# Programación Distribuida y tiempo real


### Alumnos:

- Felipe Dioguardi - 16211/4
- Julian Marques de Abrantes - 15966/0


# PRACTICA 1

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

   ▼ Sockets en C

   - Los sockets se representan mediante file descriptors, que se obtienen llamando a la función `int socket(int __domain, int __type, int __protocol)`. Esta también permite definir el dominio (`AF_UNIX`, `AF_INET`, `AF_INET6`), tipo (`SOCK_STREAM`, `SOCK_DGRAM`), y el protocolo a utilizar (TCP, UDP).
   - Se utiliza un único buffer para la recepción y el envío de datos.
   - Se utiliza la system call `int bind(int __fd, const struct sockaddr * __addr, socklen_t __len)` para vincular una dirección con un socket.
   - Se utiliza la system call `int listen(int __fd, int __n)` para indicar al proceso que escuche las conexiones del socket con file descriptor `__fd`, con un máximo de `__n` conexiones en cola.
   - Se utiliza la system call `int connect(ind __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len)` para abrir una conexión en el socked con file descriptor `__fd` con el servidor escuchando en `__addr`.

   ▼ Sockets en Java

   - Se utilizan clases para la representación de sockets
   - Existen objetos especializados para el manejo de streams, envío de datos, y recepción de datos.
   - La elección de la capa de red y de transporte se realiza utilizando la herencia de clases. Por ejemplo: `URL`, `URLConnection`, `Socket`, y

`ServerSocket` son clases que utilizan el protocolo TCP y por otro lado, `InetAddress`, `DatagramSocket` y `DatagramPacket` utilizan UDP.

- Se instancia la clase `ServerSocket` para realizar el bind entre el socket y la dirección.
- Para la recepción de datos no se hace `listen` de forma explicita.
- El cliente se conecta implícitamente con el servidor.
- Implementación mas abstracta para el programador.

En cuanto a sus similitudes:

- Es posible implementar programas que se comuniquen entre sí mediante sockets.
- Es posible implementar programas que sigan el esquema cliente/servidor.
- Utilizan los mismos nombres para varias funciones claves como `accept`, `read`, `write`, entre otras.
- Permiten cambiar el número de conexiones a encolar antes de empezar a rechazar peticiones entrantes.
- Proponen una manera sencilla de establecer conexiones TCP y UDP.
- Hacen que el programador interactúe más o menos directamente con los sockets.

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?**

      Porque los ejemplos solo muestran una comunicación simple entre 2 sockets, que no presenta ninguna de las etapas necesarias para que sea considerado un modelo cliente/servidor:

      - La comunicación no es inicializada. Una vez el servidor está escuchando, el cliente envía la información.
      - La información intercambiada no puede considerarse una petición, sino simplemente un envío.
      - La comunicación no es propiamente finalizada, ambos asumen que el intercambio termina luego del primer mensaje.

      Además, carece de algunas características típicas de estos modelos:

- El servidor no es capaz de responder a varios clientes que se conectan a la vez. Responde al primero en conectarse y el resto fallan.
- El servidor finaliza la conexión luego de recibir un mensaje, en vez de mantenerse activo escuchando mas peticiones del mismo cliente o de clientes distintos.
- El servidor no tiene un protocolo para la solicitud de recursos.
- El servidor no se mantiene activo ante la ocurrencia de errores.

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 $10^3$, $10^4$, $10^5$, $10^6$ 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).**

Modificamos los programas en C para que un proceso envíe un mensaje con un tamaño fijo y el otro lo reciba, imprimiendo la cantidad de bytes que le lleguen. Se ve que los mensajes que no superan los 65482 bytes no generan problemas. Sin embargo, los que superan ese límite son enviados por partes. Como el "Server" hace una única invocación a la función `read()`, solo lee la primer parte del mensaje, perdiendo todo el resto.

También notamos que es posible que las partes perdidas de un mensaje anterior lleguen con un delay suficiente como para ser leídas en una ejecución próxima de `read()`.


Resultados del envió de datos

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.**

   En este caso, agregamos al código un loop `do ... while()` que se asegure de ejecutar la llamada a `read()` todas las veces que sean necesarias para que todas las partes del mensaje sean concatenadas en el buffer. Para lograrlo usamos una variable `offset` que representa la cantidad total de bytes leídos de cada mensaje (y, por ende, la posición del buffer en la que se debe escribir la siguiente parte del mensaje). Sabiendo de antemano el tamaño de los mensajes a recibir, solo tuvimos que leer desde el socket hasta que `offset == tamaño`.

   Además, agregamos luego de la lectura un loop `for` para asegurar que los datos leídos son correctos.

```
do {
  bytes_read = read(newsockfd, &buffer[offset], msg_len - offset);

  if (bytes_read < 0) {
    error("Server:: ERROR reading from socket");
  }
  offset += bytes_read;

} while (offset != msg_len);

for (int j = 0; j < msg_len; j++) {
  if (buffer[j] != '$') {
    error("Server:: ERROR wrong data received");
  }
}
```

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 utilizamos una función `double dwalltime()` que nos permite obtener un timestamp con cada invocación. Del lado del cliente, guardamos esa marca antes de realizar el envío del mensaje y después de recibir la confirmación del servidor. Restamos la segunda con la primera, y obtuvimos el tiempo que ambas comunicaciones tardaron en completarse. Dividimos este número por 2, para obtener un valor aproximado al tiempo que tardó cada comunicación individual, y

repetimos este proceso para todas las comunicaciones realizadas, imprimiendo los valores en pantalla.

Ejecutando la siguiente instrucción, guardamos los valores en un archivo que luego pudiéramos abrir con Excel, para calcular la media y la desviación estándar, y graficar los resultados.

```
$ ./client.out localhost 3500 > archivo_de_tiempos
```

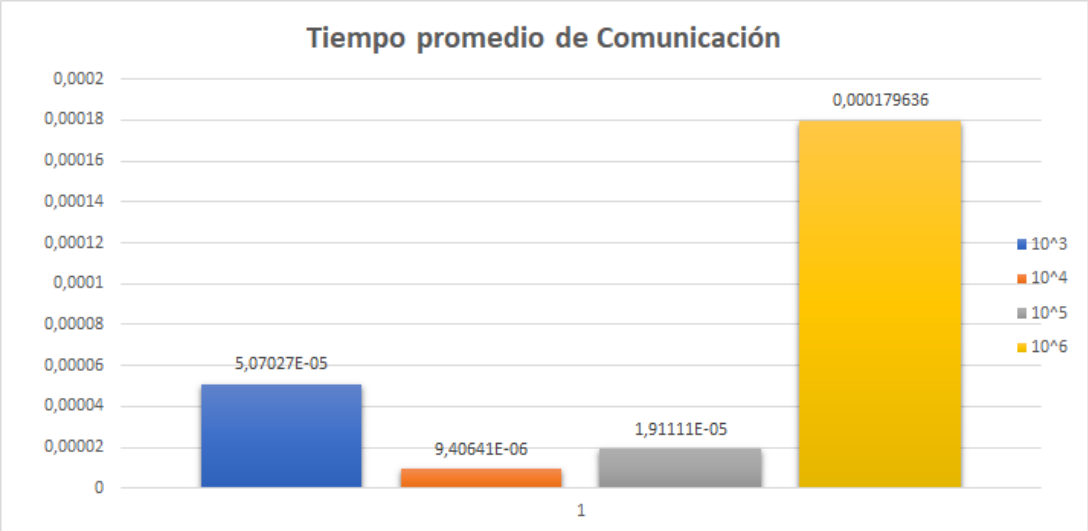


Gráfico con los tiempos promedios de comunicación del algoritmo en C con diferentes tamaños de mensajes.


Gráfico con la desviación estándar de los tiempos en base al algoritmo en C con diferente tamaño de mensajes.

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?**

   Porque el mensaje escrito por teclado puede ser guardado en un buffer mediante la función `char fgets(...)`, y el envío por socket se puede realizar llamando a la función `int write(...)`, que puede recibir el puntero al mismo buffer.

   Esto permite alocar en memoria una única variable que pueda utilizarse para ambos fines. En aplicaciones Cliente/Servidor esto permite implementar los pedidos al servidor fácilmente leyendo por teclado en una linea y enviando lo leído al socket en otra.

   Esta característica hace que sea muy sencillo implementar programas que se comuniquen a través de sockets, pero también implica que un error en el manejo de la memoria podría arrastrarse por todo el sistema.

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**

   Si, es posible de implementar un servidor de archivos remotos con sockets.

   Basándose en la arquitectura cliente/servidor es posible implementar un servidor básico en el que el cliente tiene la capacidad de enviar dos instrucciones para cargar y descargar un archivo. Mientras tanto, el servidor se encontraría en un loop en el que espera por la instrucción del cliente para enviar o recibir un archivo.

   Ambos procesos podrían mantener las dos conexiones necesarias utilizando un par de sockets cada uno.

   Sería lógico que este tipo de comunicaciones se realicen mediante el protocolo TCP, para asegurar la integridad de los datos compartidos.

   Así, la interfaz seria la siguiente:

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

   - Servidor con estado: un servidor que conserva todas las configuraciones de clientes que se conectan a él, incluso después de reiniciarlo. También mantiene información de sesiones anteriores. Es muy usado, por ejemplo, para servidores web que recuerdan la información de sus usuarios, pero

también para una gran cantidad de páginas HTTP que pretenden brindar un servicio personalizado.

- Servidor sin estado: un servidor que no retiene información de la sesión o de estado acerca de los clientes. Este tipo de servidores requieren que el cliente envíe alguna forma de identificación con cada solicitud. Los sistemas implementados con este tipo de servidor suelen ser más escalables, pues eliminan la necesidad de migrar/actualizar los datos de sesión.