Programación Distribuida y Tiempo Real
Trabajo Práctico 1
Sockets

- Joaquin Tartaruga (17023/7)

- Manuel Rubiano (17711/6)

1. a. Los sockets en C y Java comparten el mismo modelo de comunicación (TCP/IP o UDP) y el uso de IP y puertos, dando herramientas para crear y manejar conexiones.

- Nivel de abstracción:
    - **Java**: Proporciona clases como Socket, ServerSocket, y DataInputStream que abstraen muchos detalles del manejo de la red, lo que facilita el desarrollo. No es necesario controlar la creación y configuración manual de estructuras de datos para sockets.
    - **C**: Utiliza un enfoque de bajo nivel, trabajando directamente con las llamadas al sistema proporcionadas por el sistema operativo. Se usan funciones como socket(), bind(), listen(), accept(), send(), y recv(), lo que significa que se debe gestionar manualmente aspectos como la configuración de la estructura sockaddr, las direcciones IP y el manejo de errores.
- Manejo de errores:
    - **Java**: Utiliza excepciones para manejar errores. Si ocurre un problema durante la comunicación por socket, se lanza una excepción específica como IOException o SocketTimeoutException.
    - **C**: El manejo de errores se realiza mediante códigos de retorno de las funciones y el uso de la variable global errno. Se debe comprobar manualmente si cada llamada al sistema devuelve un error y luego actuar en consecuencia.
- Tipos de Sockets:
    - **Java**: Trabaja con clases Socket para conexiones cliente-servidor y ServerSocket para servidores. La creación y configuración de un socket es simple y se maneja mediante constructores de clase.
    - **C**: Los sockets se crean mediante la llamada al sistema socket(), que devuelve un descriptor de archivo.
- E/S de datos
    - **Java:** Proporciona clases de alto nivel como DataInputStream y DataOutputStream para la lectura y escritura de datos
    - **C:** Utiliza funciones de bajo nivel como send() y recv() para la transmisión de datos, que requieren que el programador maneje explícitamente los buffers de bytes.

   b. En el modelo Cliente/Servidor hay varios clientes enviando peticiones a un servidor, y éste se encarga de resolverlas; luego, se queda a la espera de otras peticiones. En el caso de sockets, cuando el servidor finaliza una tarea, se cierra la conexión y no puede volver a retomarse. Por eso decimos que implementamos una comunicación, en vez de un modelo Cliente/Servidor. Los ejemplos básicos de sockets suelen demostrar la conexión entre un cliente y un servidor de forma sencilla y directa, generalmente involucrando solo un cliente y un servidor. Esto no captura la complejidad y escalabilidad que son características de un auténtico sistema Cliente/Servidor.

2. a. Realizamos pruebas en donde mandamos mensajes que pesen desde 10^1 a 10^7 bytes. Con los mensajes con un peso menor o igual a 10^6 no tuvimos problemas, pero con 10^7 no lo logramos realizar debido a que el mismo java nos levantaba error en el socket (java.net.SocketException:). En nuestro caso, un mensaje que pesa 10^6 bytes no se llega a recibir en su totalidad, significando que

habría que iterar sobre el buffer para leer todo su contenido, por lo tanto definimos ese como el límite para las demás pruebas. Descartamos que es un envío incompleto, ya que implementamos la directiva flush() que permite enviar todos los datos que estén disponibles para enviar, sin esperar a que se llene el buffer.

Tamaño de Mensaje	Checksum Enviado	Checksum Recibido
10^3	65010	65010
10^4	650010	650010
10^5	6500010	6500010
10^6	1094795585	8519620

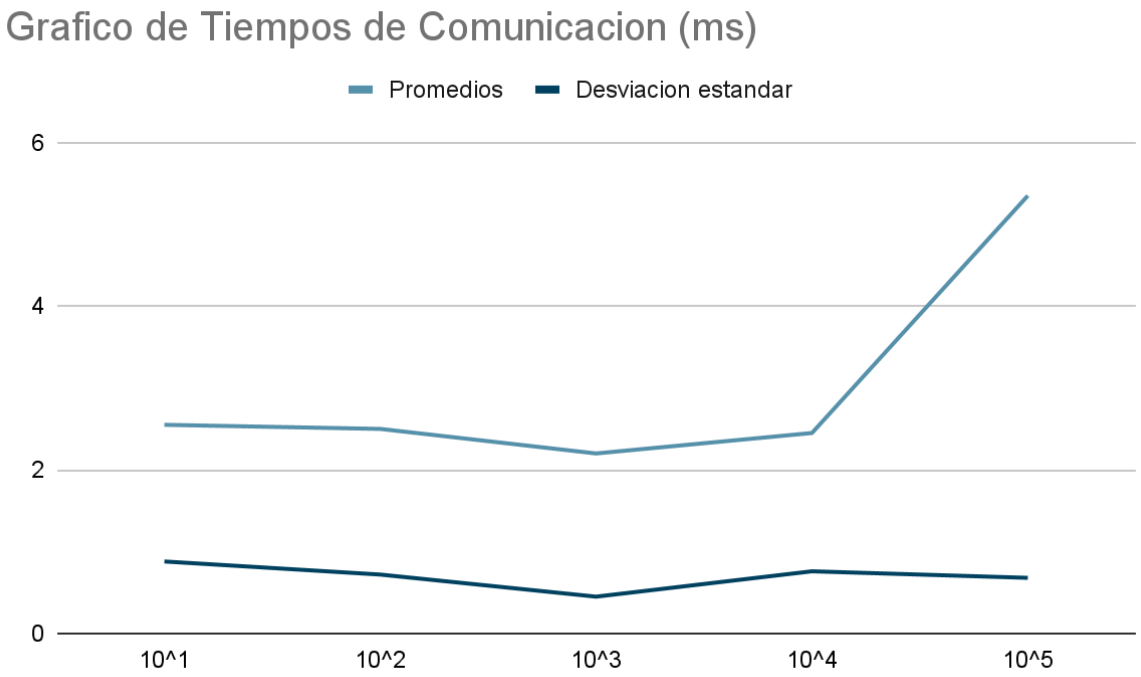
b. Para asegurarnos que el mensaje llega correctamente y completo, del lado del servidor hicimos dos verificaciones:

- Chequear que el tamaño del mensaje sea el esperado, implementando un checksum, donde el cliente envía el tamaño del mensaje y el servidor calcula el tamaño de lo recibido y verifica que correspondan ambos datos. Para verificar integridad de los datos
- Agregar un carácter especial al inicio y al final del mensaje para asegurarse que se mando completo, y luego con el servidor verificar dichos caracteres

3. a. Para este caso tomamos los tiempos del lado del cliente. Restando el momento después de la recepción con el de antes del envío. Asumimos que el tiempo de cómputo del servidor entre este intervalo es despreciable. A su vez, enviamos, por cada tamaño, 20 mensajes desde diferentes puertos, para luego calcular un promedio de éstos con una mayor precisión.

Tamaño de Mensaje	Tiempo de Comunicación (ms)	Desviación Estándar
10^1	2.70	1.25
10^2	2.35	0.68
10^3	2.85	1.35
10^4	2.80	1.21
10^5	6.10	1.37

b.



Se puede observar que la desviación estándar, es decir cuan dispersos están los valores respecto a una media, se mantiene constante y es baja (cercana a 0), lo que indica que no hay variación en los tiempos de comunicación. Por lo que hace que los tiempos de respuesta sean predecibles y consistentes.
Respecto a la proporcionalidad de tiempos hablaremos en el siguiente inciso.

c. Los tiempos no son proporcionales al tamaño del mensaje porque sino un mensaje que pese 10^2 bytes debería ser 10 veces más lento al de uno de 10^1 bytes. Esto se debe a que el buffer soporta tamaños ciertos tamaños y los puede enviar en una sola transmisión, sin embargo, en los casos como un mensaje de 10^5 bytes vemos que el buffer se sobrecarga, lo que obliga a fragmentar el mensaje, agregando un costo adicional de por hacer varias transmisiones.

d. Mensajes de 10^5 bytes:
- En Java: 1.71 ms
- En C: 0.45 ms

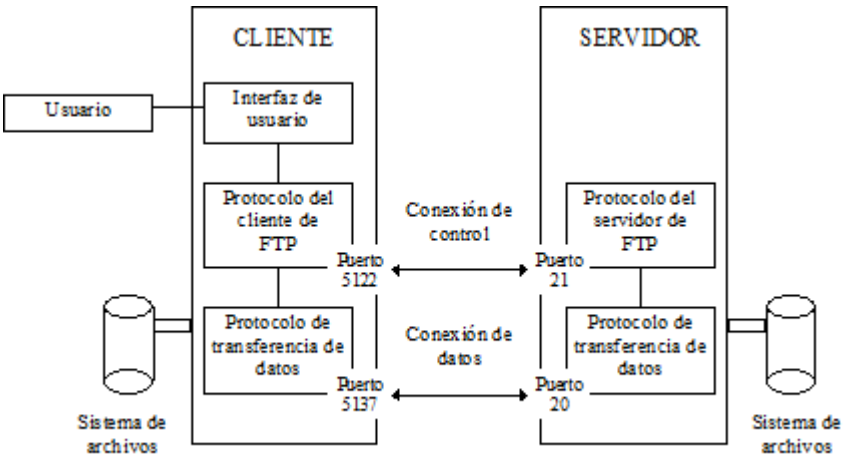
Mensaje de 10^6 bytes:
- En Java: 2.62ms
- En C: 0.90ms

C es un lenguaje de programación de bajo nivel que permite un control directo sobre la memoria y el hardware. Cuando se manejan sockets, es muy eficiente, ya que se ejecuta casi al mismo nivel que el sistema operativo. Esto significa que hay menos sobrecarga en la gestión de los datos y la transmisión es más rápida.
Java es de un nivel más alto y abstrae muchas de las operaciones de bajo nivel. Cada vez que se envían o reciben datos, Java realiza verificaciones adicionales y usa su modelo de administración de memoria (recolección de basura), lo cual puede introducir una pequeña latencia.

4. En C se puede usar la misma variable tanto para leer como para comunicar, ya que este lenguaje se maneja con punteros a buffers. Teniendo así un puntero a un buffer de tipo char[] o int(que soporta diferentes tipos de datos) que utiliza para almacenar la entrada o enviar/recibir datos a través de un socket, sin necesidad de copiar el dato a otra variable o transformar la misma. Esto hace eficiente el modelo C/S, ya que se reduce la cantidad de memoria a usar y simplifica el código, pudiendo manejar la comunicación con una sola variable. Sumado a eso, se garantiza consistencia en los datos a enviar y facilita la manipulación de los datos del buffer previo y posterior al envío.

5. Se podría implementar un sistema de archivos remotos con sockets utilizando un protocolo de FTP, Donde los clientes utilizaran comandos para realizar peticiones sobre archivos al servidor: UPLOAD (agregar), LIST (listar), DELETE (borrar), RENAME (renombrar), DOWNLOAD (descargar).
   Por el lado del cliente se podría:
   - Loguear (opcional, por cuestiones de seguridad)
   - Conectarse al servidor
   - Realizar la petición a través de comandos, si es subir un archivo se debe pasar un hash para que el servidor compruebe que fue recibido correctamente

   Por el lado del servidor:
   - Esperar conexión, se podrían limitar la cantidad de las mismas y utilizar multithreading para encargarse de varias en simultáneo
   - Recibir el comando y determinar si ese cliente tiene los permisos apropiados
   - En caso de que la petición sea de descarga se debe verificar la existencia del archivo. Si existe se crea una nueva conexión (a través de otro socket) y se transmite el archivo al cliente (cerrando la conexión una vez transferido el último dato), caso contrario se informa error
   - Si la petición es de subida, se debe verificar que haya almacenamiento para el archivo, además de verificar la existencia de algún otro con el mismo nombre. De existir, debería optarse por sobreescribir o modificar el nombre. No se debe poder hacer ninguna operación de lectura/escritura hasta que el archivo este subido por completo.



-

6. Un servidor con estado (stateful server) almacena información del cliente y de las comunicaciones establecidas, permitiendo ser consultadas en cualquier momento. En un servidor sin estado (stateless server) cada operación realizada hacia el será aislada, por lo que no va a quedar registro de ella ni de los datos intercambiados. Es decir, no se va a poder consultar la información en un futuro (por lo menos por parte del cliente), a menos que del lado del cliente se quede almacenada.

Ventajas de un stateful server

- Se pueden ofrecer respuestas más personalizadas y una experiencia de usuario mejorada al recordar detalles de las interacciones anteriores.
- Las aplicaciones que requieren un seguimiento continuo (como juegos en tiempo real o aplicaciones bancarias) pueden beneficiarse de un servidor con estado.

Desventajas

- Puede ser más propenso a errores debido a la complejidad de sincronizar estados en diferentes instancias.
- Si el servidor falla, el estado del cliente puede perderse, a menos que se almacene de manera persistente en un almacenamiento externo.

Ejemplos de uso

- Aplicaciones web tradicionales que utilizan sesiones (cookies o sesiones de servidor) para recordar la autenticación del usuario y su historial.
- Juegos multijugador en línea, donde el estado del jugador (posición, puntuación, inventario) debe ser mantenido a lo largo de la sesión de juego.

Ventajas de un stateless server

- Permite que las solicitudes se manejen por cualquier instancia del servidor, lo que resulta en una mayor tolerancia a fallos.
- Reduce la complejidad del servidor, lo que facilita el mantenimiento y el desarrollo

Desventajas

- Al no tener un estado persistente, es más difícil ofrecer experiencias de usuario personalizadas basadas en interacciones anteriores.
- El cliente debe enviar toda la información relevante en cada solicitud, lo que puede aumentar el tráfico de red si los datos son grandes.

Ejemplos

- API REST: Siguen el principio de ser sin estado, donde cada solicitud del cliente debe incluir toda la información necesaria para procesar la petición.
- Protocolos HTTP estándar