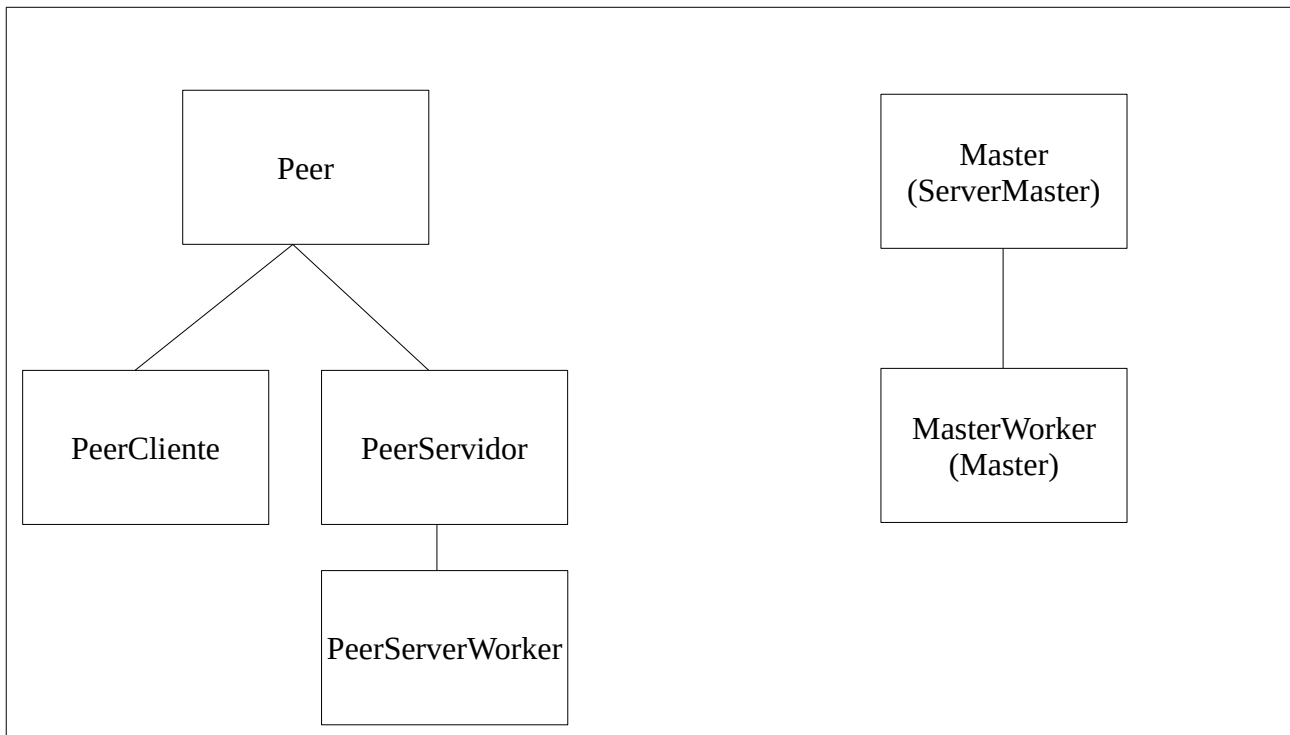


Trabajo Practico N.º 2

Sistemas distribuidos, comunicación y sincronización

Sistema P2P de carga, búsqueda 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

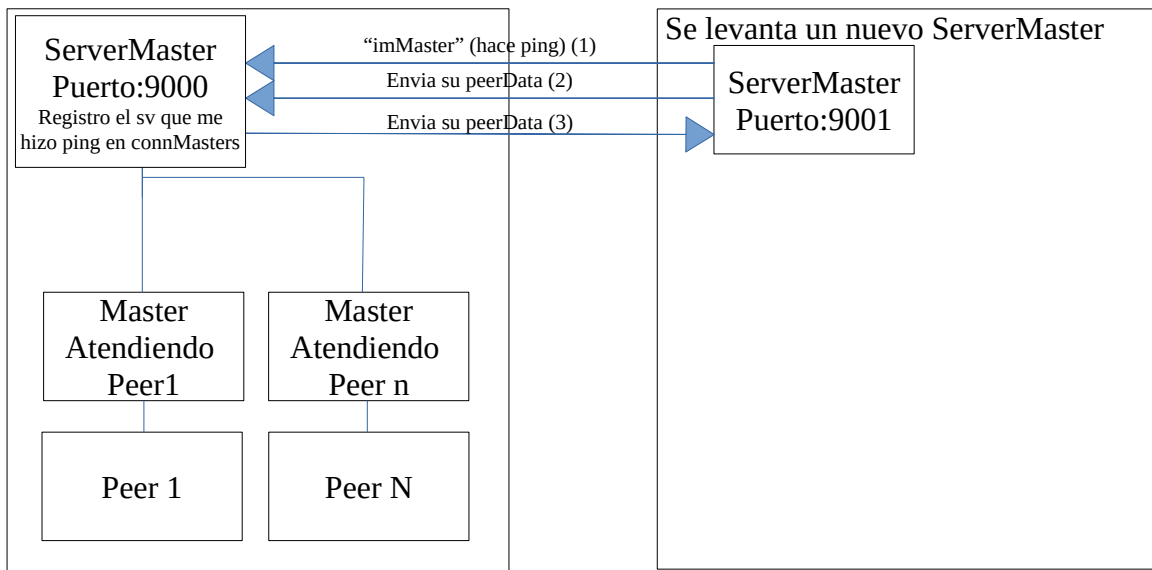
Explicacion Nodos:

- 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.

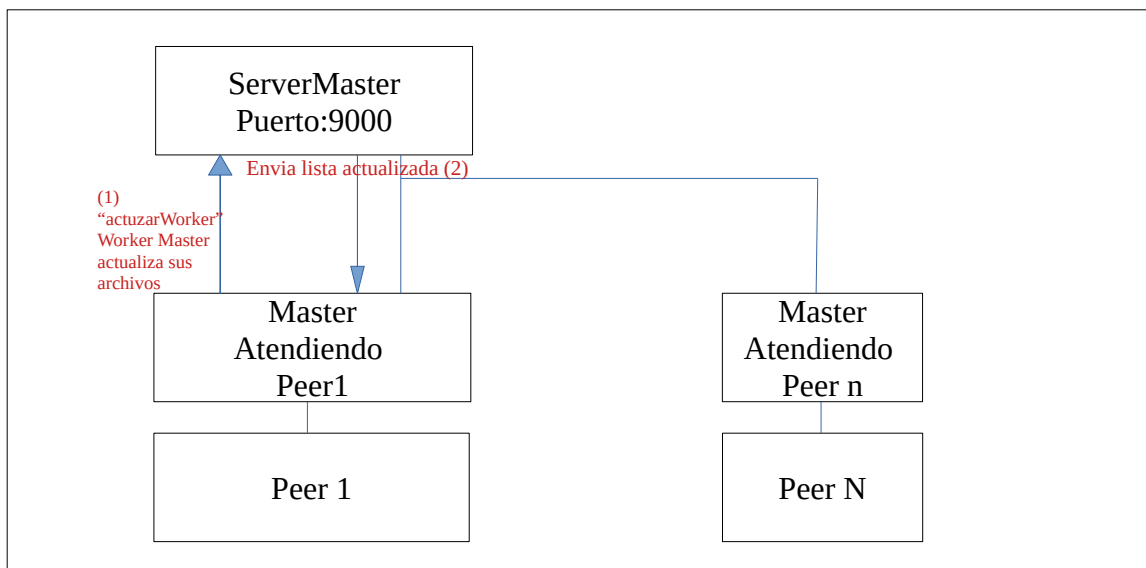
- **ServerMaster:**
 - Es el encargado de registrar sus configuraciones en un archivo (config.txt), esto lo hace mediante una clase (Config.java) que tiene la responsabilidad de leer el archivo de configuraciones y escribir nuevas configuraciones. En caso de encontrar Masters ya registrados lo notifica 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

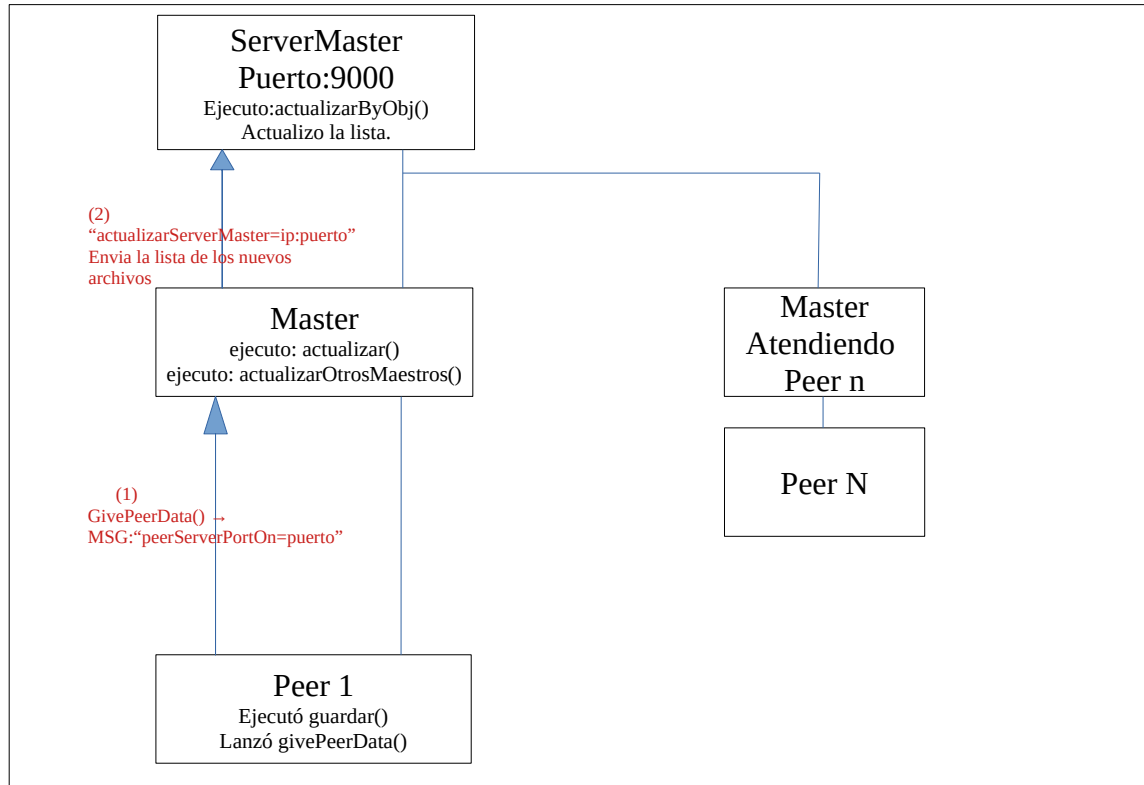
Si un servidor levanta mientras otro ya estaba corriendo (con o sin clientes)



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.

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

Main en: Banco.java

Salida:

Inicio	Continuacion
Juan Despositando \$100 en cuenta nro. 1111	----
Juan Extrayendo \$50 en cuenta nro. 1111	Fin operacion (addDinero sumé: 50)
Jose Despositando \$50 en cuenta nro. 1111	Cuenta: 1111
Jose Extrayendo \$350 en cuenta nro. 1111	Tiempo: 15:30:14.556
----	Dinero disponible: 1150 → Se esperaba 1050
Inicio operacion (discDinero resto: 50)	Autor:Jose
Cuenta: 1111	----
Tiempo: 15:30:14.516	----
Dinero disponible: 1000	Inicio operacion (addDinero sumo: 100)
Autor:Juan	Cuenta: 1111
----	Tiempo: 15:30:14.556
----	Dinero disponible: 1150
Inicio operacion (discDinero resto: 350)	Autor:Juan
Cuenta: 1111	----
Tiempo: 15:30:14.516	Jose Despositando \$50 en cuenta nro. 1111
Dinero disponible: 1000	----
Autor:Jose	Inicio operacion (addDinero sumo: 50)
----	Cuenta: 1111
----	Tiempo: 15:30:14.556
Inicio operacion (addDinero sumo: 100)	Dinero disponible: 1150
Cuenta: 1111	Autor:Jose
Tiempo: 15:30:14.516	----
Dinero disponible: 1000	----
Autor:Juan	Fin operacion (addDinero sumé: 100)
----	Cuenta: 1111
----	Tiempo: 15:30:14.596
Inicio operacion (addDinero sumo: 50)	Dinero disponible: 1250 → Entró con 1150,
Cuenta: 1111	valor correcto.
Tiempo: 15:30:14.516 → Los procesos entran	Autor:Juan
todos juntos	----
Dinero disponible: 1000	----
Autor:Jose	Fin operacion (discDinero resté: 350)
----	Cuenta: 1111Tiempo: 15:30:14.596
----	Dinero disponible: 1250 Entró con 1000, se
Fin operacion (addDinero sumé: 100)	esperaba 650.
Cuenta: 1111	Autor:Jose
Tiempo: 15:30:14.556	----
Dinero disponible: 1150 → Se esperaba 1100	
Autor:Juan	

Juan Despositando \$100 en cuenta nro. 1111	

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).*

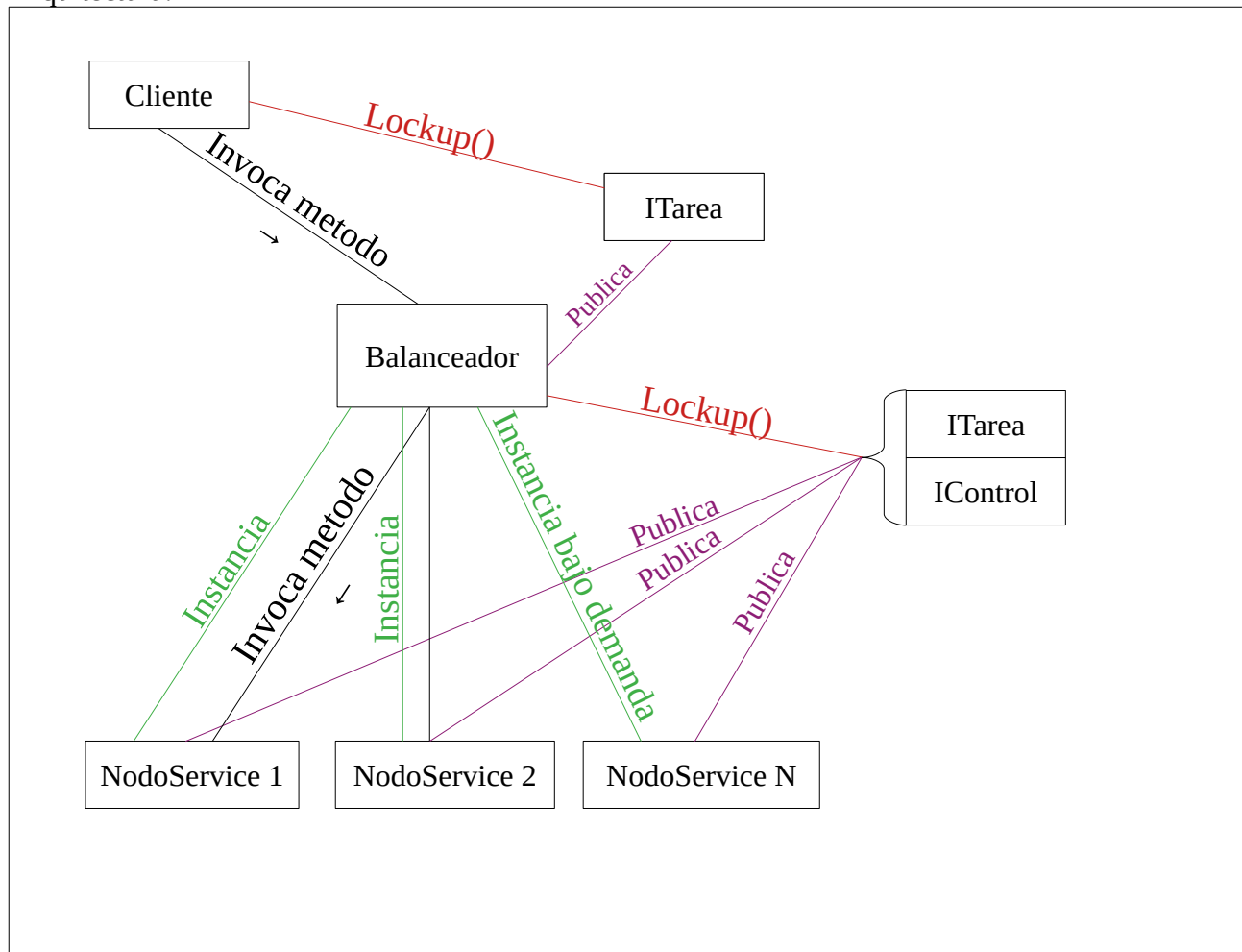
Salida:

Inicio	Continuacion
Pedro Despositando \$200 en cuenta nro. 1111	Pedro Extrayendo \$500 en cuenta nro. 1111
----	----
Inicio operacion (addDinero sumo: 200)	Inicio operacion (discDinero resto: 500)
Cuenta: 1111	Cuenta: 1111
Tiempo: 16:23:15.252	Tiempo: 16:23:16.453
Dinero disponible: 1000	Dinero disponible: 1600
Autor:Pedro	Autor:Pedro
----	----
Fin operacion (addDinero sumé: 200)	Fin operacion (discDinero resté: 500)
Cuenta: 1111	Cuenta: 1111Tiempo: 16:23:17.254
Tiempo: 16:23:15.652	Dinero disponible: 1100
Dinero disponible: 1200	Autor:Pedro
Autor:Pedro	----
----	----
Pedro Despositando \$200 en cuenta nro. 1111	Inicio operacion (discDinero resto: 150)
----	Cuenta: 1111
Inicio operacion (addDinero sumo: 200)	Tiempo: 16:23:17.254
Cuenta: 1111	Dinero disponible: 1100
Tiempo: 16:23:15.652	Autor:Juan
Dinero disponible: 1200	----
Autor:Pedro	Juan Extrayendo \$150 en cuenta nro. 1111
----	----
Fin operacion (addDinero sumé: 200)	Fin operacion (discDinero resté: 150)
Cuenta: 1111	Cuenta: 1111Tiempo: 16:23:18.055
Tiempo: 16:23:16.053	Dinero disponible: 950
Dinero disponible: 1400	Autor:Juan
Autor:Pedro	----
----	Juan Extrayendo \$150 en cuenta nro. 1111
Pedro Despositando \$200 en cuenta nro. 1111	----
----	Inicio operacion (discDinero resto: 150)
Inicio operacion (addDinero sumo: 200)	Cuenta: 1111
Cuenta: 1111	Tiempo: 16:23:18.055
Tiempo: 16:23:16.053	Dinero disponible: 950
Dinero disponible: 1400	Autor:Juan
Autor:Pedro	----
----	Fin operacion (discDinero resté: 150)
Fin operacion (addDinero sumé: 200)	Cuenta: 1111Tiempo: 16:23:18.856
Cuenta: 1111	Dinero disponible: 800
Tiempo: 16:23:16.453	Autor:Juan
Dinero disponible: 1600	----
Autor:Pedro	

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 descríballo. 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:



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