Práctica 1 - Sockets

Alumno: Cazorla, Ignacio Nicolás
N° de alumno: 13926/1

Contenidos

1) Identifique similitudes y diferencias entre los sockets en C y en Java.

Similitudes:

- La interfaz para trabajar con sockets (los métodos/funciones) es muy similar entre ambos lenguajes (más allá de las cuestiones vinculadas a cada paradigma).

Diferencias:

Entre los clientes:

- Respecto del método socket() en ambos lenguajes.

- El intercambio de información entre los procesos para ambos lenguajes toma formas distintas, es decir, se abstraen de distinta manera.
    - En C: el socket es un file descriptor, que luego podemos usar para llamadas al sistema.
    - En Java: Crea directamente la conexión. Tenemos tres clases, una para lectura, otra para escritura y el socket como tal.

Entre los servidores:

- Respecto del método socket():
- El socket generado:
    - En C: no se conecta con el cliente, sirve para generar una conexión, utilizando otro socket nuevo que genera en el momento que llega una conexión.
    - En Java: Crea el socket donde va a esperar la conexión con los clientes, y por donde va a realizarse la comunicación con los clientes.
- La espera de conexiones:
    - En C: se usa la función listen() que sirve para esperar conexiones en un socket. También se le puede especificar la cantidad máxima de conexiones que va a establecer.
    - En Java: tenemos el método Accept() el cual mantendrá el socket a la espera de una conexión.
- La asociación del socket a un puerto:
    - En C: se usa el método bind() para asociar un socket con un puerto.
    - En Java: cuando se crea el socket se le indica el puerto que debe utilizar.

Para ambos casos:

- Lectura y escritura:
    - En C: se lee/escribe una determinada cantidad de datos directamente desde/hacia el socket.
    - En Java: no se lee/escribe desde/hacia el socket. Para la lectura se debe utilizar un objeto InputStream y para la escritura un OutputStream.

2) Tanto en C[1] como en Java (directorios csock-javasock):

a.- ¿Por qué puede decirse que los ejemplos no son representativos del modelo c/s?

No son representativos del modelo cliente/servidor, porque si bien los sockets son herramientas que permiten implementar el modelo, en estos casos lo único que representan, son una forma de comunicación entre procesos.

En un modelo c/s, los agentes clientes realizan peticiones a los servidores en espera de una respuesta, en este caso sólo se conectan y se envían mensajes sin realizar ninguna petición, solo un intercambio de mensajes.

b.- Muestre que no necesariamente siempre se leen/escriben todos los datos involucrados en las comunicaciones con una llamada read/write con sockets.

Sugerencia: puede modificar los programas (C o Java o ambos) para que la cantidad de datos que se comunican sea de 10^3 , 10^4 , 10^5 y 10^6 bytes y contengan bytes asignados directamente en el programa (pueden no leer de teclado ni mostrar en pantalla cada uno de los datos del buffer), explicando el resultado en cada caso.
Importante: notar el uso de “attempts” en “...attempts to read up to count bytes from file descriptor fd...” así como el valor de retorno de la función read (del man read).

Para mostrar los bytes que fluyen en la comunicación se modificaron los sockets (en C) de la siguiente manera:

El tamaño del buffer (tanto para el cliente como para el servidor) se define como:

```
size_t arr_size = pow(10,”exp”);          //”exp” = 3,4,5 y 6
```

En el cliente:

- Se genera un buffer con tamaño arr_size, lleno con caracteres “a”.

1) Exponente 3:

El tamaño del buffer será de 1000 bytes. En este caso se envían y se reciben todos los bytes.

Servidor	Cliente
$ ./server 3000 Size of received data: 1000	$ ./cliente localhost 3000

[1] Es necesario compilar los sockets de los ejercicios 2.b ,c y d con la opción -lm, para poder usar las funciones matemáticas.

	Enter exponent n, to express data size 10^n: 3 Initializing buffer... Buffer data size to send: 1000 I got your message

2) Exponente 4:

El tamaño del buffer será de 10000 bytes. Ocurre lo mismo que en el caso anterior.

Servidor	Cliente
$ ./server 3000 Size of received data: 10000	$ ./cliente localhost 3000 Enter exponent n, to express data size 10^n: 4 Initializing buffer... Buffer data size to send: 10000 I got your message

3) Exponente 5:

El tamaño del buffer será de 100000 bytes. Sólo llegan 65482. En algunas oportunidades sólo llegan 32741 bytes.

Servidor	Cliente
$ ./server 3000 Size of received data: 65482	$ ./cliente localhost 3000 Enter exponent n, to express data size 10^n: 5 Initializing buffer... Buffer data size to send: 100000 I got your message

4) Exponente 6:

El tamaño del buffer será de 1000000 bytes. Sólo llegan 65482, como en el caso anterior.

Servidor	Cliente
$ ./server 3000 Size of received data: 65482	$ ./cliente localhost 3000 Enter exponent n, to express data size 10^n: 6 Initializing buffer... Buffer data size to send: 1000000 I got your message

En cuanto a la función **read**[2], su valor de retorno es la cantidad de bytes que pudo leer. Esta función retornará 0 (cero) si es el fin de archivo o -1 si se produce un error.
No es un error si se leen menos bytes de los esperados, sino que esto puede ocurrir debido a que hay una menor cantidad de bytes disponibles para la lectura en un momento determinado.

En este caso, estamos usando sockets de tipo stream (TCP), esto fue indicado en la creación del socket (argumento SOCK_STREAM) :

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

La cantidad de datos que son recibidos es menor debido a que TCP tiene un tamaño de transferencia de 65536 bytes[3] por defecto. Por este motivo es que no se encuentran los datos totalmente disponibles a la hora de leerlos.

c.- Agregue a la modificación anterior una verificación de llegada correcta de los datos que se envían (cantidad y contenido del buffer), de forma tal que se asegure que todos los datos enviados llegan correctamente, independientemente de la cantidad de datos involucrados.

El mecanismo que se utilizó para resolver la llegada correcta de los datos, fue:

1) Recibir en el servidor el tamaño del dato que va a enviar el cliente (almacenado en data_size), para controlar que la cantidad de bytes que se recibe coincida con los que se deseaban enviar.
2) Recibir en el servidor un hash simple (suma los valores de los caracteres ascii y lo envía) y comparar una vez recibidos todos los datos si el hash recibido coincide con el hash que se calcula en el servidor. Se debe tener en cuenta que la función de hash usada no es utilizable para una situación real, ya que los strings “abc” y “cba” producirán los mismos resultados para la función.

A continuación se observa la iteración que realiza la operación **read()** hasta que se haya recibido la totalidad de los datos:

```
while(total_data_recv < data_size){
        n = read(newsockfd,buffer,65000);
        if (n < 0) error("ERROR reading from socket");
        total_data_recv = total_data_recv + n;
        strcat(rcv_data,buffer);
        bzero(buffer,arr_size);
        printf("Total data received: %d\n ", total_data_recv);
    }
```

[2] $ man read.
[3] Se puede apreciar que se reciben 65482 bytes de datos, esto es debido a que el total incluye a la cabecera que es de 54 bytes.

d.- Grafique el promedio y la desviación estándar de los tiempos de comunicaciones de cada comunicación. Explique el experimento realizado para calcular el tiempo de comunicaciones.

En este experimento, se utilizó una comunicación entre cliente y servidor, en la cual el cliente envía un mensaje con el string “hola” y el servidor responde con el mensaje “received”.

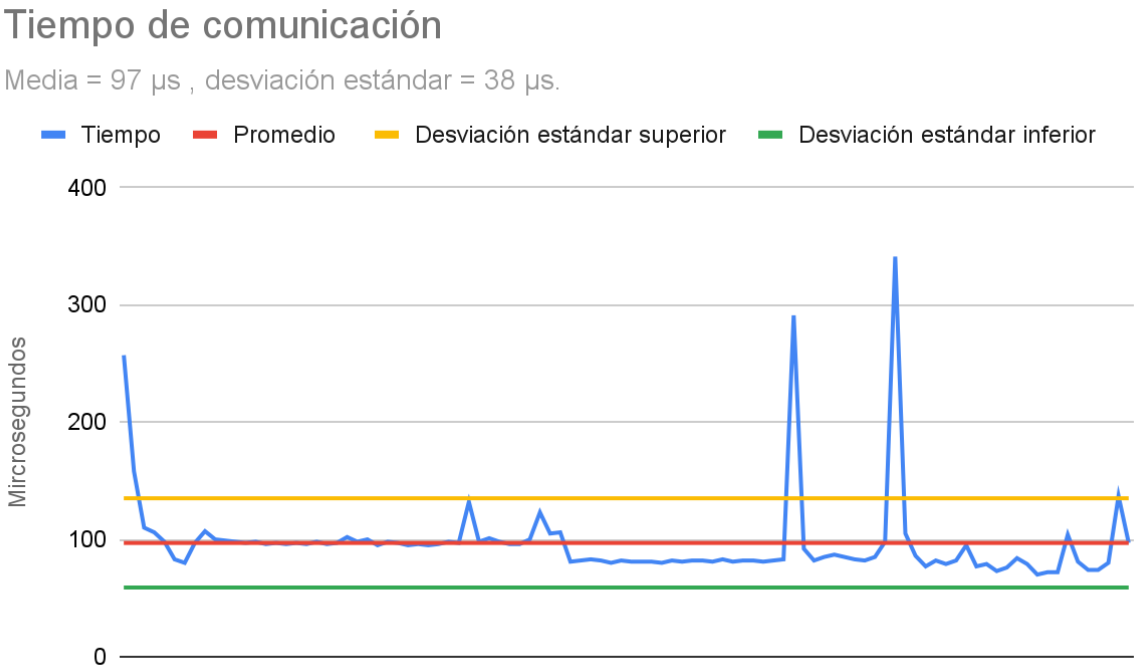
Para realizar la medición se utilizó una iteración en el cliente, dónde cada una de ellas envía y recibe un mensaje, tomando el tiempo de envío y respuesta de mensajes entre los agentes.

```
for (int i = 0; i < num_comunicaciones; i++){
       timetick = dwalltime();
       n = write(sockfd,"hello",5);
       if (n < 0) error("ERROR writing to socket");
       n = read(sockfd,buffer,256);
       if (n < 0) error("ERROR reading from socket");

       tiempo_actual = dwalltime() - timetick;

       printf("%.0f\n", tiempo_actual);
       tiempos[i] = tiempo_actual;

   }
```

El número que se estableció para num_comunicaciones es 100. Es decir, se repetirá el siguiente proceso 100 veces:

1) Obtener el tiempo actual con la función dwalltime()
2) Enviar un mensaje
3) Recibir un mensaje
4) Obtener el tiempo actual con la función dwalltime() nuevamente restando el tiempo obtenido en 1).
5) Imprimir y guardar en un arreglo de num_comunicaciones lugares, para poder realizar luego otros cálculos sobre los elementos.

Los resultados obtenidos son los siguientes:



3)¿Por qué en C se puede usar la misma variable tanto para leer de teclado como para enviar por un socket? ¿Esto sería relevante para las aplicaciones c/s?

Porque estamos usando un file descriptor, esa es toda la abstracción que hace C sobre los sockets. Las operaciones de lectura/escritura son system calls, lo que permite trabajar directamente sobre el socket, a diferencia de Java, donde estas operaciones son abstraídas por las clases de entrada y salida.

4)¿Podría implementar un servidor de archivos remotos utilizando sockets? Describa brevemente la interfaz y los detalles que considere más importantes del diseño. No es necesario implementar.

Mediante sockets se podría implementar un servidor de archivos remotos de la siguiente manera. Sería deseable tener en cuenta el funcionamiento del protocolo FTP, para implementar este sistema.

El **cliente**:

1) El cliente podría requerir loguearse (opcional)
2) Se conecta al servidor.

3) Realiza una petición por el archivo que desee descargar/subir. En caso de subir un archivo, debe enviar un hash para que el servidor pueda comprobar que el archivo fue recibido correctamente.

El **servidor**:

1) Espera por conexiones. Podría limitar la cantidad de conexiones simultáneas que puede atender.
2) Una vez que establece la conexión con el cliente, debe determinar si este tiene permiso para la acción que desea realizar.
    - En caso negativo, le informa que no puede seguir adelante. Caso contrario sigue en el paso 3.
3) Caso 1 (Petición de descarga) :
    - En caso de que exista el archivo, el servidor crea una nueva conexión (la cual podría ser un nuevo socket) y comienza la transmisión hacia el cliente. Esta finaliza cuando termina de enviar el último dato.
    - Si no existe, le informa al cliente y finaliza la conexión.

    Caso 2 (Petición de subida) : es necesario controlar:
    - Si hay espacio disponible para lo que se desea subir. Si es posible almacenar el archivo, se admite la operación.
    - Qué ocurre si hay un archivo con un nombre igual. Se sobrescribe, o se modifica el nombre?
    - No se debería permitir realizar ninguna operación (lectura/escritura) hasta que el archivo esté listo en el servidor.

    Finalmente el servidor debería poder comprobar que el archivo llegó correctamente.

Trabajando de manera similar a lo hecho hasta el momento (en C), desde la interfaz del cliente se podría hacer algo como lo siguiente, para descargar un archivo:
./client [dirección del servidor] [puerto] [--download, -d] FILENAME

Y para subida:

./client [dirección del servidor] [puerto] [--upload, -u] PATH

5)Defina qué es un servidor con estado (stateful server) y qué es un servidor sin estado (stateless server).

Un **servidor con estado**, almacena información del cliente, y/o relacionada con las comunicaciones con el cliente, tales que pueden ser consultadas en cualquier momento.

Por su parte, un **servidor sin estado**, cada operación que se realice hacia él, será una operación aislada, que no va a registrarse, ni los datos que puedan existir de por medio. Es decir, nada va a quedar almacenado como información que necesite ser consultada más tarde (por lo menos por parte del cliente).

Bibliografía

https://www.linuxhowtos.org/C_C++/socket.htm . Ejemplo de uso de sockets. [Último acceso: 7/9/2021]

https://www.redhat.com/es/topics/cloud-native-apps/stateful-vs-stateless . Sistemas con estado y sin estado. [Último acceso: 15/9/2021]