Programación Distribuida y Tiempo Real

- Trabajo Final -

Lautaro Pastorino

Facultad de Informática - Universidad Nacional de La Plata

2021

Índice

Introducción 3
Desarrollo 3
Solución de la práctica 4 en EDiPru 4
    Ejercicio 1 4
    Ejercicio 2 6
    Ejercicio 3 7
Comparación con solución en Docker 10
Mejoras posibles 10
Conclusión 11

Introducción

El objetivo del trabajo final es generar un ambiente de pruebas automatizado para JADE sobre máquinas virtuales completas.

La implementación debe ser capaz de recibir como parámetro la cantidad de máquinas virtuales que debe iniciar. Cada máquina virtual creada debe ejecutar un contenedor JADE, por lo que tienen que poder comunicarse entre sí.

Como siguiente paso, la implementación debe poder recibir como parámetro archivos de código de agentes JADE para compilar y ejecutar en el sistema distribuido.

La solución debe implementarse utilizando la herramienta Vagrant de creación de ambientes de desarrollo.

Desarrollo

La herramienta Vagrant provee una solución simple y fácil de replicar a la hora de crear máquinas virtuales para construir un ambiente de desarrollo y prueba de aplicaciones. Todas las configuraciones necesarias se realizan dentro de un único archivo llamado *Vagrantfile*. A partir de este archivo, Vagrant puede iniciar las máquinas virtuales que allí se indican de la misma forma cada vez. De esta manera, podemos replicar un ambiente de trabajo en múltiples máquinas reales sólo compartiendo un archivo.

Vagrant permite en un mismo *Vagrantfile* crear múltiples máquinas virtuales. Es más, se pueden crear un número dinámico de máquinas virtuales, recibiendo dicho número como parámetro. Dado que el *Vagrantfile* está escrito en el lenguaje de programación Ruby, podemos aprovechar las capacidades del lenguaje para realizar el siguiente ciclo:

```
for i in range(0..num_maquinas_virtuales) do
        crear maquina virtual i
end
```

Ésta es la solución clásica a la creación de múltiples máquinas virtuales en Vagrant. El problema con esta implementación radica en que es muy difícil configurar cada máquina virtual creada de forma diferente al resto, por eso, se determinó que la mejor solución era crear de forma dinámica todo el *Vagrantfile* escribiendo una entrada por cada máquina virtual requerida.

EDiPru (Entorno Distribuido de Pruebas) es una herramienta de línea de comandos escrita en Python que se encarga de obtener las necesidades del usuario y crear un *Vagrantfile* a partir de ellas. En el repositorio de EDiPru existe un documento llamado “Manual EDiPru” que puede ser consultado para aprender a utilizar la herramienta.

Solución de la práctica 4 en EDiPru

Para poder utilizar la herramienta EDiPru se debe clonar el repositorio encontrado en https://github.com/lautaropastorino/edipru.

1) Programar un agente para que periódicamente recorra una secuencia de computadoras y reporte al lugar de origen:

   a) El tiempo total del recorrido para recolectar la información.
   b) La carga de procesamiento de cada una de ellas.
   c) La cantidad de memoria total disponible.
   d) Los nombres de las computadoras.

   Comente la relación entre este posible estado del sistema distribuido y el estado que se obtendría implementando el algoritmo de instantánea.

   Para probar el experimento primero se debe crear e iniciar el entorno de pruebas. El entorno que se creará en este caso estará compuesto por 3 máquinas virtuales: una principal que ejecutará el Main-Container de JADE y dos secundarias. Para crear este entorno se debe utilizar el comando

   > $ python generator.py -v 3 -s

   o si el usuario prefiriese contar con la interfaz gráfica en la máquina virtual principal y así poder ver el comportamiento del agente, se debe utilizar el comando

   > $ python generator.py -v -g -s

   Para el último caso, se deberá iniciar manualmente el Main-Container de JADE en la máquina principal. Para esto se debe abrir una terminal dentro de la máquina virtual y escribir el comando

   > $ java -cp /jade/lib/jade.jar jade.Boot -gui

   Si todo funciona correctamente, después de unos segundos debería aparecer una interfaz donde se pueden ver los contenedores y los agentes.

   En el caso de que no se haya requerido la interfaz gráfica, el Main-Container se iniciará automáticamente en la máquina principal.

   Antes de poder ejecutarlo, se debe compilar el agente. Para realizar la compilación

primero se debe decidir en cuál de las máquinas virtuales se ejecutará (es posible hacerlo en cualquiera de las tres). Luego, se debe copiar el código fuente del agente (archivo ejercicio1/Agente.java) a la carpeta sincronizada de dicha máquina virtual (mainFS/ si se decide ejecutarlo en la máquina virtual principal, vm1FS/ si se decide ejecutarlo en la máquina virtual 1 o vm2FS/ si se decide ejecutarlo en la máquina virtual 2). De esta forma, se podrá acceder al código desde la máquina virtual elegida. Finalmente, se debe ingresar a dicha máquina virtual con el comando

> $ vagrant ssh [nombre mv]

y compilar el agente con el comando

> $ javac -classpath /jade/lib/jade.jar -d /jade/classes /[nombre mv]/Agente.java

El agente necesita que se creen 5 containers antes de ser ejecutado. Para esto se debe ejecutar 5 veces el comando

> $ java -cp /jade/lib/jade.jar jade.Boot -container -host 192.168.50.4 &

en cualquiera de las 3 máquinas virtuales iniciadas. Para poder correr dicho comando, primero se debe ingresar a la máquina virtual deseada. Con el comando

> $ vagrant ssh vm1

por ejemplo, se puede acceder a la máquina virtual 1 a través de SSH. Una vez dentro podremos ejecutar el comando anterior. El agente funcionará aunque los containers se creen en distintas máquinas virtuales. No se debe olvidar del símbolo & para que los contenedores se ejecuten en modo *background* y poder seguir utilizando la terminal.

Una vez iniciados los 5 contenedores, se podrá ingresar a la máquina virtual donde se compiló el agente con el comando

> $ vagrant ssh [nombre mv]

y ejecutarlo con el comando

> $ java -cp /jade/lib/jade.jar:/jade/classes jade.Boot -container -host 192.168.50.4 -agents ej1:ejercicio1.Agente

El agente iniciará el recorrido sobre los contenedores automáticamente, guardando antes el valor dado por *System.CurrentTimeMillis()* para luego poder medir el tiempo que tomó el recorrido. En cada contenedor visitado, el agente iniciará dos procesos Unix ejecutando dos comandos: *free -h* (brinda información sobre el uso de la memoria RAM de la computadora) y *cat /proc/cpuinfo* (imprime en pantalla los contenidos del archivo *cpuinfo*, el cual contiene información sobre el estado de la CPU). Una vez obtenida la información, se actualiza un *String* con esos datos y se pasa al siguiente contenedor.

Al llegar al mismo contenedor del que partió, el agente tomará el valor actual dado por *System.CurrentTimeMillis()* y calculará el tiempo total del recorrido. Luego imprimirá en pantalla la información recolectada en cada paso del recorrido y esperará 10 segundos antes de comenzar de vuelta.

Esta implementación podría verse como un algoritmo de instantánea imperfecto. El agente se encarga de generar un reporte del estado global del sistema recorriendo cada punto del mismo recolectando información. En un algoritmo de instantánea perfecto podríamos obtener el estado de cada parte del sistema en el mismo instante de tiempo, pero esto no es posible en una implementación real en la que no hay un reloj común a todas las partes.
En nuestra implementación transcurre demasiado tiempo entre la lectura del estado de una parte del sistema y otra. Esto puede generar que el estado global obtenido al finalizar el recorrido no sea fiel o consistente con el estado real del sistema. La precisión que necesitemos sobre esta información es lo que determinará la utilidad de nuestro algoritmo como algoritmo de instantánea.

El algoritmo de instantánea de Chandy-Lamport, a diferencia de nuestra implementación, contempla el caso en que mensajes anteriores que todavía no llegaron a su destino, afecten al estado del sistema. Si se detecta que un mensaje previo a la toma de la instantánea llega luego de haber esta sido tomada, se lo reenvía al proceso encargado de iniciar el algoritmo para que pueda hacer las modificaciones pertinentes al estado global del sistema,

2) Programe un agente para que calcule la suma de todos los números almacenados en un archivo de una computadora que se le pasa como parámetro. Comente cómo se haría lo mismo con una aplicación cliente/servidor. Comente qué pasaría si hubiera otros sitios con archivos que deben ser procesados de manera similar.

   En este caso, con un entorno de dos máquinas virtuales, una principal y una secundaria, alcanza para mostrar el funcionamiento del agente. Con el comando

   > $ python generator.py -v 2 -s

   se generará un nuevo Vagrantfile que especifica dos máquinas virtuales (es conveniente trabajar sobre el mismo directorio que se utilizó en el ejercicio 1 para reutilizar las máquinas virtuales ya instaladas y configuradas).

   El siguiente paso consiste en crear el archivo que se debe leer. Puede llevar cualquier nombre y su contenido debe ser una serie de números enteros disponiéndolos uno por fila. Por ejemplo:

   2
   4
   78
   223

5

El archivo debe ser copiado a la carpeta /mainFS para que pueda ser accedido desde la máquina virtual principal.

Luego, se debe compilar el agente. Para esto, se debe copiar el código fuente (archivo AgenteEj2.java) en la carpeta /vm1FS. Luego con el comando

> $ vagrant ssh vm1

se accede a la máquina virtual 1 a través de SSH y se compila el agente con el comando

> $ javac -classpath /jade/lib/jade.jar -d /jade/classes /vm1FS/AgenteEj2.java

Finalmente, se ejecuta el agente con el comando

> $ java -cp /jade/lib/jade.jar:/jade/classes jade.Boot -container -host 192.168.50.4 -agents ‘ej2:ejercicio2.AgenteEj2(Main-Container /mainFS/[nombre archivo])’

Si se copia y pega el comando anterior, es posible que haya un error con las comillas. Para solucionarlo, se debe volver a escribir las comillas manualmente.

El código original entregado en la práctica 4 fue modificado ya que, como el agente migra a otro contenedor, nunca se ve el resultado obtenido. El nuevo agente va a leer el archivo y vuelve al contenedor inicial para mostrar el resultado.

La ejecución del agente consiste en migrar a la computadora recibida como parámetro e intentar leer los contenidos del archivo indicado. Si el archivo no existiera se levantaría una excepción y la ejecución finalizaría.

Para realizar la misma tarea en una aplicación cliente/servidor deberíamos, desde el cliente, enviar una request al servidor indicándole el nombre del archivo sobre el que queremos realizar los cálculos. El servidor se encargaría de buscar el archivo, realizar las sumas y devolver el resultado al cliente.

Si debiéramos realizar el mismo procedimiento en distintos sitios, utilizando un modelo cliente/servidor, deberíamos implementar en cada sitio un servidor que pueda responder a nuestras consultas. En cambio, si utilizamos un modelo de migración de código, el mismo proceso se encargaría de migrar y obtener por sí mismo la información requerida. Y dado que los archivos son procesados de manera similar, no aumentaría en complejidad la programación del agente a medida que aumenta la cantidad de sitios que se deben visitar.

3) Defina e implemente con agentes un sistema de archivos distribuido similar al de las prácticas anteriores.

a.- Debería tener como mínimo la misma funcionalidad, es decir las operaciones

(definiciones copiadas aquí de la práctica anterior):

leer: dado un nombre de archivo, una posición y una cantidad de bytes a leer, retorna 1) la cantidad de bytes del archivo pedida a partir de la posición dada o en caso de haber menos bytes, se retornan los bytes que haya y 2) la cantidad de bytes que efectivamente se retornan leídos.

escribir: dado un nombre de archivo, una cantidad de bytes determinada, y un buffer a partir del cual están los datos, se escriben los datos en el archivo dado. Si el archivo existe, los datos se agregan al final, si el archivo no existe, se crea y se le escriben los datos. En todos los casos se retorna la cantidad de bytes escritos.

Comandos para el cliente:
Escritura: `java -cp /jade/lib/jade.jar:/jade/classes/ jade.Boot -container -host 192.168.50.4 -agents 'FSCA:ejercicio3.FileServerClientAgent(write docker.pdf Container-5 30000000)'`
Lectura: `java -cp /jade/lib/jade.jar:/jade/classes/ jade.Boot -container -host 192.168.50.4 -agents 'FSCA:ejercicio3.FileServerClientAgent(read docker.pdf Container-5 0 30000000)'`
Copy: `java -cp /jade/lib/jade.jar:/jade/classes/ jade.Boot -container -host 192.168.50.4 -agents 'FSCA:ejercicio3.FileServerClientAgent(copy docker.pdf Container-5)'`

Para este experimento se volverá a utilizar el Vagrantfile definido en el primer ejercicio. Para obtenerlo nuevamente, se utiliza el comando

> $ python generator.py -v 3 -s

El siguiente paso es compilar el agente servidor y el agente cliente. Primero se deben copiar los códigos fuentes a las carpetas sincronizadas de las máquinas virtuales. El código fuente del servidor se deberá copiar a /mainFS y el código fuente del cliente se deberá copiar a /vm1FS y a /vm2FS. Luego, ingresando a las máquinas virtuales mediante SSH y ejecutando el comando

> $ javac -classpath /jade/lib/jade.jar -d /jade/classes /[mv]FS/[nombre agente].java

se realiza la compilación de los archivos en las máquinas correspondientes.

Posteriormente, se debe ingresar a la máquina virtual principal y ejecutar el agente servidor con el comando

> $ java -cp /jade/lib/jade.jar:/jade/classes jade.Boot -container -host localhost -agents FSA:ejercicio3.FileServerAgent &

Finalmente, se deben crear algunos containers. Se pueden crear la cantidad que se quiera en las máquinas virtuales deseadas ingresando a ellas y ejecutando el comando

> $ java -cp /jade/lib/jade.jar jade.Boot -container -host 192.168.50.4 &

El agente cliente puede ser ejecutado en cualquiera de las dos máquinas virtuales no principales con los comandos detallados anteriormente.

El agente servidor cuando visita por primera vez un contenedor crea una carpeta llamada ~/[nombre contenedor]FS. Es allí donde se realizan las escrituras y desde donde se realizan las lecturas.

Para realizar una escritura, el cliente trata de buscar el archivo dentro de la carpeta ~/clientFS/ y por eso se debe copiar manualmente el archivo que se quiere escribir allí (si la carpeta no existe se la debe crear con ese nombre exacto).

Después de iniciarse, el agente servidor se bloquea esperando un mensaje de algún cliente mediante la instrucción *blockingReceive()*. Luego, a partir de los datos recibidos en el mensaje, determina si tiene que migrar a otro container para cumplir con la request. Una vez realizada o no la migración, el servidor verifica que exista la carpeta *contenedor-iFS/* que hará las veces de filesystem local del contenedor i. Finalmente, se determina qué acción hay que realizar (lectura o escritura de un archivo) y se la realiza. Luego de completar el pedido, el servidor se vuelve a bloquear a la espera de otra request.

Al comenzar su ejecución, el agente cliente crea la carpeta que simulará el filesystem local si esta no existe. Luego lee los argumentos recibidos por línea de comandos y determina qué acción requiere el usuario:

- Lectura: el cliente genera mensajes al servidor enviándole el archivo que se quiere leer, el contenedor en donde este archivo está ubicado, la posición inicial desde donde se debe leer el archivo y la cantidad de bytes que se deben leer. Los mensajes se seguirán generando hasta que el servidor devuelva 0 bytes leídos, indicando que el archivo llegó a su fin, o hasta que se complete la cantidad de bytes que se debían leer. La cantidad de bytes que se requiere en cada mensaje está determinada por la constante *BUFFERSIZE*.

- Escritura: luego de verificar que el archivo exista en el filesystem local, se lo lee y se determina cuántos mensajes de *BUFFERSIZE* bytes serán necesarios para escribir la cantidad de bytes indicados en el servidor. Finalmente se itera esa cantidad de veces enviando en cada iteración una request de escritura al servidor.

- Copia: el cliente comienza la comunicación con el servidor enviando una request de lectura sobre un archivo, si esa lectura es exitosa (porque el archivo existe), se crea un archivo en el filesystem local y se escriben los bytes leídos. Luego se envía al servidor una request de escritura con los bytes leídos para generar la copia remota. Este proceso se repite hasta que se finaliza la lectura del archivo remoto original.

b.- Implemente un agente que copie un archivo de otro sitio del sistema distribuido

en el sistema de archivos local y genere una copia del mismo archivo en el sitio donde está originalmente. Compare esta solución con la de los sistemas cliente/servidor de las prácticas anteriores.

Nuestra implementación implica un agente servidor y un agente cliente que se comunican a través de mensajes ACL, por lo que el funcionamiento es muy similar a las implementaciones, donde el cliente realiza los pedidos de escritura o lectura al servidor. Sin embargo, si la implementación hubiese sido con sólo un agente, éste actuaría como cliente y servidor, ya que tendría las instrucciones que tiene que realizar, pero se movería al container donde se encuentran los datos y a donde se deben escribir, pero escribiendo una parte en cada migración, de la misma manera que se realiza con buffers. También se podría tomar a los distintos containers con sus filesystems como el servidor en sí, y el agente como el cliente.
Esta implementación es claramente distinta al resto de los sistemas cliente/servidor de las prácticas anteriores, ya que en las implementaciones anteriores el cliente tiene un mínimo de conocimiento, pero delega toda la lógica al servidor y sólo envía o recibe mensajes, en cambio en ésta el cliente tiene que conectarse al enviroment, buscar los datos, leerlos y escribirlos, moviéndose por el enviroment por su cuenta.

Comparación con solución en Docker

Al utilizar el contenedor docker sucede que el programa distribuido es ejecutado en una arquitectura no distribuida. En cambio, utilizando varias máquinas virtuales completas conectadas entre sí a través de una red, podemos simular un ambiente distribuido real.

Una ventaja de utilizar Vagrant por sobre docker radica en que Vagrant trabaja sobre un proveedor de máquinas virtuales como pueden ser VirtualBox o VMware. Esto permite que las máquinas virtuales puedan ser controladas y configuradas a través de la interfaz gráfica que tanto VirtualBox como VMware proveen. Además, existe la opción de ejecutar en las máquinas virtuales un sistema operativo con interfaz gráfica y no sólo desde la línea de comandos.

Docker es una solución más liviana y que consume menos recursos ya que no trabaja con máquinas virtuales completas.

Mejoras posibles

En vagrant es posible crear y distribuir “boxes” (cajas). Una box es un formato de paquete que contiene un entorno Vagrant ya configurado. Sería interesante y productivo crear una box que ya contenga instalado el paquete openjdk-8-jdk y el entorno de escritorio utilizado para la interfaz gráfica. De esta manera el usuario de EDiPru podría empezar a trabajar una vez que se descarga la box, sin tener que instalar nada posteriormente.

La segunda parte de la consigna indicaba que la herramienta debía ser capaz de recibir como parámetro archivos de código de agentes, compilarlos y ejecutarlos automáticamente una vez que se inicien las máquinas virtuales. Esto fue implementado en la solución para el sistema operativo Linux pero actualmente no funciona en el sistema operativo Windows.

Otro aspecto a mejorar consistiría en encontrar la forma de utilizar la menor cantidad de recursos de computación para que más gente pueda utilizar la herramienta fácilmente. Esto podría encararse tratando de reemplazar la box que se utiliza actualmente con una propia más optimizada por ejemplo, o probando qué sucede si se le reserva menos memoria RAM a las máquinas virtuales.

Conclusión

EDiPru provee una solución simple y fácil de usar al problema de construir un entorno distribuido de pruebas para programas multiagente JADE. La herramienta Vagrant resulta muy útil para este caso de uso ya que provee flexibilidad y permite la posibilidad de configurar el entorno a gusto del usuario.