en ambos cliente y servidor, marcamos el servidor en escucha a nuevas conexiones, y se queda bloqueado en accept hasta que su cliente haga connect. ¿Y si otro cliente hace connect mientras el servidor no está en accept por estar atendiendo al otro cliente? En general depende de la configuración de connect, pero con el ejemplo que venimos armando se quedaría bloqueado hasta que el servidor lo acepte, cosa que no va a pasar hasta que no termine con el otro cliente, y por lo tanto, tenemos un problema.

Necesitamos de alguna herramienta que sea capaz de paralelizar tareas dentro de un mismo proceso. ¿Será que el sistema operativo nos brinda algo capaz de hacer esto?

¡Ya sé! ¡Hilos! Si bien los hilos no son llamadas al sistema relacionadas a los sockets, sí podemos usarlos para poder paralelizar las tareas que solicitan los muchos clientes que se nos van a conectar, para así poder volver lo más rápido posible al accept con el socket en listen. Lo que se me ocurre que podemos hacer es algo por este estilo:

```c
while (1) {
   pthread_t thread;
   int *socket_cliente = malloc(sizeof(int));
   *socket_cliente = accept(socket_servidor, NULL, NULL);
   pthread_create(&thread,
                  NULL,
                  (void*) atender_cliente,
                  socket_cliente);
   pthread_detach(thread);
}
```

Entonces, de esta manera estamos creando un hilo por cada conexión nueva entrante, y ese hilo tendrá comunicación únicamente con el cliente cuyo socket le estemos pasando a través de `pthread_create()` .

El malloc lo realizamos debido a que como pthread recibe una posición de memoria, si usaramos siempre el mismo int, al acceder a su posición de memoria su valor se habría pisado, y por lo tanto todos los hilos tendrían siempre el mismo socket (y probablemente generaría condiciones de carrera).

Contenido complementario

Esta guía es un resumen bastante reducido y minimalista de la **guía Beej (en inglés)**, donde obviamos muchos conceptos de redes que no interesan para esta materia. Para los interesados, o aquellos que deseen ahondar más en busca de distintas formas de implementación, les comentamos que además hay una **traducción casera de la cátedra**. Si ven alguna oportunidad de mejora en redacción o traducción estamos abiertos a escucharlos.

Notas finales

¡Y eso fue todo! Les recordamos que cualquier duda que se les presente, la pueden realizar en los **medios de consulta** correspondientes, y que hagan uso de las guías y los video tutoriales de esta página.

Historial de versiones

v1.0 Publicación Inicial