

Programación Distribuida y Tiempo

Real - Práctica 1

Autores

- Juan Cruz Cassera Botta, 17072/7
- Lucio Bianchi, 19341/8

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.

Similitudes:

1. La lógica en general es la misma. En ambos se utilizan sockets y se hacen operaciones de lectura / escritura.
2. Las librerías de sockets de ambos lenguajes nos informan si no se puede crear el socket, o no se puede escribir/leer el mensaje, para que manejemos el error.
3. Ambos corresponden a Socket Streams. Generalmente implementados sobre TCP.

Diferencias:

1. En C el socket corre directamente sobre el sistema operativo de manera bajo nivel con un file descriptor, mientras que en Java corre sobre la JVM, a más alto nivel.
2. En C se requiere especificar qué tipo de protocolo IP (IPv4, IPv6) acepta el socket al crearlo, mientras que en Java esto no se requiere.
3. Las librerías de Java resultan más sencillas de utilizar y más legibles que las de C, debido a que posee una mejor abstracción de lo que ocurre a bajo nivel.
4. En C se pueden escribir datos directamente sobre el socket, mientras que en Java no se lee ni escribe directamente en el socket, si no que se usa un intermediario InputStream y OutputStream.

b) ¿Por qué puede decirse que los ejemplos no son representativos del modelo c/s?

Los ejemplos dados no son representativos del modelo Cliente Servidor porque:

1. Este modelo implica un servidor que atiende pedidos de varios clientes, y en este caso está atendiendo pedidos de un solo cliente.
2. Además de que el cliente solo es uno, tanto el cliente como el servidor asumen que la comunicación finaliza luego de un solo mensaje, cuando debería ser variable.
3. Si tuviéramos varios clientes, el servidor los atendería secuencialmente y no concurrentemente, lo cual no es usual en este modelo.
4. En el modelo C/S, el cliente realiza una petición y el servidor la "resuelve" y le devuelve el resultado. Esto no ocurre en el ejemplo, ya que simplemente se trata de un intercambio de mensajes y nada más. No hay una petición y resolución de la misma.

2) Desarrolle experimentos que se ejecuten de manera automática donde:

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

Sugerencia: modifique 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).

Lo que hicimos fue, como el inciso sugiere, que el cliente le envíe al servidor 10^3 , 10^4 , 10^5 y 10^6 bytes en un mismo buffer. Lo que hallamos (**en Java**) es que:

- El servidor **siempre** lee todos los datos recibidos si fueron de 10^3 o 10^4 bytes.
- El servidor **a veces** no lee todos los datos recibidos si fueron de 10^5 bytes.
- El servidor **nunca** lee todos los datos en su totalidad si fueron de 10^6 bytes.

Output del servidor:

```
sock> java ServerEj2 8080
Recibidos 1000 bytes para el buffer de tamaño: 1000
Recibidos 10000 bytes para el buffer de tamaño: 10000
Recibidos 100000 bytes para el buffer de tamaño: 100000
Recibidos 131071 bytes para el buffer de tamaño: 1000000
```

Output del cliente:

```
vasock> java ClientEj2 localhost 8080
Enviando un buffer de tamaño: 1000 bytes
Enviando un buffer de tamaño: 10000 bytes
Enviando un buffer de tamaño: 100000 bytes
Enviando un buffer de tamaño: 1000000 bytes
```

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.

i)

En primera instancia modificamos el programa del inciso 2)a) para validar que la totalidad de los mensajes sean leídos por el servidor. Para lograrlo, hacemos un bucle en el servidor que contabiliza la cantidad de bytes leídos, y hace operaciones de read() hasta que la cantidad total de bytes leídos sea mayor o igual al tamaño del mensaje.

Output del servidor:

```
Recibidos 1000 bytes para el buffer de tamaño: 1000  
Recibidos 10000 bytes para el buffer de tamaño: 10000  
Recibidos 100000 bytes para el buffer de tamaño: 100000  
Recibidos 1000000 bytes para el buffer de tamaño: 1000000
```

Output del cliente:

```
Enviando un buffer de tamaño: 1000 bytes  
Enviando un buffer de tamaño: 10000 bytes  
Enviando un buffer de tamaño: 100000 bytes  
Enviando un buffer de tamaño: 1000000 bytes
```

ii)

Luego, para asegurarnos que los mensajes lleguen de forma segura y sin alteraciones utilizamos un cálculo de checksum de tipo CRC (Cyclic Redundancy Check) que provee la librería de java.util.zip.CRC32. El cliente calcula el checksum del contenido de su buffer y lo envía luego de calcularlo, y el servidor una vez lo recibe computa el checksum de su lado y verifica que sea el mismo que envió el cliente.

Output del servidor:

```
istribuida y Tiempo Real\Práctica\1\src\jvasock> java ServerEj2B 8080
Checksum recibido: 1961098049
Checksum computado: 1961098049
Checksum computado: 1961098049
Checksum computado: 1961098049
Checksum computado: 1961098049
Checksum computado: 1961098049
Checksum computado: 1961098049
Recibidos 1000 bytes para el buffer de tamaño: 1000
Checksum recibido: 3523200252
Checksum computado: 3523200252
Recibidos 10000 bytes para el buffer de tamaño: 10000
Checksum recibido: 2865713097
Checksum computado: 2865713097
Recibidos 100000 bytes para el buffer de tamaño: 100000
Checksum recibido: 1635920155
Checksum computado: 1635920155
Recibidos 1000000 bytes para el buffer de tamaño: 1000000
```

Output del cliente:

```
istribuida y Tiempo Real\Práctica\1\src\jvasock> java ClientEj2B localhost 8080
Enviando un buffer de tamaño: 1000 bytes
Enviando checksum: 1961098049
Enviando un buffer de tamaño: 10000 bytes
Enviando checksum: 3523200252
Enviando un buffer de tamaño: 100000 bytes
Enviando checksum: 2865713097
Enviando un buffer de tamaño: 1000000 bytes
Enviando checksum: 1635920155
```

3) Tiempos y tamaños de mensajes

a) Como en el caso anterior, desarrolle y documente experimentos para evaluar/obtener los tiempos de comunicaciones para tamaños de mensajes de 10^1 , 10^2 , 10^3 , 10^4 , 10^5 y 10^6 bytes. Tener en cuenta la cantidad total de datos que se transfieren entre los procesos en el experimento para estimar el tiempo de comunicaciones.

Inicialmente, definimos cómo calcular el tiempo que toma un mensaje: tenemos t_0 que ocurre justo antes que el cliente envíe el mensaje al servidor, y t_1 que ocurre luego que el cliente recibe la confirmación por parte del servidor de que ya leyó su mensaje. Entonces el tiempo es simplemente $t_1 - t_0$.

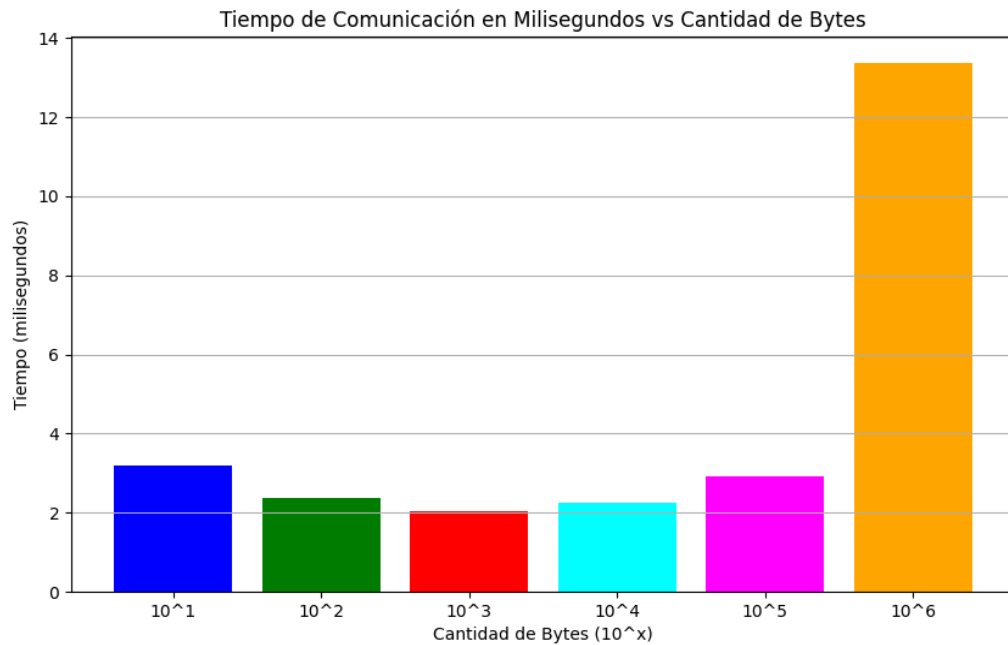
Para calcular el tiempo de comunicación de cada uno de los 6 tamaños de mensajes, lo que hicimos fue calcular 100 veces para cada uno de los tamaños el tiempo que tarda una comunicación, y luego obtuvimos el promedio de esas 100 iteraciones.

Tiempos promedio obtenidos (en milisegundos):

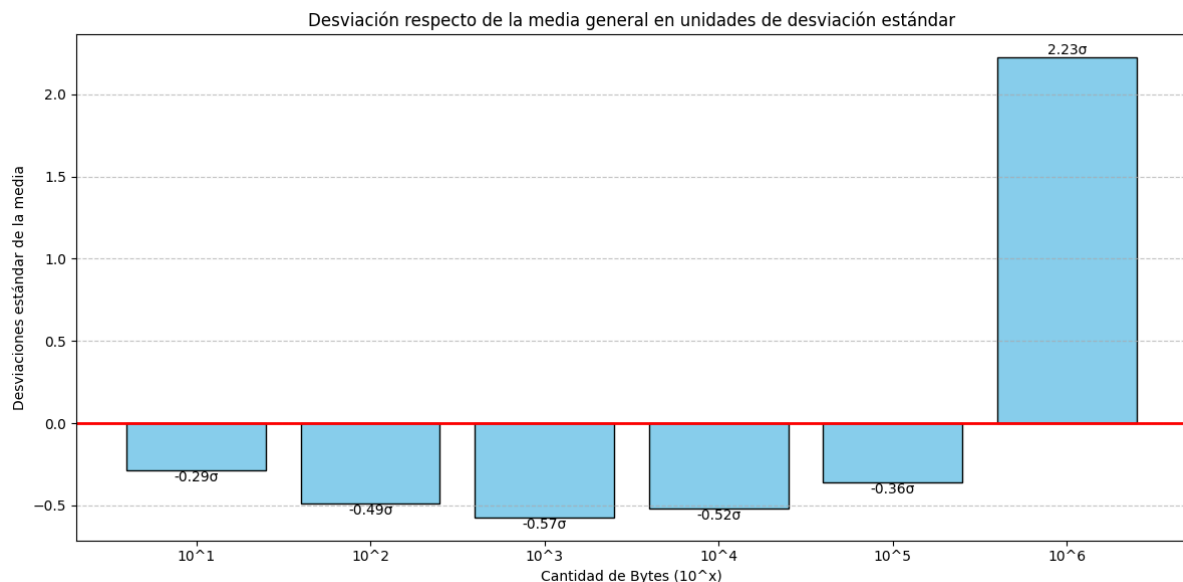
```
n Distribuida y Tiempo Real\Práctica\1\src\javadoc> java ClientEj3 localhost 8080
Tamaño en bytes: 10 - Tiempo promedio de ejecución: 3,1913100 milisegundos
Tamaño en bytes: 100 - Tiempo promedio de ejecución: 2,3777500 milisegundos
Tamaño en bytes: 1000 - Tiempo promedio de ejecución: 2,0404300 milisegundos
Tamaño en bytes: 10000 - Tiempo promedio de ejecución: 2,2617700 milisegundos
Tamaño en bytes: 100000 - Tiempo promedio de ejecución: 2,9031000 milisegundos
Tamaño en bytes: 1000000 - Tiempo promedio de ejecución: 13,3698300 milisegundos
```

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

Promedio:



Desviación estándar:



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.

Efectivamente, los tiempos no son proporcionales, e incluso podemos ver que en ciertos casos un tamaño de mensaje mayor tarda menos tiempo.

Esto podría deberse a cómo el sistema operativo trabaja con las operaciones `read()` y `write()` a bajo nivel, detrás de estas abstracciones. Una posible causa sería que el SO esté agrupando y enviando de a grupos de bytes grandes, por lo que la cantidad de bytes puede no impactar tanto en el tiempo de comunicación cuando los bytes enviados son pocos.

d) Compare los tiempos de comunicaciones para los tamaños 10^5 y 10^6 bytes usando C y Java.

Cantidad de bytes comunicación	Java (tiempo en milisegundos)	C (tiempo en milisegundos)
10^5	2,9031000	0,095
10^6	13,3698300	0,798

Los resultados obtenidos en C son bastante mejores que los de Java. Por lo general C suele ser más rápido que Java por su mucho menor overhead, y en este caso las operaciones de sockets provistas por C parecen ser más performantes.

4) ¿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?

Cuando se pasa por parámetro un arreglo en C, en realidad se le está pasando un puntero al primer elemento. En C tanto la función para leer por teclado *fgets()*, como el *write()* que utilizamos para enviar por socket reciben un puntero. Esto nos permite utilizar la misma variable para leer de teclado y para enviar por un socket, por ejemplo:

```
char buffer[256];  
fgets(buffer, sizeof(buffer), stdin);  
write(sockfd, buffer, strlen(buffer));
```

Lo que ocurre es que el área de memoria donde está alocado el buffer se llena con los datos que ingresa el usuario vía *fgets()*, y luego el *write()* se dirige a esa misma área de memoria para obtener los datos a escribir en el socket.

Para aplicaciones cliente/servidor, este enfoque puede ser muy conveniente ya que permite reutilizar variables y buffers para diferentes propósitos, simplificando el manejo de datos y evitando desperdicio de memoria al declarar menos variables.

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 implementar un servidor de archivos remotos basándonos en la arquitectura Cliente Servidor. Un ejemplo sería el protocolo FTP que corre sobre TCP. Podríamos implementar una interface similar, con las operaciones necesarias:

CONNECT: El cliente establece una conexión con el servidor.

LIST-ALL: Enumera los archivos disponibles.

DOWNLOAD <filename>: Solicita un archivo específico.

UPLOAD <filename>: Sube un archivo específico al servidor.

DELETE <filename>: Elimina un archivo del servidor.

CLOSE: El cliente cierra la conexión con el servidor.

Algunos de los detalles importantes en cuanto al diseño serían:

- **Conexión requerida:** el cliente no podrá ejecutar ningún comando si no hizo un CONNECT primero.
- **Uso de sockets TCP:** permitiendo una comunicación confiable y ordenada entre el cliente y el servidor.
- **Servidor concurrente:** permitiendo manejar pedidos de múltiples clientes simultáneamente. Se podría lograr mediante el uso de PThreads en C, o de Virtual Threads en Java.
- **Manejo de errores:** implementar mensajes de error para enviarle al cliente en caso que algo falle.
- **Seguridad:** comunicaciones encriptadas mediante SSL/TLS.





6) Explique y justifique brevemente ventajas y desventajas de un servidor con estado respecto de un servidor sin estados.

Se dice que un servidor “tiene estado” cuando mantiene información sobre cada cliente a lo largo de distintas peticiones.



En cambio, un servidor “sin estado” es uno que no mantiene esta información sobre ningún cliente.

Debajo listamos ventajas (marcadas con ) y desventajas (marcadas con ) de ambos.

Servidor con estado

-  **Persistencia:** como el servidor almacena datos de los clientes que le envían peticiones, esto le permite personalizar las respuestas basadas en esta información histórica.
-  **Simplicidad del lado del cliente:** debido a la persistencia del servidor, el cliente no necesita enviar todos los datos cada vez que interactúa con el servidor, si no muchos menos.
-  **Mayor uso de recursos de hardware:** mantener la información de los clientes, sobre todo si son muchos, implica un incremento significativo en el uso de la memoria, disco y CPU del servidor.
-  **Dificultad en la escalabilidad:** crear copias del servidor en servidores nuevos para equilibrar la carga de peticiones se vuelve complejo por el hecho de sincronizar la información de los clientes entre todos estos servidores.

Servidor sin estado

-  **Simplicidad del lado del servidor:** un servidor sin estado es mucho más simple de administrar debido a que no necesita “recordar” nada de las peticiones de cada cliente.
-  **Escalabilidad simplificada:** cuando el sistema escala y se agregan nuevos servidores para balancear la carga, como no es necesario que estos servidores sepan nada de los clientes, cualquier servidor puede

aceptar cualquier petición en cualquier momento sin mayores inconvenientes.

- **✗ Menor personalización:** como el servidor no “conoce” ni identifica a ningún cliente, la experiencia de los clientes se vuelve la misma para cada uno. Por ejemplo, el manejo de sesiones se dificulta mucho.
- **✗ Mayor inconveniencia para el cliente:** los clientes deben enviar toda la información relacionada a la petición en esa misma petición, lo cual puede causar que estas peticiones se vuelvan muy grandes y engorrosas de crear.

Anexo

Ejercicio 3A: *Tiempo promedio en milisegundos para cada uno de los tamaños (resolución en Java)*

Cantidad de bytes comunicación	Tiempo promedio en milisegundos
10^1	3,1913100
10^2	2,3777500
10^3	2,0404300
10^4	2,2617700
10^5	2,9031000
10^6	13,3698300