UNIVERSIDAD
NACIONAL
DE LA PLATA

Práctica nro. 4

Docentes: Fernando Tinetti, Gastón Marón, Alan Castelli, Julián Delekta.

Materia: Programación Distribuida y Tiempo Real.

Integrantes: Juliana Delle Ville, Marianela Diaz Catán.

Índice

Ejercicio 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.

AgenteMovil2

Esta clase define el comportamiento del agente.

1. **Métodos:**
    a. `setup()`: agrega los contenedores a un arrayList, con el objetivo de recorrerlos luego mediante esta estructura. Comienza a calcular el tiempo para determinar cuanto se tardo en la recopilación de la información. Inicia el proceso de recorrido, moviendo el agente al primer contenedor mediante la invocación del método `doMove();`
    b. `afterMove()`: define el comportamiento del agente en cada contenedor luego de la migración. En cada contenedor recopila el nombre de las computadoras/contenedores, la información de carga de procesamiento, la memoria en uso y la memoria disponible, almacenando los datos para luego informarlos al finalizar el recorrido. Si el contenedor es el último de la lista, luego de recopilar los datos propios del contenedor, el agente migra nuevamente el contenedor inicial e informa el tiempo total del recorrido y los datos recopilados en todos los contenedores.

Comandos de ejecución

Para ejecutar este ejemplo se deben ejecutar los siguientes comandos:

1. Posicionarse dentro del directorio “Entrega 4/1”

2. Compilación:
```
javac -classpath lib/jade.jar -d classes AgenteMovil2.java
```

3. GUI:
```
java -cp lib/jade.jar:classes jade.Boot -gui
```

4. Agente:
```
java -cp lib/jade.jar:classes jade.Boot -gui -container -host localhost
-agents Ejercicio1:AgenteMovil2
```

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

Dado que esta solución, recorre cada nodo uno a uno, y almacena la información de cada nodo en distintas instancias de tiempo, los estados que se informen al finalizar los recorridos, probablemente no corresponden a un estado coherente en el sistema.

La información recopilada para esta solución, no refleja el estado real de cada nodo al finalizar el recorrido de la recopilación de datos, esto se debe a la naturaleza misma de los sistemas distribuidos, los cuales no son sistemas estáticos, sino que su estado muta constantemente.

Por otro lado, implementar un algoritmo de instantánea, implicaría una foto de todo el sistema en un momento específico, esto permitiría saber de forma más fidedigna, cuál es el estado real de todo un sistema, en un tiempo dado.

Ejercicio 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.

AgenteSuma

Esta clase define el comportamiento del agente.

1. **Parámetros**:
    a. `fileName`: archivo a leer.
    b. `computadora`: computadora o contenedor, donde se encuentra el archivo a procesar.
2. **Métodos**:
    a. `setup()`: válida que se hayan recibido los parámetros requeridos, luego mediante el método `doMove()`, migra el agente hacia el contenedor destino o `computadora`, donde se encuentra el archivo que debe procesar.
    b. `afterMove()`: agrega un behaviour con el método `addBehaviour()`, el cual tendrá la lógica de acción del agente. Para esto, define una clase interna `CalculateSumBehaviour`, la cual define en el método action() lee el archivo y realiza la suma de los número que contiene. Al finalizar informa en consola, el valor de la suma de todos los números del archivo.

Comandos de ejecución

Para ejecutar este ejemplo se deben ejecutar los siguientes comandos:

1. Posicionarse dentro del directorio “Entrega 4/2”

2. Compilación:

```
javac -classpath lib/jade.jar -d classes AgenteMovil2.java
```

3. GUI:

```
java -cp lib/jade.jar:classes jade.Boot -gui
```

4. Agente:

```
java -cp lib/jade.jar:classes jade.Boot -gui -container -host localhost
-agents "agenteMovil:AgenteMovil('data.txt','Main-container')"
```

Comente cómo se haría lo mismo con una aplicación cliente/servidor.

Para una aplicación Cliente-Servidor, se deberían realizar las siguientes operaciones/tareas:

1. El cliente realiza una petición HTTP al servidor.
2. El servidor realiza la búsqueda del archivo.
3. El servidor lee el archivo.
4. El servidor envía la respuesta al cliente.
5. El cliente recibe la respuesta e imprime el resultado o lo guarda.

Comente qué pasaría si hubiera otros sitios con archivos que deben ser procesados de manera similar

Si se encontrase varios sitios con partes del archivo a buscar, existen dos esquemas posibles:

1. No existe un balanceador:
   - 1.1. El cliente tiene una lista con los servidores a consultar.
   - 1.2. Mientras no termine de leer la lista
     - 1.2.1. Lee el servidor.
     - 1.2.2. Realiza una petición HTTP al servidor.
     - 1.2.3. El servidor busca el fragmento.
     - 1.2.4. El servidor lee el fragmento.
     - 1.2.5. El servidor envía la respuesta con el fragmento al cliente.
     - 1.2.6. El cliente guarda el fragmento.
   - 1.3. El cliente realiza el merge de todos los fragmentos
   - 1.4. El cliente lee el archivo e imprime el resultado o lo guarda.

2. Existe balanceador:
   - 2.1. El cliente realiza la petición al balanceador
   - 2.2. El balanceador conoce los servidores a preguntar, por lo que realiza las peticiones HTTP correspondientes a los servidores.
   - 2.3. El balanceador realiza un merge de los fragmentos
   - 2.4. El balanceador devuelve el resultado al cliente
   - 2.5. El cliente imprime el resultado o lo guarda

Ejercicio 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.

Class AgenteMovil

Define el comportamiento del agente:

1. **Parámetros**:
    a. `mode`: indica el tipo de operación a realizar. Acepta modo “r” (lectura) y “w” (escritura).
    b. `sourcePath`: path del sistema de archivos local.
    c. `targetHost`**:** contendor destino.
    d. `targetPath`: path destino.
    e. `position`: posición de inicio de lectura.
    f. `amount`**:** cantidad de bytes a leer o escribir.
2. **Métodos**:
    a. `setup()`: válida recibir para cada modo los parámetros requeridos.
        i. Modo “r”, es obligatorio recibir `sourcePath`, `targetHost`, `targetPath`, `position` y `amount`.
        ii. Modo “w”, es obligatorio recibir `sourcePath`, `targetHost`, `targetPath` y `amount`.

    Por otro lado, este método determina para mode “r” moverse al contenedor destino, donde se encuentra el archivo que se desea leer, y para mode “w” primero, mediante el método de clase `FTP.read()`, copia en la variable de instancia `content`, el contenido del archivo local, que se desea escribir en el contenedor destino, o “servidor” y luego indica al agente que migre.
    b. `afterMove()`: indica que acciones debe realizar el agente luego de migrar.
        i. Modo “r”, valida en qué contenedor se halla:
            1. Si el contenedor no es el origen, significa que está en “servidor”, por lo tanto almacena el la variable de instancia content, como resultado de la invocación del método de clase `FTP.read()`, el archivo indicado en el parámetro `targetPath`

2. Si el contenedor es el contenedor origen (`sourceHost`), significa que ya leyó el contenido en el sistema de archivos destino o server (Storage-server), por lo tanto, escribe en el sistema de archivos local (Storage-client) mediante el método de clase `FTP.write()`, el contenido guardado en la variable de instancia `content`. Luego finaliza retornado código 0.

ii. Modo “w”, valida en qué contenedor se halla:

1. Si el contenedor no es el origen, escribe escribe en el sistema de archivos distribuido (Storage-server) mediante el método de clase `FTP.write()`, el contenido guardado en la variable de instancia `content`, el cual posee el contenido del archivo local que se especificó en el parámetro `sourcePath`.
2. Si el contenedor es el origen, significa que finalizó la operación de escritura exitosamente. Finaliza el retornando código 0.

Class FTP

Define los métodos que simularán las operaciones de lectura y escritura de un sistema remoto de archivos:

1. `read()`:
   a. **Parámetros**:
      i. `name`: String que representa el nombre del archivo a leer.
      ii. `position`: la posición desde la cual se comenzará a leer el archivo.
      iii. `amount`: la cantidad de bytes a leer en el archivo solicitado.
   b. **Comportamientos**: Lee un archivo desde una posición dada, hasta alcanzar la cantidad de bytes indicada e imprime en consola, la cantidad de bytes que se pudieron leer.
   c. **Retorno**: arreglo de bytes con el contenido leído.
2. `write()`:
   a. **Parámetros**:
      i. `name`: nombre del archivo a escribir.
      ii. `content`: arreglo de bytes, posee el contenido a escribir.
   b. **Comportamiento**: abre el archivo indicado en caso de existir, caso contrario lo crea. Se posiciona al final del archivo para no sobreescribirlo en caso de que no esté vacío, luego escribe el contenido y finalmente imprime en consola la cantidad de bytes efectivamente escritos.

Comandos de ejecución

Para ejecutar este ejemplo se deben ejecutar los siguientes comandos:

5. Posicionarse dentro del directorio “Entrega 4/3”

6. Compilación:

```
javac -classpath lib/jade.jar -d classes a/Ftp.java a/AgenteMovil.java
```

7. GUI:

```
java -cp lib/jade.jar:classes jade.Boot -gui
```

8. Agente para lectura:

```
java -cp lib/jade.jar:classes jade.Boot -gui -container -host localhost
-agents "am:AgenteMovil(r, Main-Container, a/Storage-client/fatality.mp4,
a/Storage-server/fatality.mp4, 0, 50000)"
```

9. Agente para escritura:

```
java -cp lib/jade.jar:classes jade.Boot -gui -container -host localhost
-agents "am:AgenteMovil(w, Main-Container, a/Storage-client/fatality.mp4,
a/Storage-server/fatalitycopiab.mp4, 50000)"
```

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.

Class AgenteMovil

Define el comportamiento del agente:

1. **Parámetros**:
   a. `sourcePath`: path del sistema de archivos local.
   b. `targetHost`**:** contendor destino.
   c. `targetPath`: path destino.
   d. `backupPath`: path de copia/backup del archivo en server (Storage-server)
   e. `amount`**:** cantidad de bytes a leer o escribir.
2. **Métodos**:
   a. `setup()`:
      i. válida recibir los parámetros necesarios para realizar la operación requerida.
      ii. Crea un objeto `SequentialBehaviour copies`, al cual luego agrega 2 subcomportamientos `CopyBehavior`.
      iii. Agrega el subcomportamiento `localCopy`, el cual representa la copia en el sistema local. Para este caso crea una instancia de `CopyBehavior`, pasándole como parámetro `sourcePath`, el path "local" (Storage-client), `sourceHost` el contenedor "local o cliente", `targetHost` el contenedor "destino o server" (donde se debe ir a buscar el archivo) y `targetPath` el path donde se encuentra el archivo en el contenedor "server"

iv. Agrega subcomportamiento `remoteCopy`, correspondiente a la copia en el sistema “remoto”. Para este caso invierte los parámetros, dado que debe realizar la misma acción que localCopy, por lo tanto, crea una instancia de `CopyBehavior`, pasando como parámetro las siguientes variables de instancias de la clase `AgenteMovil`:
    1. `targetPath` (este valor se le asignará a la variable de instancia `sourcePath` al comportamiento) el path “remoto” (Storage-server) donde se hará la copia,
    2. `targetHost` el contenedor “remoto o server” (este valor se le asignará a la variable de instancia `sourceHost` al comportamiento), donde se hará la copia.
    3. `sourceHost` el contenedor “local o client” (este valor se le asignará a la variable de instancia `targetHost` al comportamiento), donde está el archivo a copiar.
    4. `sourcePath` el path “local o client” (este valor se le asignará a la variable de instancia `targetPath` al comportamiento), donde está el archivo a copiar.
v. Invoca el método `addBehavior(copies)`, donde copies posee los 2 subcomportamientos antes mencionados.

3. **Clases Anidadas:**
    a. `Class CopyBehavior`:
    Subclase de la clase Behavior.
        i. **Constructor**: Requiere recibir los siguientes parámetros:
            1. `sourcePath`: path en el contenedor donde se copiara el archivo
            2. `sourceHost`: contenedor donde se copiara el archivo
            3. `targetHost`: contenedor donde esta el archivo a copiar
            4. `targetPath`: path del archivo a copiar
        ii. **Métodos**:
            1. `action()`: define el comportamiento a realizar por el agente.
                a. Si el contenedor es el destino, lee el archivo y guarda el resultado en una variable de instancia content para luego copiarlo en el contenedor origen, setea la variable fileRead en true, para indicar que el archivo fue leído, luego se mueve al contenedor origen.
                b. Si el contenedor es el origen y aún no se leyó el archivo (`fileRead = false`) entonces se mueve al contenedor destino para realizar la lectura, caso contrario copia en el directorio indicado en la variable de instancia `sourcePath`, el contenido almacenado en la variable de instancia `content`.

2. `done()`: verifica si finalizó la ejecución de cada behaviour, comprobando si realizó la copia, mediante la variable booleana `copia`

Class FTP

Mismos métodos y comportamiento que en el ejercicio 3.a

Comandos de ejecución

Para ejecutar este ejemplo se deben ejecutar los siguientes comandos:

1. Posicionarse dentro del directorio “Entrega 4/3”

2. Compilación:
```
javac -classpath lib/jade.jar -d classes b/Ftp.java b/AgenteMovil.java
```

3. GUI:
```
java -cp lib/jade.jar:classes jade.Boot -gui
```

4. Agente:
```
java -cp lib/jade.jar:classes jade.Boot -gui -container -host localhost
-agents “am:AgenteMovil(Main-Container, b/Storage-client/fatality.mp4,
b/Storage-server/fatality.mp4, b/Storage-server/copiafatality.mp4, 50000)”
```

Comparación soluciones cliente/servidor

De los modelos cliente/servidor implementados en las prácticas, se puede observar diferencias comparados con la implementación con JADE:

1. Las soluciones realizadas con agentes presentan mayor facilidad a la hora de implementar la transmisión de los datos.
2. En la transmisión de datos:
    a. gRPC transmite los datos usando HTTP/2 como protocolo de transporte, gracias a Protocol Buffer, donde se definen las estructuras de datos de mensajes y servicios utilizados en la comunicación.
    b. Sockets, para la solución implementada, utiliza TCP como protocolo de red, el cual garantiza la entrega de datos.
    c. La solución con JADE tiene su propio protocolo de comunicación entre contenedores, además para la implementación, se utilizaron variables de instancia para almacenar los datos, que luego utilizará al ir migrando de contenedores.
3. La solución con JADE, es menos representativa del modelo cliente/servidores en lo que respecta a la transferencia de datos entre el cliente/servidor.
4. En JADE no existe un proceso cliente y otro proceso servidor al mismo tiempo intercambiando datos, sino que el agente migra de contenedores, cumpliendo el rol de cliente y servidor, realizando las tareas indicadas para el contenedor en el que se encuentra.

5. El nivel de abstracción de JADE es mayor que al de gRPC. Lo único que se indica es una referencia a un contenedor, luego JADE se encargará de gestionar la comunicación y las transferencias.
6. JADE cuenta por defecto con una implementación propia de protocolo de transmisión de datos.
7. Tanto en sockets como gRPC, existe un proceso estático especializado (servidor) que espera peticiones (request) y realiza su tarea en base a información proporcionada. En JADE, el agente es el que migra y recolecta la información.
8. Por la naturaleza estática del servidor, tanto en gRPC como en Sockets, si se “cae”, el servicio queda suspendido. En la implementación con JADE, si se “cae” un contenedor, si el agente no se encuentra en el mismo, sigue operando.
9. El agente puede realizar diferentes tareas intercambiables mediante Behaviour, lo que brinda versatilidad y facilidad de uso.
10. Respecto a JADE, la movilidad de código, puede implicar overhead extra, debido a la transferencia del código y del estado del agente.