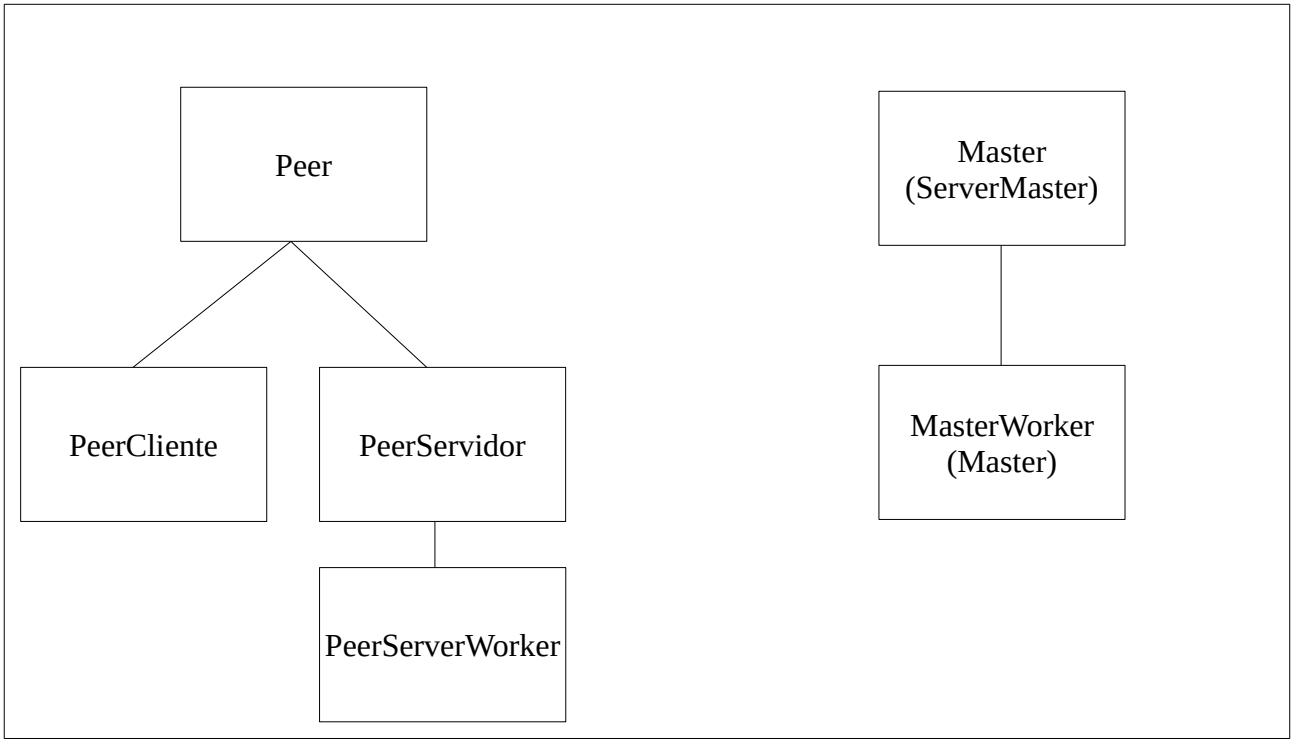
41409 – Sistemas distribuidos y Programación Paralela

Trabajo Practico N.º 2

Sistemas distribuidos, comunicación y sincronización

Sistema P2P de carga, busqueda y descarga de archivos

Arquitectura general



Como correr el ejercicio:

Ir a .\TP2\src\main\java\jusacco\TP2\punto1\run
Donde se encuentran los archivos java listos para ser ejecutados:
Master1.java, … , MasterN.java
Peer1.java, … , PeerN.java

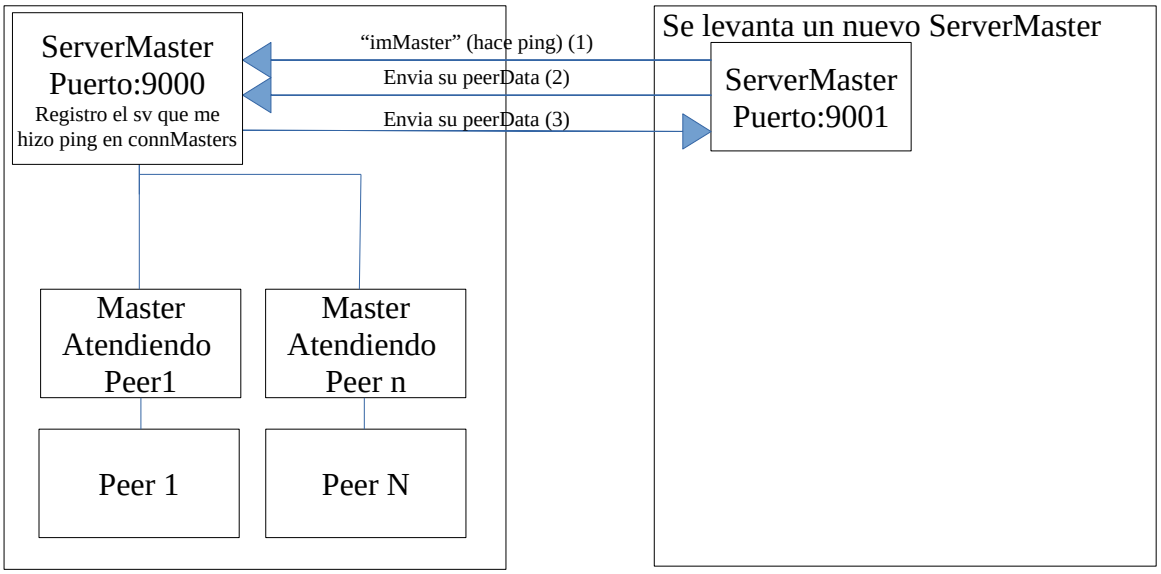
Explicacion Nodos:

- Peer:
  - Encargado de cargar las configuraciones tales como:
    - Buscar y cargar archivo de configuraciones basico para conectarse a Masters
    - Cargar los archivos de su directorio (crear el directorio si hace falta)
  - Hacer ping a todos los master que levanto para ver cuales están disponibles.
  - Conectarse al ultimo master que esta disponible.
  - Crear el hilo de cliente (PeerClient.java).
  - Crear el hilo de servidor (PeerServer.java).
  - Hacer un Thread join a PeerClient, para terminar el proceso cuando el cliente se cierre, de esta manera, el PeerServer será dado de baja cuando el cliente se cierre.
- PeerCliente:
  - Enviar archivos disponibles a el Master conectado (metodo givePeerData()).
  - Ofrecer un menu para poder interactuar con el servidor Master(opcion de buscar archivos, recibir el lugar donde se encuentra y descargarlo).
  - Guarda el archivo descargado, renombrandolo si hace falta (ya existe otro del mismo nombre)
- PeerServidor:
  - Espera conexiones nuevas (cuando se recibe una, implícitamente es un pedido de descarga).
  - Obtenida una nueva conexión lanza el worker para atender al cliente. (PeerServerWorker.java).
- PeerServerWorker:
  - Lee por socket el pedido del cliente, las opciones a recibir pueden ser:
    - buscar=nombre de archivo a buscar.
    - descargar=nombre de archivo a descargar.
  - Se encarga de leer el archivo en el file system, traducirlo a bytes, crear un objeto de tipo Archivo (en el que se encuentran los bytes a enviar).
  - Enviar el objeto Archivo.
  - Devolver un mensaje de error en caso de que el archivo que desea descargar no existe.

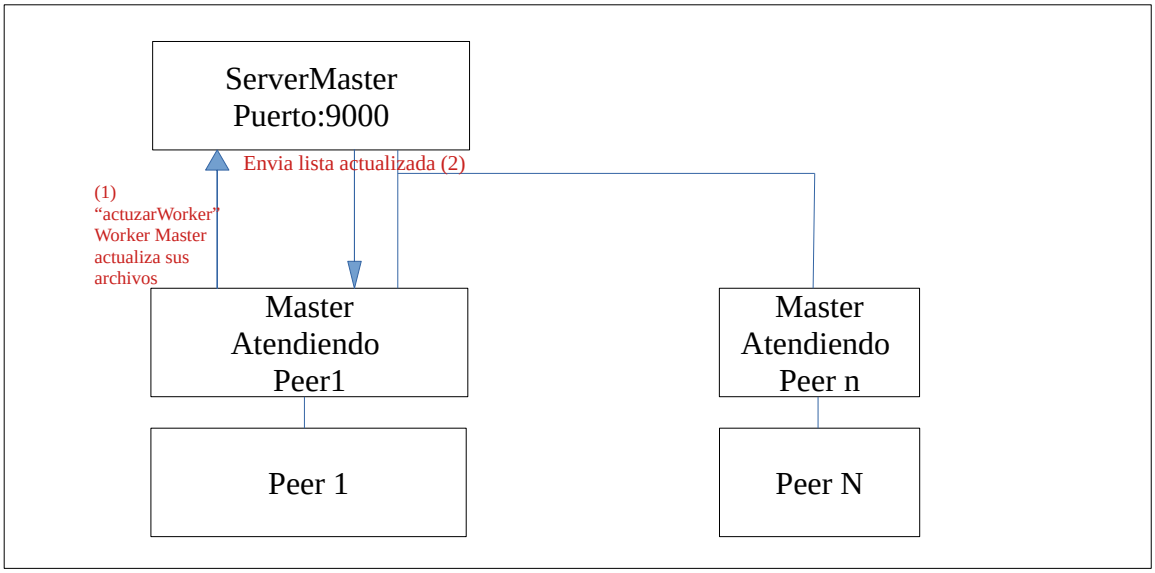
- ServerMaster:
  - Es el encargado de registrar sus configuraciones en un archivo (config.txt), esto lo hace mediante una clase (Config.java) que tiene la responsabilidad de leer el archivo de configuraciones y escribir nuevas configuraciones. En caso de encontrar Masters ya registrados lo notifica para que el nuevo ServerMaster lo registre en su lista de Masters conocidos(ArrayList otherMasters).
  - Posee un metodo para hacer ping a todos los master que conoce, en caso de que el ping sea devuelto lo registrara como un Master actualmente conectado (ArrayList connMasters)
  - Por ultimo esperara conexiones nuevas a través de socket, debido a que tanto Masters(workers) como Peers pueden comunicarse con el, lee un primer mensaje para detectar que se requiere de el. Los mensajes pueden ser:
    - “imMaster”: Si se recibe este mensaje quiere decir que otro master le hizo Ping, en este caso, al master que le hizo ping lo registra en connMasters.
    - “imClientPinging”: Un cliente consulto para saber si esta conectado o no. Cierra la conexión inmediatamente, ya que solo se trata de un ping.
    - “actualizarServerMaster”: Un worker Maestro esta enviando su lista de peers+archivos, por tanto, utilizo el metodo actualizarByObj (busco los peer que tengo y los comparo con los datos que recibí.
      - Si el peer esta: me quedo con los archivos nuevos que este master no tenía.
      - Si el peer no esta: agrego toda la lista de archivos junto con el peer.
    - “peerServerPortOn”: este mensaje es para notificar que un peer server esta en el puerto indicado, si esto sucede, quiere decir que hay un peer client que inició. Registro los archivos del nuevo peer mediante requestClientData(ip,port) y se lanza un hilo para atender al peer cliente.
    - “actualizarWorker”: Utilizado para actualizar la lista de archivos del worker. Fue implementado debido a que un ServerMaster puede tener multiples clientes peer (múltiples worker atendiéndolos).
    - “bajaPeer”: Borra todos los archivos de dicho peer.
- Master:
  - Atiende a un cliente que abrio socket contra el. Lee lo que este le envíe y procede. El cliente le puede enviar:
    - “buscar=nombre archivo”: Buscará en su lista el archivo indicado, si no lo encuentra devuelve “no existe”. Si encuentra mas de uno los envia a todos mediante un mensaje del tipo: dueñoArchivo//@t//nombre archivo. Hasta que no tenga mas para enviar y envía “.END” para notificar que no hay mas.
    - “cerrarConn”: Si recibe esto borra todas las entradas en su peerData para el cliente que cierra la conexión y cierra la conexión.
    - “peerServerPortOn”: Al igual que en ServerMaster, este metodo se encarga de actualizar el contenido del peer y de replicarlo a los Masters conectados.

Escenarios

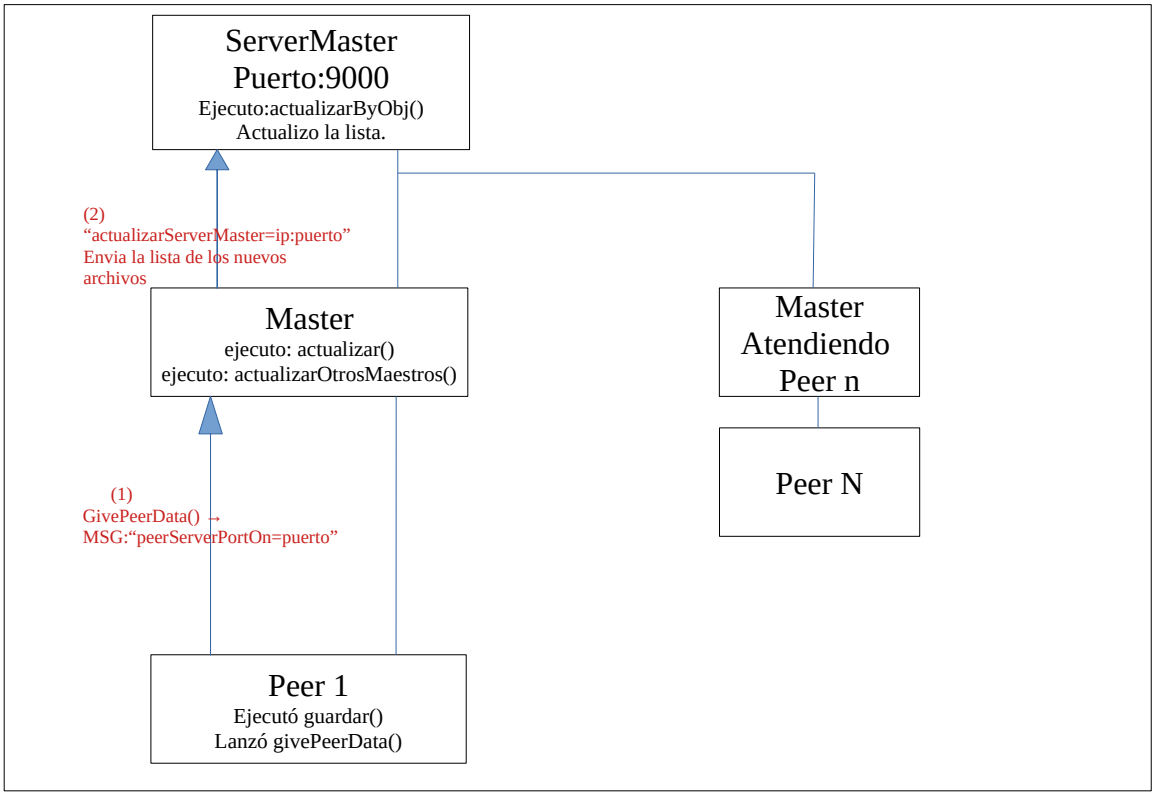
Si un servidor levanta mientras otro ya estaba corriendo (con o sin clientes)



Si le piden buscar un archivo a un worker Master primero actualiza su lista



Si un peer descarga un archivo actualiza su lista y la lista del serverMaster



Critica al modelo:
Cada vez que un nodo modifica, descarga o se da de baja los master deben replicar la modificación a el resto de masters. En sistemas de pocos nodos esto no es un problema, pero a medida que se incrementa la cantidad de nodos y servidores masters esto comienza a tener relevancia. Tener una red completamente centralizada o una red completamente distribuida tiene sus problemas en los dos extremos.

- Red centralizada: al tener una red que depende de un nodo (que maneja una lista de archivos y los peer propietarios de los mismos), estoy creando una debilidad, ya que si el nodo central se cae el sistema deja de funcionar
- Red distribuida: el problema esta en la cantidad de trafico que se genera para conocer a los demás nodos, para solucionar esto, se podría implementar una cantidad de saltos máximos en el flooding que se realiza para conocer nodos y sus respectivos archivos. Para tener la certeza de que un archivo sigue estando en el nodo que recibí la información que lo tenía debería de estar consultando continuamente. Por lo tanto este caso de red tiene tendencias a ser inconsistente si es que no se esta todo el tiempo consultando por cambios en los peer.

Mejora del modelo:
Hacer un modelo híbrido en donde la red es centralizada y distribuida, juntando lo mejor de cada modelo. Cada peer tendrá una lista de Nodos indice e irá probando para ver que nodo esta disponible, cuando encuentre uno, se conecta y podrá realizar peticiones. Los nodos indice responderán las peticiones de sus peer, y luego, entre nodos indice se replicarán su información.

Proceso bancario de extracciones y depósitos

Problema: *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 mseg entre que consulta el saldo actual, y lo actualiza con el nuevo valor. El proceso que realiza una extracción tarda 80 mseg entre que consulta el saldo (y verifica que haya disponible) y lo actualiza con el nuevo valor*

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.

Codigo en: .\TP2\src\main\java\jusacco\TP2\punto2\sinSync

Main en: Banco.java

Salida:

Inicio

```
Juan Despositando $100 en cuenta nro. 1111
Juan Extrayendo $50 en cuenta nro. 1111
Jose Despositando $50 en cuenta nro. 1111
Jose Extrayendo $350 en cuenta nro. 1111
----
Inicio operacion (discDinero resto: 50)
Cuenta: 1111
Tiempo: 15:30:14.516
Dinero disponible: 1000
Autor:Juan
----
----
Inicio operacion (discDinero resto: 350)
Cuenta: 1111
Tiempo: 15:30:14.516
Dinero disponible: 1000
Autor:Jose
----
----
Inicio operacion (addDinero sumo: 100)
Cuenta: 1111
Tiempo: 15:30:14.516
Dinero disponible: 1000
Autor:Juan
----
----
Inicio operacion (addDinero sumo: 50)
Cuenta: 1111
Tiempo: 15:30:14.516 → Los procesos entran todos juntos
Dinero disponible: 1000
Autor:Jose
----
----
Fin operacion (addDinero sumé: 100)
Cuenta: 1111
Tiempo: 15:30:14.556
Dinero disponible: 1150 → Se esperaba 1100
Autor:Juan
----
Juan Despositando $100 en cuenta nro. 1111
```

Continuacion

```
----
Fin operacion (addDinero sumé: 50)
Cuenta: 1111
Tiempo: 15:30:14.556
Dinero disponible: 1150 → Se esperaba 1050
Autor:Jose
----
----
Inicio operacion (addDinero sumo: 100)
Cuenta: 1111
Tiempo: 15:30:14.556
Dinero disponible: 1150
Autor:Juan
----
Jose Despositando $50 en cuenta nro. 1111
----
Inicio operacion (addDinero sumo: 50)
Cuenta: 1111
Tiempo: 15:30:14.556
Dinero disponible: 1150
Autor:Jose
----
----
Fin operacion (addDinero sumé: 100)
Cuenta: 1111
Tiempo: 15:30:14.596
Dinero disponible: 1250 → Entró con 1150, valor correcto.
Autor:Juan
----
----
Fin operacion (discDinero resté: 350)
Cuenta: 1111Tiempo: 15:30:14.596
Dinero disponible: 1250 Entró con 1000, se esperaba 650.
Autor:Jose
----
```

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

Codigo en: .\TP2\src\main\java\jusacco\TP2\punto2\conSync

Main en: Banco.java

Se agrego “synchronized (this.cuenta){}” a cada operación que modificaba los valores de la cuenta.

*Nota: se multiplico *10 los tiempos dados con el fin de poder visualizar la salida de una mejor manera (si se le da tiempos bajos, es altamente probable que el hilo realice muchas operaciones para un solo cliente, sin dejar entrar a otro cliente).*

Salida:

Inicio

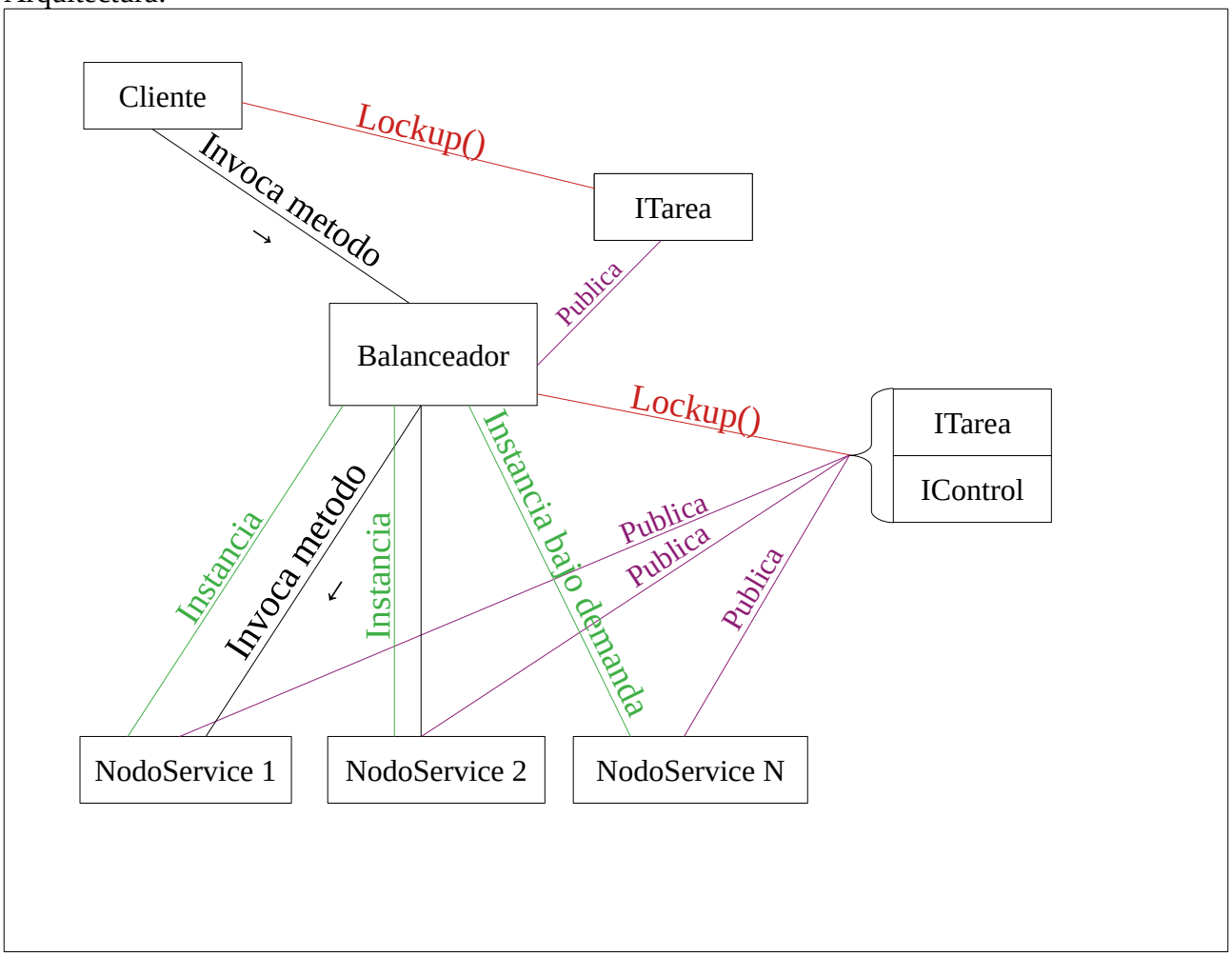
```
Pedro Despositando $200 en cuenta nro. 1111
----
Inicio operacion (addDinero sumo: 200)
Cuenta: 1111
Tiempo: 16:23:15.252
Dinero disponible: 1000
Autor:Pedro
----
----
Fin operacion (addDinero sumé: 200)
Cuenta: 1111
Tiempo: 16:23:15.652
Dinero disponible: 1200
Autor:Pedro
----
Pedro Despositando $200 en cuenta nro. 1111
----
Inicio operacion (addDinero sumo: 200)
Cuenta: 1111
Tiempo: 16:23:15.652
Dinero disponible: 1200
Autor:Pedro
----
----
Fin operacion (addDinero sumé: 200)
Cuenta: 1111
Tiempo: 16:23:16.053
Dinero disponible: 1400
Autor:Pedro
----
Pedro Despositando $200 en cuenta nro. 1111
----
Inicio operacion (addDinero sumo: 200)
Cuenta: 1111
Tiempo: 16:23:16.053
Dinero disponible: 1400
Autor:Pedro
----
----
Fin operacion (addDinero sumé: 200)
Cuenta: 1111
Tiempo: 16:23:16.453
Dinero disponible: 1600
Autor:Pedro
----
```

Continuacion

```
Pedro Extrayendo $500 en cuenta nro. 1111
----
Inicio operacion (discDinero resto: 500)
Cuenta: 1111
Tiempo: 16:23:16.453
Dinero disponible: 1600
Autor:Pedro
----
----
Fin operacion (discDinero resté: 500)
Cuenta: 1111Tiempo: 16:23:17.254
Dinero disponible: 1100
Autor:Pedro
----
----
Inicio operacion (discDinero resto: 150)
Cuenta: 1111
Tiempo: 16:23:17.254
Dinero disponible: 1100
Autor:Juan
----
Juan Extrayendo $150 en cuenta nro. 1111
----
----
Fin operacion (discDinero resté: 150)
Cuenta: 1111Tiempo: 16:23:18.055
Dinero disponible: 950
Autor:Juan
----
Juan Extrayendo $150 en cuenta nro. 1111
----
----
Inicio operacion (discDinero resto: 150)
Cuenta: 1111
Tiempo: 16:23:18.055
Dinero disponible: 950
Autor:Juan
----
----
Fin operacion (discDinero resté: 150)
Cuenta: 1111Tiempo: 16:23:18.856
Dinero disponible: 800
Autor:Juan
----
```

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:
- Simulación de carga de nodos. Creación dinámica de conexiones de clientes y pedidos de atención al servicio publicado.
- 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.
- Definición de umbrales de estado {sin carga, normal, alerta, crítico}
- 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.

Arquitectura:



El cliente solicita al balanceador realizar una tarea, el balanceador debe asignarle a algun nodo de los disponibles la tarea, para ello revisa una lista que posee con los nodos y las tareas que esta realizando, cada nodo cuenta con estados definidos como constantes, para este ejercicio se definio:

SIN_CARGA → 0 Tareas
NORMAL → 3 Tareas
ALERTA → 4 Tareas
CRITICO → 5 Tareas

Si el nodo se encuentra en estado “Sin Carga ” o “Normal” se le asigna la tarea directamente y se registra la misma en la lista.
Si el Nodo se encuentra en “Alerta” si es posible se creara un nuevo nodo, para ello se lee el archivo nodosDinamicos.json que posee las direcciones disponibles para levantar los nuevos NodoService. Si se pudo crear un nuevo Nodo se asigna a este la tarea y se registra en la lista. Si no fue posible la creacion del nuevo Nodo la tarea se asigna al nodo que se encontraba en “Alerta”.
Si el nodo se encuentra en estado “Critico” no puede atender mas tareas, ante lo cual se intenta crear nuevos nodos, si no es posible se espera un determinado tiempo y para que algun nodo se libere y pueda atender nuevamente.
Para la asignacion de las tareas el Balanceador mantiene la lista ordenada y siempre asigna la tarea al nodo que menos tareas este resolviendo.
Una vez asignada la tarea mediante RMI se solicita la resolucion y se espera por una respuesta, la cual luego se envia al cliente por RMI.
Siempre el Balanceador mantiene activos 2 Nodos, si se tiene una cantidad mayor de activos y alguno se encuentra en estado “Sin Carga” se da de baja, para ello utiliza la interface IControl.

Para correr el ejercicio primero se debe ejecutar el balanceador de carga que se encuentra en: .\TP2\src\main\java\jusacco\TP2\punto3\run\BalanceadorInstance.java

Luego, se deben ejecutar los clientes. Estos se pueden ejecutar de dos maneras:

- Individualmente: .\TP2\src\main\java\jusacco\TP2\punto3\run\MultiplesClientes.java
- Multiples: .\TP2\src\main\java\jusacco\TP2\punto3\run\MultiplesClientes.java

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 la máscara, y genere un nuevo archivo con el resultado.

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.

a) Para correr el ejercicio se debe ejecutar el main que se encuentra en:

.\TP2\src\main\java\jusacco\TP2\punto4\centralizado\Main.java

Utiliza una imagen que (por defecto) esta en:

.\TP2\repositorioImagenes\

b & c) Se implemento un sistema de colas:

- queueTrabajos: En esta cola el Servidor maestro coloca todos los trozos de imágenes que previamente corto.
- queueEnProceso: En esta cola los worker colocan los bytes de la imagen con la cual están trabajando.
- queueTerminados: En esta cola los worker ingresan los trozos de imágenes ya procesadas con el filtro.

<u>Funcionamiento:</u>

1. El Maestro inicia, crea las colas, publica sus servicios RMI y queda a la espera.
2. El cliente inicia, invoca la función que publico el Maestro.
3. El Maestro toma el tiempo de inicio, purga las colas, troza las imágenes en una cantidad definida, publica todos los trozos en queueTrabajos e instancia una cantidad definida de Workers. Se quedará esperando que la cantidad de mensajes de la cola “queueTerminados” sea igual a la cantidad de trozos que el realizo.
4. El worker una vez instanciado pregunta la cantidad de trabajos que hay (cantidad de mensajes de la cola “queueTrabajos”, luego pregunta la cantidad de mensajes de la cola “queueTerminados” (este será su punto de corte). Toma un mensaje de la cola de trabajos, compara si los bytes de ese mensaje son iguales a los bytes de cada mensaje de la cola “en proceso”, si no encuentra que haya uno igual, coloca esos bytes en la cola en proceso y empieza a realizar el aplique del filtro. Si encuentra que hay uno igual, toma otro mensaje de la cola “queueTrabajos”. Una vez que la cantidad de mensajes de terminados es igual a trabajos termina su ejecución.
5. El Maestro trae todos los mensajes de queueTerminados y los ordena (gracias a un atributo de la clase Imagen que se utiliza para saber que nro. de trozo es). Une todos los trozos ordenadamente y devuelve al cliente la imagen con el filtro Sobel.

Performance:

Proceso centralizado (Single Thread):

Se realizaron pruebas en el centralizado trozando la imagen y procesando la entera.

Resultados:

Imagen: 6016x4016					
Cantidad de cortes	Sin corte	1500	500	100	10
Cantidad de workers	Delta time	Delta time	Delta time	Delta time	Delta time
Centralizado	7.03	5.68	5.71	5.82	6.55

Imagen: 940x444					
Cantidad de cortes	Sin corte	1500	500	100	10
Cantidad de workers	Delta time	Delta time	Delta time	Delta time	Delta time
Centralizado	0.37	0.44	0.43	0.42	0.39

Proceso multi-thread + colas:

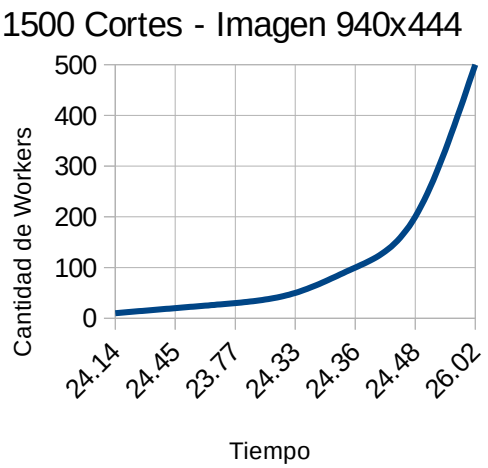
Imagen: 6016x4016				
Cantidad de cortes	1500	500	100	10
Cantidad de workers	Delta time	Delta time	Delta time	Delta time
10	35.07	14.63	6.34	4.42
20	33.14	14.28	6.37	4.46
30	32.39	14.57	6.23	4.32
50	32.28	14.47	6.04	4.41
100	32.09	14.31	6.29	4.95
200	32.36	14.67	8.20	4.58
500	34.11	15.73	13.81	6.97

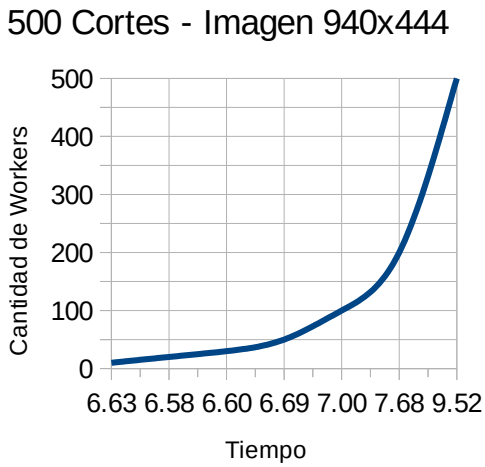
Imagen: 940x444	Delta time	Delta time	Delta time	Delta time
C. workers \ C. cortes	1500	500	100	10
10	24.14	6.63	1.49	0.46
20	24.45	6.58	1.18	0.53
30	23.77	6.60	1.22	0.47
50	24.33	6.69	1.25	0.54
100	24.36	7.00	1.42	0.60
200	24.48	7.68	1.58	0.91
500	26.02	9.52	3.10	2.05

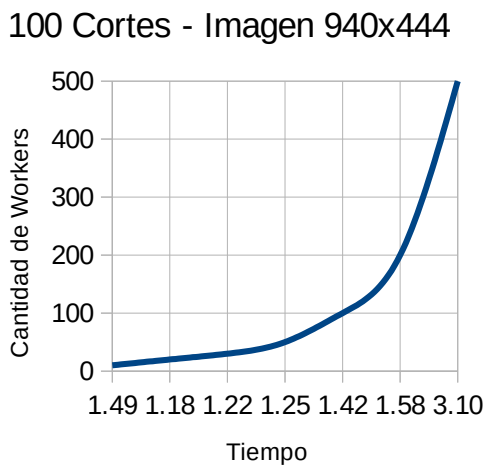
*Todos los valores dentro de Delta time se encuentran en segundos.

Graficas:

Imagen de 940x444 pixeles.







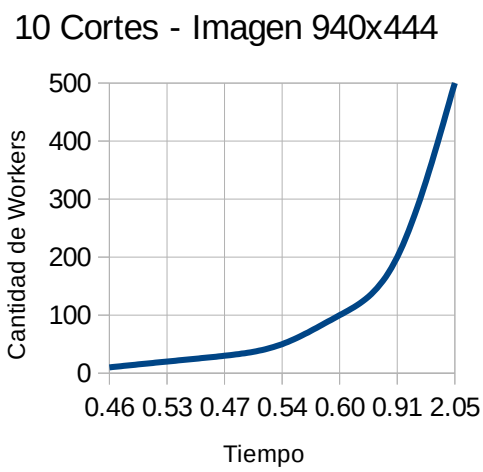


Imagen de 6016x4016 pixeles.