Programación Distribuida y Tiempo Real
Práctica 1

JUAN CRUZ RIGOL N°leg.: 16119/0

1) Teniendo en cuenta al menos los ejemplos dados (puede usar también otras fuentes de información, que se sugiere referenciar de manera explícita).

a.- Identifica similitudes y diferencias entre los Sockets en C y en Java.

- El manejo de Sockets entre ambos lenguajes opera de manera similar, ya que ambos llaman a las mismas SysCalls para realizar las comunicaciones.

- Tanto C como Java cumplen el siguiente flujo: Crear Socket > Asignar los datos representativos del mismo según se sea Cliente o Servidor > Realizar la conexión > Leer/Escribir datos al Socket.

- Ambos lenguajes utilizan por defecto comunicaciones bloqueantes.

- En C, la operativa con Sockets es indiferente para clientes como servidores.

- Se crea un Socket con socket() y según lo que devuelva la llamada se procede o se maneja error, no hay diferencia entre clientes y servidores.

- Luego, un cliente utiliza connect() para conectarse al puerto indicado de una dirección IP indicada (el socket servidor) y una vez conectado el cliente puede realizar sus operaciones read() o write().

- Por el lado del servidor, el mismo debe realizar un bind() para setear el file descriptor retornado por socket(), listen() para determinar la cantidad de conexiones a esperar, y accept() para esperarlas.

- Por su parte, Java si realiza una distinción entre la operativa del cliente y el servidor, teniendo el cliente que realizar un Socket() para la conexión, y el servidor un ServerSocket() y Accept(). Los resultados en ambos son los mismos, pero las semánticas entre ambas funciones no son aparentes al usuario (hace por debajo lo mismo que en C para cliente/servidor respectivamente), una es utilizada para cliente y la otra para servidor.

- Otra pequeña diferencia entre los sockets en ambos lenguajes es que C trabaja directamente con datos, es decir que para utilizarlos hay que manejar manualmente los buffers, etc.; mientras que Java trabaja con InputStreams y OutputStreams, que se encargan de eso automáticamente.

Como conclusión se puede afirmar que el manejo de **Sockets** en ambos lenguajes es semánticamente equivalente (con una pequeña excepción). Sin embargo, donde ocurren las diferencias es en la sintaxis, más específicamente en los pasos 2 y 3 del flujo.

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

Esto se debe a que un Cliente/Servidor real tiene un Cliente, es decir un extremo de la conexión que solo realiza peticiones, y un Servidor (con o sin estado), que solo sirve datos a las peticiones que reciba, pudiendo provenir las mismas de diferentes fuentes, al mismo tiempo.

Los ejemplos son, más bien, una conexión 1 a 1 entre un Sender y un Receiver, donde el sender envía datos y espera un ack, y el receptor recibe datos y envía un ack. Al trabajar con protocolos bloqueantes la comunicación no es "fluida" como lo sería un cliente/servidor, y el servidor solo puede responder a una petición por vez por lo cual no hay concurrencia.

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).

El motivo del fallo durante la primera entrega fue que en C, a la hora de imprimir strings, el comando printf continua leyendo memoria hasta encontrar un '\0' que lo termine.Al no tener dicho carácter "cerrando" los strings en mis pruebas, me tiraba basura de memoria hasta encontrarlo aleatoriamente.

El problema se solucionó tomando el último byte del buffer y utilizando el carácter '\0' para el mismo.

Excerpt from man read:

"According to POSIX.1, if *count* is greater than **SSIZE_MAX**, the result is implementation-defined; see NOTES for the upper limit on Linux."

Esto significa que read solo lee hasta SSIZE_MAX, en caso de que la información en el file descriptor ocupe más espacio el comportamiento de read depende de la implementación especifica al OS.

Excerpt from man read:

"On Linux, **read**() (and similar system calls) will transfer at most 0x7ffff000 (2,147,479,552) bytes, returning the number of bytes actually transferred. (This is true on both 32-bit and 64-bit systems.)"

Continuando la lectura del manual, se encuentra que en Linux la operación read solo puede leer hasta 2 a la 16 bytes, caso contrario simplemente retorna los bytes leídos y deja los excedentes. Sin embargo, los resultados obtenidos muestran un truncado a partir de los 10 a la 5 bytes.

Excerpt from baeldung:

"The maximum size of a TCP packet is 64K (65535 bytes)."

Es muy probable que la limitación de tamaño para estas pruebas este siendo el tamaño máximo de paquete TCP/IP.

Excerpt from man read:

"On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number."

Read retorna la cantidad de bytes leídos, entonces printeando dicho valor se puede obtener la cantidad de bytes comunicados. Esto servirá para verificar la hipótesis.

client.c 1 | server.c | run_e2.sh | server_output.txt | client_output.txt

server_output.txt

```
1  Read 9 bytes of buffer from fd
2  Read 99 bytes of buffer from fd
3  Read 999 bytes of buffer from fd
4  Read 9999 bytes of buffer from fd
5  Read 32741 bytes of buffer from fd
6  Read 98223 bytes of buffer from fd
7
```

client.c 1 | server.c | run_e2.sh | server_output.txt | client_output.txt

server_output.txt

```
1  Read 9 bytes of buffer from fd
2  Read 99 bytes of buffer from fd
3  Read 999 bytes of buffer from fd
4  Read 9999 bytes of buffer from fd
5  Read 32741 bytes of buffer from fd
6  Read 32741 bytes of buffer from fd
7
```

client.c 1 | server.c | run_e2.sh | server_output.txt | client_output.txt

server_output.txt

```
1  Read 9 bytes of buffer from fd
2  Read 99 bytes of buffer from fd
3  Read 999 bytes of buffer from fd
4  Read 9999 bytes of buffer from fd
5  Read 32741 bytes of buffer from fd
6  Read 98223 bytes of buffer from fd
7
```

client.c 1 | server.c | run_e2.sh | server_output.txt | client_output.txt

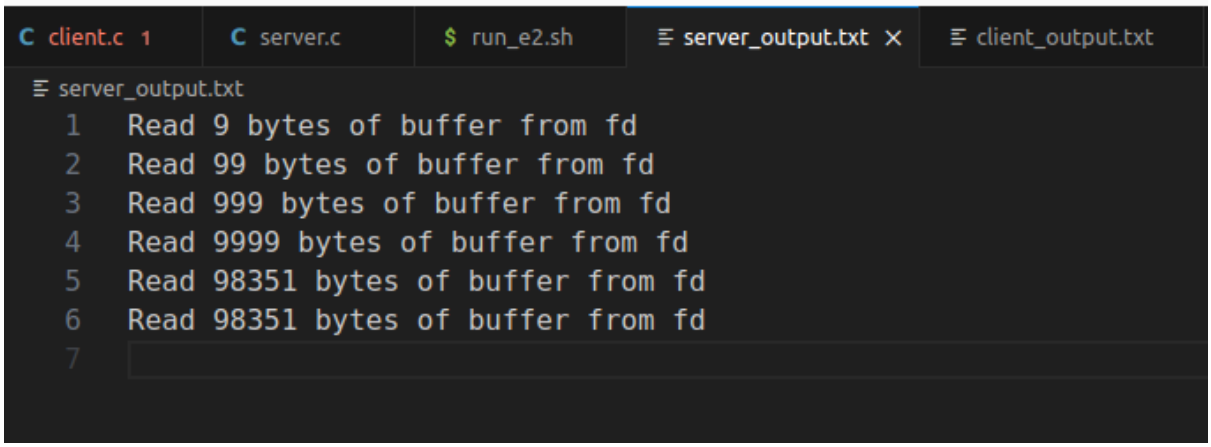
server_output.txt

```
1  Read 9 bytes of buffer from fd
2  Read 99 bytes of buffer from fd
3  Read 999 bytes of buffer from fd
4  Read 9999 bytes of buffer from fd
5  Read 32741 bytes of buffer from fd
6  Read 65482 bytes of buffer from fd
7
```

Se realizaron 4 pruebas, con 4 diferentes resultados. En todos hay una constante, a partir de 10 a la 5 bytes enviados no se reciben todos los bytes, sino que solo se reciben de a grupos de 32741 bytes (65kb y 98kb son múltiplos).

Por tanto se llega a la conclusión de que la comunicación esta siento fragmentada por el protocolo TCP/IP con un tamaño máximo de paquete de 32Kb aproximadamente, y como el programa es “no bloqueante” en lo que se refiere a esperar que se termine el envió de los paquetes, a veces procede antes de que lleguen en su totalidad.

Añadiendo una espera de 10 segundos antes de realizar el read muestra los siguientes resultados.



Esto significa que hay dos limitaciones en juego: tanto el tamaño máximo de salida de TCP/IP (alrededor de 2Gb) que determinara cuantos writes deberá hacer el emisor para enviar la totalidad de sus datos; como el tamaño máximo de buffer de entrada TCP/IP, que determinará cuantos reads deberá efectuar el consumidor para leer la totalidad de los datos enviados en un write.

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 lleguen correctamente, independientemente de la cantidad de datos involucrados.

Como resultado de la investigación al respecto, se encuentra que los paquetes enviados por el emisor que exceden el tamaño máximo del buffer de entrada quedan “encolados” esperando ser leídos.

Por lo cual, la mecánica para el envío de cantidades de datos mayores a las permitidas en una sola comunicación es la siguiente:

- El emisor (cliente) entablará la comunicación con el receptor (servidor) comunicando el tamaño total del volumen de datos a enviar, y esperará un ACK del servidor. Este valor quedara guardado en ambos solo mientras la comunicación ocurra, como una manera de verificar el progreso de la misma. A su vez, ambos iniciaran un contador que medirá la cantidad de bytes enviados/recibidos durante la comunicación, respectivamente.
- Tanto el cliente como el servidor entraran en un doble loop cada uno, donde el más externo será para la verificación del contenido enviado y el más interno para el envío en sí.
- Del lado del cliente, el loop interno se repetirá hasta que se hayan enviado todos los datos a enviar (man write: “On success, the number of bytes written is returned.”). Para tracker ello, se acumula el valor de retorno del write en la variable nr_bytes_sent. En cada iteración, el cliente intentará enviar los bytes restantes a enviar del buffer en su totalidad, desde el punto en el que el último write no pudo enviar. Los bytes restantes son buff_size - nr_bytes_sent, y el índice del cual enviar el buffer es buffer + nr_bytes_sent.
- El loop externo enviará un checksum realizado sobre el buffer en su totalidad al servidor, y esperará la respuesta del mismo. Si el servidor rechaza el checksum, se repite el loop interno, sino se escapa con una comunicación exitosa.
- Del lado del servidor, el loop interno lee el buffer de entrada hasta agotar la comunicación del cliente, es decir que el read devuelva 0 (o que bytes_received sea igual a bytes_torecv en caso de que la comunicación del cliente sea más chica que el tamaño máximo de envío). Al leer el buffer completo, se realiza un ACK al cliente para que continúe la comunicación, y dependiendo de si se recibió el buffer completo o no se vuelve al loop interno o se prosigue a la etapa de verificación.
- En el loop de verificación, el servidor espera recibir un checksum del cliente. Al recibirlo, calcula el checksum del payload recibido y los compara, si son iguales acepta y finaliza la comunicación, y si son distintos rechaza, notifica al cliente, y entra de nuevo en el loop interno.

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.

Para tomar las mediciones temporales se utilizó el método **gettimeofday** de la librería **sys/time.h**, utilizando struct timevals start y end para el tiempo inicial y el final respectivamente.

```
double calculate_time(struct timeval start, struct timeval end, int repetitions)
{
    return ((end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6) / repetitions;
}
```

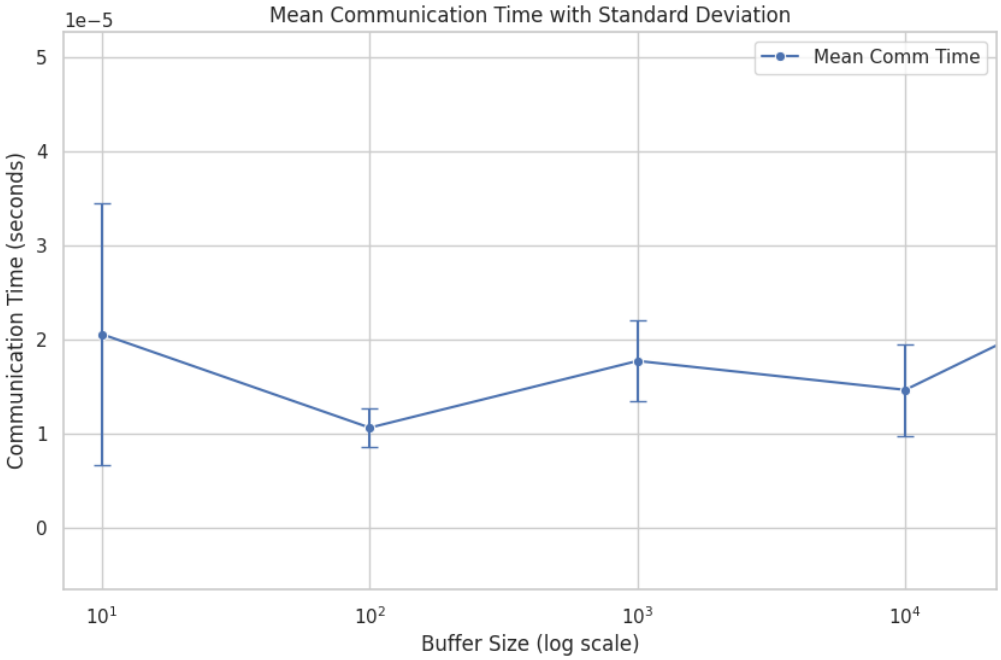
La toma inicial de tiempo se efectúa luego de la inicialización de variables para la comunicación, justo antes de realizar el primer ciclo read/write para la cantidad de datos a enviar. La toma final de tiempo se realiza inmediatamente luego de confirmar el checksum. Ambas mediciones son realizadas en el cliente.

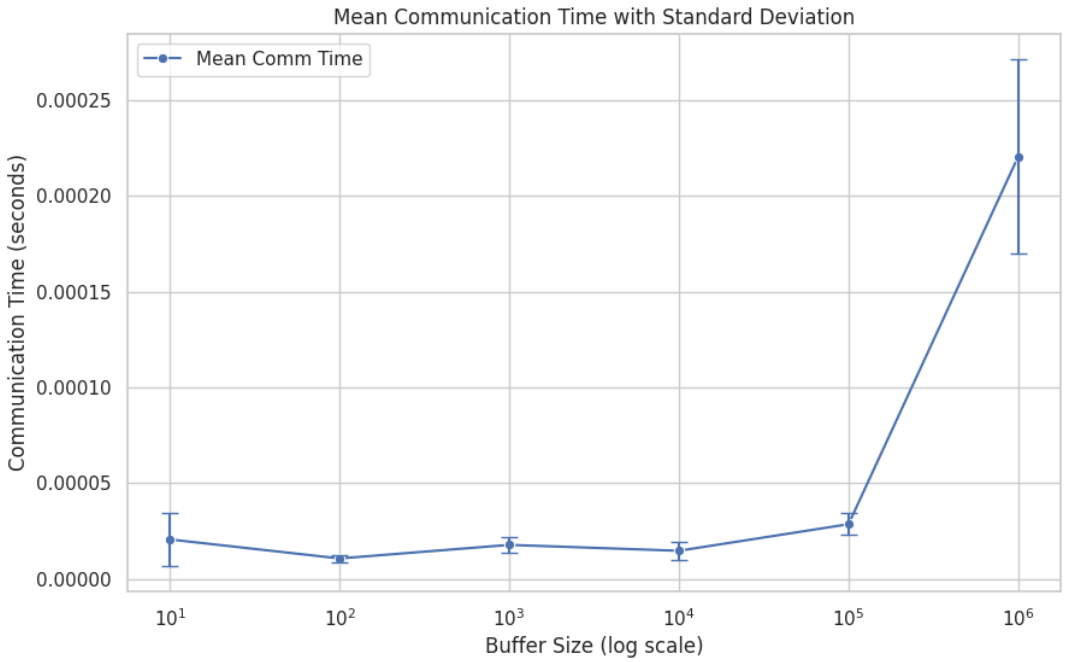
La variable repetitions es la cantidad de ciclos read/write realizados durante la ejecución del programa, multiplicada por dos.

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

Se realizan 100 mediciones por tamaño, registrando el número de medición, tamaño, y tiempo total elapsado de la misma.

Finalmente, las mediciones son guardadas en un .csv que luego es pasado a un script Python para calcular y graficar los resultados utilizando pandas y matplotlib.





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).

Esto se debe a que las redes están optimizadas para transmitir paquetes de un tamaño máximo determinado. Cuando los tamaños de los mensajes son menores, el tiempo de transmisión no varía significativamente porque el paquete se envía de una sola vez y no necesita dividirse.

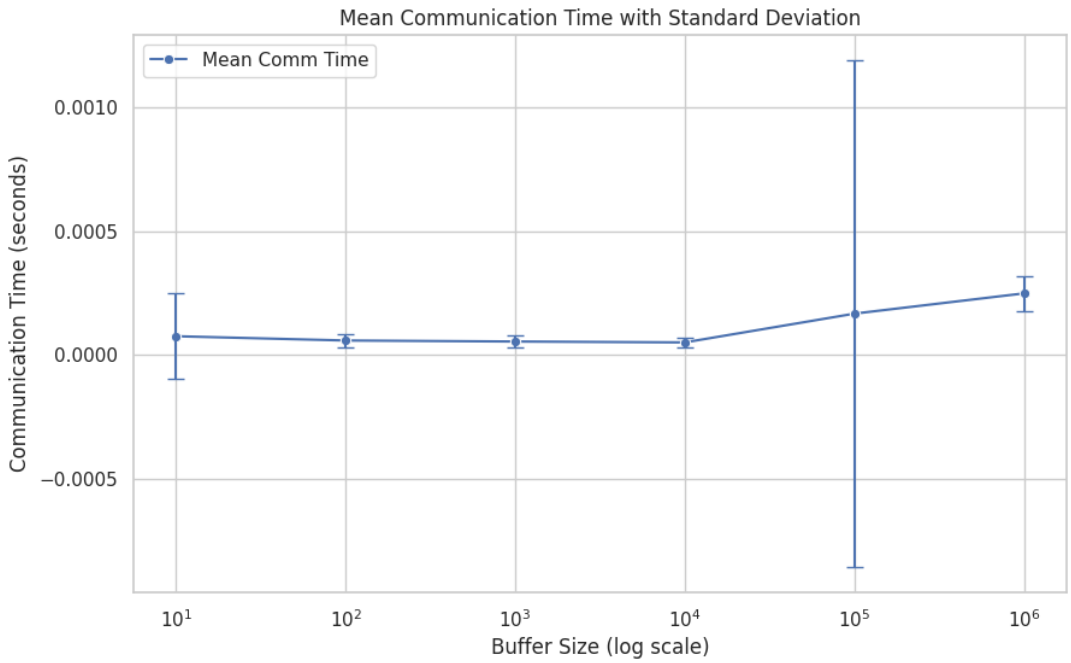
En los casos más chicos 10^1, 10^2, 10^3 y 10^4 el mensaje puede entrar completamente dentro de un único paquete, lo que resulta en tiempos similares entre ellos, ya que no hay fragmentación de paquetes.

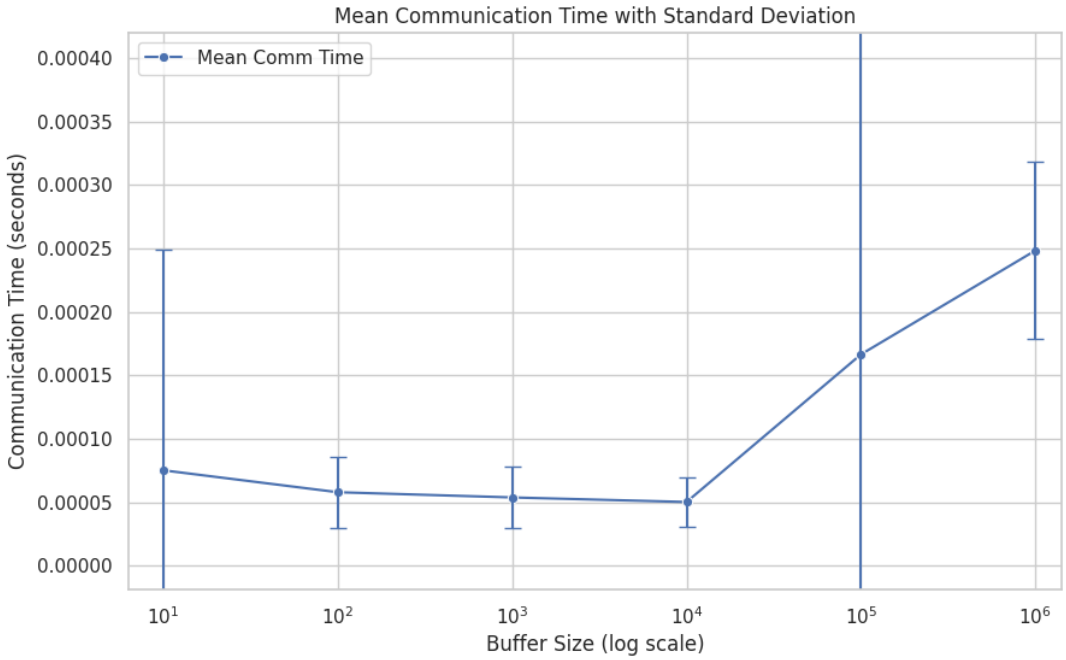
Cuando el tamaño del mensaje excede el máximo (10^5 y 10^6 bytes) el mensaje debe dividirse en varios paquetes para su envío, lo que introduce más latencia debido a la fragmentación y la reensamblación de los paquetes.

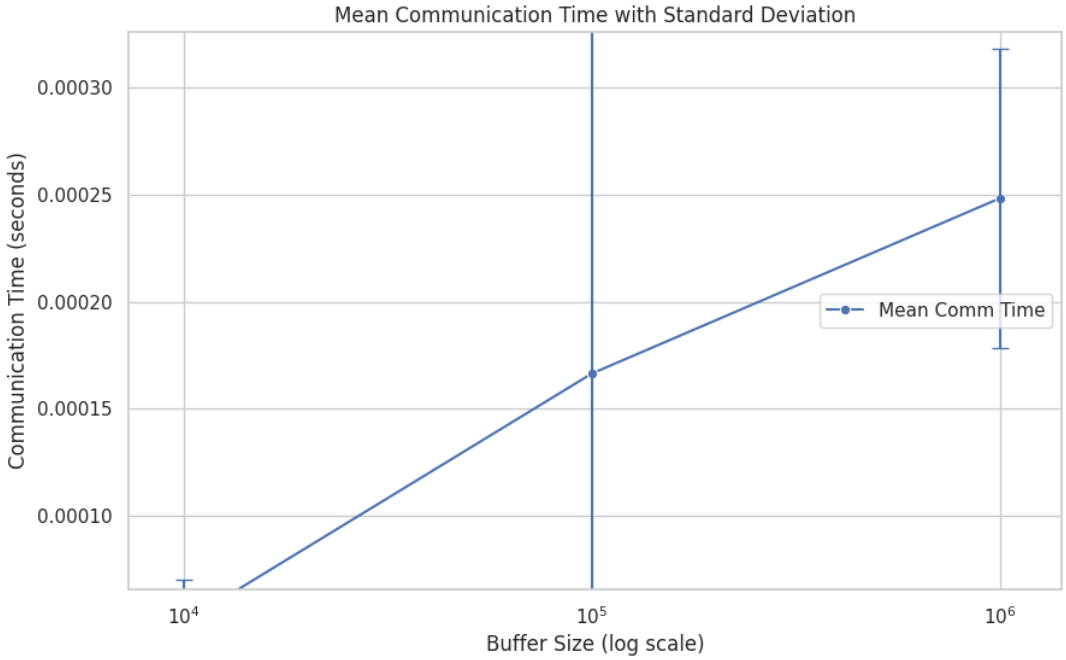
Entre los resultados observados, se nota como las primeras mediciones para cada tamaño registran tiempos más elevados, lo que causa un crecimiento de la desviación estándar.

d.- Compare los tamaños de comunicación para los tamaños 10^5 y 10^6 bytes usando C y Java.

Se puede observar, como demuestran los gráficos a continuación, que C opera mucho más rápido en tamaños elevados que Java. Esto se debe tanto a que C es un lenguaje de más bajo nivel, como al manejo de bufferes automático que implementa Java (añade overhead),y el overhead causado por la serialización de los datos a enviar por Java.







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

Lo mencionado en el item se debe a que en C, las funciones que leen de teclado y envían datos a un socket (scanf y write respectivamente) utilizan buffers de memoria. Por lo tanto, lo que se pasa a la función es una dirección de memoria que contiene el dato.

En el modelo cliente servidor, esto ayuda a mantener el código limpio y fácil de entender, mejorando la eficiencia debido a que no se necesitan múltiples buffers para las comunicaciones, puede utilizarse solo uno.

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.

El servidor entra en un bucle infinito esperando conexiones. Al conectarse un cliente, le manda al mismo el listado de comandos que espera recibir. Luego se bloquea a esperar que el cliente envíe la operación deseada, y según la misma da respuesta.

El cliente se conecta al servidor, e inmediatamente recibe el listado de opciones del mismo. El usuario elige cual usar, y el cliente la comunica al servidor y espera respuesta (en los casos que sea necesaria). Al terminar las operaciones, el usuario le indica al cliente para que el mismo mande el cierre de la conexión al servidor.

Listado de operaciones:

LIST: Lista los archivos en el directorio del servidor.
UPLOAD <filename>: Inicia la subida de un archivo desde el cliente al servidor.
DOWNLOAD <filename>: Inicia la descarga de un archivo desde el servidor al cliente.
DELETE <filename>: Elimina un archivo en el servidor.
EXIT: El cliente le indica al servidor que quiere cerrar la conexión.

FTP como adaptar a esto
El servidor es creado detallando sus parámetros principales, tales como el puerto de conexión, la cantidad de conexiones entrantes que maneja, y el tamaño máximo de cada comunicación (se referirá al conjunto de éstos parámetros como configuración).

Dicha configuración es almacenada y está a disposición del servidor en todo momento (puede ser la representación hexa de los valores concatenados, con un delimitador entre ellos como 0xFF por ejemplo).

El servidor solo recibe una conexión por vez, es decir que al conectarse un cliente el servidor se quedará escuchando al mismo hasta que la conexión se cierre.

Al recibir una conexión, el servidor realiza un acknowledgement de la misma enviando su configuración al cliente, y esperando luego un acknowledgement del mismo (mímica del three way handshake) que detalle la operación a realizar.

Según la operación recibida (recibir o enviar un archivo) el servidor opera de la siguiente manera:

- Al recibir un archivo, el servidor espera recibir dentro de la operación la cantidad de segmentos que conforman el archivo a recepcionar (en caso de un archivo que exceda el máximo tamaño de comunicación) y el tipo de archivo, y se queda escuchando hasta que lleguen todos los mensajes que contengan los segmentos.

- Al recibir cada mensaje de segmento, el servidor primero realiza una verificación de integridad sobre el mismo (esto se puede hacer con un bit de paridad añadido a cada segmento, por ejemplo), y si la verificación es exitosa guarda los mismos en un buffer donde a su vez los va ensamblando y recomponiendo el archivo original.

- Si el segmento se vio corrupto, entonces el servidor termina de recibir todos los segmentos, los descarta, y comunica de la situación al cliente.

- Al enviar un archivo, el servidor lo segmenta, inserta en el primer segmento la metadata necesaria ya detallada, añade bits de paridad en cada segmento, y los envía al cliente. En el último segmento se inserta una señalización de finalización. Luego, el servidor espera un mensaje de recepción del cliente, que indique si el archivo fue enviado exitosamente o hubo corrupción, en cuyo caso el servidor repite el proceso (hasta que el archivo llegue completo).

Disclaimer: Se puede optimizar mucho más, por ejemplo en áreas como manejo de corrupción de los segmentos, reenviando solo los segmentos afectados en vez del archivo completo. No se planteó de una manera más optimizada ya que excede mi capacidad de plantear el desarrollo en marco teórico, sin realizar una implementación y observar métricas y resultados.

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

Un servidor con estado es un servidor que mantiene información sobre la conexión o el cliente al que atiende, a lo largo de múltiples interacciones, por lo que recuerda el contexto de las interacciones previas y puede utilizarlas para futuras solicitudes.

Esto hace que las interacciones continuas vean una eficiencia mejorada, al no necesitar contexto para realizarse. A su vez permite interacciones complejas, como cadenas largas de solicitudes que requieran recordar datos entre las mismas.

El coste es un mayor consumo de recursos y coste del servidor, al necesitar mayores prestaciones para mantener los estados. A su vez, en servidores distribuidos, todos deben tener acceso al estado de cada cliente, por lo que la escalabilidad de los mismos se complejiza.

Un servidor sin estado no guarda información de las interacciones. Cada solicitud se maneja de manera independiente, conteniendo toda la información necesaria para procesarla.

Esto hace que las solicitudes del cliente deban tener metadata que las describa, como autenticación o datos de sesión. A su vez hacen falta más comunicaciones para resolver solicitudes, por ejemplo, si el límite fuese 100 bytes y se tiene un archivo de 98 bytes, al añadirse los headers el archivo sobrepasa el límite necesitando dos mensajes en vez de uno.

Sin embargo, los servidores sin estado son más escalables (tanto en coste como facilidad) y más tolerantes al fallo de los mismos (ya que no se pierde ningún contexto si fuesen a fallar).