Trabajo Practico Nº02

Sistemas distribuidos, comunicación y sincronización

Fecha de entrega: 22/05/2020

https://gitlab.com/pitta1881/sdypp-2020
Ledesma Damián – 155825 – damiledesma67@gmail.com
Salinas Leonardo – 104478 – Sal_chueco@hotmail.com
Pittavino Patricio – 121476 - pitta1881@gmail.com

1) Desarrolle una red P2P de carga, búsqueda y descarga de archivos siguiendo las siguientes pautas:

- Existen dos tipos de nodos, Maestros y Extremos. Los primeros, son servidores centralizados replicados (al menos 2 nodos) que disponen del listado actualizado de los nodos extremos y se encargan de gestionar la E/S de los peers. Los segundos cumplen dos funciones en el sistema: realizan consultas (como clientes) y atienden solicitudes (como servidores).

- Funcionamiento:

-- Cada extremo dispone de un parámetro definido en un archivo de inicialización con las direcciones IP de los nodos Maestros. Al iniciarse se contacta con un maestro el cual funciona como punto de acceso al sistema e informa cuáles son los archivos que dispone para compartir. Luego, está atento a trabajar en dos modos (cliente y servidor)

-- Como cliente, deriva consultas al nodo maestro y una vez obtenida la respuesta, seleccionará el/los recursos que desee descargar y se contactará con el par correspondiente para descargar el/los archivo/s.

-- Como servidor, recibe la consulta, revisa si matchea la consulta con alguno de los recursos disponibles y devuelve los resultados al nodo que solicitó resultados.

A partir de los conceptos vistos en la teoría, critique este modelo y presente mejoras en su propuesta.

La resolución se encuentra en: **.\TP2\TP2-Ej1**

Nodo Maestro (\MastersSide)	**Nodo Extremo** (\ExtremosSide)	
	Como Cliente (\AsClient)	**Como Servidor** (\AsServer)
Maestro.java	ExtremoAsClient.java	ExtremoAsServer.java
Extremo.java		ExtremoAsServerHandler.java
ExtremoHandler.java		

Los recursos se encuentran en: **.\TP2\TP2-Ej1\src\main\resources**

Nodo Maestro (\Maestros\ServerXXXXXX)	**Nodo Extremo** (\Extremos\ExtremoXXXXXX)	
	Como Cliente (\data)	**Como Servidor** (\log)
ListaExtremos.json	Archivos para compartir	myLog.log
myLog.log		

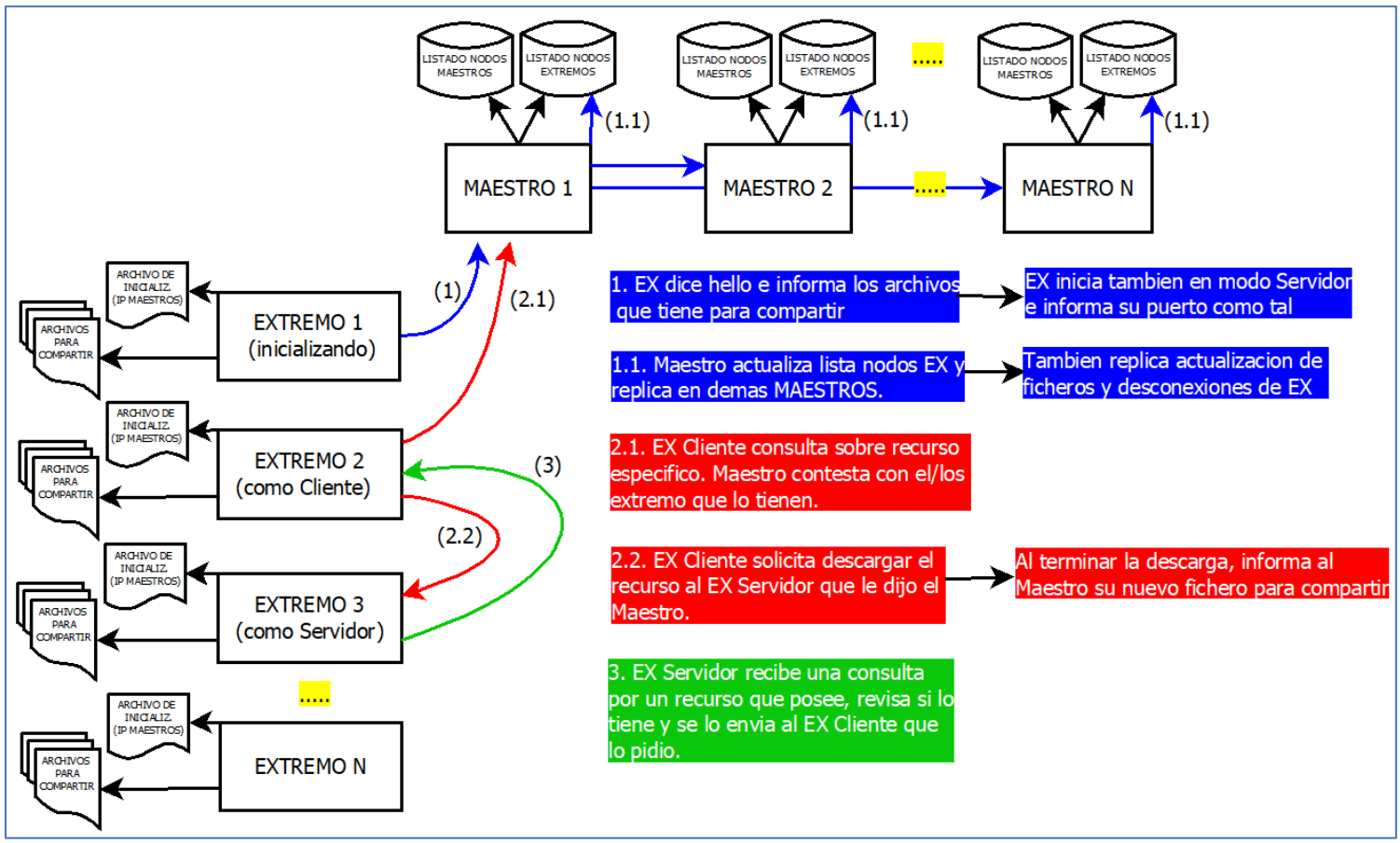
Pasos a seguir para iniciar la red P2P:

1. **Modificar el archivo de inicialización**: navegar hasta el archivo:
   .\TP2\TP2-Ej1\src\main\resources\inicializacion-ListaMaestros.json
   y modificarlo, indicando IP de donde correrán los Servidores Maestros.
2. **Iniciar Nodo/s Maestro/s:** se debe ejecutar la clase Maestro.java, se intentará iniciar en el Puerto 10000, y en caso de estar ocupado, continuará con el siguiente, esto permitirá iniciar múltiples Servidores bajo una misma IP. Al iniciar se intentará conectar a otros Maestros para pedir su lista de extremos, sino iniciará vacía. Si se inicia en otra maquina distinta a donde se iniciarán los Extremos, recordar buscar su IP y actualizar el archivo de inicialización.

3. **Para iniciar Nodos Extremos:** ejecutaremos la clase ExtremoAsClient.java, quien leerá el archivo de inicialización para intentar conectarse a un Nodo Maestro. Una vez conectado y para simular archivos para compartir, se randomizará una serie de archivos de la carpeta:
   .\TP2\TP2-Ej1\src\main\resources\Extremos\TodosLosArchivos\
   por lo que cada extremo puede llegar a tener archivos iguales que NO serán listados para descargar. Al iniciar correctamente el Extremo como Cliente, se iniciará también como Servidor, intentando a partir del puerto 20000.

Una vez iniciado el Extremo como Cliente y como Servidor, el Nodo Maestro al que se conectó, replicará a los demás Maestros esta información actualizada del Extremo (IP y PortAsServer). También replicará actualización de ficheros y desconexión de extremos.

La red P2P funciona de la siguiente manera:



Hay que tener en cuenta que el Maestro 1 tiene una gran carga de trabajo ya que todos los Extremos se dirigen a él, y luego replica datos nuevos a los demás Maestros, por lo que como propuesta central de mejora podemos sugerir que cuando un Extremo intente conectarse a un Maestro, le pregunte cuál es su carga de trabajo, y en caso de ser mayor que cierto valor, intentar por el siguiente hasta conectar con un Maestro que no supere el máximo de carga.

Como propuesta de mejora secundaria podemos sugerir que en el Punto 2.1 cuando el Maestro contesta con la lista de los Extremos que poseen el recurso, ordene dicha lista por alguna variable como por ejemplo menor latencia, o menor carga del nodo, etc.

2) Un banco tiene un proceso para realizar depósitos en cuentas, y otro para extracciones. Ambos procesos corren en simultáneo y aceptan varios clientes a la vez. El proceso que realiza un depósito tarda 40 ms entre que consulta el saldo actual, y lo actualiza con el nuevo valor. El proceso que realiza una extracción tarda 80 ms entre que consulta el saldo (y verifica que haya disponible) y lo actualiza con el nuevo valor.

La resolución se encuentra en: **.\TP2\TP2-Ej2**

ServerSide (\ServerSide) y (\log)	**ClientSide** (\ClientSide)
ServerMain.java	ClienteMain.java
ServerImplementer.java	Cliente.java
RemoteInt.java (iface)	
ClienteBanco.java	

Los recursos se encuentran en: **.\TP2\TP2-Ej2\Punto A\src\main\resources**
.\TP2\TP2-Ej2\Punto B\src\main\resources

ServerSide
.\data\Saldos.json
.\log\myLog.log

Para ejecutar el programa debemos iniciar primero el Servidor (ServerMain.java), el cual creará dos Servicios, "Deposito-Services" y "Extraccion-Services". Luego iniciaremos el Cliente (ClientMain.java) quien iniciará dos clientes (Threads) automáticamente, ambos operarán sobre el mismo cliente de banco (idCliente = 1001; saldoInicial = $0), pero el primero hará una operación de Deposito de $500 y el segundo una operación de Extracción de $250. Para que el Sistema Operativo ejecute en orden los dos Threads, se añade la sentencia Thread.sleep(1000) entre cada Start() y se aumenta el tiempo de sleep de los procesos de Deposito (40ms -> 1400ms) y Extracción(80ms -> 1800ms).
Lógicamente el Cliente 2 debería poder extraer sin problemas la cantidad de dinero solicitada.

a) Escribir los dos procesos y realizar pruebas con varios clientes en simultáneo, haciendo operaciones de extracción y depósito. Forzar, y mostrar cómo se logra, errores en el acceso al recurso compartido. El saldo de la cuenta puede ser simplemente un archivo de texto plano.

.\TP2\TP2-Ej2\Punto A

En una primera versión, al ejecutar nuestro programa obtenemos lo siguiente según el orden de respuesta, lo cual no satisface nuestro ideal:

```
Persona 1 > El saldo Actual del Cliente 1001 es: $0.0
Persona 1 > Depositando dinero..
Persona 2 > El saldo Actual del Cliente 1001 es: $0.0
Persona 2 > Error al Extraer dinero. No dispone de saldo disponible en su cuenta.
Persona 1 > Su Deposito de $500 fue realizado correctamente.
Persona 1 > Su nuevo saldo es: $500.0
```

Revisando el Log del Servidor:

```
[2020-05-06 20:02:50] [INFORMACIÓN] Mensaje Recibido de 192.168.0.28 (Persona 1) -> Consulta Saldo Cliente 1001
[2020-05-06 20:02:50] [INFORMACIÓN] Operacion Interna  -> Entro Operacion Consulta Cliente 1001
[2020-05-06 20:02:50] [INFORMACIÓN] Operacion Interna  -> Salgo Operacion Consulta Cliente 1001
[2020-05-06 20:02:50] [INFORMACIÓN] Mensaje Recibido de 192.168.0.28 (Persona 1) -> Operacion Deposito Cliente 1001
[2020-05-06 20:02:50] [INFORMACIÓN] Operacion Interna  -> Entro Operacion Deposito Cliente 1001
[2020-05-06 20:02:50] [INFORMACIÓN] Mensaje Recibido de 192.168.0.28 (Persona 2) -> Consulta Saldo Cliente 1001
[2020-05-06 20:02:50] [INFORMACIÓN] Operacion Interna  -> Entro Operacion Consulta Cliente 1001
[2020-05-06 20:02:50] [INFORMACIÓN] Operacion Interna  -> Salgo Operacion Consulta Cliente 1001
[2020-05-06 20:02:51] [INFORMACIÓN] Operacion Interna  -> Salgo Operacion Deposito Cliente 1001
```

Operación (1): el servidor recibe orden de consulta de saldo (línea 1) y la ejecuta completa (líneas 2 y 3).
Operación (2): el servidor recibe orden de depósito (línea 4), comienza a ejecutar (línea 5) y queda procesando por 1400ms.
Operación (3): mientras tanto, el servidor recibe otra orden de consulta de saldo (línea 6) y la ejecuta completa (líneas 7 y 8), pero aquí encontramos un error ya que está leyendo un dato viejo, debido a que la operación 2 aun está en proceso y no actualizó el valor del saldo.
Se completa la Operación 2 (línea 9).
Notamos que cuando se opera sobre un objeto compartido, éste debe ser bloqueado para impedir su acceso concurrente y evitar lecturas basura.

B) ESCRIBIR UNA SEGUNDA VERSIÓN DE LOS PROCESOS, DE FORMA TAL QUE EL ACCESO AL RECURSO COMPARTIDO ESTÉ SINCRONIZADO. EXPLICAR Y JUSTIFICAR QUÉ PARTES SE DECIDEN MODIFICAR

.\TP2\TP2-Ej2\Punto B

La solución al problema de lecturas basura está en sincronizar tanto las lecturas como las escrituras en dicho objeto, atomizando cada proceso y bloqueando su acceso a los demás. Para ello se añade la sentencia "synchronized (listaClientes)" cada vez que voy a hacer uso de dicho objeto en un método (Consulta Saldo, Depósito, Extracción).
Ejecutamos nuevamente ServerMain.java y ClientMain.java:

```
Persona 1 > El saldo Actual del Cliente 1001 es: $0.0
Persona 1 > Depositando dinero..
Persona 2 > El saldo Actual del Cliente 1001 es: $500.0
Persona 2 > Extrayendo dinero..
Persona 1 > Su Deposito de $500 fue realizado correctamente.
Persona 1 > Su nuevo saldo es: $500.0
Persona 2 > Su Extraccion de $250 fue realizada correctamente.
Persona 2 > Su nuevo saldo es: $250.0
```

Ahora la sincronización ordena la entrada y salida al recurso compartido y obtenemos un resultado válido.

Mientras tanto en el log del Servidor observamos que por mas que se reciban nuevas consultas, las operaciones son atómicas y se acumulan secuencialmente por orden de llegada.

```
[2020-05-06 21:18:51] [INFORMACIÓN] Mensaje Recibido de 192.168.0.28 (Persona 1) -> Consulta Saldo Cliente 1001
[2020-05-06 21:18:51] [INFORMACIÓN] Operacion Interna  -> Entro Operacion Consulta Cliente 1001
[2020-05-06 21:18:51] [INFORMACIÓN] Operacion Interna  -> Salgo Operacion Consulta Cliente 1001
[2020-05-06 21:18:51] [INFORMACIÓN] Mensaje Recibido de 192.168.0.28 (Persona 1) -> Operacion Deposito Cliente 1001
[2020-05-06 21:18:51] [INFORMACIÓN] Operacion Interna  -> Entro Operacion Deposito Cliente 1001
[2020-05-06 21:18:52] [INFORMACIÓN] Mensaje Recibido de 192.168.0.28 (Persona 2) -> Consulta Saldo Cliente 1001
[2020-05-06 21:18:53] [INFORMACIÓN] Operacion Interna  -> Salgo Operacion Deposito Cliente 1001
[2020-05-06 21:18:53] [INFORMACIÓN] Operacion Interna  -> Entro Operacion Consulta Cliente 1001
[2020-05-06 21:18:53] [INFORMACIÓN] Operacion Interna  -> Salgo Operacion Consulta Cliente 1001
[2020-05-06 21:18:53] [INFORMACIÓN] Mensaje Recibido de 192.168.0.28 (Persona 2) -> Operacion Extraccion Cliente 1001
[2020-05-06 21:18:53] [INFORMACIÓN] Operacion Interna  -> Entro Operacion Extraccion Cliente 1001
[2020-05-06 21:18:55] [INFORMACIÓN] Operacion Interna  -> Salgo Operacion Extraccion Cliente 1001
```

3) Construya una red flexible y elástica de nodos (servicios) la cual se adapte (crece /decrece) dependiendo de la carga de trabajo de la misma. El esquema será el de un balanceador de carga y nodos detrás que atienden los pedidos. Para ello deberá implementar mínimamente:

La resolución se encuentra en: **.\TP2\TP2-Ej3**

ServerSide		**ClientSide** (\ClientSide)
LoadBalancer (\LoadBalancer)	**Nodo** (\NodoServer)	
LoadBalancerMain.java	Nodo.java	ClienteMain.java
LoadBalancerImplementer.java	NodoImplementer.java	Cliente.java
RemoteIntBalancer.java	RemoteNodoServ.java	
	Clima.java – FechaHora.java – etc.	

- Simulación de carga de nodos. Creación dinámica de conexiones de clientes y pedidos de atención al servicio publicado.

En primer lugar, se debe ejecutar la clase LoadBalancerMain.java, el cual iniciará un Servidor RMI (Balanceador de Carga) bajo el cual los Clientes harán las peticiones. Al iniciar, también lo harán dos Nodos(primarios), sobre los cuales se repartirán las tareas solicitadas. Luego debemos iniciar la clase ClienteMain.java quien creará x Clientes que harán peticiones automáticas cada cierto lapso de tiempo.

- Protocolo de sensado para carga general del sistema. Elija un criterio para detectar esa carga y descríbalo. Ejemplo: cantidad de clientes en simultáneo que el servicio puede atender. Si se excede esa cantidad, el punto de entrada (balanceador de carga) crea dinámicamente un nuevo servicio.

El balanceador usa el método *Weighted Least Connections,* en el cual cada Nodo tiene un limite de **Peticiones** simultaneas y un limite de **Peso** soportado. Al recibir una petición se sabe cual es su peso, por lo que se consulta a cada nodo si lo puede aceptar, si un nodo excede alguna de las dos variables, rechaza la petición y la pasa al próximo nodo. Si no hubiera mas nodos, la petición quedará en espera ***wait()*** a que se cree un nuevo nodo ***notifyAll()*** o que uno de los existentes termine una tarea pendiente y desocupe tanto el peso como las peticiones actuales ***notify()***.

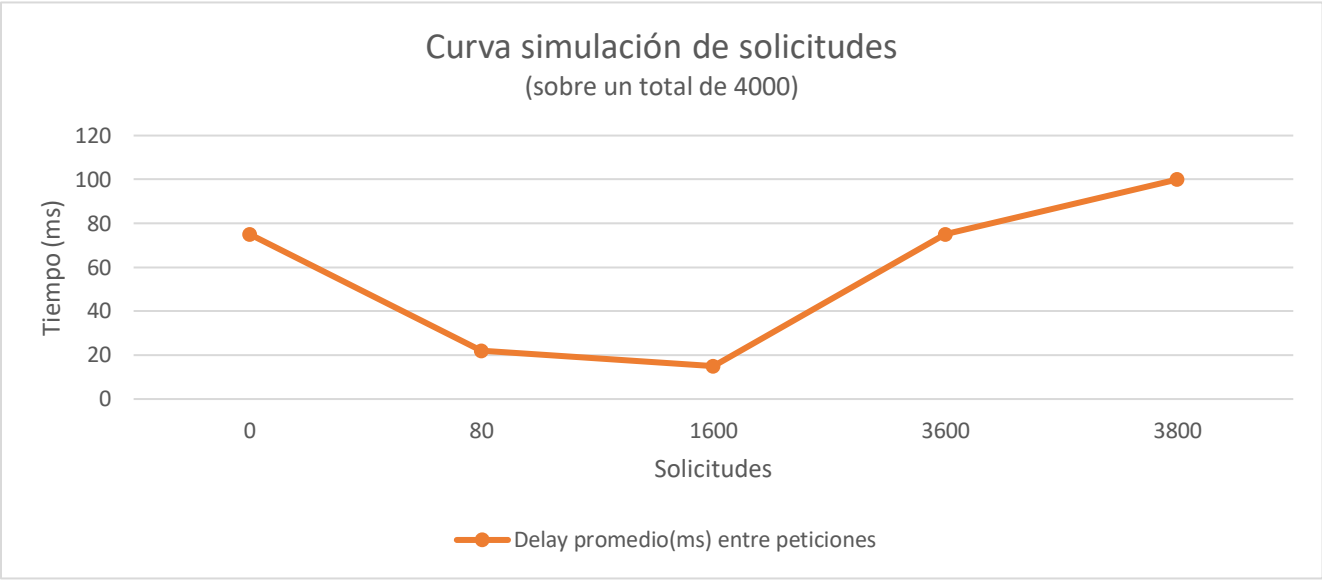
- Definición de umbrales de estado {sin carga, normal, alerta, crítico}

	PETICIONES		**PESO**
SIN CARGA	0%	&&	0%
NORMAL	>0%	||	>0%
ALERTA	>60%	||	>60%
CRITICO	>80%	||	>80%

- Creación, puesta en funcionamiento de los servicios nuevos, y remoción de ellos cuando no sean más necesarios. Para esto el balanceador puede contar con una lista de IPs donde los servicios están instalados y pueden correr. De forma tal que, arrancando inicialmente con 2 servicios en 2 nodos distintos, el sistema escala dinámicamente en función de la carga del sistema, usando los nodos listados en ese archivo de configuración. Si fuera necesario, puede haber más de un servicio en un mismo nodo. El servicio debe ser multi thread.

Para crear un nuevo nodo, se calcula la carga general del sistema, haciendo un promedio entre peticiones actuales y peso actual en cada nodo, si esta carga general es mayor a 55%, se lanza la orden de crear un nuevo nodo, y si es menor a 13%, se elimina un nodo cuyo estado sea Sin Carga.

Para la correcta visualización del funcionamiento se simula la carga de 4000 solicitudes según el siguiente gráfico:



- El 2% de solicitudes inicial:
  - Representa las solicitudes 1-80.
  - Delay promedio de 75ms entre cada solicitud.
  - Comienza a cargar muy lentamente al balanceador, si es necesario irá creando nuevos nodos y también puede eliminar los que acaba de crear.
- El 38% de solicitudes siguiente:
  - Representa las solicitudes 81-1600.
  - Delay promedio de 22ms entre cada solicitud.
  - Se empieza a saturar el balanceador, crea nuevos nodos y debido a la gran carga que tiene el balanceador nunca elimina ninguna.
- El 50% de solicitudes siguiente:
  - Representa las solicitudes 1601-3600.
  - Delay promedio de 15ms entre cada solicitud.
  - Se sigue enviando mas carga todavía, se siguen creando nuevos nodos.
- El 9% de solicitudes siguiente:
  - Representa las solicitudes 3601-3800.
  - Delay promedio de 75ms entre cada solicitud.
  - Disminuye la carga del sistema a menos del umbral mínimo, los nodos comienzan a quedar ociosos y se van eliminando de a poco.
- El 1% de solicitudes final:
  - Representa las solicitudes 3801-4000.
  - Delay promedio de 100ms entre cada solicitud.
  - La carga del sistema es menor que el umbral mínimo, se eliminan todos los nodos ociosos y solo quedan los dos principales recibiendo las peticiones.

4) El operador de Sobel es una máscara que, aplicada a una imagen, permite detectar (resaltar) bordes. Este operador es una operación matemática que, aplicada a cada pixel y teniendo en cuenta los pixeles que lo rodean, obtiene un nuevo valor (color) para ese pixel. Aplicando la operación a cada pixel, se obtiene una nueva imagen que resalta los bordes.

A) Desarrollar un proceso centralizado que tome una imagen, aplique una máscara, y genere un nuevo archivo con el resultado.

La resolución se encuentra en: **.\TP2\TP2-Ej4\Punto A**

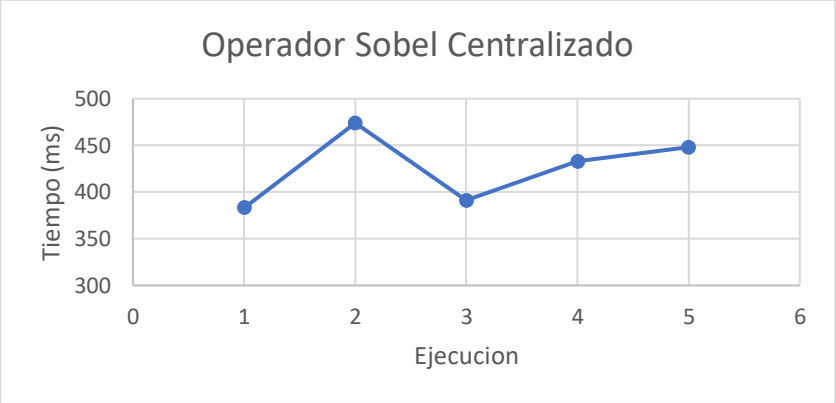
Se construyó la lógica para aplicar la máscara a una imagen de entrada. Se utilizó un método (llamado getGrayScale en nuestro ejercicio) que obtiene el valor de oscuridad del pixel, para determinar si se trata de un pixel que forma parte de un borde o no.

La ejecución del proceso centralizado, demora entre 380ms y 500ms.

Tamaño de imagen pequeña: 870x555 (1Mb)

Ejecucion	Tiempo (ms)
1	383
2	474
3	391
4	433
5	448

PROMEDIO= 425.8ms



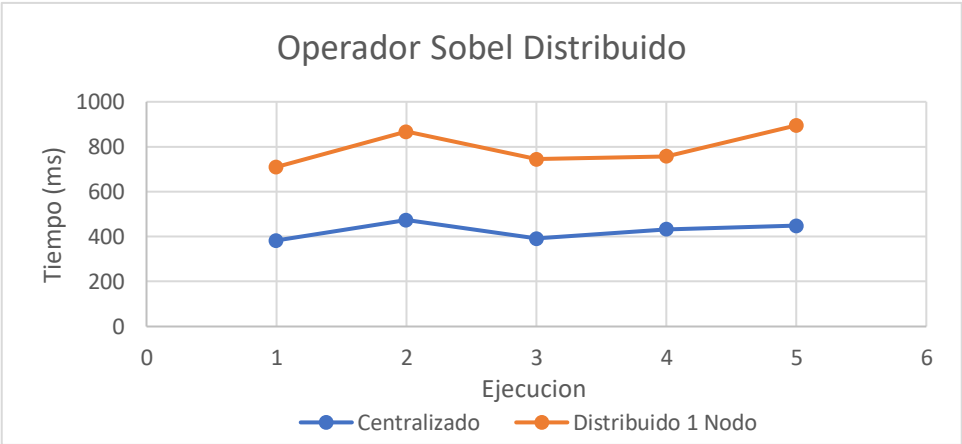
B) Desarrolle este proceso de manera distribuida donde se debe partir la imagen en N pedazos, y asignar la tarea de aplicar la máscara a N procesos distribuidos. Después deberá juntar los resultados. Se sugiere implementar los procesos distribuidos usando RMI. A partir de ambas implementaciones, comente los resultados de performance dependiendo de la cantidad de nodos y tamaño de imagen.

C) Mejore la aplicación del punto anterior para que, en caso de que un proceso distribuido (al que se le asignó parte de la imagen a procesar) se caiga y no responda, el proceso principal detecte esta situación y pida este cálculo a otro proceso.

Se desarrolló la lógica utilizando RMI. El ServerImpl se encarga de realizar la operación Sobel. A continuación, una gráfica que representa los tiempos de ejecución (ms) de una implementación mediante RMI con un nodo.

1 NODO	
Ejecucion	**Tiempo (ms)**
1	710
2	867
3	745
4	758
5	895

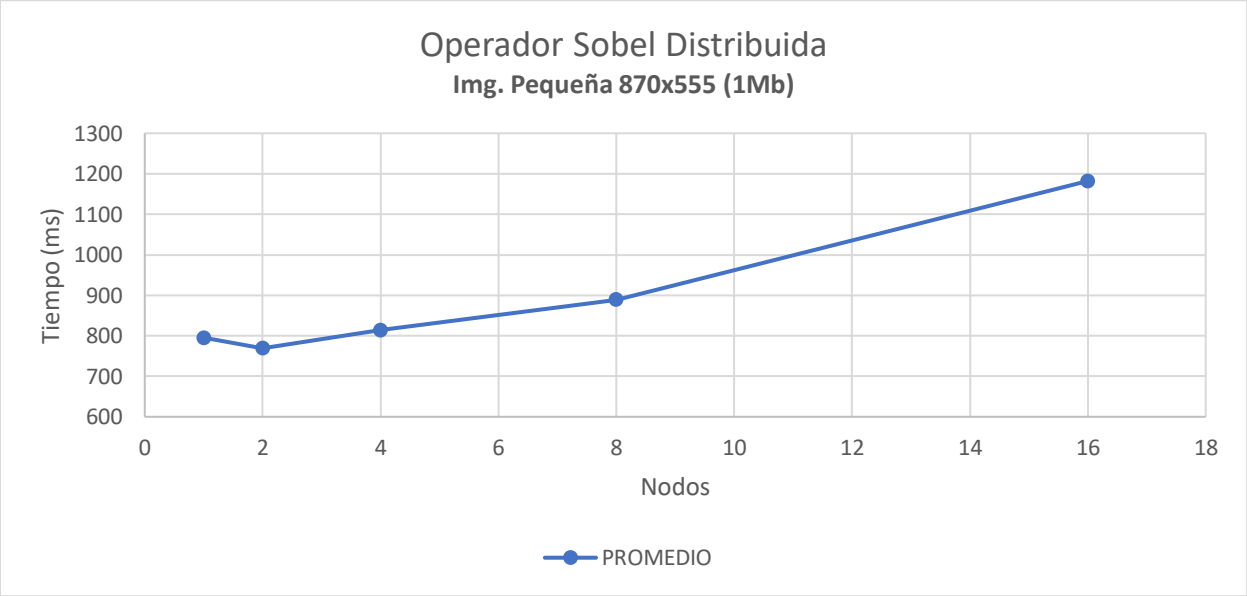
PROMEDIO= 795ms



Claramente, los tiempos aumentan, esto es un resultado esperado debido al Overhead agregado de enviar la imagen al servidor, y respuesta del servidor con el resultado.

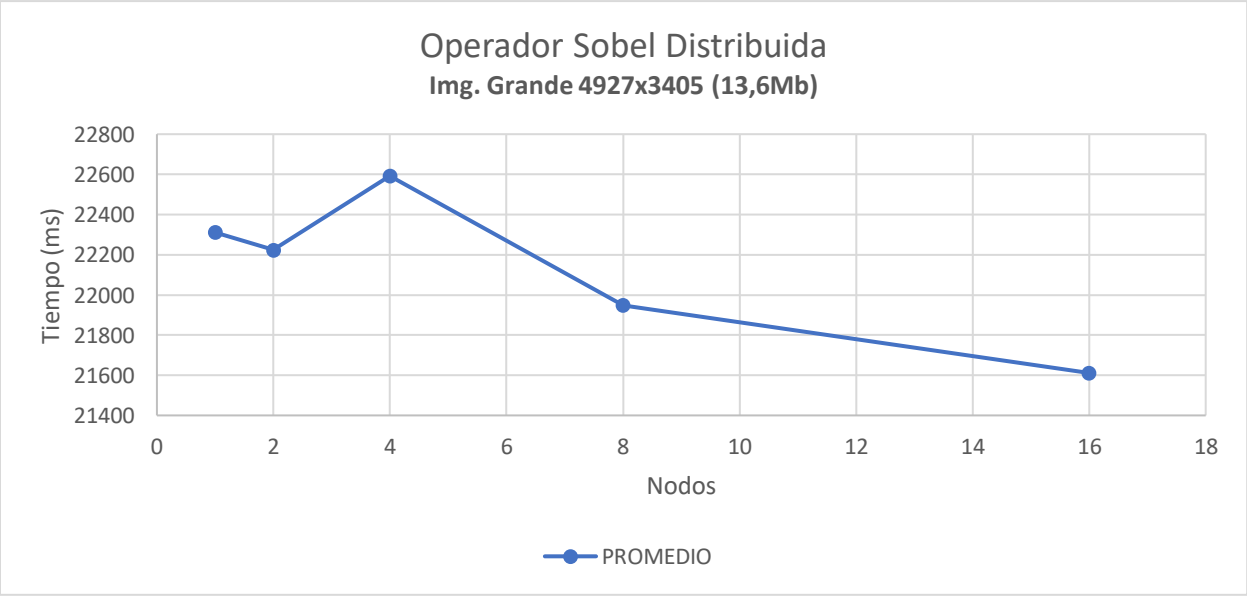
Tamaño de imagen pequeña: 870x555 (1Mb)

NODOS	MIN	MAX	PROMEDIO
1	710	895	795
2	725	870	769.2
4	766	897	813.8
8	853	934	888.8
16	1057	1344	1182.2



Tamaño de imagen grande: 4927x3405 (13,6Mb)

NODOS	MIN	MAX	PROMEDIO
1	20589	22951	22312
2	20553	23025	22224
4	20516	23453	22592
8	20863	23380	21948.4
16	20513	23000	21611.2



Conclusión:
No siempre a más procesos menor tiempo de respuesta. El tiempo de respuesta se ve afectado por varios aspectos del contexto en el cual se ejecuta el sistema. Siempre se debe encontrar un balance entre tiempo de respuesta aceptable y número de procesos requeridos.

Ejecución:
Para ejecutar dirigirse a target y correr los dos .jar:
java –jar TP02_EJ4b_Servidor-1
java –jar TP02_EJ4b_Cliente-1