Practica 1

Inti María Tidball 17612/3
Juan Pablo Sanchez Magariños 13238/3

1. **Identifique similitudes y diferencias entre los sockets en C y en Java.**
   Si estudiamos las implementaciones de sockets en C y en Java, podemos observar que ambos proporcionan un mecanismo para la comunicación entre procesos en redes, con soporte para varios tipos de protocolos como TCP y UDP. Con sockets se permite abrir y cerrar conexiones, y enviar y recibir datos.
   Sin embargo, notamos que tienen varias diferencias. Java es más transparente al usuario, y encapsula muchas características para hacerlo más simple para el uso, escondiendo varios elementos de la implementación en el Factory. Mientras que C es más complejo, pero ofrece un control más detallado sobre la configuración de sockets a nivel del SO. Podemos observar que Java tiene una API orientada a objetos para trabajar con sockets, mientras que C utiliza una API basada en funciones y estructuras, por sus orientaciones como lenguajes. Similarmente, ya que Java es un lenguaje que corre sobre una máquina virtual, y tiene una funcionalidad que seleccionar código para ser interpretado o compilado a bytecode, es lógico que arroje excepciones, mientras que C, que es un lenguaje compilado con una implementación más cercana al SO, utiliza códigos de retorno y variables globales para manejar errores. Se podría decir que los sockets en Java son más portables debido a la naturaleza portátil de la plataforma Java, pero también menos eficientes por el overhead que produce tener las capas de software necesarias para correr la máquina virtual de Java, mucho más alejadas del hardware que un lenguaje directamente compilado como C.

2. **Tanto en C como en Java**

   a. **¿Por qué puede decirse que los ejemplos no son representativos del modelo c/s?**
      Ya que no se cumplen los pasos que definen a este tipo de modelo:
      - La comunicación no se **inicializa**, simplemente el servidor escucha y el cliente envía la información.
      - Dicha información difícilmente constituye una **petición**, es simplemente un envío.
      - Tampoco se **finaliza** protocolarmente la comunicación, tanto el cliente como el servidor asumen que el envío de información concluye después del primer mensaje.

   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 103, 104, 105 y 106 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).**

Las llamadas read y write pueden no transferir todos los datos en una sola operación. Se agrega un parámetro de entrada al cliente y al servidor, para poder setear la cantidad de datos que se van a enviar (modificación del tamaño del buffer), se aloca la memoria para ese tamaño y se llena el buffer con un patrón repetido de letras para verificar el envío correcto.
En casos mayores a 10^5 se perciben errores con respecto a la recepción debido a los buffers.

```
$ ./server 4000 1000
Bytes recived: 1000
$ ./server 4000 10000
Bytes recived: 10000
$ ./server 4000 100000
Bytes recived: 32741
$ ./server 4000 1000000
Bytes recived: 32741
$
```

```
$ ./client localhost 4000 1000
Sending 1000 bytes
$ ./client localhost 4000 10000
Sending 10000 bytes
$ ./client localhost 4000 100000
Sending 100000 bytes
$ ./client localhost 4000 1000000
Sending 1000000 bytes
$
```

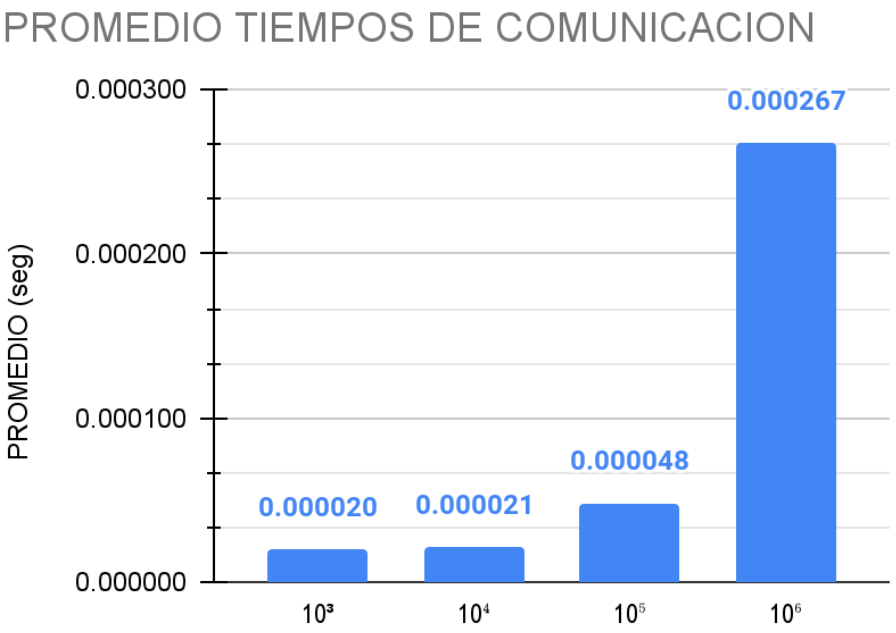
c. **Agregue a la modificación anterior una verificación de llegada correcta de los datos que se envían (cantidad y contenido del buffer), de forma tal que se asegure que todos los datos enviados llegan correctamente, independientemente de la cantidad de datos involucrados.**

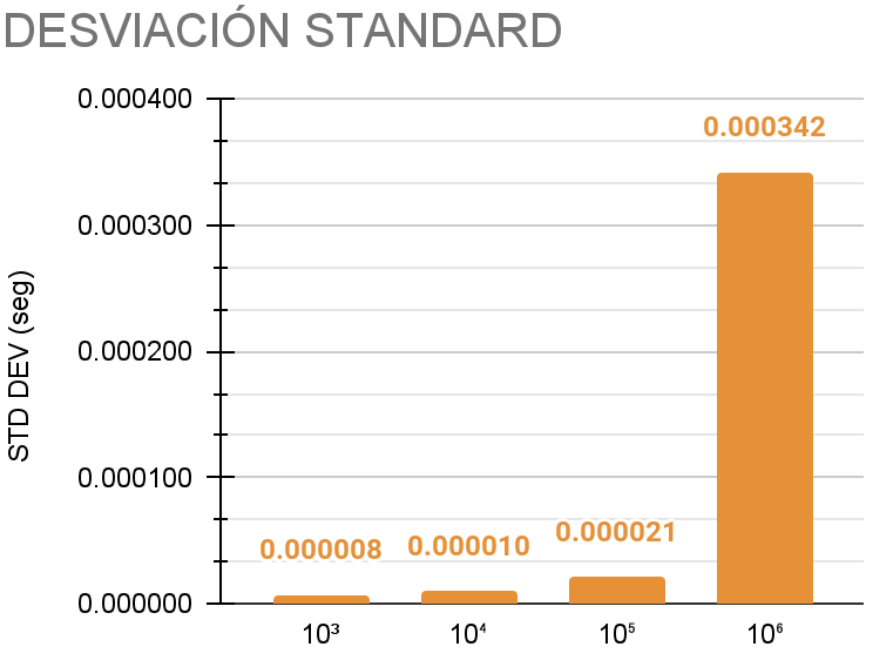
Se agrega una modificación en donde se implementa en el servidor un loop do while, que repetirá hasta haber recibido todos los datos (cuyo tamaño sabemos de antemano). Entonces se va registrando la cantidad de bytes leídos en una variable y en la siguiente iteración se concatena en el buffer, utilizando la variable antes mencionada para apuntar a la dirección correspondiente.

```
$ ./server 4000 1000
Bytes recived: 1000
$ ./server 4000 10000
Bytes recived: 10000
$ ./server 4000 100000
Bytes recived: 100000
$ ./server 4000 1000000
Bytes recived: 1000000
$
```

```
$ ./client localhost 4000 1000
Sending 1000 bytes
$ ./client localhost 4000 10000
Sending 10000 bytes
$ ./client localhost 4000 100000
Sending 100000 bytes
$ ./client localhost 4000 1000000
Sending 1000000 bytes
$
```

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.



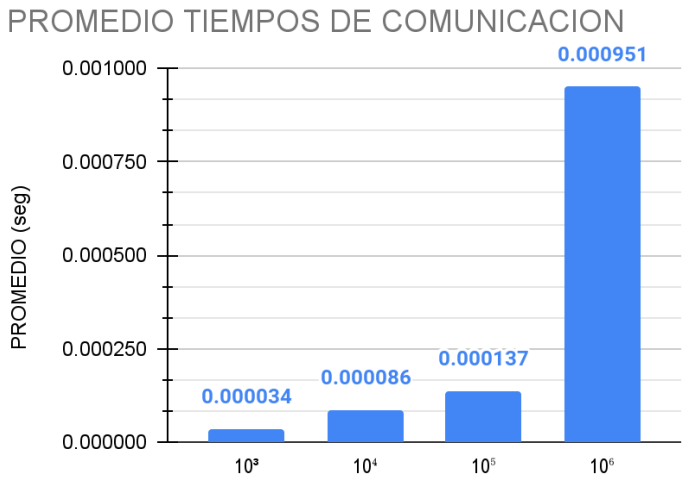


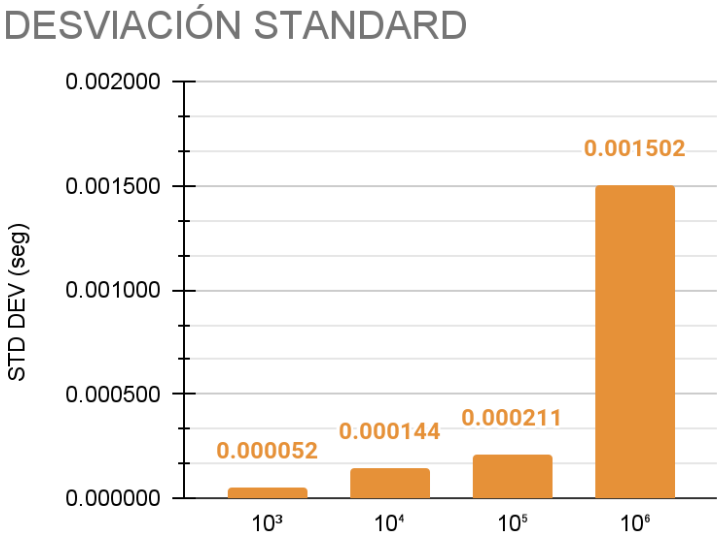
Para calcular el tiempo de cada comunicación se utilizaron dos variables de tipo `clock_t` llamando a la función `clock()` inmediatamente antes de que el cliente envíe el mensaje, y después de de que el mismo reciba la confirmación del servidor.

Se restan estos dos tiempos y se dividen por dos, ya que se tiene en cuenta la ida y la vuelta. En este caso hicimos un pequeño script de bash para poder ejecutar el

programa muchas veces seguidas y que imprima solamente estos tiempos, en un formato que luego fue sencillo de importar en una planilla para realizar la estadística.

Para hacer estos cálculos quitamos la verificación de datos, implementada en el punto anterior, ya que notamos que perjudicaba los tiempos





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 todo lo que es variable es un flujo de bytes en memoria, entonces una misma variable podría ser usada para leer del teclado o para enviar datos a través de un socket. O sea, una variable es esencialmente un área de memoria que se puede leer o escribir invariablemente de las librerías o llamadas del SO que luego utilizan sus datos.
   Por ejemplo, una variable en C puede almacenar los datos que guarda **scanf** o **fgets** al leer del teclado y luego enviar los datos almacenados en la misma usando un **send** o un **write** para enviar en un socket. En ambos casos se están leyendo o escribiendo

bytes en memoria. Después el SO o las bibliotecas tienen los estándares o protocolos definidos de cómo se manejan esos bytes.
En una aplicación cliente/servidor, podrías tener una estructura de datos que almacena bytes en función de la entrada del usuario (**scanf** o **fgets**) y luego los envía tal cual a través de un socket a un servidor (**send** o **write**), lo cual podría simplificar bastante el código.

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.**
   Se podría implementar un servidor de archivos remotos con sockets basándose en la arquitectura cliente/servidor. El cliente tendría que implementar dos envíos de peticiones: uno para subir un archivo y otro para descargarlo. Por otro lado, el servidor estaría siempre a la espera de estas peticiones. Cada proceso utilizará dos sockets para mantener las dos conexiones necesarias para el envío/recepción de las peticiones, que utilizarían el protocolo TCP. Así la integridad de los datos compartidos estaría garantizada.

5. **Defina qué es un servidor con estado (stateful server) y qué es un servidor sin estado (stateless server).**
   El Stateful server mantiene información sobre cada sesión con el cliente, tiene ‘memoria’ de sesiones anteriores, el Stateless no mantiene ninguna información sobre interacciones pasadas, cada request es una transacción independiente.