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:
      - En un modelo C/S completo, la comunicación se **inicializa** a través de un proceso de "handshake" o saludo, que puede incluir varios pasos. En los ejemplos dados, la comunicación no se inicializa, simplemente el servidor escucha y el cliente envía la información.
      - En un modelo C/S, el cliente envía una **petición** al servidor, que luego procesa la petición y envía una respuesta. En los ejemplos dados, se hace un envío de datos sin estructura, lo cual difícilmente constituye una petición.
      - Tampoco se **finaliza** protocolarmente la comunicación, tanto el cliente como el servidor asumen que el envío de información concluye después del primer mensaje. En una arquitectura C/S la comunicación se finaliza de manera ordenada, liberando recursos y cerrando conexiones.

      Además no cumplen con algunas de las características del modelo c/s:
      - El servidor no es *multi-tenant*, es decir, no es capaz de responder a varios clientes que se conecten a la vez. Le da respuesta al primero que se conecte y el resto fallará.
      - El servidor no posee un protocolo mediante el cual se soliciten los recursos.

- El servidor no se mantiene activo ante la ocurrencia de errores, si bien se informa de los mismos, no se rescatan ni se vuelve a intentar.

Considerando los sockets stream cuando utilizan TCP en la capa subyacente de transporte. Se plantea la duda - se podría decir que hay una implementación del modelo C/S en esta capa inferior de la arquitectura de manera más abstracta y de bajo nivel? Se analiza lo siguiente: TCP tiene una implementación que **inicializa** la comunicación con un three-way-handshake utilizando SYN, SYN-ACK, ACK para concretar una conexión; la parte de **petición y respuesta** tiene menos sentido en este caso, se reciben flujos de datos y se recepcionan con ACK; se **finaliza** la comunicación de forma ordenada con el four-way-handshake con FIN, ACK, FIN, ACK. No hay una interfaz 'de usuario' pero sí hay claros protocolos con respecto a lo que se puede pedir, como y en qué orden, y que se espera a cambio. Pero al mismo tiempo, la lógica específica para manejar peticiones y respuestas que se espera de una arquitectura de C/S generalmente se implementa en una capa de aplicación superior.

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).

NOTA: Para realizar los experimentos, hicimos un scripts de bash "run.sh" que se encuentra en la carpeta de cada punto, que compila y corre los ejercicios y se modifica ligeramente para cada sección. Este script compila y ejecuta un cliente y el servidor en C, probando diferentes tamaños de búfer - envía como argumento tanto al cliente como al servidor el tamaño del buffer que debe alocar, corriendo un bucle por número de exponente. El servidor se ejecuta en segundo plano usando &, y el cliente se conecta a él después de una breve pausa (sleep 2) para asegurarse de que el servidor esté listo.

Estos ejercicios se corrieron en nuestras máquinas las cuales tienen las siguientes especificaciones:
Juan: Pop OS 22.10, Lenovo B50-80, 4 CPUs, 16 GB de RAM, Intel i7
Inti: Debian 11, Lenovo T420 Thinkpad, 4 CPUs, 12G de RAM, Intel i5

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. Luego se imprime en el cliente los bytes enviados y en el servidor los bytes recibidos, información que se obtiene de los valores de retorno de las instrucciones write y read, respectivamente. Para evitar el error "Address

already in use" que puede ocurrir al conectarnos seguidamente al mismo puerto, habilitamos la opción de reutilizar el mismo puerto en el socket del lado del servidor con la función setsockopt() con SO_REUSEADDR, y la variable opt seteada en 1.

Vemos que en casos mayores a 10^5 se la cantidad de bytes recibidos en el servidor difiere con respecto a los enviados:

```
$ ./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
$
```

Si utilizamos el script **2b/run.sh**, ejecutar fácilmente casos, pasando un argumento que indica el exponente máximo en la secuencia de potencias de 10 con las que se ejecuta la comunicación. En el siguiente caso se pasó el argumento 12:

```
Buffer size 10^6=1000000 bytes
Sending 1000000 bytes
Bytes recived: 65482

Buffer size 10^7=10000000 bytes
Sending 10000000 bytes
Bytes recived: 32741

Buffer size 10^8=100000000 bytes
Sending 100000000 bytes
Bytes recived: 32741

Buffer size 10^9=1000000000 bytes
Sending 1000000000 bytes
Bytes recived: 32741

Buffer size 10^10=10000000000 bytes
Sending 1410065408 bytes
Bytes recived: 81610

Buffer size 10^11=100000000000 bytes
Sending 1215752192 bytes
Bytes recived: 81610

Buffer size 10^12=1000000000000 bytes
./run.sh: line 6: 351428 Segmentation fault      (core dumped) ./server 4000 $size
./run.sh: line 6: 351434 Segmentation fault      (core dumped) ./client localhost 4000 $size
```

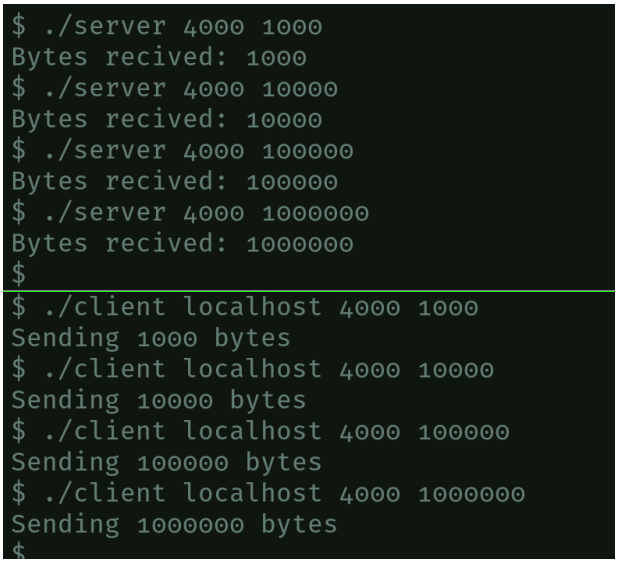
```
Buffer size 10^4=10000 bytes
Sending 10000 bytes
Bytes recived: 10000

Buffer size 10^5=100000 bytes
Sending 100000 bytes
Bytes recived: 32741

Buffer size 10^6=1000000 bytes
Sending 1000000 bytes
Bytes recived: 32741

Buffer size 10^7=10000000 bytes
./run.sh: line 6: 679248 Segmentation fault      ./client localhost 4000 $size
```

Observamos que la cantidad de bytes recibidos no siempre es la misma. Es decir que no siempre se corta la comunicación en el mismo punto, vemos valores como 32741 (el más común), 65482 y 81610 bytes.
También vemos que a partir de 10^{10} se envían menos datos de los que debería, pero otra vez, la cantidad no es constante.
Por otra parte, a partir de 10^{12} se genera un error Segmentation fault, debido a que no es posible alocar la cantidad de memoria solicitada. Esta cantidad depende del uso de las regiones de memoria contiguos en la máquina, por lo que en otra computadora se veían en 10^{7}.

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.**

Para validar el contenido, inicialmente se implementó una funcionalidad que chequeaba que el contenido enviado era el mismo que se recibía. Esto se logra, como puede verse en el directorio **2c/verif-manual**, agregando en en el servidor un loop **do while**, que repetirá hasta haber recibido todos los datos (cuyo tamaño sabemos de antemano, ya que se trata de una secuencia de potencias de 10). 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.

De esta manera, se puede ver como el servidor imprime que recibió la cantidad de bytes correctamente:



Como nos pareció que esta forma de validar la información estaba muy trivial, además de atada a saber la el tamaño del envío de datos y su contenido, en una segunda aproximación se implementan dos versiones de una función auxiliar 'calculate_checksum' para hacer el chequeo correcto de la recepción de los datos utilizando diferentes algoritmos, se se encuentran en los directorios **2c/md5** y **2c/sha512**.

El primero es el algoritmo MD5 de la librería de openssl <openssl/md5.h>. Se crean dos archivos adicionales checksum.h donde se declara la firma de la función, y el archivo checksum.c donde se define la implementación de la misma.

La función recibe un arreglo de datos **const char *data**, el tamaño del mismo **size_t len**, y tiene un parámetro extra **unsigned char *checksum** de al menos **MD5_DIGEST_LENGTH** bytes donde se almacena el resultado de la comprobación y es el valor de retorno de la función.

La función tiene una variable de contexto md5 de tipo **MD5_CTX** de la librería de openssl, que se inicializa, y luego se utiliza para calcular la suma de comprobación y finalmente guarda el resultado en el parámetro **checksum**.

Hay información sobre la librería de openssl md5 en MD5

Y las funciones utilizadas MD5_Init, MD5_Update, MD5_Final

Aquí se visualiza un screenshot donde el cliente imprime el checksum de los datos antes del envío, y el servidor imprime el checksum al recibir los datos:

```
❯ ./run.sh
Buffer size 1000 bytes
Checksum before sending: 7644672d049290f0390d9c993c7d343d
Sending 1000 bytes
Checksum after receiving: 7644672d049290f0390d9c993c7d343d
Bytes recived: 1000

Buffer size 10000 bytes
Checksum before sending: 0f53217fc7c8e7f89e8a8558e64a7083
Sending 10000 bytes
Checksum after receiving: 0f53217fc7c8e7f89e8a8558e64a7083
Bytes recived: 10000

Buffer size 100000 bytes
Checksum before sending: 5793f7e3037448b250ae716b43ece2c2
Sending 100000 bytes
Checksum after receiving: 5793f7e3037448b250ae716b43ece2c2
Bytes recived: 100000

Buffer size 1000000 bytes
Checksum before sending: 48fcdb8b87ce8ef779774199a856091d
Sending 1000000 bytes
Checksum after receiving: 48fcdb8b87ce8ef779774199a856091d
Bytes recived: 1000000
```

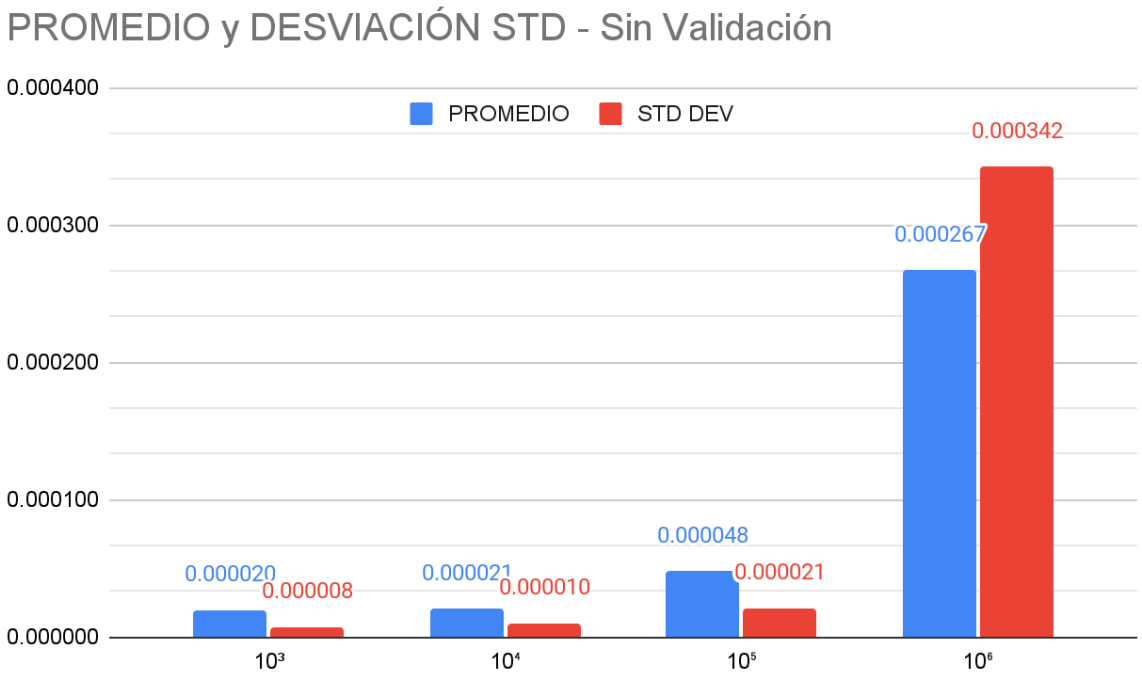
Como la funcionalidad MD5 está deprecada, se hace una segunda una versión con SHA512 cuya implementación es muy similar a la de MD5, las diferencias son que se usa alternativamente la librería de openssl <openssl/sha.h>, el tipo de dato **SHA512_CTX** y la constante **SHA512_DIGEST_LENGTH**, generando un checksum más largo y más seguro. Se puede leer sobre esta funcionalidad en SHA512. Así se visualiza la validación en la terminal:

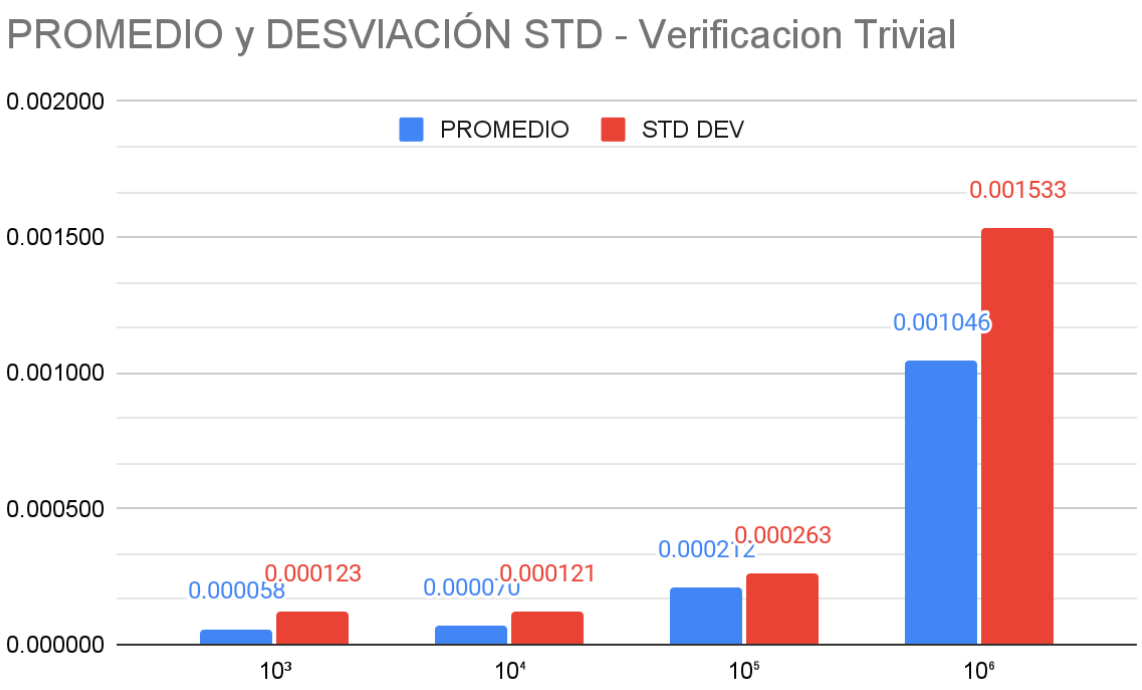
```
❯ ./run.sh
Buffer size 1000 bytes
Checksum before sending: 329c52ac62d1fe731151f2b895a00475445ef74f50b979c6f7bb7cae349328c1d4cb4f7261a0ab43f936a24b000651d4a82
4fcdd577f211aef8f806b16afe8af
Sending 1000 bytes
Checksum after receiving: 329c52ac62d1fe731151f2b895a00475445ef74f50b979c6f7bb7cae349328c1d4cb4f7261a0ab43f936a24b000651d4a8
24fcdd577f211aef8f806b16afe8af
Bytes recived: 1000

Buffer size 10000 bytes
Checksum before sending: 5b40cbafad64f231f8396e38af5aa754eae8ce61beca208f13e4145abedee8493869f4155148928bd15ed66116966497d79
bbc64307bb09cc42388b958a11a85
Sending 10000 bytes
Checksum after receiving: 5b40cbafad64f231f8396e38af5aa754eae8ce61beca208f13e4145abedee8493869f4155148928bd15ed66116966497d7
9bbc64307bb09cc42388b958a11a85
Bytes recived: 10000

Buffer size 100000 bytes
Checksum before sending: 50fb8b5b66d3d2391d5d3ca0f65972eb746d391c0cc67bc56fac62b46bfa57f842b239ca797e0003ff571ee1a4bf9143646
41c00bd219f1bb914199970ea427a
Sending 100000 bytes
Checksum after receiving: 50fb8b5b66d3d2391d5d3ca0f65972eb746d391c0cc67bc56fac62b46bfa57f842b239ca797e0003ff571ee1a4bf914364
641c00bd219f1bb914199970ea427a
Bytes recived: 100000

Buffer size 1000000 bytes
Checksum before sending: 990fed5cd10a549977ef6c9e58019a467f6c7aadffb9a6d22b2d060e6989a06d5beb473ebc217f3d553e16bf482efdc4dd9
1870e7943723fdc387c2e9fa3a4b8
Sending 1000000 bytes
Checksum after receiving: 990fed5cd10a549977ef6c9e58019a467f6c7aadffb9a6d22b2d060e6989a06d5beb473ebc217f3d553e16bf482efdc4dd
91870e7943723fdc387c2e9fa3a4b8
Bytes recived: 1000000
```

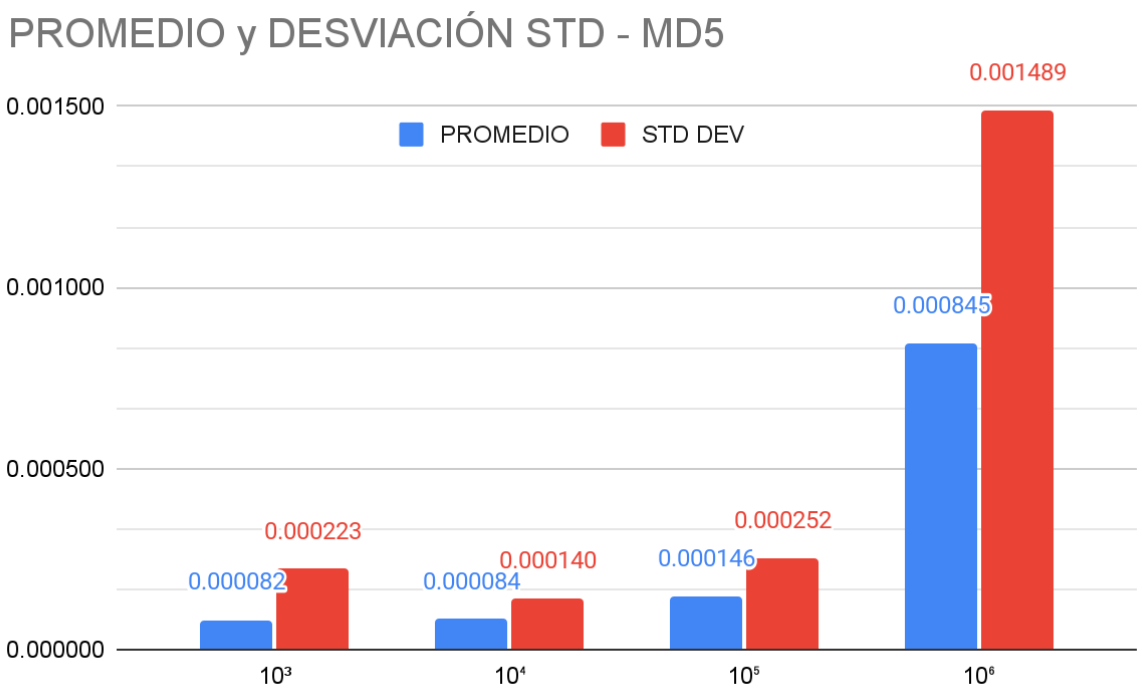
d. Grafique el promedio y la desviación estándar de los tiempos de comunicaciones de cada comunicación. Explique el experimento realizado para calcular el tiempo de comunicaciones.

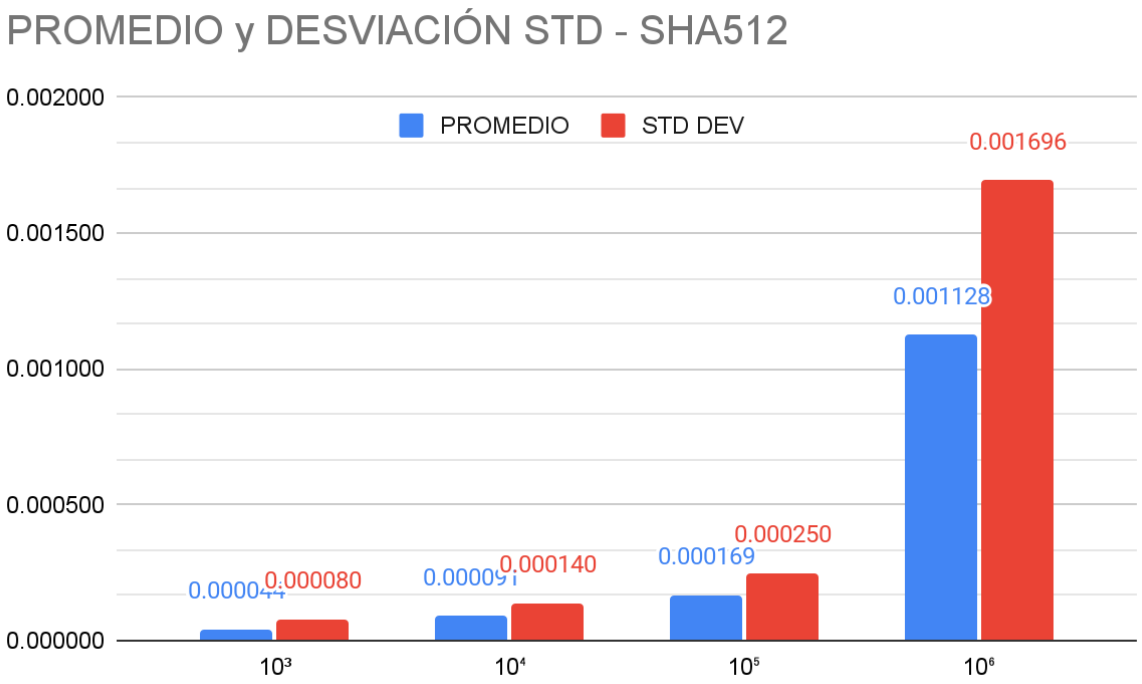
Para calcular el tiempo de cada comunicación se utilizaron dos variables de tipo clock_t llamando a la función clock() (de la librería <time.h>), inmediatamente antes de que el cliente envíe el mensaje, y después de de que el mismo reciba la confirmación del servidor.

Se restan estos dos tiempos y se dividen por dos, ya que se tiene en cuenta la ida y la vuelta del mensaje. En este caso el script de bash (**2d/run.sh**) permite ejecutar los programas 100 veces seguidas con 10^3, 10^4, 10^5 y 10^6 bytes de datos. Además modificamos el código para que imprima solamente los tiempos medidos seguido de una tabulación. Agregando un salto de línea (`echo ""`) obtenemos la salida en un formato que luego fue sencillo de importar en una planilla para realizar la estadística. Decidimos hacer las estadísticas para cada uno de los experimentos ejecutados previamente:

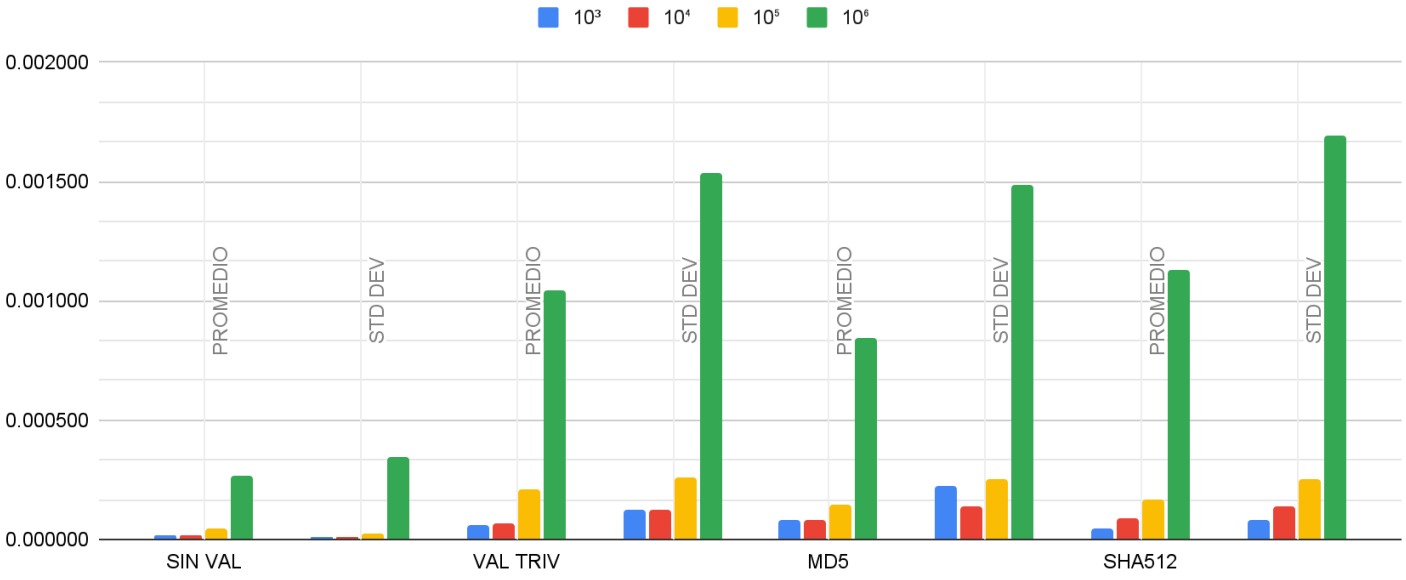



PROMEDIO y DESVIACIÓN STD - Verificacion Trivial
0.002000
0.001500
0.001000
0.000500
0.000000
PROMEDIO
STD DEV
0.000058
0.000123
0.000070
0.000121
0.000212
0.000263
0.001046
0.001533
10³
10⁴
10⁵
10⁶


PROMEDIO y DESVIACIÓN STD - MD5
0.001500
0.001000
0.000500
0.000000
PROMEDIO
STD DEV
0.000082
0.000223
0.000084
0.000140
0.000146
0.000252
0.000845
0.001489
10³
10⁴
10⁵
10⁶



Vemos que la validación agrega un overhead, como era de esperarse. También se puede apreciar una mayor varianza en los los experimentos con validación.



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 independientemente de las librerías o llamadas del SO que luego utilizan los datos allí almacenados.

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 una interfaz de usuario, que podría ser una línea de comandos donde se le presentan opciones como "Subir archivo", "Listar archivos" y "Descargar archivo" que hacen peticiones a una API en el servidor.

   Para manejar diferentes tipos de peticiones, el servidor podría implementar una estructura switch que, dependiendo del tipo de petición ("UPLOAD", "LIST", "DOWNLOAD"), ejecutaría diferentes bloques de código.

   UPLOAD. El cliente solicita la subida de un archivo. Primero le envía al servidor el nombre del archivo, luego envía los datos. El servidor crea en el fs un archivo con dicho nombre y dicho contenido. Podría devolver un error en caso de que el nombre ya existiese.

   DOWNLOAD. El cliente solicita bajar un archivo, enviando el nombre del mismo. El servidor recibe el nombre del archivo y ejecuta la lógica de búsqueda del archivo, enviando el contenido al cliente, o enviando un error en caso de que no exista.

   LIST. El cliente solicita una lista de archivos. El servidor genera una lista de los archivos existentes en el momento de la petición y se la envía al cliente.

   Cada proceso utilizará 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.

   RFC para FTP (con state diagrams e standard de implementación):
   RFC 959: File Transfer Protocol
   Información sobre FPT de Apache:
   Introduction to the FTP Protocol - Apache HTTP Server Version 2.4

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.

El mantenimiento de sesiones es crucial en el contexto de uso de autenticación y autorización, por ejemplo. El servidor debe mantener información sobre la sesión durante la conexión. Esto evita que el usuario tenga que volver a autenticarse con cada operación que realice, y pueda almacenar para la autorización, información sobre los permisos del usuario en la sesión.

Adicionalmente, el uso de state permite la implementación de monitoreo y auditoría y llevar adelante un seguimiento más detallado de los usos del mismo. También facilita el envío ya que no es necesario volver a enviar toda la información del lado del cliente en cada comunicación.

Sin embargo, esto genera un nivel alto de acoplamiento entre cliente y servidor, implementaciones muy complejas, y hace difícil la recuperación ante errores.

En un servidor de archivos distribuido se utiliza el estado para mantener información, por ejemplo, sobre autenticación, navegación en los directorios, descargas en proceso, conexiones, entre otras cosas.

Por otro lado, los servidores stateless se suelen considerar más robustos. Estos son fácilmente escalables, tienen tolerancia a fallos y alta disponibilidad. La implementación también es más simple y más segura. Esto también los hace más portátiles y mantenibles. Podemos tener varias instancias ya que sabemos que van a dar la misma respuesta, y esto también nos permite usar servicios de caché.

Ejemplos de usos de servers Stateless y Stateful:

Stateless - microservices, server web HTTP, UDP, DNS, SMPT
Stateful - implementacion con RPC, ejemplos TCP, FTP, Telnet

Ref:
Defining Stateful vs Stateless Web Services | Nordic APIs |
Stateless vs Stateful Servers (with examples) | by Ahmed Ossama | Medium