Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Programación distribuida y tiempo real

Entrega 1

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Ejercicio 1

C	Java
Utiliza file descriptors para representar los sockets.	Utiliza clases para representar los sockets.
Los datos se envían mediante el mismo puntero al buffer de bytes.	En java se utilizan diferentes objetos que manejan el streams de datos y se encargan de mandar/recibir datos.
La elección del protocolo de red y transporte se establecen como parámetros de la función **socket()** la cual crea un socket. En este código se elige AF_INET para utilizar la familia IPv4 y SOCK_STREAM para TCP.	La elección de la capa de red y de transporte se realiza por herencia de clases. Por ejemplo para usar UDP se necesita heredar de DatagramSocket.
Se vincula el file descriptor del servidor con la dirección y el puerto de forma explícita con la función **bind().**	No hace el bind de forma explícita para el programador, simplemente se instancia la clase ServerSocket.
Se utiliza el **listen** para esperar una conexión con el cliente.	No hace el listen de forma explícita para el programador, solo realiza el accept.
Se utiliza del lado del cliente la system call **connect()** la cual conecta el file descriptor con la dirección especificada.	No hace el connect de forma explícita para el programador.
Se establece la cantidad máxima de 5 conexiones, mediante la función **listen().**	Por default, la clase ServerSocket acepta 50 conexiones aunque esto se puede parametrizar.
Se ve que deja la administración de memoria como responsabilidad del programador	La implementación se la ve con más abstracción para el programador.
Ambos siguen la misma interfaz para la implementación de sockets.	
Ambos utilizan las llamadas write para enviar mensajes y read para recibir los mensajes.	
Ambos están usando TCP como protocolo de transporte e IPv4 como protocolo de red.	
Tanto la librería socket.h como java.net aporta funciones/métodos suficientes para poder implementar un esquema C/S.	

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Ejercicio 2

Inciso A

Puede decirse que los ejemplos no son representativos del modelo C/S por las siguientes razones:

- ➜ En los ejemplos el servidor recibe una única petición y luego finaliza. En cambio en el modelo c/s el servidor siempre está a la espera de nuevas peticiones.
- ➜ No se tiene manejo de los errores que podrían ocurrir cuando se envíe un mensaje que no entre en el buffer, por ejemplo cuando el mensaje enviado no fue recibido o cuando el mensaje enviado tiene datos corruptos.
- ➜ En el modelo c/s los servidores tienen varios recursos que pueden ser accedidos a través de una interfaz, y los usuarios realizan peticiones para poder utilizar los mismos. En los ejemplos no existe ningún tipo de recurso en el servidor y los usuarios no solicitan nada, simplemente envían un mensaje.

Inciso B

Para mostrar que no necesariamente siempre se leen/escriben todos los datos involucrados en las comunicaciones utilizando las llamadas **read/write** se realizaron los códigos *2b/Client.java y 2b/Server.java* en donde el cliente envía 4 mensajes, cada uno de tamaño 10^3, 10^4, 10^5 y 10^6 bytes respectivamente. Luego el servidor imprime cuantos bytes recibió utilizando el valor que retorna la llamada **read**.
El output de los scripts fueron los siguientes:

```
root@ac1e56547eb3:/pdytr/practica-1/ejercicios/2b# java Client localhost 3000
Bytes send: 1000
----------------------------------------
Bytes send: 10000
----------------------------------------
Bytes send: 100000
----------------------------------------
Exception in thread "main" java.net.SocketException: Connection reset
        at java.net.SocketOutputStream.socketWrite(SocketOutputStream.java:118
)
        at java.net.SocketOutputStream.write(SocketOutputStream.java:159)
        at java.io.DataOutputStream.write(DataOutputStream.java:107)
        at Client.main(Client.java:61)
```

```
root@ac1e56547eb3:/pdytr/practica-1/ejercicios/2b# java Server 3000
Bytes read: 1000
Expected bytes: 1000
----------------------------------------
Bytes read: 10000
Expected bytes: 10000
----------------------------------------
Bytes read: 65482
Expected bytes: 100000
----------------------------------------
Bytes read: 32741
Expected bytes: 1000000
----------------------------------------
```

Se puede ver que el cliente al enviar el mensaje de 10^5 bytes el servidor solo recibe 65482, y al enviar 10^6 bytes solo se leen 32741 bytes.
También se puede notar una excepción en el cliente, esto sucede porque el servidor solo recibió 32741 bytes y por el flujo del código cerró su lado de la conexión mientras el cliente está enviando los bytes restantes.

Esto nos resultó raro, entonces leímos la documentación de la syscall **read.** La cual nos dice que comienza a leer los bytes una vez que la información esté disponible, particularmente cuando se utiliza sockets TCP esto se hace para optimizar la conexión (https://linuxhint.com/read_syscall_linux/).
Llegamos a la conclusión que lo que está pasando es que el cliente al mandar una información tan grande (mayor a 10^5 bytes) va a ser fragmentada por TCP, por lo tanto el cliente va a mandar N porciones de la información en segmentos TCP. Luego el read va a leer la información que esté disponible pero puede ser que el cliente no haya terminado de mandar la información, por lo tanto se corre el riesgo de no leer la misma cantidad que se ha enviado. Eso es lo que sucede por el lado de la red, pero también juegan otros factores como el tamaño del buffer que cada Sistema Operativo aloja para la syscall **read**.

La implementación se encuentra en los siguientes archivos:
- ➜ Script del cliente: *2b/java/Client.java*
- ➜ Script del servidor: *2b/java/Server.java*
- ➜ Para compilar ejecutar: *javac <archivo.java>*

Adicional

Para ver esto más en detalle realizamos los siguientes códigos: *2b/adicional/Server.java* **-** *2b/adicional/Client.java*. El script del cliente sigue igual al ejercicio anterior, mientras que el script del servidor al leer el mensaje se quedará en un bucle hasta leer todos los bytes que le ha enviado el cliente, para ello se acumula el valor que retorna la llamada **read** el cual nos permite saber cuántos bytes fueron leídos. También realizamos una función **correctBytes** la cual nos indica cuántos son correctos de los recibidos, es decir que se correspondan con el mensaje que envió el cliente.

En el output (*2b/adicional/output.txt*) podemos ver que para leer 10^5 bytes se necesitaron 3 lecturas y para leer 10^6 bytes se necesitaron 9 lecturas. También podemos ver que en el buffer se están mezclando los bytes finales del mensaje de 10^5 y los primeros bytes del mensaje de 10^6.

Todos estos problemas son solucionados en el próximo inciso.

La implementación se encuentra en los siguientes archivos:
- ➜ Script del cliente: *2b/adicional/java/Client.java*
- ➜ Script del servidor: *2b/adicional/java/Server.java*
- ➜ Para compilar ejecutar: *javac <archivo.java>*

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Inciso C

Para solucionar el tamaño del mensaje que se ve en el ejercicio anterior utilizamos la función **read()** dentro de un loop, el cual finaliza una vez que se haya leído el mensaje completo.
En cada iteración la función **read()** nos retorna la cantidad de bytes leídos, esa cantidad la sumamos a una variable **desplazamiento** la cual usamos para desplazarnos por el buffer para no leer los mismos datos ni la misma cantidad de datos en la siguiente iteración. El loop finaliza una vez que la variable **desplazamiento** sea igual al tamaño del mensaje.
La implementación es la siguiente:

Java

```java
int totalBytesRead = 0;

while (totalBytesRead < bufferSize)
{
    int bytesRead = fromclient.read(
                        buffer,
                        totalBytesRead,
                        bufferSize - totalBytesRead
                    );

    if ( bytesRead < 0 )
    {
        System.err.println("Error to read buffer");
        System.exit(1);
    }

    totalBytesRead += bytesRead;

}
```

C

```c
despla = 0;

while(despla < bufferSize)
{
    n = read(newsockfd, buffer + despla, bufferSize - despla);

    if (n < 0) error("ERROR reading from socket");

    despla += n;
    printf("Tamaño de desplazamiento: %ld\n", despla);
}
```

Luego, para solucionar el problema de integridad de datos decidimos utilizar un checksum. En particular optamos por MD5 el cual es un mecanismo criptográfico que recibe una cadena de bytes y retorna un hash de 128 bits.

La implementación se encuentra en los siguientes archivos:

- **Java:**
  - Script del cliente: *2c/java/Client.java*
  - Script del servidor: *2c/java/Server.java*
  - Para compilar ejecutar: *javac <archivo.java>*
- **C:**
  - Script del cliente: *2c/c/client.c*
  - Script del servidor: *2c/c/server.c*
  - Ejecutar el script *2c/requirement.sh* para instalar las dependencias.
  - Para compilar ejecutar: *gcc <archivo.c> -o <ejecutable> -lssl -lcrypto*

Explicación del código

Realizamos el siguiente protocolo de comunicación:

- Primero se envían/reciben 4 bytes del mensaje cuyo contenido es el tamaño del mensaje.
- Segundo se envían/reciben 16 bytes cuyo contenido es el hash del checksum generado por MD5.
- Tercero se envía/recibe el contenido del mensaje.

Para el envío del tamaño, checksum y contenido del mensaje hicimos 3 llamadas distintas a la función **write(),** aunque también se podría hacer en una sola llamada utilizando un buffer de bytes y calculando el desplazamiento mediante el protocolo propuesto.

El cliente escribe el mensaje (en los algoritmos cargamos un buffer con carácter mediante un loop para no tener que ingresar el mensaje por teclado) y antes de enviarlo se calcula su tamaño y el checksum. Particularmente para el cálculo del checksum en C usamos la librería *md5 de openssl*. Mientras que para Java se utilizó la librería MessageDigest del paquete java.security.
El servidor recibe primero un mensaje de 4 bytes que indica el tamaño del mensaje, después un mensaje de 16 bytes que contiene el checksum del mensaje y por último recibe el contenido mensaje. Una vez recibido el mensaje completo el servidor calcula el checksum del mensaje recibido y lo compara con el checksum recibido previamente.

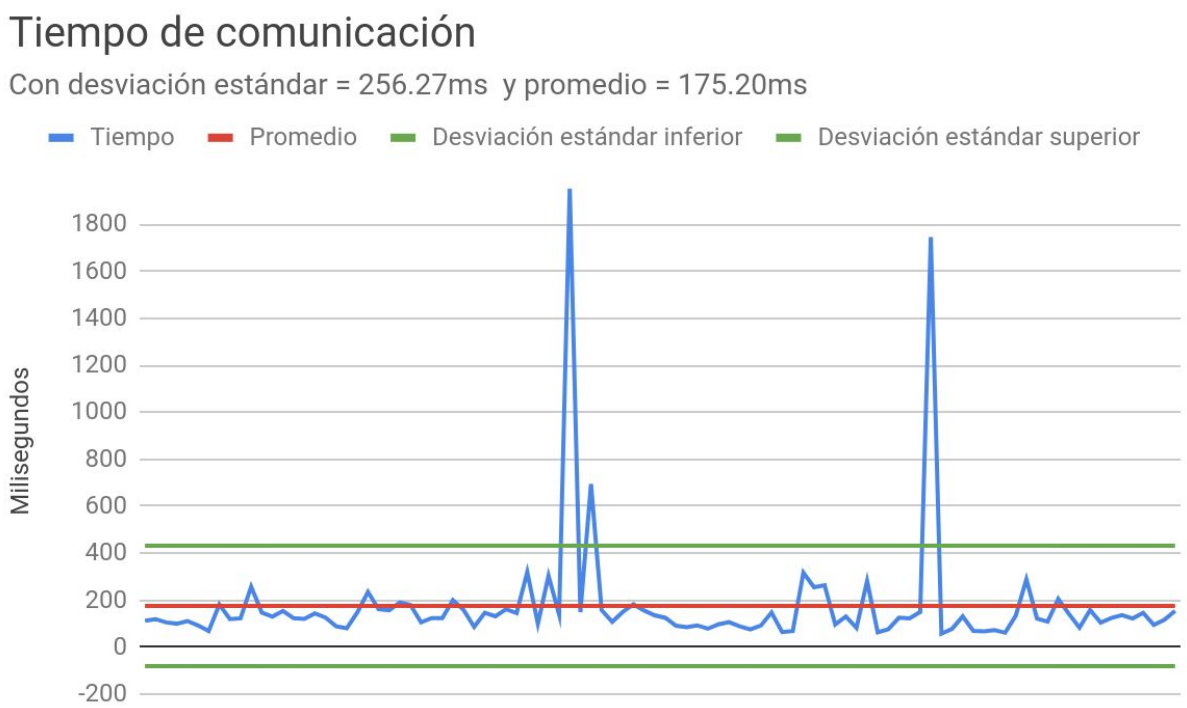
Inciso D

Para calcular el tiempo de una comunicación hicimos la siguiente simulación:

El cliente manda un mensaje, en donde previamente comienza a contar el tiempo y se queda esperando una respuesta del servidor.
Desde el lado del servidor se ejecuta un **sleep** de 1 segundo que simula algún procesamiento de datos que haría el servidor en un escenario real y al finalizar envía un mensaje al cliente.
En el tiempo transcurrido ocurrieron 2 comunicaciones: El envío del mensaje del cliente y el envío de respuesta del servidor; por lo tanto si dividimos ese tiempo calculado por 2 entonces nos daría el tiempo aproximado de una comunicación, es decir una estimación.
Se llega a la siguiente fórmula:

$$Tiempo\ de\ comunicación = \frac{tiempo\ total\ calculado - tiempo\ de\ procesamiento\ del\ servidor}{2}$$

Para calcular el promedio y la desviación estándar armamos un script *2d/metrics.sh* que ejecuta 100 comunicaciones y almacena los tiempos calculados en el archivo *2d/metrics.csv*.
Con dichos datos armamos la siguiente gráfica en donde se puede apreciar el promedio y la desviación estándar:



En la gráfica se puede ver que una comunicación promedio aproximadamente tarda 175 milisegundos con una desviación estándar de 256 milisegundos aproximadamente.

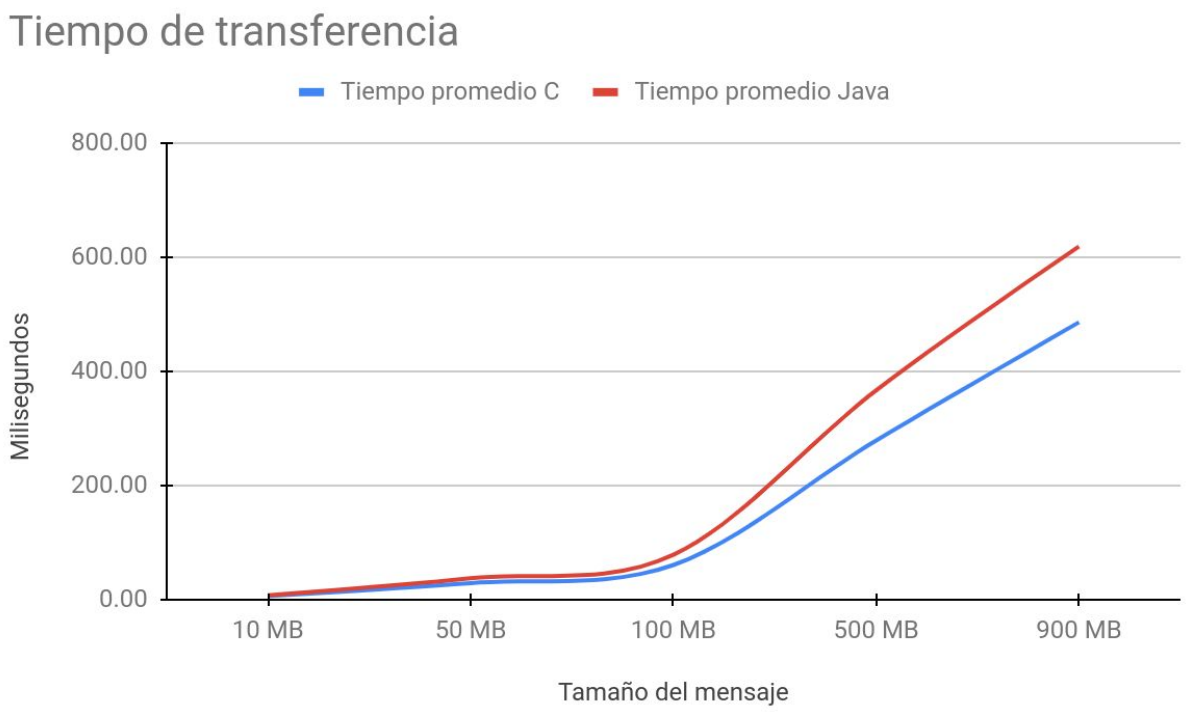
La implementación se encuentra en los siguientes archivos:

- Script del cliente: *2d/client.c*
- Script del servidor: *2d/server.c*
- Script para tomar los tiempos: *2d/metrics.sh*

Adicional

Cómo ejercicio adicional decidimos agregarle el parámetro de tamaño a la comunicación y medir cuánto tarda enviar cierto tamaño de mensaje. Acá hay que aclarar que para enviar una gran cantidad de datos probablemente se necesiten muchas comunicaciones como vimos en el ejercicio **2b**, por lo cual no se estaría tomando el tiempo de 1 comunicación como en el ejercicio **2d**, si no de las tantas comunicaciones que lleve enviar un tamaño grande de mensaje.

Lo implementamos tanto en Java como en C para después comparar los tiempos de cada lenguaje. Los scripts son simples, es una extensión del ejercicio **2c** en donde se envía el tamaño, checksum y el contenido del mensaje con la particularidad de que antes de enviar los 3 writes se comienza a tomar el tiempo y una vez que el servidor procesó la gran cantidad de datos que le fueron enviados envía una respuesta al cliente y se deja de tomar el tiempo. Luego se armó un shell script en dónde se enviaron mensajes 1MB hasta 900MB, este proceso se repitió 100 veces. Los tiempos se guardaron en una hoja de plantilla y se graficaron los promedios de la duración de la transferencia por tamaño de mensaje:



La implementación se encuentra en los siguientes archivos:

- C: Script del cliente: *2d/adicional/client.c*
- C: Script del servidor: *2d/adicional/server.c*
- Java: Script del cliente: *2d/adicional/client.java*
- Java: Script del servidor: *2d/adicional/server.java*
- Shell script para tomar los tiempos: *2d/adicional/metrics.sh [-c -java]*
  No se recomienda correr este script a menos que se quiera probar el funcionamiento ya que envía gran cantidad de datos muchas veces y puede llegar a tardar mucho.

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Ejercicio 3

En C para leer por teclado se usa la función **fgets** la cual guarda el mensaje escrito en un buffer de caracteres mediante un puntero y la función **write** que utilizamos para enviar datos a través del socket recibe un puntero y una cantidad de bytes a escribir. Por lo tanto podríamos usar el mismo puntero para recibir datos a través de la entrada estándar y enviar datos a través del socket como se ve en el siguiente código:

```c
char buffer[256];

// Captura la entrada estándar en la variable buffer
fgets(buffer, 255, stdin);

// Envía un mensaje a través del socket
n = write(sockfd, buffer, strlen(buffer));
```

Esto es una gran ventaja ya que podemos alocar memoria sólo una vez, y utilizar esa misma variable para comunicar luego de leer del teclado. También nos provee una gran flexibilidad para el modelo cliente/servidor ya que podríamos implementar los pedidos del cliente fácilmente en 2 líneas: una línea que capture la entrada estándar (**fgets**) y otra para el envío a través del socket (**write**). *Por ejemplo si tendríamos un servidor FTP, podríamos pedir fácilmente la lista de archivos con sólo escribir el comando “ls” del lado del cliente.*

La desventaja es que el control pasa a ser responsabilidad del programador, por lo tanto hay que tener precaución en el manejo de memoria.

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Ejercicio 4

Sí, podríamos implementar un servidor de archivos utilizando sockets. Un caso conocido es FTP que funciona a través de TCP y está basado en la arquitectura Cliente/Servidor. La interfaz principal de nuestro sistema de archivos remoto sería la siguiente:

Orden	Acciones del Cliente	Acciones del Servidor
load: Subir un archivo al sistema de archivos remoto.	**1.** Enviar como string de 255 bytes el string “load <filename>” **2.** Enviar el tamaño del archivo en 4 bytes. **3.** Enviar un checksum del contenido y nombre del archivo en 16 bytes. **4.** Enviar el contenido del archivo en bytes. **5.** Recibir la respuesta del servidor.	**1.** Recibir la orden como string en un buffer de 255 bytes. **2.** Recibir el tamaño del archivo en un buffer de 4 bytes. **3.** Recibir el checksum en un buffer de 16 bytes. **4.** Recibir el contenido del archivo. **5.** Calcular el checksum del nombre y contenido del archivo, luego compararlo con el recibido. **6.** Si el checksum coincide → Enviar como respuesta un código de éxito de 4 bytes y guardar el archivo. **7.** Sino → Enviar como respuesta un código de error de 4 bytes y descartar el archivo.
download: Recuperar un archivo desde el sistema de archivos remoto.	**1.** Enviar como string de 255 bytes el string “download <filename>” **2.** Recibir el tamaño del archivo en un buffer de 4 bytes. **3.** Recibir el checksum en un buffer de 16 bytes. **4.** Recibir el contenido del archivo. **5.** Calcular el checksum del nombre y contenido del archivo, luego compararlo con el recibido. **6.** Si el checksum coincide → Guardar el archivo. **7.** Sino → Descartar el archivo.	**1.** Recibir la orden como string en un buffer de 255 bytes. **2.** Buscar el archivo por el nombre recibido. **3.** Enviar el tamaño del archivo en 4 bytes. **4.** Enviar un checksum del contenido y nombre del archivo en 16 bytes. **5.** Enviar el contenido del archivo en bytes.

Detalles de implementación

Los **clientes** enviarán órdenes en formato de string de acuerdo a la interfaz que fue declarada.

El **servidor** esperará en un loop infinito esperando a que le llegue una solicitud, como dicha solicitud va a ser un string, podrá identificar la solicitud a través de un **switch**. Luego hará la operación de acuerdo a la interfaz declarada y le retornará la respuesta al cliente.

* Para enviar el mensaje se puede implementar con la llamada **write**.
* Para recibir el mensaje se puede implementar con la llamada **read.**

Lopes Canadell, Ramiro - 15263/2
Milla, Andrés - 14934/6

Ejercicio 5

Un *servidor sin estado* consiste en no guardar información sobre el estado actual del cliente. En cambio un *servidor con estado* debe guardar el estado actual y la información de la sesión.
Como ventaja de los servidores con estado encontramos que son ideales para esquemas unicast ya que no se necesita enviar la información con el estado en cada paquete. Un ejemplo de estos servidores sería el típico servidor Web en donde se debe guardar la sesión del usuario.
Por otra parte los servidores sin estado son especiales para esquemas multicast, tienden a ser más desacoplados y ser más fáciles de mantener/diseñar ya que no hay que mantener un estado. Un ejemplo de estos servidores sería una API, en donde simplemente se piden requerimientos y el servidor responde sin guardar ningún estado.