Programación Distribuida
y
Tiempo Real

Práctica 1

AÑO 2022 – 2° Semestre

Lenguaje utilizado

C

Referencias

https://www.programacion.com.py/noticias/sockets-en-c-parte-i-linux

https://redespomactividad.weebly.com/modelo-cliente-servidor.html

https://www.xataka.com/basics/ftp-que-como-funciona

https://refactorizando.com/stateful-vs-stateless-arquitectura

1) **Teniendo en cuenta al menos los ejemplos dados (puede usar también otras fuentes de información, que se sugiere referenciar de manera explícita):**
   a. **Identifique similitudes y diferencias entre los sockets en C y en Java.**
   b. **Por qué puede decirse que los ejemplos no son representativos del modelo c/s?**

a) A la hora de desarrollar los sockets, sean en C o en Java, estos cuentan con las mismas (o muy similares) funcionalidades y utilizan el mismo modelo Cliente/Servidor. Es decir, que en ambas implementaciones se necesitan de procesos que hagan papel de cliente y procesos que hagan papel de servidor para el envío y recibo de peticiones. Y, por lo tanto, la utilización de direcciones y puertos para poder establecer una comunicación.

En ambos casos, los sockets pueden utilizar los protocolos de transporte TCP (Orientados a la conexión Socket Stream) y UDP (No orientados a la conexión Socket Datagram).

Algunas diferencias que se pueden destacar en los sockets dependiendo del lenguaje son los siguientes:

Socket en C	Socket en Java
Como ya se conoce, en Unix todo es un fichero, por lo que un socket no es la excepción. Por lo que a la hora de enviar y/o recibir datos por la red, tan solo se realizan escrituras y lecturas sobre un fichero “especial”. Esto se debe a que cuenta con unas características particulares, por lo que utiliza comandos o funciones especiales. Por ejemplo: Se utiliza la función socket() para crear un nuevo socket.	Los sockets brindados por Java son clases provistas mediante la librería java.net. Por un lado, tenemos la clase Socket que se utiliza para implementar la conexión desde el lado del cliente y por el otro SocketServer, que nos permite manipular la conexión desde el lado del servidor.
El manejo de socket en C puede ser considerado como bajo nivel, ya que se deben tener en cuenta ciertas configuraciones que deben ser realizadas manualmente para poder	El manejo de socket en Java puede ser considerado de alto nivel, y se debe a la capa de abstracción que se utiliza al manipular las clases brindadas por la propia librería y que permite obviar ciertas

implementar correctamente el funcionamiento de los sockets.	configuraciones que desde ya vienen resueltas por defecto mediante el llamado de métodos.
Tanto del lado del cliente como del servidor, el comando Socket() devuelve un file descriptor del socket que luego puede ser utilizado para llamadas al sistema. Adicionalmente, del lado del cliente se encuentra el comando Connect() que se utiliza para conectar el socket a una dirección IP/puerto especifico. Y del lado del servidor, el comando Bind() para asociar un socket con un puerto de la máquina.	Se ahorra a la hora de las configuraciones con el mensaje Socket() de la clase Socket donde crea directamente la conexión y que lleva como parámetros la IP y el puerto donde escucha el servidor. Por el lado del servidor, se cuenta con el mensaje ServerSocket() de la clase del mismo nombre donde crea un socket donde se va a esperar la conexión con los clientes y lleva como parámetro el puerto por donde va a escuchar peticiones. Adicionalmente cuenta con el mensaje Accept() que se utiliza para mantener el socket a la espera de una conexión.
Los comandos brindados como Read() y Write() tanto para el servidor como para el cliente funcionan respectivamente para leer y escribir una cantidad determinada de datos desde el socket.	Los comandos brindados por las clases como Read() y Write() no leen o escriben respectivamente directo desde el Socket, sino que deben utilizar unas clases para dichas operaciones brindadas por el paquete java.io. El comando Read() utiliza la clase InputStream relacionado al socket y Write() utiliza la clase OutputStream relacionado al socket.

b) La arquitectura cliente/servidor es un modelo donde las tareas se reparten entre los clientes quienes realizan las peticiones a un servidor quien es el proveedor de recursos o servicios. Lo que significa que el servidor debe estar a disposición en todo momento para poder aceptar los requerimientos de uno o muchos procesos clientes.
   En la conexión deben cumplirse los siguientes pasos:
   - Inicialización
   - Envío/recepción de peticiones
   - Finalización

Justamente en los ejemplos brindados, una vez que el servidor termina de responder la petición del cliente, este finaliza. Por lo que no se respetan los pasos de conexión, ya que falta el mensaje de cierre de conexión. Además de no brindar la posibilidad de atender futuras peticiones de posibles clientes tras finalizar el servidor.

Por lo tanto, no se estaría cumpliendo completamente con la metodología de cliente/servidor.

2) **Desarrolle experimentos que se ejecuten de manera automática en una máquina virtual (mv) con Vagrant donde**
   a. **Se muestre que no necesariamente siempre se leen/escriben todos los datos involucrados en las comunicaciones con una llamada read/write con sockets.**
   Sugerencias: 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 estudiar los distintos escenarios donde se realiza la comunicación sobre distintas cantidades de datos, se decidió por trabajar teniendo como base la implementación de los sockets brindados en el lenguaje de programación C.

Se modificó la forma de asignar espacio a los buffers utilizados, creando una constante (BUFFER_SIZE) cuyo valor será el espacio que se desea manejar para el buffer. Por lo que se utilizara esta constante en vez de asignar manualmente es espacio cada vez que se requería.

Otra modificación fue utilizar una función memset(), que nos permite de forma automática, llenar de caracteres el espacio que se le asigna al buffer.

Realizando las pruebas, donde se cambiaba el valor de la constante BUFFER_SIZE por los valores solicitados, se pudo observar que se leen correctamente todos los caracteres

enviados desde el cliente hacia el servidor en los casos 10^3, 10^4.
En cambio, en los casos desde 10^5 en adelante, tan solo son leídos 32741 caracteres (o 65482 caracteres en algunos casos).

b. 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.

Para asegurar si los datos enviados por el socket cliente llegan correctamente al servidor, se deben validar la cantidad de caracteres y contenido que llega desde el buffer.

Para esta solución, se agrega a la anterior implementación un par de nuevas validaciones de parte del servidor que permite de forma automática verificar en qué condiciones llego el mensaje enviado por el cliente.

Del lado del socket cliente, se agrega dos nuevos mensajes que envía el servidor, el primero envía el tamaño del mensaje del buffer previamente enviado, y el segundo mensaje envía un hash generado por el contenido del mensaje del buffer.

Del lado del socket servidor, se agregan dos recepciones de mensajes más para atender los nuevos mensajes del cliente, con el primer mensaje compara el tamaño mandado por el cliente con el tamaño del buffer obtenido, y con el segundo mensaje compara el hash enviado por el cliente con el hash generado del lado del servidor con el contenido del buffer obtenido.

Se utiliza una función para generar los hashes en ambos sockets llamada sdbm.

```
Static unsigned sdbm(unsigned char *str)
```

3) Tiempo y tamaño de mensajes

a. Como en el caso anterior, desarrolle y documente experimentos en mv para evaluar obtener los tiempos de comunicación para tamaños de mensajes de 10^1, 10^2, 10^3, 10^4, 10^5 y 10^6 bytes.

Para poder medir los tiempos que se toman en las comunicaciones entre los procesos cliente y servidor, se utiliza la función clock() de la librería <time.h> de C.

Del lado del socket cliente, se empieza a contar el tiempo justo antes de mandar el mensaje al servidor y termina de contar una vez reciba una respuesta por parte del servidor.

Del lado del socket servidor, se empieza a contar el tiempo cuando se recibe un mensaje del cliente y este deja de contar una vez envíe una respuesta al cliente.

Además, se busca solucionar el escenario donde se quiere mandar grandes cantidades de datos desde el socket cliente utilizando un iterado (do - while) del lado del socket servidor donde leerá y almacenará en su buffer “concatenando” los fragmentos del mensaje recibido hasta obtener el tamaño completo del mensaje.

Con los datos recopilados, se saca un promedio con cada uno de los resultados utilizando la siguiente formula:

$$(T1 – T2) / 2$$

Siendo T1 el tiempo obtenido por el socket cliente y T2 el tiempo obtenido por el socket servidor. Y en base a este dato se calcula el promedio y desviación estándar para cada uno de los casos solicitados.

b. Grafique el promedio y la desviación estándar para cada uno de los tamaños del inciso anterior.

Promedios obtenidos en cada una de las pruebas solicitadas:

Desviación estándar obtenida en cada una de las pruebas solicitadas:

c. Provea una explicación si los tiempos no son proporcionales a los tamaños (ej: el tiempo para 10^2 bytes no es diez veces mayor que el tiempo para 10^1) bytes.

Los tiempos de comunicación que se utiliza para el envío de mensaje entre los sockets clientes con el socket servidor no varía según la proporcionalidad del tamaño del mensaje que se esté enviando.

Esto se debe principalmente a la capacidad de información que puede enviar el socket en un solo mensaje. Como se estudió en ejercicios anteriores, los sockets tienen como límite de envío de 32741 (o 65482 en algunos casos) caracteres por mensaje.

Por lo tanto, no variara el tiempo que implementara el socket para el envío de información inferior a su límite de caracteres, explicando así que los tiempos que demoraría para enviar información de 10^1 o 10^2 bytes serían muy similares y no proporcionales entre sí.

A medida que el mensaje a enviar sea mayor, y este supere el límite que implantan los sockets, comenzaran a crecer los tiempos requeridos en la comunicación para que el servidor sea capaz de recopilar completamente la información que el socket cliente le envía fraccionado en “paquetes”. Esto se puede observar en las gráficas, el considerable aumento de tiempo necesario para comunicar datos con un tamaño de 10^6.

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

Por un lado, tenemos a la función fgets() que nos permite, la recopilación o entrada de información desde teclado. La estructura de la función es la siguiente:

fgets(buffer,long,stdin)

Donde la variable buffer es un puntero, que referencia un espacio de memoria, donde se almacenara la información recibida desde teclado.

Por el otro lado, la función write() que nos permite escribir una determinada cantidad de datos en el socket. La estructura de la función es la siguiente:

write(sockfd, buffer, long)

Donde la variable buffer es un puntero, que referencia un espacio de memoria cuya longitud (cantidad de bytes) está dado por el valor almacenado en el tercer parámetro de la función. Y que la información almacenada en la variable es mandada por el socket asociado a la variable sockfd.

Por lo tanto, se puede observar que la variable utilizada para el almacenamiento de información desde teclado, como la variable utilizada para el envío de información son compatibles en tipos de datos (ya que la mismas son punteros a espacios de memoria), por lo que no existe ningún problema para que se aproveche el mismo espacio de memoria en una sola variable.

Aunque se debe poseer ciertos cuidados si se decidiera por utilizar una sola variable para ambos trabajos. Ya que se pueden presentar ciertos inconvenientes a la hora de desarrollar una arquitectura de cliente/servidor, debido a se podría presentar sobreescrituras o perdidas de información entre el periodo de recepción de datos desde teclado hasta el envío de información.

5) **¿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.**

Se podría pensar en una implementación de un servidor de archivos remotos mediante la utilización de sockets basándose en protocolos existentes que realizan esta misma tarea, que es la de un servidor que almacena y distribuye diferentes tipos de archivos entre clientes y que permite el acceso remoto a los mismos.

Un protocolo muy clásico y conocido (aunque no es tan utilizado como en sus inicios), el cual se adapta a nuestra problemática y que puede ser utilizado como ejemplo, es el protocolo FTP (File Transfer Protocol) que desde la década del 80 utiliza la versión TPC/IP y permite la transferencia directa de archivos desde un dispositivo a otro. Los datos son enviados a través de los puertos 20 y 21 (que son los asignados en todos los equipos para llevar a cabo transferencia de archivos). Y las conexiones en este protocolo tienen una relación de cliente/servidor, por lo que puede ser implementado utilizando sockets.

Entonces, para implementar mi propio servidor de archivos se puede optar por una “alternativa” basándose en FTP. Por lo que se contaría para la comunicación, con un protocolo TCP/IP.

El conjunto de comandos fundamentales que se tendrían a disposición en esta “implementación” serian aquellos que permitan una navegación sobre directorios dentro del servidor y aquellos comandos que nos permitan el manejo de archivos.

Comandos:

Navegación sobre directorios	
ls – Listar archivos	Devuelve una lista de todos los archivos que se encuentren dentro del directorio donde se encuentre actualmente.
cd – Cambia de directorio	Se utiliza para cambiar desde un directorio a otro diferente. Utilizando el comando sin ningún argumento, se posicionará como directorio actual, al directorio principal del servidor. Si se utiliza con un argumento, se posicionará el cliente en el directorio del servidor indicado por el argumento.
pwd – Ubicación actual	Devuelve la dirección o path absoluto del directorio actual del servidor en la que se encuentra el cliente.
Manejo de archivos	
add – Agregar archivo	Envía un archivo local del cliente al servidor. El argumento que acompaña al comando es la dirección donde se encuentra el archivo local que se quiere mandar al servidor, y este se carga sobre el directorio donde se encuentre posicionado el cliente en ese momento sobre el servidor.
get – Obtener archivo	Se obtiene localmente un archivo remoto desde el servidor. El argumento que acompaña al comando es el nombre del archivo que se encuentra en el directorio

	donde este posicionado el cliente en ese momento sobre el servidor.
delete – Borrar archivo	Se borra un archivo remoto que se encuentre en el servidor. El argumento que acompaña al comando es el nombre del archivo que se encuentra en el directorio donde este posicionado el cliente en ese momento sobre el servidor.

6) **Defina que es un servidor con estado (stateful server) y qué es un servidor sin estado (stateless server).**

Stateful server: Aplicaciones que mantienen el estado e información en el que se encontraba el cliente en su último uso del sistema, donde se puede utilizar el mismo servidor para realizar el procesamiento de todas las peticiones sobre la misma información de estado o donde se puede compartir dichas sesiones con demás servidores o servicios si así lo necesitaran.

Es decir, que recuerdan la información del cliente entre una y las posteriores peticiones que se realicen sobre el servidor. Permitiéndole, de esta forma, mantener un seguimiento de las acciones que realicen los clientes.

Requieren de algún tipo de sitio en el que puedan almacenar la información de una manera persistente. Por lo que aplicaciones que utilicen este tipo suelen ser bases de datos como, por ejemplo, aplicaciones bancarias.

Stateless server: Aplicaciones que están pensadas para realizar únicamente una función y que (a diferencia del caso anterior) no almacenan información sobre las operaciones o clientes anteriores ni se hace referencia a ellos. Por lo que cada operación se lleva a cabo desde cero y es completamente independiente de las demás.

Al no contar con información del cliente, las diferentes peticiones realizadas pueden ser respondidas mediante diferentes servidores. Por lo que se aumenta y favorece la resiliencia y elasticidad de los sistemas.

Aplicaciones que utilizan este tipo de diseño de servicios suelen ir ligados a una arquitectura de microservicios y contenedores. Además de no depender de un sistema de almacenaje persistente.