



UNIVERSIDAD  
NACIONAL  
DE LA PLATA

## Práctica nro. 1

**Docentes:** Fernando Tinetti, Gastón Marón, Alan Castelli, Julián Delekta.

**Materia:** Programación Distribuida y Tiempo Real.

**Estudiantes:** Juliana Delle Ville, Marianela Diaz Catán.

# Índice

<b>1. Identifique similitudes y diferencias entre los sockets en C y en Java.....</b>	<b>2</b>
<b>2. Tanto en C como en Java (directorios csock-javasock):.....</b>	<b>3</b>
a. ¿Por qué puede decirse que los ejemplos no son representativos del modelo c/s?.....	3
b. Muestre que no necesariamente siempre se leen/escriben todos los datos involucrados en las comunicaciones con una llamada read/write con sockets.....	3
C.....	4
JAVA.....	5
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 lleguen correctamente, independientemente de la cantidad de datos involucrados.....	8
C.....	8
JAVA.....	9
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.....	10
Diseño en C.....	10
Diseño en JAVA.....	10
Fórmulas empleadas.....	10
<b>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?.....</b>	<b>12</b>
<b>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.....</b>	<b>13</b>
<b>5. Defina qué es un servidor con estado (stateful server) y qué es un servidor sin estado (stateless server).....</b>	<b>16</b>
<b>6. Referencias.....</b>	<b>16</b>

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

Cliente	
C	JAVA
La función <code>socket()</code> devuelve un <i>file descriptor</i> para hacer syscalls al sistema	El método <code>socket()</code> crea la conexión <b>directamente</b> , recibe como parámetros la IP y el puerto.
La función <code>connect()</code> se usa para conectarse a una dirección IP o puerto específico	No tiene la función, sino que <code>socket</code> la incluye.
La función <code>read()</code> lee una cantidad determinada de datos desde el <i>socket</i>	El método <code>read()</code> no lee directo de <i>socket</i> sino que tiene que realizar un <i>InputStream</i>
La función <code>write()</code> escribe una cantidad determinada de datos en el <i>socket</i>	El método <code>write()</code> no lee directo de <i>socket</i> sino que tiene que realizar un <i>OutputStream</i>
Puede utilizarse el mismo buffer tanto para leer como para enviar datos a través del <i>socket</i>	Utiliza objetos específicos para cada operación. <i>DataInputStream</i> y <i>DataOutputStream</i>
Puede implementarse la solución tanto del lado del cliente como del servidor utiliza la misma biblioteca <code>socket.h</code>	Para la implementación del cliente utiliza una clase ( <code>java.net.Socket</code> ) distinta que para la implementación del servidor ( <code>java.net.ServerSocket</code> )
Si los datos no pueden ser escritos o leídos en una sola operación no involucra un error	Si se supera el tamaño máximo de bytes transmitidos se produce una excepción
Los sockets tienen límites en la cantidad de datos que envían/reciben	
Los métodos/funciones brindados por las bibliotecas, permiten simular un esquema cliente/servidor	

Servidor	
C	JAVA
La función <code>socket()</code> devuelve un <i>file descriptor</i> para hacer syscalls al sistema. Este <i>socket</i> no se conecta con el cliente, sino que se encarga de crear	Se usa la clase <code>serverSocket()</code> , que puede recibir como parámetro el puerto. Crea el <i>socket</i> donde se va a esperar la conexión de los clientes.

### las conexiones

La función `bind()` asocia un `socket` con un puerto

La asociación del puerto está incluida en constructor `serverSocket()`

La función `listen()` es usada para esperar una cantidad determinada de conexiones entrantes en un `socket`

El método `accept()` es el encargado de esperar por conexiones y aceptarlas

La función `accept()` es usada para esperar las conexiones desde los clientes. Retorna un nuevo `socket` (file descriptor), mediante el cual se realiza la conexión.

El método `accept()` es el encargado de esperar por conexiones y aceptarlas

La función `read()` lee una cantidad determinada de datos desde el `socket`

El método `read()` no lee directo de `socket` sino que tiene que realizar un `InputStream`

La función `write()` escribe una cantidad determinada de datos en el `socket`

El método `write()` no lee directo de `socket` sino que tiene que realizar un `OutputStream`

Los sockets tienen límites en la cantidad de datos que envían/reciben

Los métodos/funciones brindados por las bibliotecas, permiten simular un esquema cliente/servidor

## 2. Tanto en C como en Java (directorios `csock-javasock`):

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

En el modelo **cliente/servidor** el proceso del servidor debe esperar la solicitud entrante de un cliente y asegurar que dicha solicitud sea atendida, luego de atender esta solicitud, debe esperar una solicitud siguiente. En los ejemplos de c y java, se observa que una vez se establece la comunicación entre el cliente y el servidor, el servidor atiende la solicitud del cliente e inmediatamente finaliza el programa. Este comportamiento no es representativo del modelo **cliente/servidor**, donde el proceso server debería quedarse a la espera de una nueva solicitud.

- 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`).

## C

Para realizar las siguientes comprobaciones, se modificaron los ejemplos **client.c** y **server.c** agregando una directiva **#define** llamada **buf\_size** la cual establece el tamaño del buffer. Además, en el ejemplo **client.c**, se agregó la función **memset()** la cual asigna **buf\_size** cantidad de caracteres (para este ejemplo se asignó el carácter **a**) al buffer. En el proceso **server.c** se agregó la función **printf()** la cual informa la cantidad de caracteres que leyó el servidor del buffer (esta información se obtuvo de la función **read()** la cual retorna la cantidad de bytes leídos).

Bytes	S.O	Resultado
$10^3$	Windows	<pre>root@DESKTOP-0NM1LAD:~# ./server 4000 Cantidad de caracteres leídos: 1000</pre>
	Linux	<pre>./server 4000 Cantidad de caracteres leídos: 1000</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela c % ./server 4000 Cantidad de caracteres leídos: 1000</pre>
$10^4$	Windows	<pre>root@DESKTOP-0NM1LAD:~# ./server 4000 Cantidad de caracteres leídos: 10000</pre>
	Linux	<pre>./server 4000 Cantidad de caracteres leídos: 10000</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela c % ./server 4000 Cantidad de caracteres leídos: 10000</pre>
$10^5$	Windows	<pre>root@DESKTOP-0NM1LAD:~# ./server 4000 Cantidad de caracteres leídos: 100000</pre>
	Linux	<pre>./server 4001 Cantidad de caracteres leídos: 65482</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela c % ./server 4000 Cantidad de caracteres leídos: 48996</pre>
$10^6$	Windows	<pre>root@DESKTOP-0NM1LAD:~# ./server 4000 Cantidad de caracteres leídos: 49152</pre>
	Linux	<pre>./server 4000 Cantidad de caracteres leídos: 65482</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela c % ./server 4000 Cantidad de caracteres leídos: 65328</pre>
32767 (SSIZE_MAX)	Windows	

	Linux	<pre>./server 4000 Cantidad de caracteres leídos: 32767</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela c % ./server 4000 Cantidad de caracteres leídos: 32767</pre>
32770	Windows	<pre>root@DESKTOP-0NM1LAD:~# ./server 4000 Cantidad de caracteres leídos: 32770 root@DESKTOP-0NM1LAD:~#</pre>
	Linux	<pre>./server 4002 Cantidad de caracteres leídos: 32741</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela c % ./server 4000 Cantidad de caracteres leídos: 32770</pre>

Para el caso de los procesos **client.c** y **server.c** el valor más grande que podrá ser escrito para la función **write()** o leído con la función **read()** en una sola operación, están delimitados por el tipo `ssize_t` **SSIZE\_MAX**. Este valor se puede obtener mediante el comando **getconf SSIZE\_MAX**, el cual para las pruebas realizadas sobre sistemas operativos linux, windows(wsl) y macOS, arrojaron 32767 bytes.

Todos los sistemas mostraron un comportamiento esperado en la lectura/escritura cuando se respetó el límite máximo **SSIZE\_MAX**, sobrepasado este valor la cantidad de caracteres leídos en una operación comenzó a variar de manera indefinida.

Cabe destacar que la utilización de un **count** mayor a **SSIZE\_MAX** no involucra error en la ejecución. El parámetro **count** determina la cantidad de bytes que pueden ser leídos de forma atómica, si **count** supera **SSIZE\_MAX** la cantidad de caracteres a ser leídos en una sola operación no pueden determinarse, pudiendo variar dependiendo las restricciones que impone el sistema operativo. Por otro lado, las funciones **read()** y **write()** también pueden verse afectadas por una interrupción del sistema, lo cual tampoco involucra un error, pero en caso de que tal suceso ocurriese durante la ejecución de una de estas operaciones, podrían leerse/escribirse menos caracteres de los esperados.

## JAVA

Para realizar las siguientes comprobaciones se ingresaron por teclado los tamaños del byte buffer siguiendo.

A diferencia de C, estos ejemplos se ejecutan sobre una capa adicional que es la JVM, por lo tanto, se supuso que la capacidad de envío sería menor, ya que existe otro capaz de control adicional al S.O

Bytes	S.O	Resultado
$10^3$	Windows	<pre>erial de Practica\javasock&gt; java Server 4000 Here is the message: 1000</pre>
	Linux	<pre>\$ java Server.java 4000 Here is the message: 1000</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela java % java Server.java 4000 Validacion: true Bytes que se leyeron: 1000</pre>
$10^4$	Windows	<pre>erial de Practica\javasock&gt; java Server 4000 Here is the message: 10000</pre>
	Linux	<pre>\$ java Server.java 4000 Here is the message: 10000</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela java % java Server.java 4000 Validacion: true Bytes que se leyeron: 10000</pre>
$10^5$	Windows	<pre>erial de Practica\javasock&gt; java Server 4000 Here is the message: 100000 steps: I got your message</pre>
	Linux	<pre>\$ java Server.java 4000 Here is the message: 100000</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela java % java Server.java 4000 Validacion: true Bytes que se leyeron: 100000</pre>
$10^6$	Windows	<pre>erial de Practica\javasock&gt; java Server 4000 Here is the message: 1000000</pre>
	Linux	<pre>\$ java Server.java 4000 Here is the message: 1000000</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela java % java Server.java 4000 Validacion: true Bytes que se leyeron: 1000000</pre>
$10^7$	Windows	<pre>PS C:\Users\Maiev\Downloads&gt; java -cp Socket-20230821\1730402-001\1. Socket\Material de Practica\j t 4000 Please enter the message: ñ Exception in thread "main" java.net.SocketException: Se ha anulado una conexión establecida host.</pre>
	Linux	<pre>\$ java Server.java 4000 Here is the message: 10000000</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela java % java Client.java localhost 4000 Please enter the message: p Exception in thread "main" java.net.SocketException: Broken pipe</pre>
$10^9$	Windows	<pre>Here is the message: 1000000000 strrrsp I got your message PS C:\Users\Maiev\Downloads&gt; java -cp Socket-20230821\1730402-001\1. Socket\Material de Practica\j Please enter the message: f Exception in thread "main" java.net.SocketException: Se ha anulado una conexión establecida host. at java.base/sun.nio.ch.NioSocketImpl.implWrite(NioSocketImpl.java:418)</pre>

Mayores a $10^9$	Linux	<pre>er.java 4000 Here is the message: 1000000000</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela java % java Client.java localhost 4000 Please enter the message: pepe Exception in thread "main" java.net.SocketException: Broken pipe</pre>
	Windows	<pre>PS C:\Users\Maiev\Downloads\1. Socket-20230621T173040Z-001\1. Socket(Material de Practica)jav Client.java:45: error: integer number too large byte[] buffer = new byte[1000000000000]; ^</pre>
	Linux	<pre>\$ java Server.java 4000 Server.java:58: error: integer number too large buffer = new byte[1000000000000]; ^</pre>
	macOS	<pre>marianela@MacBook-Air-de-Marianela java % java Client.java localhost 4000 Client.java:61: error: integer number too large buffer = new byte[1000000000000]; ^</pre>

Los procesos server y client de JAVA mostraron un comportamiento correcto hasta tamaño de datos de  $10^6$ , por lo tanto se ingresaron tamaños superiores, verificando que el proceso cliente a partir de tamaños mayor o iguales a  $10^7$  genera una excepción que indica el cierre abrupto de la conexión. Pese a la excepción generada en el cliente, se observó que el proceso servidor no presentaba fallos ni inconsistencias con tamaños superiores a  $10^6$ , concluyendo que este comportamiento puede ser debido al manejo de recursos que hace la JVM en conjunto con el S.O. Por último con el objetivo de comprobar si el proceso servidor podría comenzar a fallar a partir de un cierto tamaño de datos, se probó hasta el tamaño límite permitido por el compilador, establecido en  $10^9$  bytes, a partir de este valor el programa ya no compila.

Respecto al error en el cliente para tamaños de datos superiores o iguales a  $10^7$ , se concluyó que era producto del del buffer. Si bien el servidor recibe todos los datos y no se interrumpe la ejecución, en el proceso cliente si se interrumpe por la generación de una excepción. Este comportamiento se creyó que podía tratarse de cómo la JVM gestiona la memoria, donde la misma, al detectar que quiere copiarse un buffer muy grande (aproximadamente 1GB de memoria) hacia el servidor, el cual que supera la cantidad máxima de bytes soportados (JAVA usa punteros de 32 bytes, así que soporta  $2^{31} - 1$  direcciones según [oracle](https://www.oracle.com/technetwork/java/faq-136031.html)), causa el cierre abrupto de la conexión, generando una excepción en el proceso cliente.

Por otro lado, con el objetivo de complementar la conclusión expuesta, se consultó a chat GPT, el cual expone que este comportamiento puede estar dado por:

- Problema de recursos: Si el cliente está utilizando demasiada memoria o recursos, esto podría llevar a problemas en la



comunicación y al cierre abrupto de la conexión. Asegúrate de que el cliente no esté consumiendo una cantidad excesiva de recursos.

- **Buffer Overflow:** En tu código del cliente, estás creando un buffer muy grande (`byte[] buffer = new byte[1000000000];`) y luego estás intentando escribirlo completo en el servidor. Si este buffer es demasiado grande para el sistema, podría causar un desbordamiento de memoria o recursos y resultar en la terminación de la conexión.
- **Limitaciones del sistema operativo:** Algunos sistemas operativos tienen limitaciones en la cantidad de datos que pueden enviarse o recibirse en una sola operación de socket. Si el buffer es demasiado grande, podría exceder estas limitaciones y causar la desconexión.
- **Problemas en la JVM:** Aunque menos común, los problemas en la máquina virtual de Java (JVM) también pueden causar comportamientos inesperados. Asegúrate de estar utilizando una versión actualizada y estable de Java.

Finalmente luego de haber ejecutado el programa sobre un sistema nativo Linux, se comprobó que se realizó sin inconvenientes la transmisión incluso de  $10^9$  bytes, tanto del lado del cliente como del servidor, por lo tanto la evidencia indica que la limitación está dada por el propio S.O.

- 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 lleguen correctamente, independientemente de la cantidad de datos involucrados.](#)

## C

Para la verificación de los datos que se envían, en **client.c** se realizaron las siguientes modificaciones:

1. Se transfiere mediante el socket la cantidad de bytes que luego se le enviará.
2. Se espera validación de recepción.
3. Se calcula la función de hash sobre los datos a enviar y se almacena para su posterior envío.
4. Se transfiere el mensaje a través del socket.
5. Se espera validación de recepción.
6. Se envía la función de hash.
7. Se espera validación de recepción.

Para la verificación de los datos que se reciben, en **server.c** se realizaron las siguientes modificaciones:

1. Se recibe la cantidad de bytes del mensaje que luego que se recibirá y se almacena dicho valor para posteriormente comprar.
2. Se envía mensaje de validación de recepción a través del socket.

3. Lee el socket (realizando desplazamiento sobre el buffer si tiene que leer más de una vez) hasta que la cantidad de bytes leídos sea la misma que el proceso client le indico previamente.
4. Se envía mensaje de validación de recepción a través del socket.
5. Se calcula función de hash y almacena el valor calculado.
6. Se recibe el valor de hash calculado por el proceso client.
7. Verifica si el hash generado en el server es el mismo que el hash recibido por el proceso server
8. Se envía mensaje de validación de recepción a través del socket.

## JAVA

Para los procesos de JAVA se diseñó un mecanismo similar al de C, pero simplificando las verificaciones. En este caso no se verifica el checksum ni tampoco la cantidad de bytes. La estrategia de hash que se usó fue MD5. El cliente a su vez le manda el tamaño de buffer por comodidad para hacer las pruebas.

### Desde cliente.java:

1. Se cambió la implementación de forma tal que el cliente le mande al servidor el buffer por comodidad a la hora de realizar pruebas. Como primer paso le paso el buffer.
2. Se genera el checksum del archivo a enviar.
3. Luego se le envían los datos al servidor de la siguiente forma:
  - 3.1. Checksum
  - 3.2. buffer

Por simplicidad se evitaron las verificaciones de ida y vuelta. El checksum usado es una implementación sencilla del algoritmo MD5. Tiene dos métodos uno que encripta por la clave MD5 y otro que lo desencripta y se fija si es válido:

- Generate: Este método toma como entrada una matriz de bytes message y devuelve una matriz de bytes que representa la suma de comprobación MD5 del mensaje. Inicializa una instancia de MessageDigest con el algoritmo MD5, la actualiza con el mensaje de entrada y luego genera la suma de comprobación.
- checksumToString: toma una matriz de bytes checksum y la convierte en una representación de cadena hexadecimal.
- isValid: Este método verifica si la suma de comprobación coincide con la suma de comprobación del mensaje dado. Convierte la suma de comprobación en una cadena hexadecimal y el mensaje en una nueva suma de comprobación MD5. Luego, compara las dos cadenas de suma de comprobación y devuelve true si son iguales.

### Del lado del servidor.java:

1. lee el buffsize.
2. se itera para poder leer todo el archivo completo
3. lee el checksum
4. se genera el checksum de los datos recibidos y se compara

- 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.

#### Diseño en C

Para realizar el cálculo de tiempo de comunicación, se decidió realizar las medidas desde el lado del proceso cliente para el programa realizado sobre C. La metodología implementada consiste en una función la cual llamamos **dwalltime()**, la cual comienza a calcular el tiempo en la primera comunicación y finalmente toma el tiempo en la última comunicación. Por otro lado, se decidió calcular con la misma función **dwalltime()** en el proceso servidor solo el tiempo de cálculo de la función de hash sobre el mensaje leído, todo el resto del procesamiento fue despreciado, ya que consideramos que no representa un tiempo significativo.

El cálculo del tiempo de comunicación comprende la resta de la última comunicación menos el tiempo de la primera comunicación, menos el tiempo de procesamiento del hash en el proceso servidor.

#### Diseño en JAVA

Para este caso se decidió tomar checksum como parte de la comunicación entre los procesos. Para tomar los tiempos se usó nanoTime y las mediciones de la misma se hicieron del lado del cliente. Las mediciones se toman por cada write() y read(), despreciando el tiempo en el medio del trabajo de servidor.

#### Fórmulas empleadas

Una vez obtenido los tiempos total para cada tamaño de datos, se realiza el promedio sobre los valores obtenidos con la siguiente fórmula:

$$\text{Tiempo Comunicación } i = \frac{\text{Tiempo Total}}{n}$$

donde **n** representa la cantidad de comunicaciones (read y write) en el proceso client.

Finalmente, para calcular el tiempo aproximado medio de cada comunicación, se dividió entre 2 el tiempo promedio total, estimando que el tiempo de ambas comunicaciones son aproximadamente iguales.

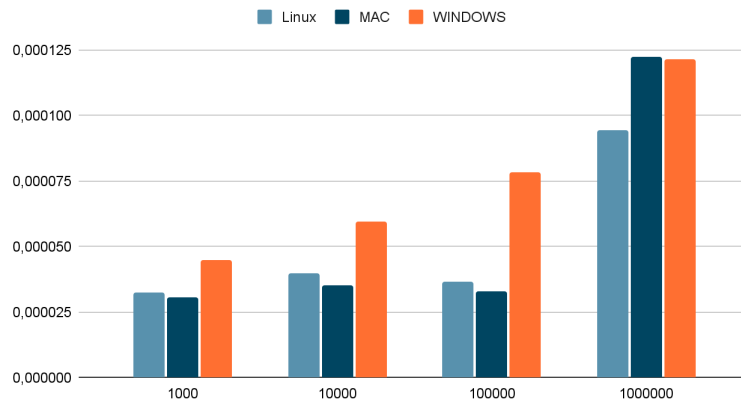
La fórmula utilizada es la siguiente:

$$\text{Tiempo Total Promedio} = \frac{\sum_i^n \text{Tiempo comunicacion } i}{n}$$

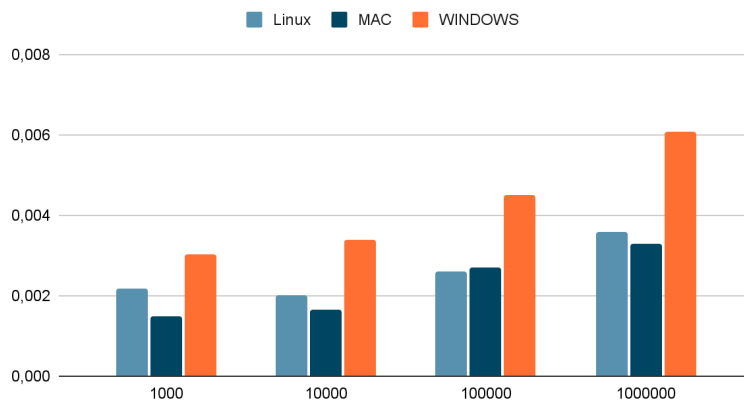
donde **n** representa el tamaño de la muestra utilizado.

El siguiente gráfico muestra el promedio de tiempo de comunicación para tamaños de datos de  $10^3$ ,  $10^4$ ,  $10^5$  y  $10^6$  y el Promedio:

Promedio de comunicación C



Promedio de comunicación JAVA

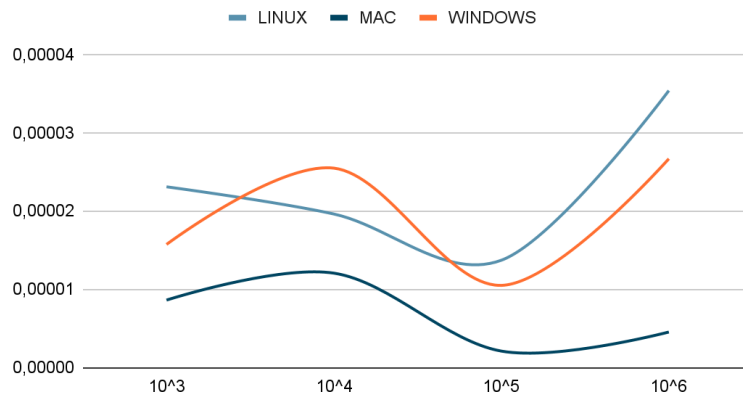


Finalmente para el cálculo de la desviación estándar se utilizó la siguiente fórmula:

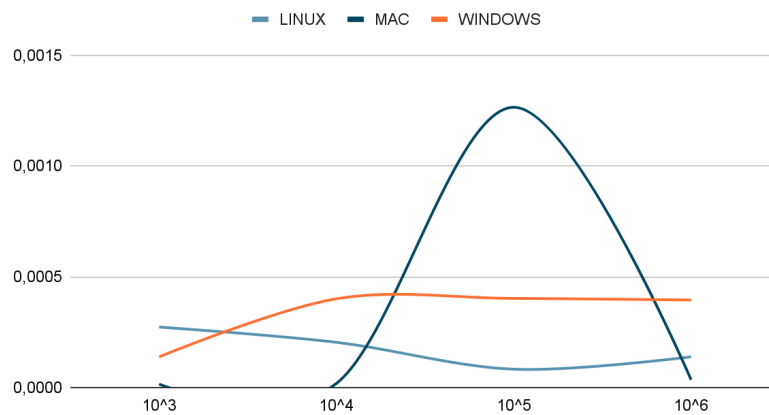
$$S = \sqrt{\frac{\sum_i^n (\text{Tiempo comunicacion } 10^i - \text{Tiempo Total Promedio})^2}{n}}$$

El siguiente gráfico ilustra la desviación obtenida para tamaños de datos de  $10^3$ ,  $10^4$ ,  $10^5$  y  $10^6$ :

Desviación estandar C



Desviación Estandar JAVA



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?

En C se puede utilizar la misma variable tanto para leer como para escribir porque es un puntero que apunta a una posición de memoria determinada. La función **\*fgets(char \*s, int size, FILE \*stream)** almacena los valores recibidos en el buffer luego del último valor presente en el mismo. Por otro lado, la función **ssize\_t write(int fd, const void \*buf, size\_t count)** escribe una cantidad de bytes en el socket, desde el buffer apuntado, hasta la cantidad determinada por el parámetro **count**.

El uso de una única variable "buffer" tanto para almacenar los valores leídos de teclado como para enviar los datos a través del socket es relevante especialmente si se cuenta con un equipo o infraestructura con poca capacidad de memoria disponible. En este caso, cada proceso o thread que está atendiendo a un cliente, reutilizan su espacio evitando tener un buffer privado para recibir y otro para enviar.

La ventaja que tiene este enfoque es que se reutiliza el espacio, consumiendo menos recursos de memoria, pero por otro lado, la desventaja de este enfoque es la dificultad de su programación, ya que el programador debe hacerse responsable de la sincronización de los threads/procesos para evitar cualquier tipo de interferencia entre los mismos, así como la administración/manejo adecuado de la memoria.

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.

Un modelo de servidor de archivos remoto basado en la interfaz del protocolo FTP, se podría implementar para manejar de forma básica la transferencia de archivos, mediante los comandos **GET**, **PUT**, **LIST** de la siguiente forma:

LIST		
Paso	Cliente	Servidor
1.		En espera de una petición mediante la función <code>accept()</code> .
2.	Mediante la función <code>write()</code> se envía un buffer con el contenido "LIST directorio" al servidor y queda a la espera de respuesta del servidor mediante el uso de la función <code>read()</code> .	
3.		Recibe una petición mediante la función <code>read()</code> , procesa el pedido y le envía al cliente con un <code>write()</code> el contenido del directorio solicitado.
4.	El cliente recibe la petición y procede a leer los datos recibidos, finalmente termina.	
5.		Se queda a la espera de nueva petición mediante la función <code>accept()</code>

GET		
Paso	Cliente	Servidor
1.		En espera de una petición mediante la función <code>accept()</code> .
2.	Mediante la instrucción <code>write()</code> se transmite al socket el contenido "GET archivo", luego queda a la espera de la respuesta del servidor con la función <code>read()</code> .	
3.		El servidor detecta la petición y la recibe con la función <code>read()</code> . Extrae la información la parsea, ubica el archivo requerido y envía contenido del mismo con función <code>write()</code> .
4.	El cliente recibe la respuesta del servidor con la función <code>read()</code>	
5.		Envía un código hash para verificar integridad de envío con función <code>write()</code> y se queda a la espera de recibir un "ok" de parte del cliente con un <code>read()</code> .
6.	Lee hash del servidor con <code>read()</code> , aplica función de hash sobre el contenido del archivo recibido y compara los valores. Si está correcto hace un <code>write()</code> con respuesta "ok" y termina. Caso contrario escribe "not ok", y vuelve al paso de esperar el archivo.	
7.		Se queda a la espera de nueva petición mediante la función <code>accept()</code>

PUT		
Paso	Cliente	Servidor
1.		En espera de una petición mediante la función <code>accept()</code> .
2.	Mediante la instrucción <code>write()</code> envía al socket el contenido "PUT archivo" y se queda esperando la respuesta del servidor en un <code>read()</code>	
3.		Detecta el pedido y hace un <code>read()</code> . Le responde al usuario haciendo un <code>write()</code> "ok" y se queda esperando en un <code>read()</code> al archivo
4.	Hace un <code>write()</code> del archivo completo	
		El servidor hace un <code>read()</code> para recibir el archivo
5.	Aplica función de hash sobre el contenido del archivo y envía el hash con <code>write()</code> al servidor y espera con <code>read()</code> .	
6.		Hace <code>read()</code> para recibir el hash del cliente y luego aplica función de hash sobre el archivo recibido y compara el hash con el recibido del cliente. Si está correcto le envía un "ok", caso contrario envía un "not ok" mediante operación <code>write()</code>
7.	Si recibe respuesta ok termina, caso contrario vuelve al paso enviar archivo.	
8.		Espera nueva petición mediante <code>accept()</code> .



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

**Stateful server:** Este tipo de diseño mantiene información de forma persistente en sus usuarios. Este tipo de información necesita ser explícitamente eliminada por el server. La ventaja es que tiene mayor rendimiento que un servidor stateless por su capacidad de actualizar constantemente el estado. Como desventaja, al momento de un crash se debe recuperar su estado total. Es decir la tabla (cliente,archivo) y no hay garantías si la información que se recupera es la última versión.

**Stateless server:** Este tipo de diseño no mantiene la información de estado de los clientes, y puede cambiar su propio estado sin tener información del cliente. Algunos servidores “sin estado” mantienen información de los clientes, pero la pérdida de esta información no involucra la interrupción del servicio. Una forma de stateless servers son los servers soft state, los cuales mantienen la información/estado del cliente durante un periodo de tiempo.

6. Referencias

Tiempo comunicación C (punto 2d):

[https://docs.google.com/spreadsheets/d/1\\_FxCkY9j\\_w7bkGju-s\\_fn1WUmbSywgxhaKaw4zNqEkA/edit#gid=0](https://docs.google.com/spreadsheets/d/1_FxCkY9j_w7bkGju-s_fn1WUmbSywgxhaKaw4zNqEkA/edit#gid=0)

Tiempo comunicación JAVA (punto 2d):

[https://docs.google.com/spreadsheets/d/12on\\_GhDxy6ggGQQxHMRgo3cOsYclloVjJRawYi9fxYo/edit#gid=0](https://docs.google.com/spreadsheets/d/12on_GhDxy6ggGQQxHMRgo3cOsYclloVjJRawYi9fxYo/edit#gid=0)