Practica 2

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

1)

Primera Prueba (verificación manual):

Se corre el socket usando el lenguaje c, con el cliente y el servidor en dos pc diferentes de la misma red. Los corremos usando el programa de 2c de la práctica 1 con la verificación manual, porque no tienen la librería libcrypto.so.3 para usar la verificación con checksum. Se debe tener en cuenta que esta verificación manual recorre el byte array que se envía como dato para verificar que es correcto (sabiendo de antemano lo que se envió).
Por esta razón, los tiempos son mayores en general en este código que en los otros. Por lo tanto, se deben comparar con los tiempos de verificación manual anteriores y no con los de otros métodos de verificación cuyos tiempos son menores.

Las computadoras están en la misma red local 163.10.54.0

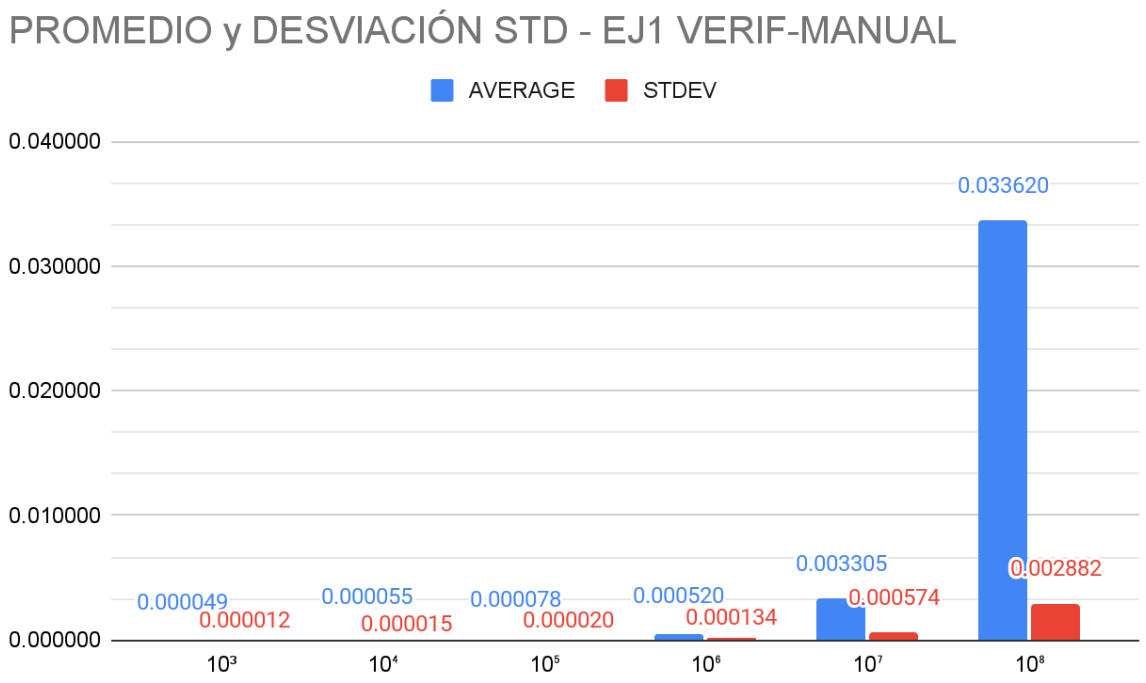
Como en la TP1 2d, corremos el programa 100 veces con datos de tamaños de 10^3, 10^4, 10^5, 10^6, 10^7 y 10^8 bytes. Utilizamos la siguiente planilla de cálculo para registrar los tiempos: de cada ejercicio:

https://docs.google.com/spreadsheets/d/1lnUBDRxzgXN_NVP-IsYNSaeTBOtIIQYsi-ZO7_PvWBE

Las PCs usadas para este ejercicio tienen las siguientes características:

PC CLIENTE:
IP 163.10.54.153
SO Ubuntu 20.04
CPU(s): 4
Nombre del modelo: Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz
RAM: 7,8G

PC SERVER:
IP: 163.10.54.146
SO Ubuntu 20.04
CPU(s): 4
Nombre del modelo: Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz
RAM: 7,8G



Segunda Prueba (checksum):

Luego, para correr otra prueba usando los checksum, creamos una pequeña LAN hogareña entre una notebook pop os (derivado de ubuntu) y una raspberry PI 4 corriendo raspbian (derivado de debian). La red es 192.168.0.0 y se tienen los IP 192.168.0.5 del notebook linux y 192.168.0.0.18. Usando scp se copiaron los archivos al pi, y se tomaron los tiempos de comunicación entre el cliente y el servidor.

Como en la TP1 2d, corremos el programa 100 veces con datos de tamaños de 10^3, 10^4, 10^5, 10^6 y 10^7 bytes.

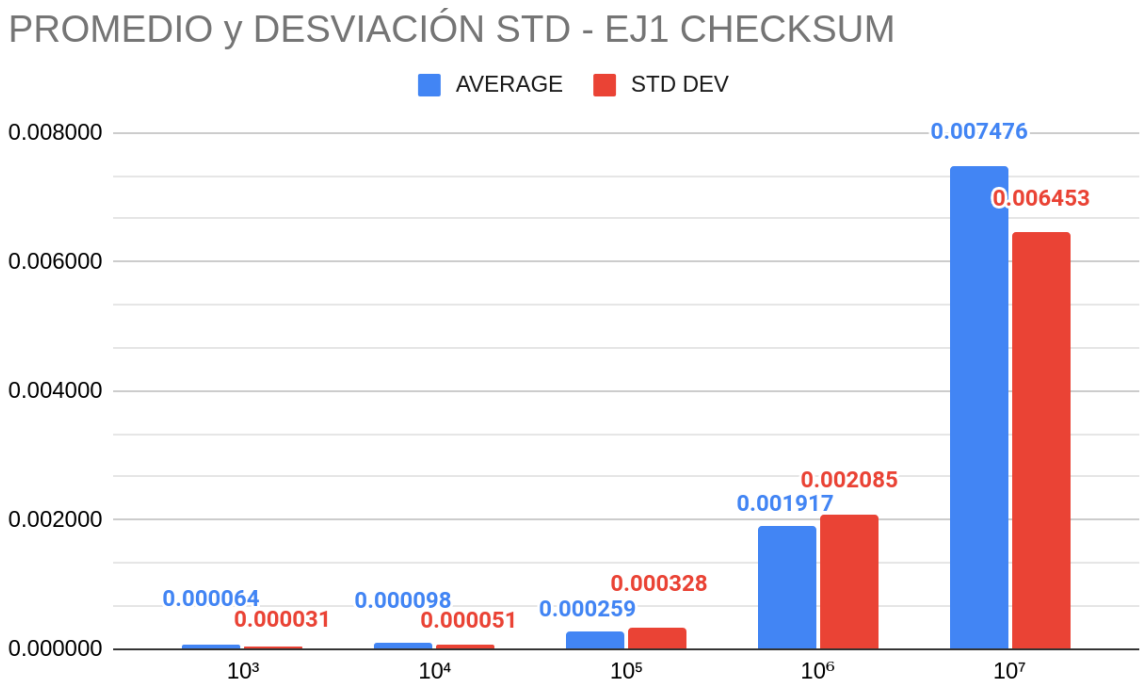
Como no usamos un while loop para verificar los datos, estos tiempos son menores que con la verificación manual.

PC CLIENT:
IP: 192.168.0.5
Lenovo Ideapad
SO Pop_OS! Ubuntu 22.04
CPU(s): 4
Nombre del modelo: Intel(R) Core(™) i7-6500U @ 2.5GHz
RAM: 11,4G

PC SERVER:
IP:192.168.0.18
Raspberry PI 4
SO Raspbian

Programación Distribuida y Tiempo Real
Facultad de Informática - Universidad Nacional de La Plata

CPU(s): 4
Nombre del modelo: ARM Cortex-A72 processor @ 1.5 GHz
RAM: 3,12G



Algunas conclusiones:

Nos enfrentamos con el problema de que no hay una sincronización que se pueda hacer desde dos SO diferentes para asegurar que los programas se corran de manera iterativa y debemos recurrir a usar otros métodos para poder asegurarnos de sincronizar el envío y recibimiento de datos.

Verificamos que para correr los programas correctamente desde diferentes máquinas debemos usar el método de un mecanismo de loop que permite al servidor recibir todos los datos de cada cliente en su totalidad antes de recibir datos de otro cliente, lo cual en la misma máquina no era necesario por las cualidades especiales de la interfaz de loopback (127.0.0.1).

La interfaz de loopback es una interfaz de red virtual que utiliza el software de red de una computadora para permitirle comunicarse consigo misma. La dirección IP asignada a la interfaz de loopback es generalmente 127.0.0.1 y es conocida como localhost. Cuando las comunicaciones se hacen a través de la interfaz de loopback el sistema operativo maneja todo el tráfico de red internamente a nivel de software, sin interactuar con el hardware de red.

En nuestro código anterior, usábamos scripts de shell para correr servidores y clientes de manera iterativa en la misma maquina, donde el servidor puede manejar un cliente a la

vez. Después de aceptar una conexión con accept(), el servidor maneja esa conexión, y una vez que se cierra, vuelve a accept() para esperar otra conexión. Sin embargo, el sistema operativo tiene un mecanismo de cola para manejar múltiples solicitudes de conexión entrantes, y listen(sockfd,5); configura el tamaño de esta cola. Al estar los procesos en la misma máquina, y usar la interfaz de loopback, todas las conexiones se manejan localmente, e incluso si varios clientes intentan conectarse al mismo tiempo, el sistema operativo puede poner en cola las conexiones entrantes y el servidor puede manejarlas una por una de manera eficiente.

Por otro lado, el tiempo de las conexiones es mayor cuando se corren el cliente y el servidor en máquinas diferentes. Cuando comparamos estos resultados con los que obtuvimos en el TP1 corriendo ambos programas de cliente y servidor en la misma computadora, se hace evidente la latencia agregada por la comunicación en la capa de red.

2)

Ya verificamos en el ejercicio 1, que para correr los programas desde diferentes máquinas debemos usar el método de loop que permite al servidor recibir todos los datos del cliente en su totalidad previo a verificar si los datos son correctos.

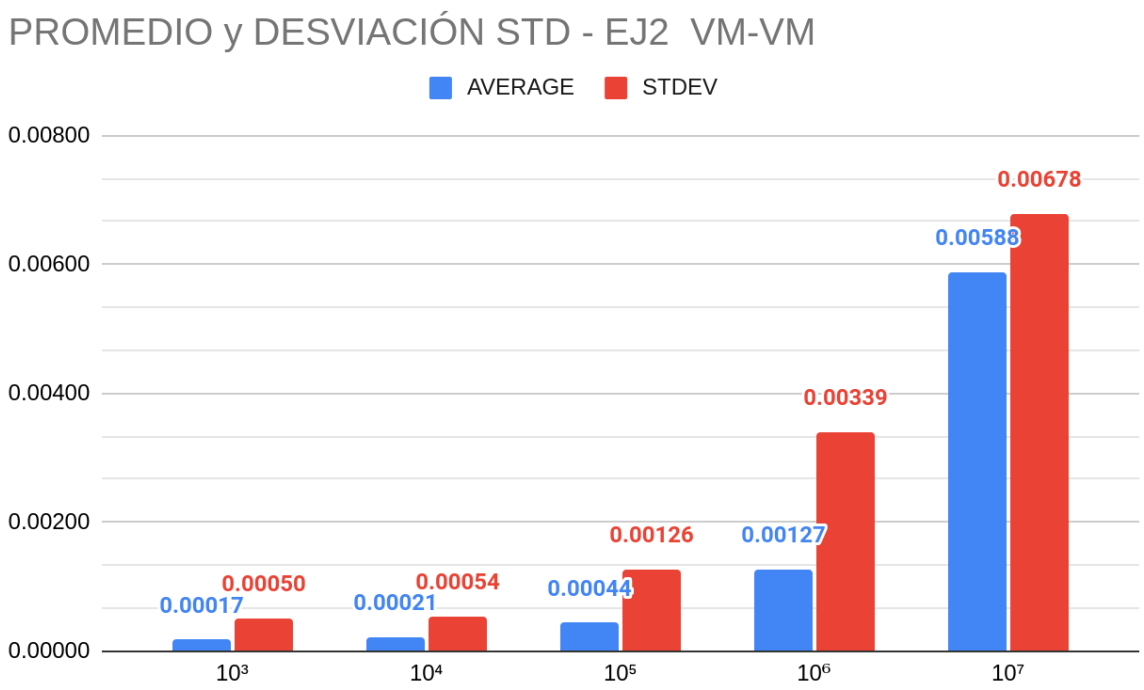
Al vagrantfile se le agrega ‘libssl-dev’ al comando `sudo apt-get install`, para poder acceder a las librerías de openssl que usamos para los checksums.

También se le quita la instalación del default-jdk ya que no estaremos usando Java en este ejercicio.

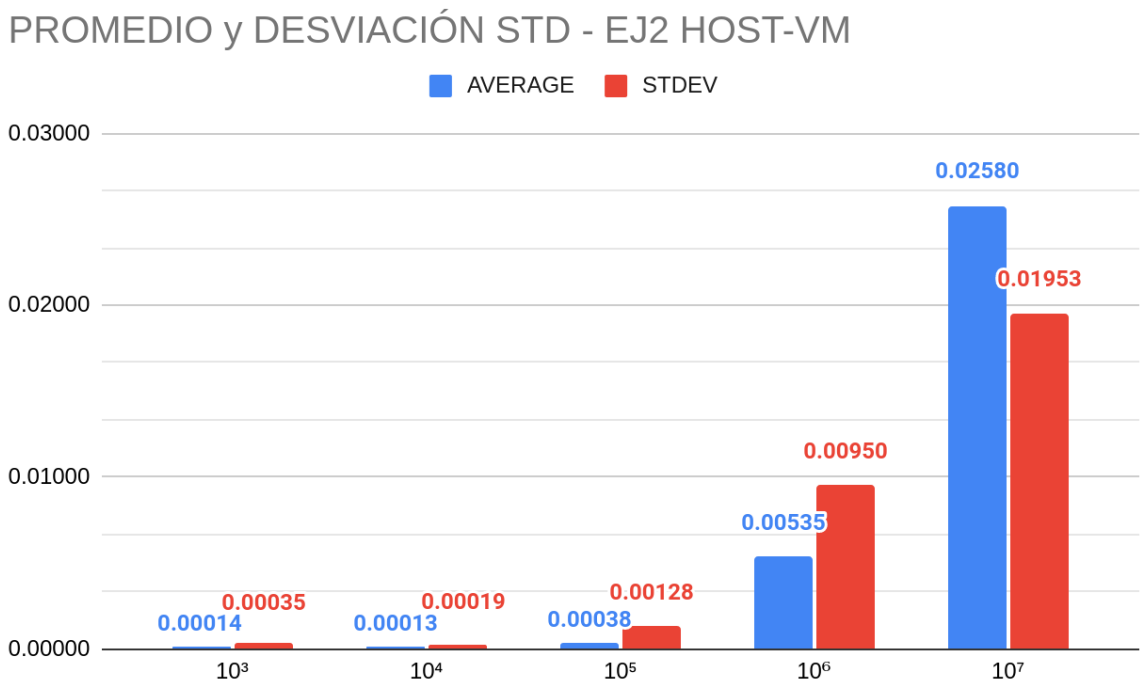
En la primera parte de este ejercicio se comunica un cliente y un servidor en dos máquinas vagrant, vm1 y vm2. Para entrar a cada una se usa el comando `vagrant ssh vm1` y `vagrant ssh vm2` respectivamente.
Se genera una IP estática desde el vagrantfile, lo cual permite automatizar los ejercicios.

En este ejercicio hicimos una serie de experimentos, cada integrante del grupo corrió los scripts necesarios para levantar las máquinas virtuales en su máquina personal y pudimos medir los tiempos tanto de vm a vm como de host a vm y vice versa,con 100 iteraciones de cada uno de tal manera de poder comparar estadísticamente los casos.

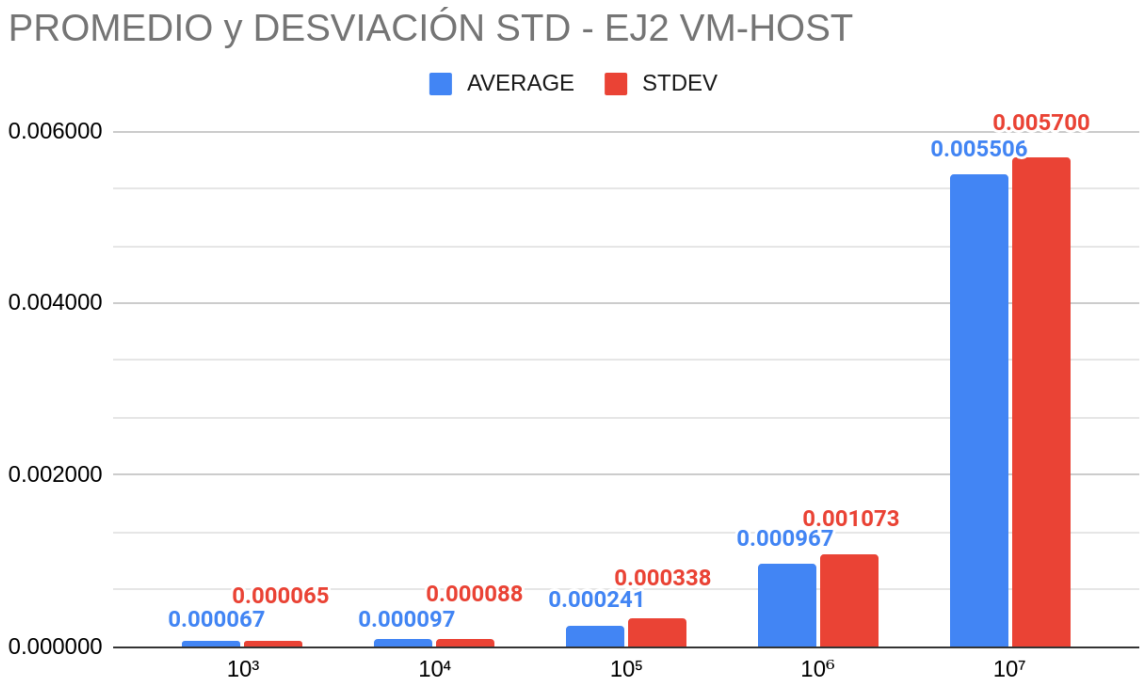
VM a VM:



Server Host, Client VM:



Server VM, Client Host:



3)

Considerando la aplicación práctica de esta entrega, este ejercicio parece estar dirigido a analizar los aspectos de sincronización y temporización de la comunicación cliente-servidor, en particular, analizar y comprobar si el read y write son sincrónicos o asincrónicos.
La referencia es esta filmina:

- En C
  - "Server" - "Client"

srvr: socket() - bind() - listen() - accept() - read() - write()
clnt: socket() - connect() - write() - read()

srvr: socket() - bind() - listen() - accept() - read() - write()
clnt: socket() - connect() - write() - read()

Asincrónico
Sincrónico
Unica vez para todas las comunicaciones
¿Sincrónico o Asincrónico?

Tomando esto en consideración, analizamos algunos posibles puntos de prueba, que se explican aquí con pseudocódigo para cada punto.

Para estos experimentos decidimos bajar la cantidad de pruebas de 100 a 10, dado que se está buscando probar la robustez del sistema por un lado, y para maximizar los tiempos de experimento por el otro considerando el número de pruebas y el delay agregado.

Se decide correr los experimentos con la opción del ejercicio anterior en el cual el servidor corre en el host y el cliente en la vm.

NOTA: Vale aclarar que la función que utilizamos anteriormente para medir el tiempo, ***cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC / 2;*** *calcula el tiempo de CPU utilizado por el proceso, no el "tiempo de pared" transcurrido. O sea, mide el tiempo que la CPU realmente está ejecutando el proceso, no, por ejemplo, el tiempo que mide un reloj en la pared. Al introducir un sleep de 10 segundos se obliga a que el proceso se duerma y no use el CPU por 10 segundos. Este tiempo no se refleja en el tiempo de CPU utilizado por el proceso.*

Con el fin de explorar con mayor detalle el efecto de estos delays en el funcionamiento, agregamos una función, dwalltime(), que nos permite tomar el tiempo de reloj para poder ver reflejados los 10 segundos agregados, además de tener el tiempo de CPU.

Corremos el programa 10 veces por experimento con datos de tamaños de 10^3, 10^4, 10^5 , 10^6 y 10^7 bytes.

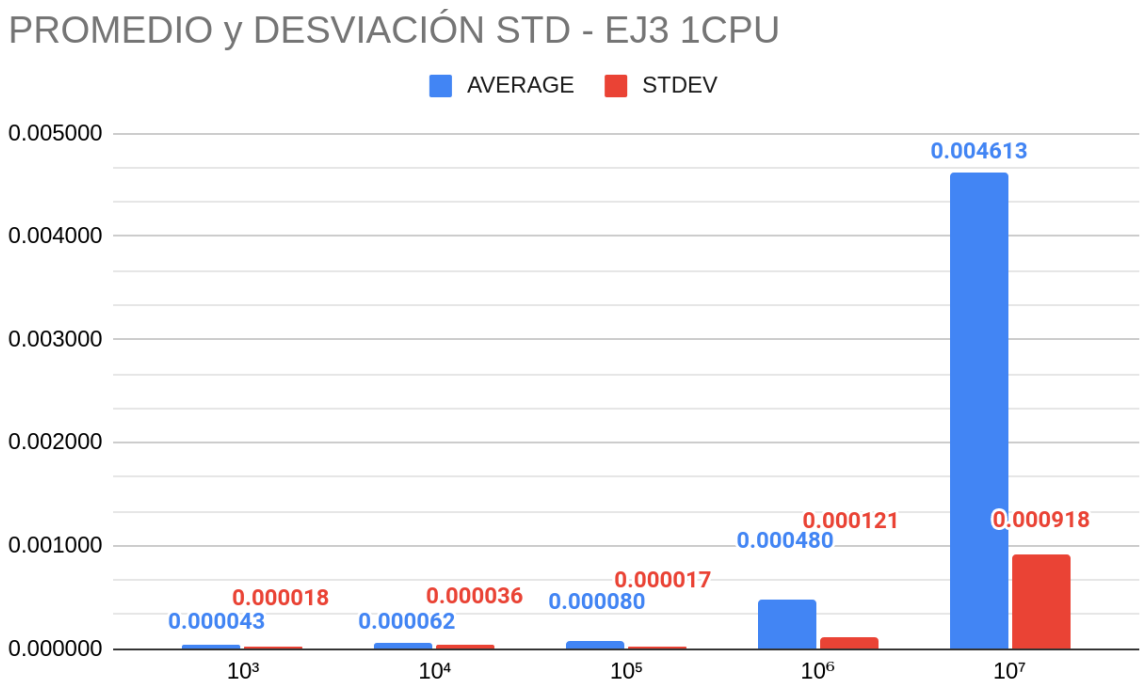
Para el Cliente

1. Antes de intentar conectarse:

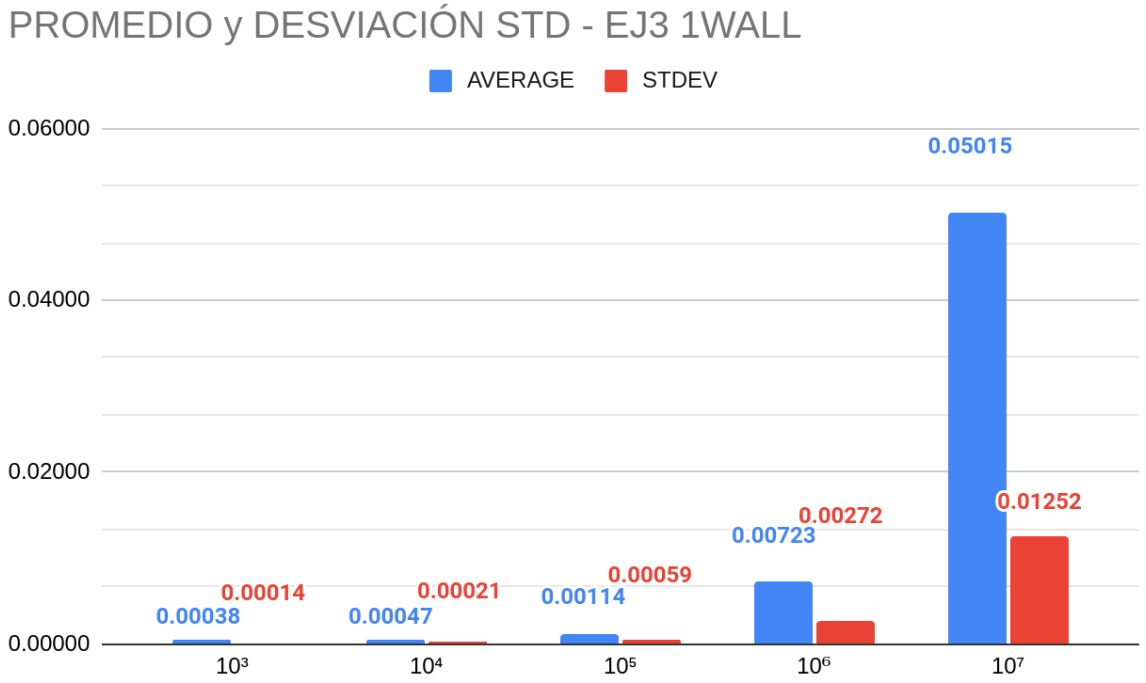
```
Do sleep 10
  While (connect(socket, address) no tire error)
```

- No se perciben errores: Esto sugiere que el cliente y el servidor manejan bien el retraso introducido antes de intentar la conexión.
- No se percibe diferencia en el tiempo de comunicación a nivel CPU o pared: Esto es dado que no hay un retraso en el tiempo de envío o recibimiento de datos.
- Se percibe una mayor lentitud al correr el programa que no influye en el tiempo de comunicación.
- Para cada conexión, hay una espera mayor que es la que se indicó en el código que confirma que el retraso introducido está funcionando como se esperaba.

CPU Time:



WALL Time:



2. Antes de enviar datos:

```
Sleep 10
write(socket, data, size)
```

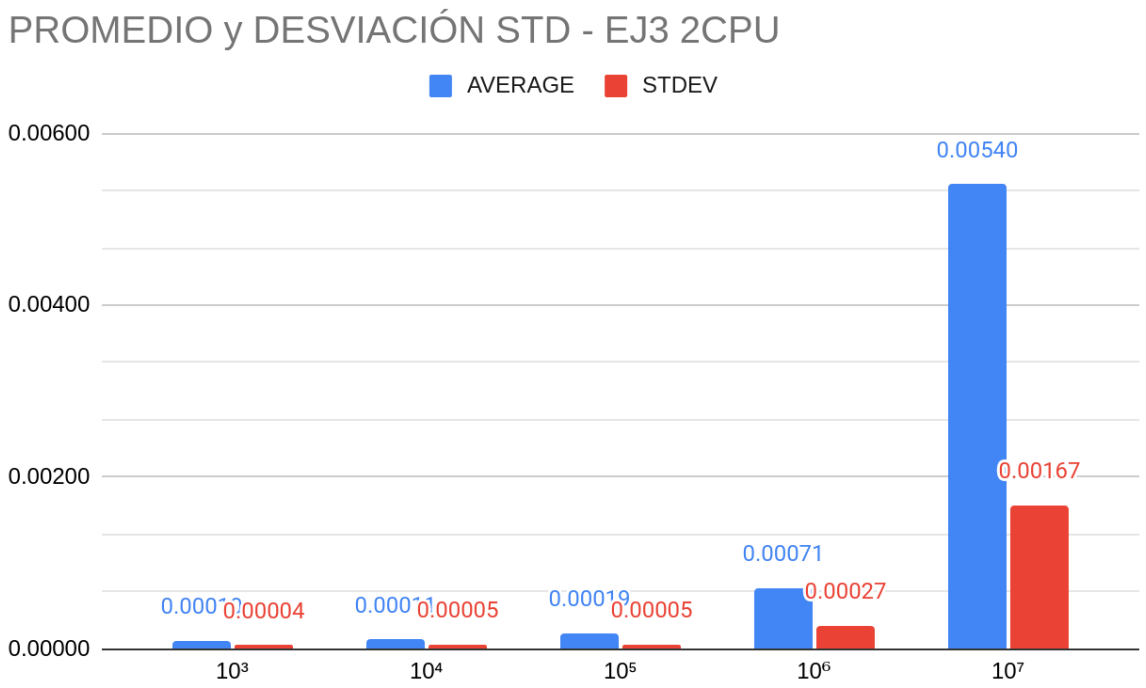
- No se perciben errores en la comunicación: Todos los datos se envían correctamente.

- Se percibe un incremento en los tiempos de comunicación tanto en tiempo pared como en tiempo de CPU. Esto es esperable para el tiempo de pared, pero el tiempo CPU es
- Como ejemplo para el incremento de los tiempos de CPU, el medio de tiempos de operación de envío de un buffer de 10^6 era 0.0004798 segundos, y ahora, con el retraso, el medio de envío del mismo buffer es de 0.0007072 segundos. Estos valores en este caso nos dan un incremento del 47.39%.
- Puede ser que la condición de espera luego de la conexión establecida genere actividad de CPU adicional directamente conectada con el proceso, pero puede que haya alguna actividad de red o CPU adicional en alguna capa subyacente.
- Se corre htop, se busca el proceso y se usa s (para strace) para investigar la actividad de CPU, pero no se encuentra actividad adicional de esta manera. En el siguiente log, se pueden ver las llamadas de manera esperada: **connect**, seguido de **gettime**, seguido de **nanosleep** (sleep 10), seguido de **write, read** y **gettime**. No se hacen adicionales llamadas que justifiquen el tiempo adicional de CPU, aunque podría haber alguna actividad de red o CPU adicional en alguna capa subyacente del sistema operativo o del stack de red, posiblemente debido a la transición de dormir a despertar:

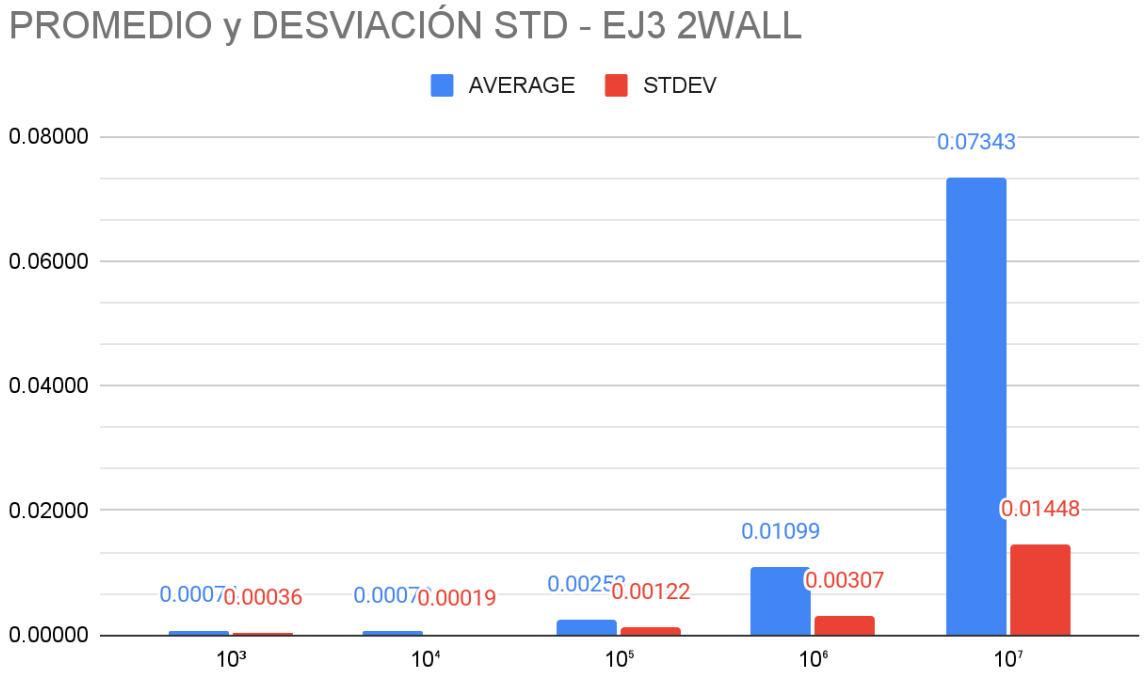
```
145002 clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffd9401da20) = 0
145002 connect(3, {sa_family=AF_INET, sin_port=htons(4003), sin_addr=inet_addr("192.168.56.11")}, 16) = 0
145002 clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=12623328}) = 0
145002 clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=10, tv_nsec=0}, 0x7ffd9401da20) = 0
145002 write(3, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., 1000000) = 1000000
145002 read(3, "I got your message", 18) = 18
145002 clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=13617827}) = 0
```

- Al agregar un retraso antes de enviar los datos, el tiempo de wall total medido desde el punto de vista del cliente (desde antes de enviar hasta después de recibir la respuesta) se incrementó por el valor del retraso ya que el **sleep** está en el medio de estos dos puntos de comunicación.

CPU Time:



Wall Time:



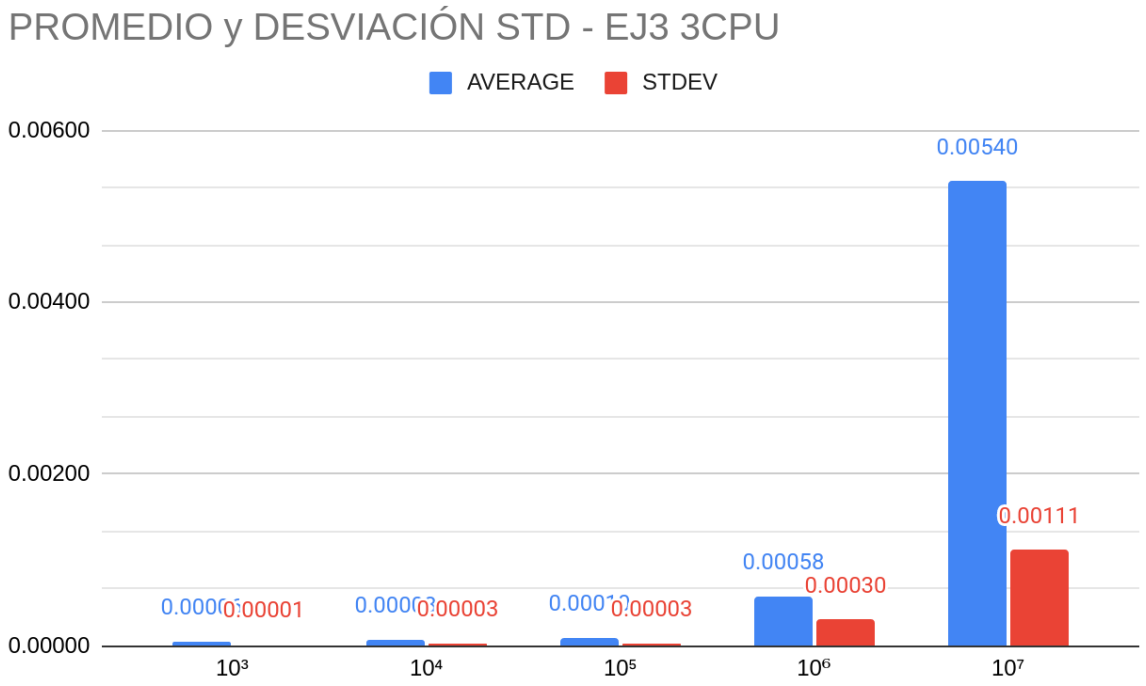
3. Luego de enviar datos, pero antes de leer la respuesta:

```
write(socket, data, size)
Sleep 10
read(socket, response, size)
```

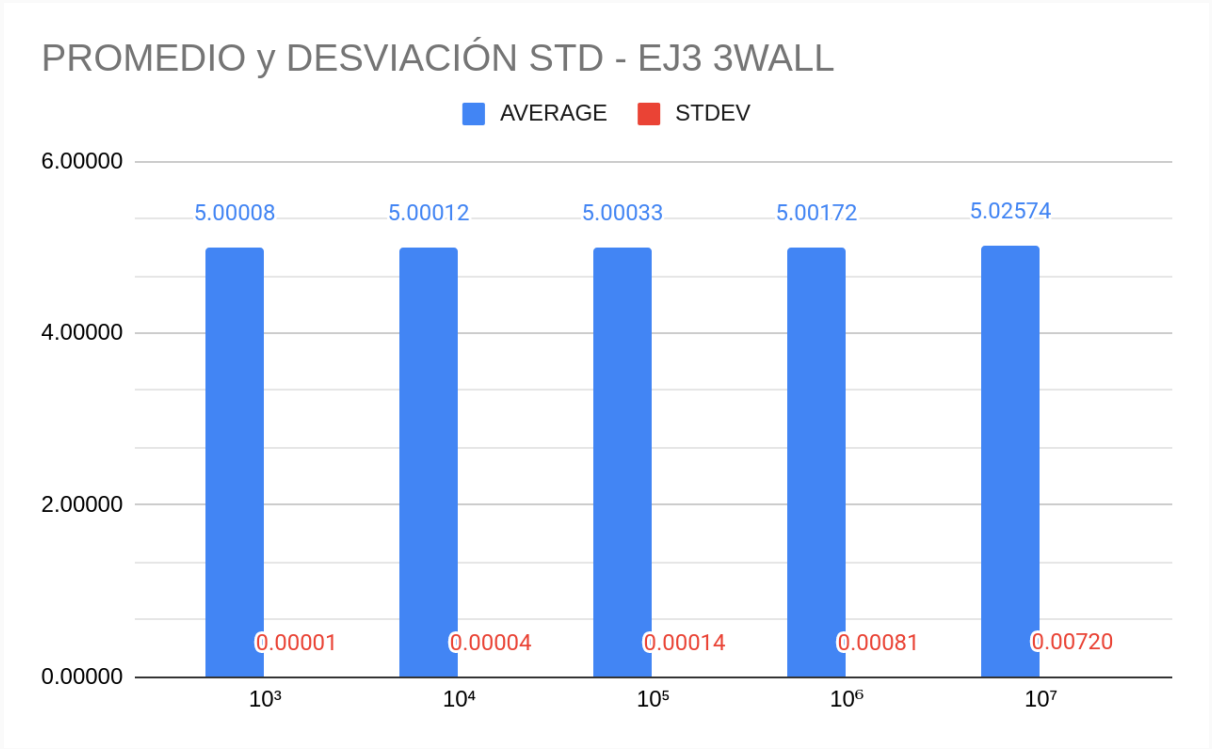
- No se perciben errores en la comunicación o en los tiempos de comunicación a nivel CPU, pero sí un gran incremento en los tiempos de pared.

- Al agregar un retraso después de enviar los datos pero antes de recibir una respuesta, el tiempo total medido desde el punto de vista del cliente (desde antes de enviar hasta después de recibir la respuesta) se incrementó por el valor del retraso, ya que el **sleep** está en el medio de estos dos puntos de comunicación.

CPU Time:



WALL Time:



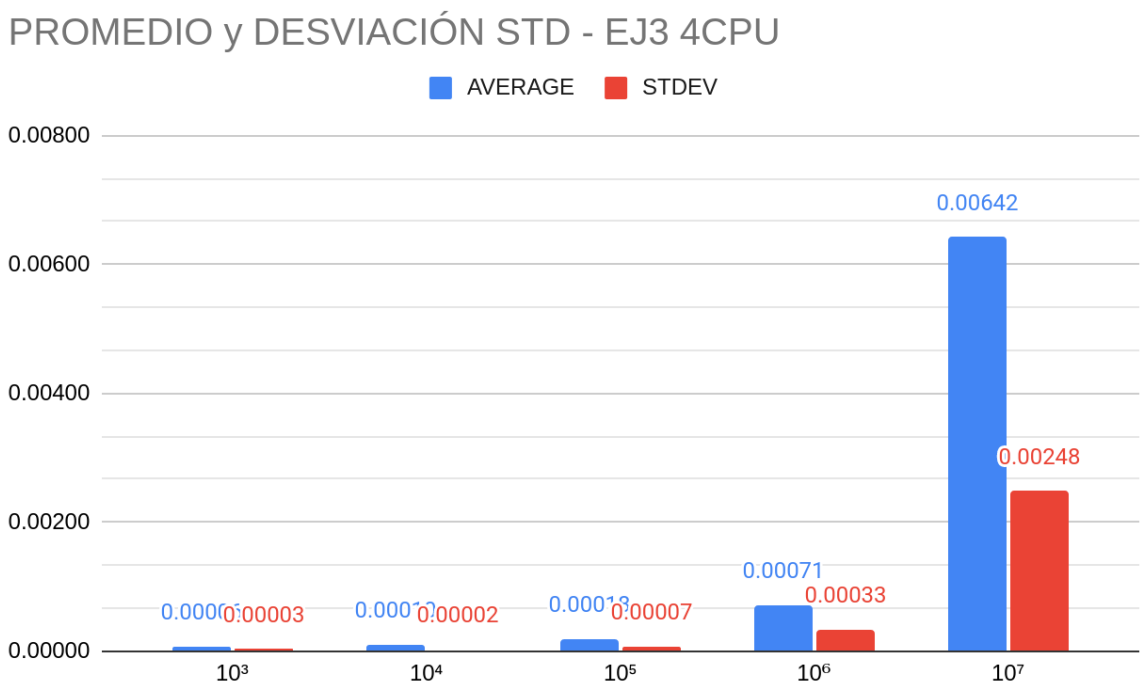
Para el Servidor

4. Antes de aceptar la conexión:

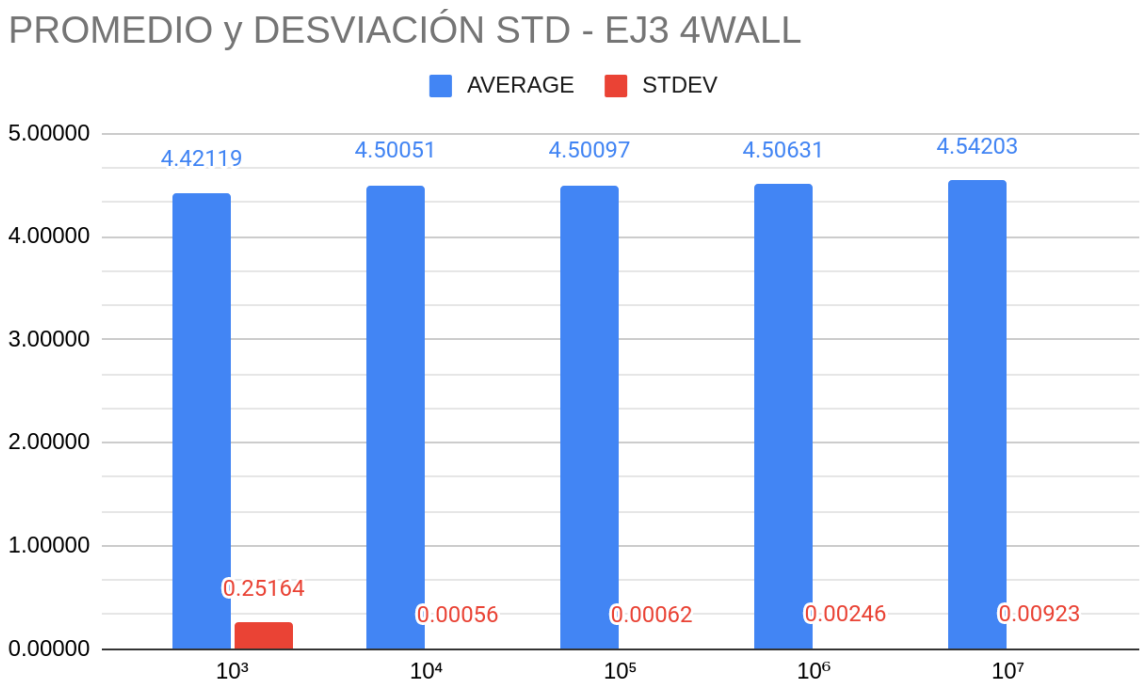
```
Sleep 10
nuevo_socket = accept(socket, address, client)
```

- No se perciben errores en la comunicación, genera un retraso en los tiempos medidos en el cliente, pero no de un tiempo proporcional a los 10 segundos enteros, sino de un periodo menor (aprox el 80%).
- Al usar el loop do while en el **connect**, se entiende que siendo este un punto de sincronización, el cliente debe esperar a que esté listo el servidor, así evitando errores.
- Es posible que esta espera esté agregando el tiempo de espera al tiempo total, aunque el tiempo medido en el cliente antes del **write** no debería verse afectado por el **sleep** en el servidor, ya que el **write** en el cliente no comenzará hasta que el **connect** haya sido exitoso.

CPU Time:



Wall Time:

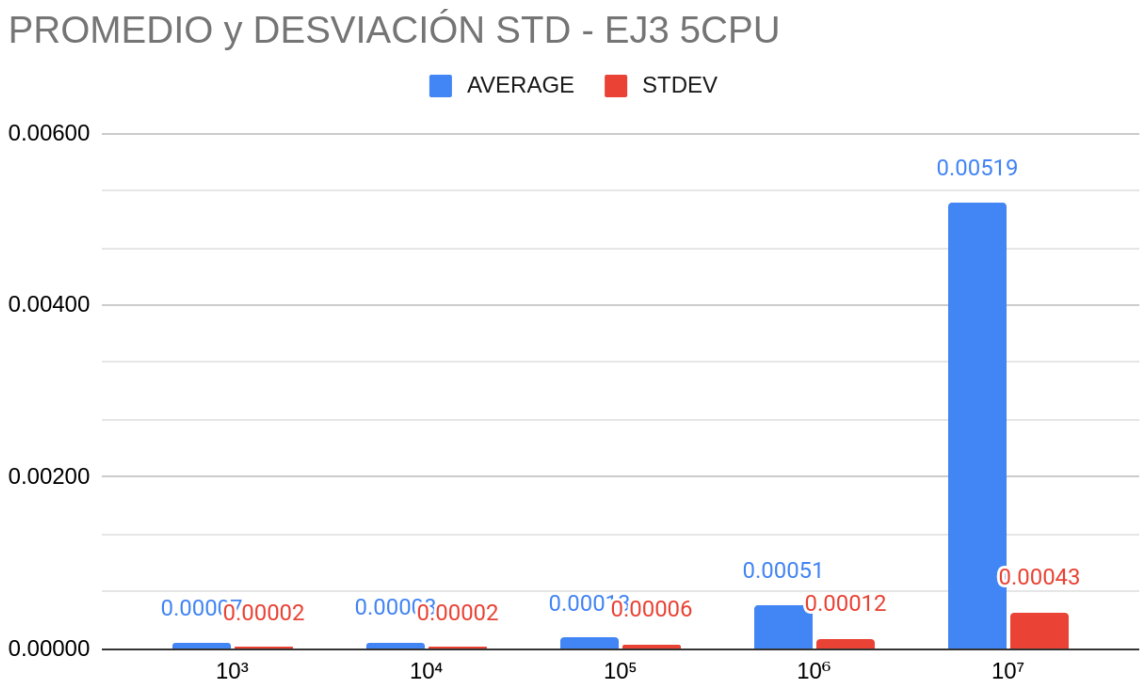


5. Luego de aceptar la conexión pero antes de leer los datos:

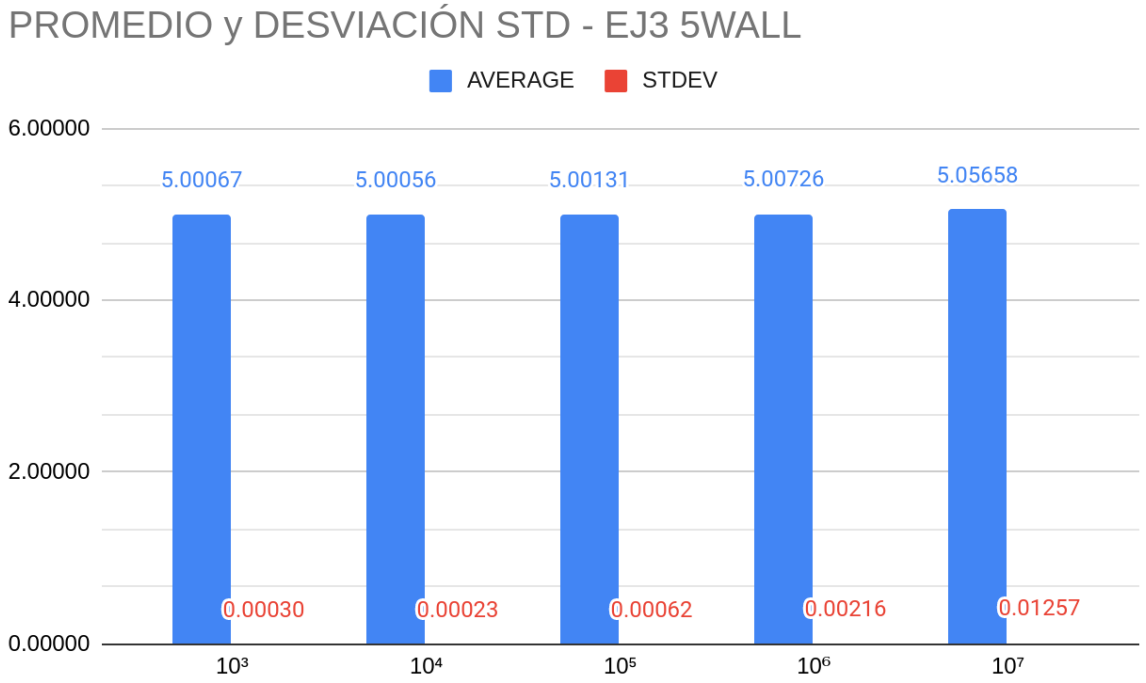
```
nuevo_socket = accept(socket, address, client)
Sleep 10
```

- Sin errores en la sincronización ni los datos enviados, pero si, se evidencia la latencia creada por el delay, y una medida pequeña más.
- Es comprensible el retraso ya que el punto de sincronización de **connect** y **accept** ya se logró; en este punto el cliente puede enviar datos con **write** y debe esperar al servidor, que no está en condiciones de leer con **read** hasta haber terminado su periodo de sleep 10.
- Restando los 10 segundos agregados por el **sleep** al wall time, se nota que hay una mayor latencia en la comunicación, de una proporción menor pero significativa.

CPU Time:



Wall Time:



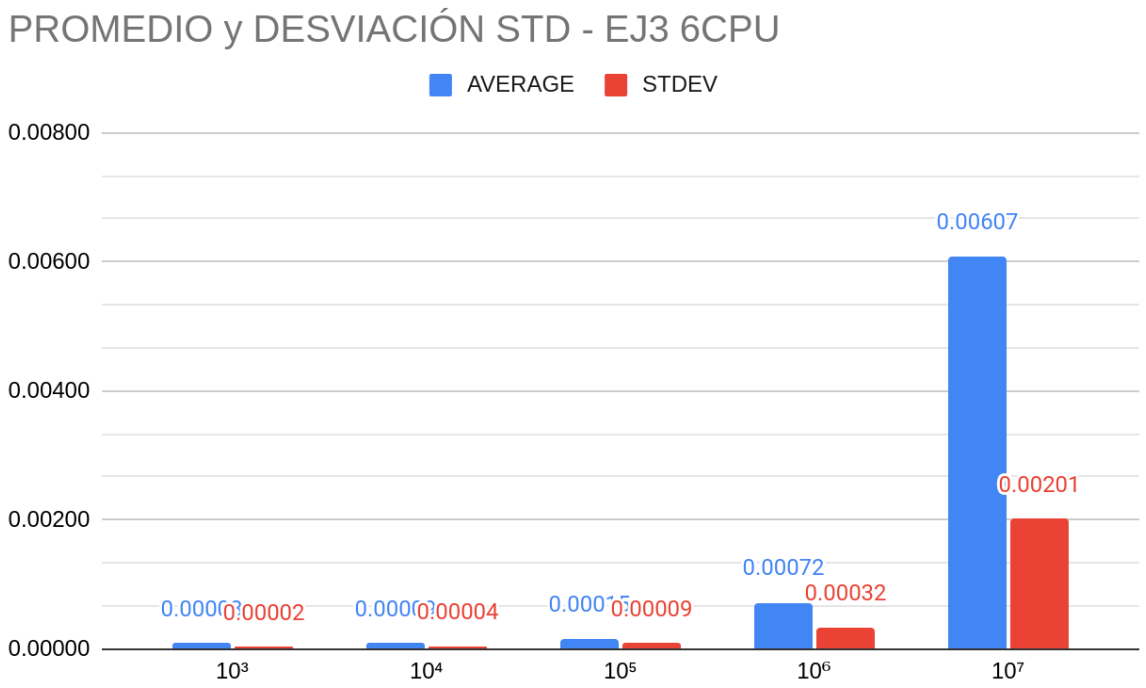
6. En el proceso de leer los datos (?)

```
do read(nuevo_socket, data, size)
  Sleep 10
  while (no se leen todos los datos);
```

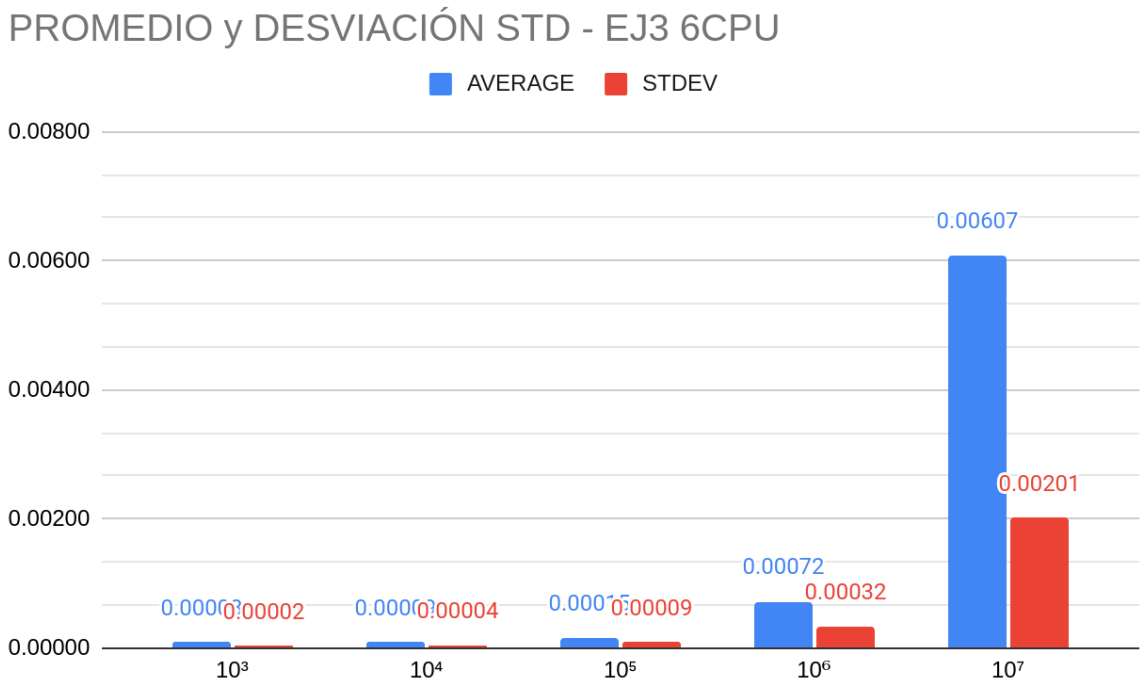
- Se comporta de manera esperable: Los tiempos de comunicación crecen de manera proporcional con el tamaño del buffer enviado, ya que se agregan 10 segundos para cada bloque no leído.

- Entendemos que el número de iteraciones depende del tamaño del buffer enviado y el buffer disponible por el SO.
- No se perciben cambios significativos en los tiempos de CPU.

CPU Time:



Wall Time:



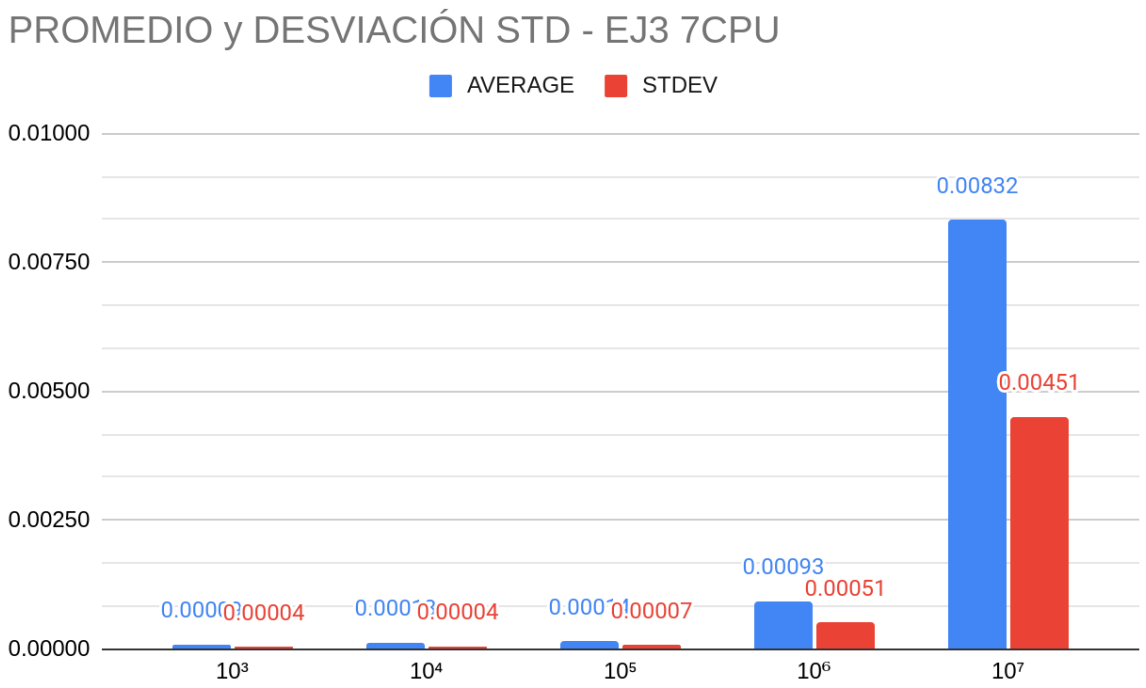
7. Luego de leer los datos pero antes de enviar una respuesta:

```
do read(new_socket, data, size)
```

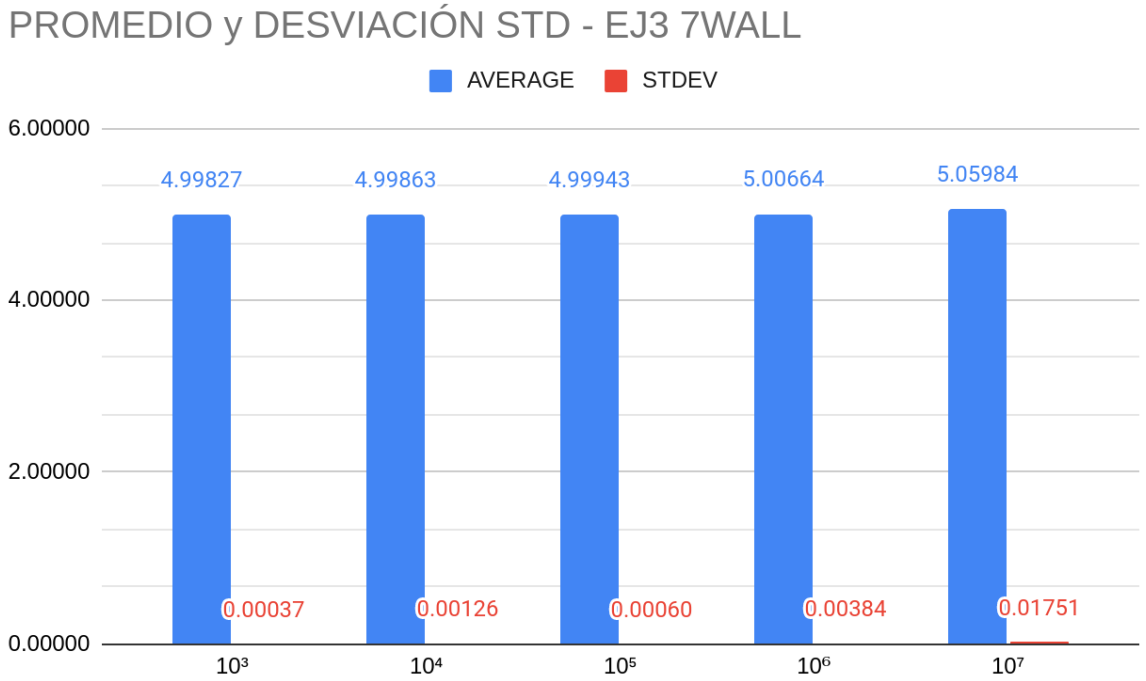
```
  while (no se leen todos los datos);
Sleep(10)
```

- No hay errores en la comunicación, el retraso es el esperado y corresponde al delay agregado. Al estar entre un **read** y un **write**, sabemos que interfiere directamente en la medición del tiempo de pared.

CPU Time:



Wall Time:



Para el Cliente y el Servidor

8. Antes del connect en el cliente y del accept en el servidor

Cliente:

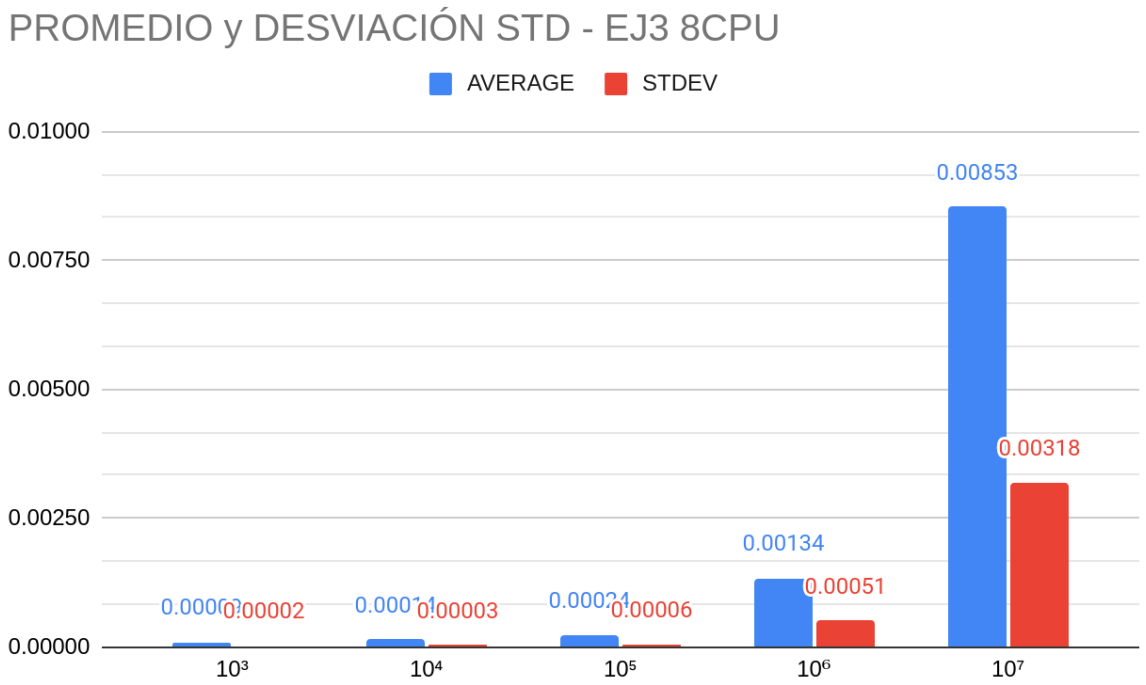
```
Do sleep 10
 While (connect(socket, address) no tire error)
```

Servidor:

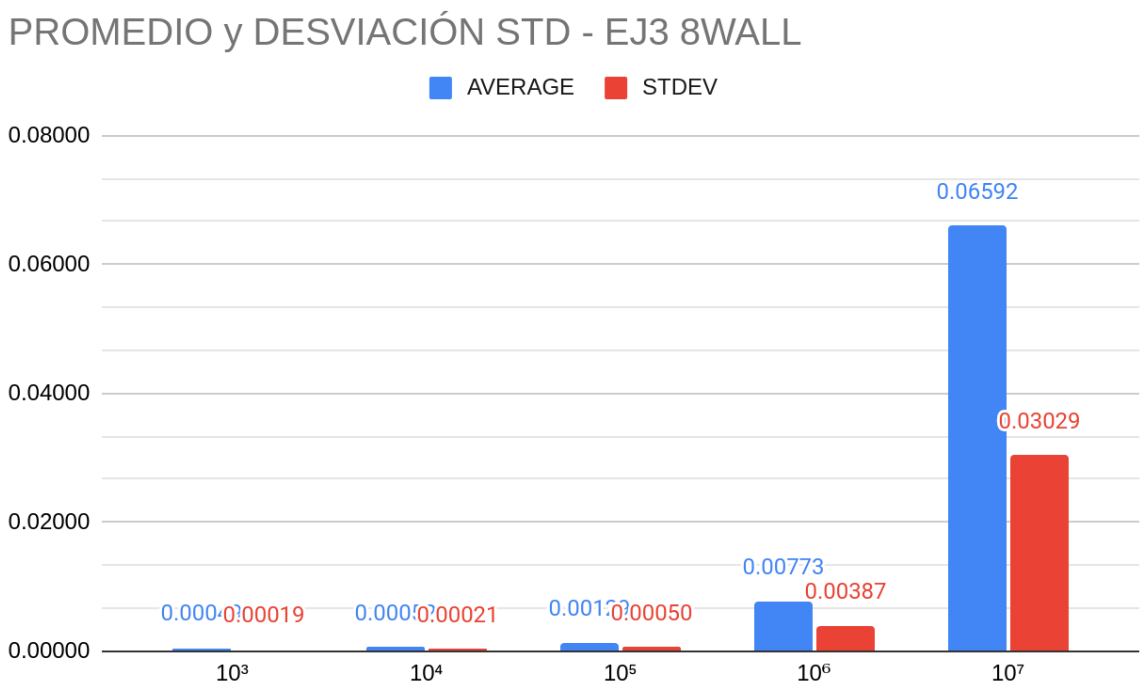
```
Sleep 10
nuevo_socket = accept(socket, address, client)
```

- No hay errores: Hay un incremento en los tiempos de comunicación con wall time que no es proporcional al delay de **sleep** 10, no llega al segundo, en ninguno de los casos. En este caso, a diferencia de cuando era solo el server que hacia el wait, los incrementos son menores, capaz por que el cliente tambien esta esperando antes de enviar y se sincroniza mejor el momento del **write**.
- El delay en tiempo pared es relativamente significativo siendo aprox entre 5x y 10x veces mayor al tiempo sin delay.
- El tiempo CPU es mayor a los experimentos sin delay.

CPU Time:



Wall Time:



9. Luego del write del cliente y del read en el servidor

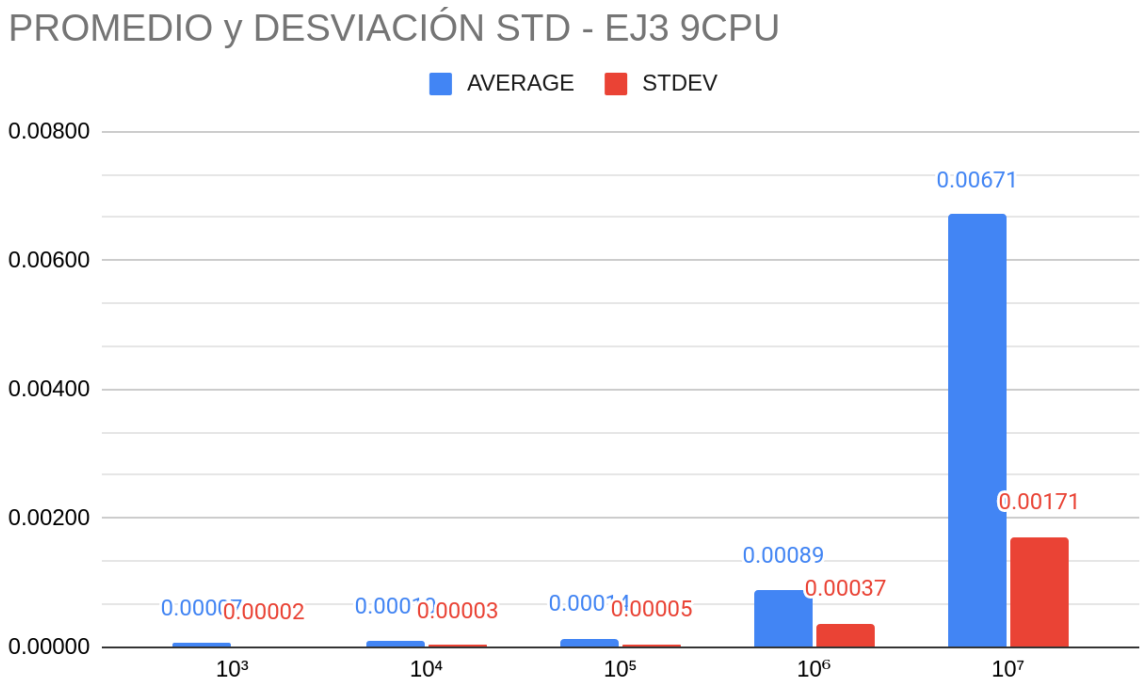
Cliente:
write(socket, data, size)
Sleep 10

Servidor:

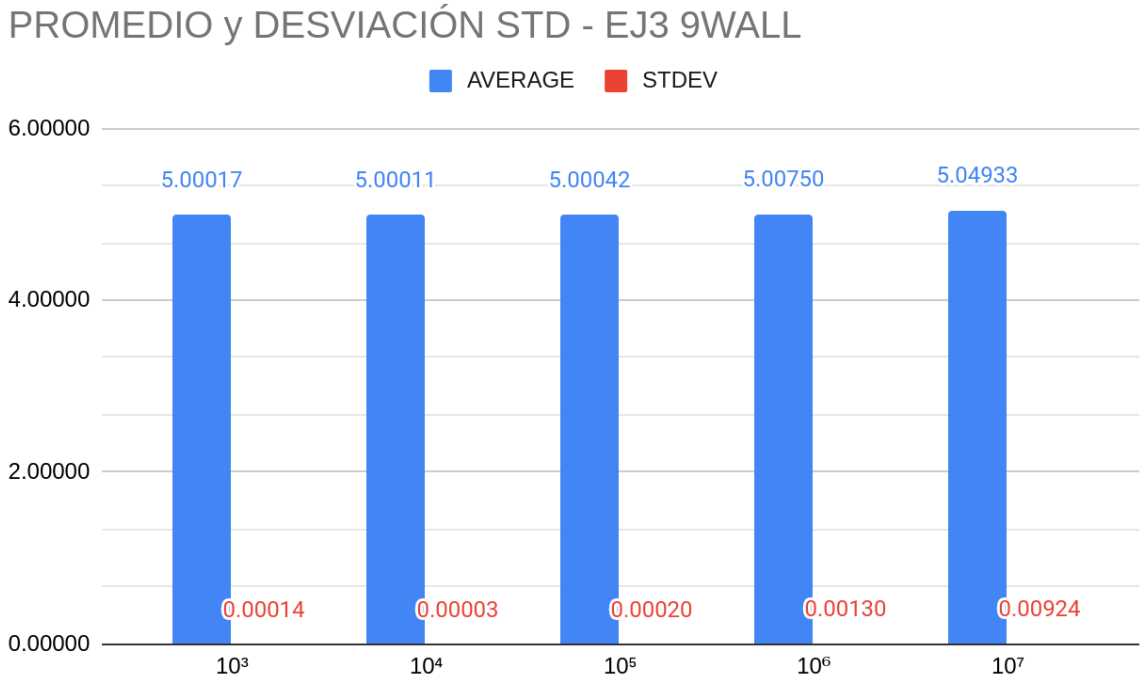
```
do read(new_socket, data, size)
  while (no se leen todos los datos);
Sleep(10)
```

- No hay errores en la comunicación: No se percibe una mayor o doble de demora a los otros experimentos, sino la demora correspondiente a un solo sleep de 10 segundos.
- Los tiempos confirman la forma que influye la sincronización implícita de C en el read y write en la ejecución del programa y en los tiempos de comunicación.

CPU Time:


PROMEDIO y DESVIACIÓN STD - EJ3 9CPU
AVERAGE
STDEV
0.00800
0.00600
0.00400
0.00200
0.00000
0.00671
0.00171
0.00089
0.00037
0.00002
0.00003
0.00005
10³
10⁴
10⁵
10⁶
10⁷

Wall Time:


PROMEDIO y DESVIACIÓN STD - EJ3 9WALL
AVERAGE
STDEV
6.00000
4.00000
2.00000
0.00000
5.00017
5.00011
5.00042
5.00750
5.04933
0.00014
0.00003
0.00020
0.00130
0.00924
10³
10⁴
10⁵
10⁶
10⁷