sockets

1)Sockets en C:
Cada socket es representado por un file descriptor. Los pasos necesarios para establecer un socket en el lado del cliente son los siguientes:

- Crear un socket con la system call **socket()**
- Conectar el socket a la dirección del servidor mediante la system call **connect()**
- Para enviar y recibir datos hay varias maneras de hacer esto, pero la más simple es usar las system calls **read()** y **write().**

Los pasos necesarios para establecer un socket en el lado del servidor son los siguientes:
- Crear un socket con la system call **socket()**
- Enlazar el socket a una dirección utilizando la system call **bind().** Para un socket de servidor en Internet, una dirección consiste en un número de puerto en la máquina host.
- Escuchar las conexiones con la system call **listen()**
- Aceptar una conexión con la system call **accept()**. Esta llamada generalmente se bloquea hasta que un cliente se conecta con el servidor.
- Enviar y recibir datos con system calls **read()** y **write()**.

Sockets en java

El paquete **java.net** en la plataforma Java proporciona una clase, Socket,
que implementa un lado de una conexión bidireccional entre su programa Java y otro programa en la red. La clase Socket se encuentra en la parte superior de una implementación dependiente de la plataforma, ocultando los detalles de cualquier sistema en particular de su programa Java. Al usar la clase **java.net.Socket** en lugar de depender del código nativo, los programas Java pueden comunicarse a través de la red de manera independiente de la plataforma.

Además, **java.net** incluye la clase **ServerSocket,** que implementa un socket que los servidores pueden usar para escuchar y aceptar conexiones a clientes.
Las lecturas y escrituras al socket se resuelven utilizando input y output streams

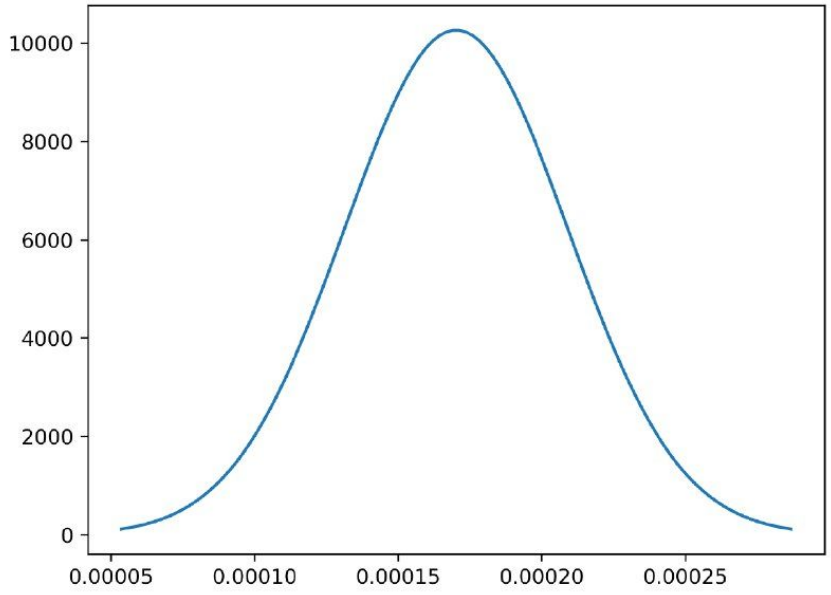
2a)en el modelo **cliente servidor** generalmente se consta de 3 pasos:
- **inicialización o handshake.**
- envío y recepción de datos.
- cierre de conexión.
En este caso solo hay envío y recepción de datos, una vez que termina esto, se termina la conexión, no hay un saludo entre el cliente y el servidor, ni un mensaje indicando el cierre de la misma, a su vez el servidor cierra y no se queda escuchando esperando otra petición, los servidores se quedan siempre a la espera de nuevas conexiones.

2b)El tamaño del **buffer** tanto del cliente como del servidor se fue modificando con diferentes valores para ver hasta qué cantidad de datos podía leer, se probó con los siguientes valores:
- 1000.
- 10000.
- 100000.
Pudo leer correctamente los **10000 caracteres**, pero cuando se probó con el valor **100000**, solo pudo leer **65482 caracteres**

2c)Se agregó en el **cliente** y en el **servidor** una función llamada **hash**, la cual genera un hash de los datos recibidos para después utilizarlo al realizar un **checksum**.
Del lado del cliente se genera un hash a partir de los datos del buffer, el cual junto con el tamaño de los datos va a ser enviado al servidor para validar si es que los datos llegaron correctamente.
Del lado del servidor, una vez recibido los datos del cliente, se calcula el hash de los datos recibidos, y se compara con el hash del cliente, también se compara el tamaño de los datos recibidos con el tamaño del cliente y se informa si concuerdan o difieren, aunque realmente no es necesario, dado que si llegaron bien los datos, se podría utilizar el hash solo y demostrar que llegaron todos los datos correctamente

2d)En este punto se busca calcular el tiempo que tarda una comunicación ida y vuelta entre el cliente y servidor utilizando un buffer de 4 bytes.
Para esto vamos a utilizar la función **dwalltime()**, que nos va a decir el tiempo de toda la comunicación y luego vamos a dividir por 2 para calcular el tiempo de cada una de las comunicaciones individuales.
Luego vamos a calcular la media que es la suma de todos los tiempos dividido la cantidad total de tiempos y una vez hecho esto vamos a calcular la desviación estándar.
Los tiempos tomados se encuentran en el archivo **buffer_time_data**

Media: 0.017021
Desviacion estandar: 3.88611e-05



3)Para enviar datos por un socket se necesita una variable **void ***, entonces, se puede enviar cualquier puntero a un tipo de datos en **C**, a través del mismo, y dado que para leer de teclado lo que se necesita es una variable de tipo **char *,** se podría utilizar la misma variable para las dos cosas . Pero hay que contar con ciertas precauciones si se busca usar la misma variable tanto para leer como para enviar datos, dado que lo que podría pasar es que ocurra alguna sobreescritura o pérdida de los mismos.

4) Se puede implementar y ya existen protocolos de manejo de archivos utilizando sockets, estos son **FTP** y **TFTP.**
Para mi implementacion me voy a basar en el protocolo **FTP:**

En **FTP** el cliente realiza tanto para el inicio y cierre de conexión un saludo de 3 vías (**3-way handshake**)
las instrucciones principales serían:
- LS: Lista los archivos.
- ADD: Agregar un archivo.
- GET: Baja un archivo.
- DELETE: Elimina un archivo.

El servidor, en principio, se encuentra a la espera de alguno de los comandos dichos anteriormente y si recibe otra cosa, indica que es un comando equivocado

LS - Listar archivos
El cliente envía el comando **LS** y luego, el servidor contesta con la lista de archivos que posee.
Se debe verificar el tamaño de los datos enviados desde el servidor para asegurarse que lleguen todos datos.

ADD - Enviar un archivo
El cliente envía el comando **ADD**, junto con el path relativo al archivo a subir. el cliente, se encarga verificar la existencia del archivo y, de existir, lo abre.
Luego comprueba el tamaño del archivo y lo envía al servidor. el cual solo puede datos de a paquetes de 1024 bytes, en caso de que el tamaño del archivo sea menor o igual a eso, se enviará en un solo paquete, y si es mayor, se tendrá que enviar a través de varios paquetes hasta completar el tamaño total. es Para realizar la comprobación archivo, se envía un checksum, el cual el servidor debe contrastar contra un **checksum** del archivo que le llegó.

GET - Descargar un archivo
El cliente envía el comando **GET** seguido del nombre del archivo que desea.
El servidor comprueba la existencia del archivo, de existir, primero abre el archivo y comprueba su tamaño, al igual que en el **ADD** solo se pueden enviar paquetes de a 1024 bytes, así que si el tamaño del archivo es menor o igual a eso, se enviará en un solo paquete, y si es mayor, se tendrá que enviar a través de varios paquetes hasta completar el tamaño total. Luego genera un **checksum** del archivo. Por

último, envía el archivo y el **checksum** por el socket para comprar si se recibió correctamente.

DELETE - Eliminar un archivo
El cliente envía el comando **DELETE** seguido del archivo que desea eliminar. El servidor comprueba que exista el archivo, y lo elimina.

5)Un **servidor con estado**, es aquel que recuerda el estado en el que había quedado el sistema en la anterior conexión de un cliente, sin la necesidad de mantener la conexión. Esto quiere decir, por ejemplo, un servidor con estado es aquel que puede mantener una página web.

Los **servidores sin estado**, por otro lado, no mantienen ninguna información de estado para el usuario. Cada solicitud es completamente independiente de la solicitud anterior. La arquitectura sin estado más común que usa **HTTP** es **REST (REpresentational State Transfer)** que se usa para diseñar API web. Los servidores sin estado aún pueden identificar al usuario si la solicitud al servicio incluye una identificación de usuario única que se asignó anteriormente al usuario. Pero ese ID de usuario deberá pasarse en cada solicitud, a diferencia del caso de los servidores con estado que mantienen este ID de usuario en la sesión y los datos de la solicitud no necesariamente deben contener este ID.