Ramiro Lopes Canadell, 15263/2
Andrés Milla, 14934/6

Programación distribuida y tiempo real

Entrega 4

Ramiro Lopes Canadell, 15263/2
Andrés Milla, 14934/6

Ejercicio 1

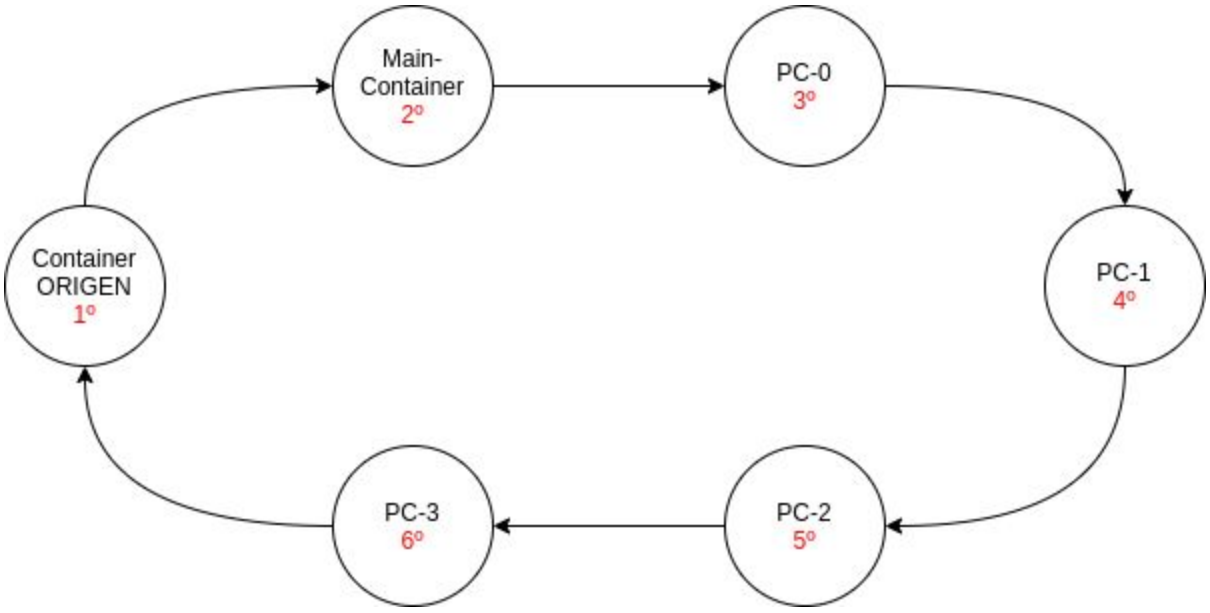
Implementación

Para implementar el agente que recolecta la información de las computadoras, comenzamos con crear computadoras adicionales mediante el método `crearComputadoras()`.

Luego, encapsulamos la información que se desea recolectar en una clase llamada **Informacion**. La misma tendrá la información asociada a cada computadora, por lo tanto, creamos un objeto Información por cada computadora y la guardamos en una lista ordenada de la siguiente forma:

- **0:** Container origen
- **1:** Main-Container
- **2:** Computadoras adicionales

En este mismo orden se realizará el recorrido, es decir, se comenzará desde el container origen, luego al main-container y a las computadoras adicionales. Finalmente se termina el recorrido al llegar al container origen:

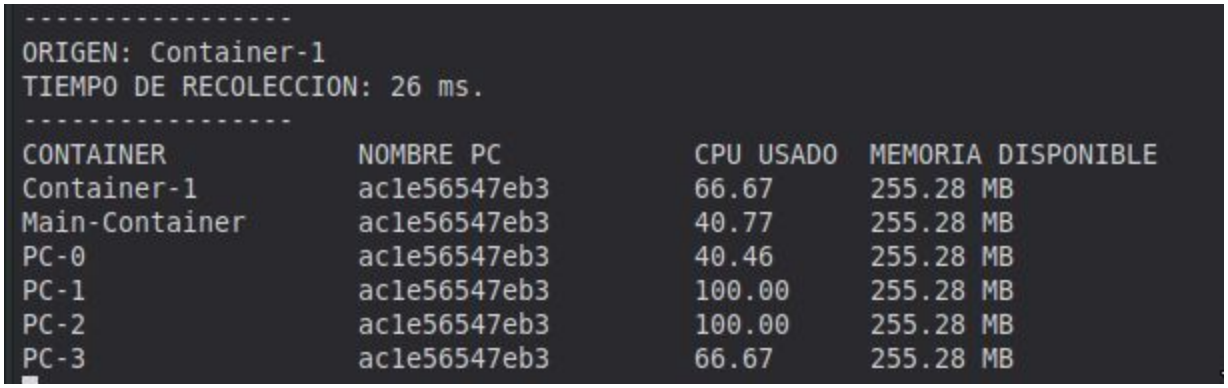


Durante el recorrido se va actualizando la información del container actual mediante el método `Informacion.actualizar()`, en donde usamos la librería `OperatingSystemMXBean` del

paquete `com.sun.management` para calcular el porcentaje de CPU usado en todo el sistema y la memoria disponible en bytes. Luego, para saber el nombre del host, utilizamos la librería `InetAddress` del paquete `java.net`.
Finalmente, cuando se llega al container origen se informan los datos recolectados a través de la salida estándar.

Para tomar el tiempo del recorrido, se comienza tomando el tiempo antes del primer movimiento desde el origen al main-container y se termina de contar cuando se llega nuevamente al origen. Además, una vez que se llega al origen se espera 1 segundo y comienza de nuevo el recorrido, esto permite que el script se ejecute de forma periódica tal como lo pide el enunciado.

- ➡ El código fuente se encuentra en el directorio *ejercicios/ejercicio-1/src*.
- ➡ Para ejecutar seguir las instrucciones del archivo *ejercicios/ejercicio-1/README.md.*



```
-----------------
ORIGEN: Container-1
TIEMPO DE RECOLECCION: 26 ms.
-----------------
CONTAINER           NOMBRE PC          CPU USADO  MEMORIA DISPONIBLE
Container-1         ac1e56547eb3       66.67      255.28 MB
Main-Container      ac1e56547eb3       40.77      255.28 MB
PC-0                ac1e56547eb3       40.46      255.28 MB
PC-1                ac1e56547eb3       100.00     255.28 MB
PC-2                ac1e56547eb3       100.00     255.28 MB
PC-3                ac1e56547eb3       66.67      255.28 MB
```

Estado global

El estado del sistema distribuido que se obtiene es consistente como el algoritmo de instantánea de Chandy-Lamport, ya que se recorren todas la computadoras de manera cíclica y secuencialmente sin tener problemas de interferencias o inconsistencias.

También hay que destacar que lo hacen de diferente forma, ya que este algoritmo permite comunicar el estado mediante un Agente que va recolectando la información y manda todo el estado interno del programa a través de la red. En cambio, el algoritmo de Chandy-Lamport va comunicando el estado a través de marcas en los canales de red que le permiten lograr la consistencia de los datos.

Por último aclaramos que el estado de este algoritmo es consistente siempre y cuando se cumplan las hipótesis del algoritmo de Chandy-Lamport sobre el estado de la red: *No fallan ni*

Ramiro Lopes Canadell, 15263/2
Andrés Milla, 14934/6

---

*los canales ni los procesos: **la comunicación es fiable** por lo que todos los mensajes se reciben intactos y una única vez.*

Ejercicio 2

Implementación

Para realizar este ejercicio implementamos un agente que se mueve a un container diferente (al **Main-Container** por ejemplo) el cual va a tener un archivo con números, y realiza la suma en dicho container. Luego de hacer la suma vuelve al container origen e imprime el resultado.

El archivo con números se encuentra en *ejercicio-2/database/numeros.csv*.

En el **setup** el agente guarda en una variable el nombre del container origen para poder regresar y también se inicializa en cero la variable que almacenará la suma.

En la función **afterMove** puede ocurrir lo siguiente:

- Si el agente se encuentra en un container diferente al origen significa que debe realizar la suma, por lo tanto abre el archivo de números y lo procesa, guardando el resultado en la variable previamente inicializada en el **setup**. Una vez terminada la suma y cerrado el archivo se mueve al container origen utilizando la funciona *migrarAgente()*.

- Si el agente se encuentra en el container origen significa que la suma ya se realizó, por lo tanto lo único que hace es imprimir el resultado de la suma.

➡ El código fuente se encuentra en el directorio *ejercicios/ejercicio-2/src*.
➡ Para ejecutar seguir las instrucciones del archivo *ejercicios/ejercicio-2/README.md.*

Cliente/Servidor

En un esquema cliente/servidor, el servidor tendría el archivo de números y el cliente le solicitaría el archivo mediante un `read`. Luego, el cliente deberá hacer el procesamiento y realizar la suma.
En cambio, en el esquema de la implementación en jade no se realiza ninguna petición, el agente se encarga de migrar a la computadora que posee el archivo, realizar la suma y regresar a la computadora origen con el resultado. Por lo tanto, se ven notoriamente 2 paradigmas de programación diferentes.

Ramiro Lopes Canadell, 15263/2
Andrés Milla, 14934/6

Múltiples archivos distribuidos

Para esto implementamos un agente que va recorriendo varias computadoras, las cuales tienen cada una un archivo con números, y realiza la suma de cada uno de los archivos. Una vez finalizado el recorrido regresa al container origen e imprime el resultado.
El recorrido del agente es circular, similar al recorrido del ejercicio 1, pero con la diferencia de que en este caso el recorrido no es periodico (se hace una única vez).

Los archivos con números se encuentran en *ejercicio-2-adicional/database/.* Los archivos tienen el mismo nombre que su container, por ejemplo el Main-Container posee el archivo *Main-Container.csv*

En el **setup** el agente realiza las siguientes acciones:

- Crea 3 containers adicionales utilizando el método `crearComputadoras()`.Los nombres de los contenedores se encuentran en una lista llamada *COMPUTADORAS_ADICIONALES.*
- Se agregan los nombres del *Main-Container* y del container origen dentro de la lista de containers.
- El agente migra al primer container de la lista de container mediante el método *migrarAgente()*.

En el **afterMove** puede ocurrir lo siguiente:

- Si el agente se encuentra en un container diferente al origen significa que debe realizar la suma del archivo de ese container, por lo tanto abre el archivo de números y lo procesa, guardando el resultado en un arreglo de números donde se almacenarán los resultados de cada suma. Una vez terminada la suma y cerrado el archivo se mueve al siguiente container de la lista de containers.

- Si el agente se encuentra en el container origen significa que la suma de todos los archivos ya se realizó, por lo tanto lo único que hace es imprimir el resultado de la suma de cada uno de los archivos, y de la suma total, utilizando el método *loguearInformacion()*.

➡ El código fuente se encuentra en el directorio *ejercicios/ejercicio-2-adicional/src*.
➡ Para ejecutar seguir las instrucciones del archivo *ejercicios/ejercicio-2-adicional/README.md.*

Ramiro Lopes Canadell, 15263/2
Andrés Milla, 14934/6

Ejercicio 3

Inciso A

Implementación

Para implementar tanto la lectura como la escritura del sistema de archivos, aprovechamos la migración del código y creamos una clase **Transferencia**, en la cual se va a mantener el estado de cada lectura y escritura de los bytes del archivo. El mismo está compuesto por los siguientes atributos:

- **Ventana**: Representa la porción de bytes en la cual va a ser transferido el archivo a través de la red. La misma será de 1024 bytes (1 KiB).

- **Total de bytes leídos**: Representa la cantidad de bytes que se han leído hasta el momento del archivo. Inicialmente será 0.

- **Contenido**: Representa el buffer de bytes en dónde se depositará el contenido leído de cada porción del archivo. Inicialmente estará vacío.

- **Bytes faltantes**: Representa la cantidad de bytes que faltan escribir, particularmente se utiliza para condición de fin. Es decir, cuando la cantidad de bytes faltantes sea 0 entonces la transferencia terminará. Inicialmente estará en ***null***, ya que al estar en un sistema distribuido no podemos saber la cantidad de bytes faltantes hasta visitar el container que tiene el archivo.

Implementación de la lectura **local**

Particularmente para la lectura local, creamos un método `Transferencia.leer` el cual recibe un path del archivo que se desea leer en el sistema de archivos actual del agente. El algoritmo es el siguiente:

1. Se abre un stream de datos del contenido del archivo que se desea leer.

2. Se posiciona en la posición del contenido, la cual está dada por la cantidad total de bytes leídos hasta el momento.

3. Se deposita la información en el buffer. El tamaño del buffer va a ser siempre del tamaño de la ventana excepto en el último caso, ya que si el tamaño de archivo no es

múltiplo del tamaño de la ventana diferirá. Por lo tanto, se toma el mínimo entre el tamaño de la ventana y la diferencia del tamaño del archivo - total bytes leídos.

4. Finalmente, se actualiza la cantidad total de bytes leídos y los bytes faltantes.

Implementación de la escritura **local**

Para la escritura del archivo local, creamos un método `Transferencia.escribir` el cual recibe el path del archivo que se desea escribir y el modo de escritura (**true** indica append, **false** indica destructiva). El algoritmo es simple:

1. Se abre un stream del archivo en el modo que se indicó como parámetro.
2. Se escribe el buffer que contiene el objeto de la transferencia.
3. Se cierra el stream de datos.

Implementación de las operaciones **remotas**

Para la implementación de las operaciones remotas utilizamos los comportamientos que brinda *JADE*: Creamos un comportamiento **TransferenciaBehaviour**, el cual transferirá un archivo que reside en el container destino hacia el container origen, el mismo puede ser instanciado con los siguientes parámetros:

1. **Container origen:** Será el container en el cual se ejecuta el script inicialmente.

2. **Container destino**: Será el container que tiene el archivo que se desea transferir al origen.

3. **Path origen:** Representa la ubicación en donde quedará el archivo transferido en el container origen.

4. **Path destino:** Representa la ubicación del archivo en el container destino que se desea transferir al container origen.

El comportamiento está en el método `TransferenciaBehaviour.action` y se comportará de la siguiente forma:

- Si el agente está ubicado en el container origen, entonces se usa el método `Transferencia.escribir` con los siguientes parámetros:

    1. El path especificado del container origen.

---

2. El modo de escritura, el cual inicialmente será destructivo para generar el archivo en el origen y luego será append para escribir los fragmentos del archivo al final.

Una vez hecho esto, se actualiza la condición de fin para el método `done`, la cual es el resultado de evaluar si la cantidad de bytes faltantes es 0. Por último, se mueve el agente al container destino.

- Por otro lado, si el agente está ubicado en el container destino, entonces se usa el método `leer` con el path destino que fue especificado en la instanciación. Luego, se mueve el agente al container origen.

Este comportamiento se mantendrá hasta que el método `done` devuelva True, el mismo retorna la condición que es actualizada al escribir en el origen, como fue mencionado anteriormente.

Después creamos otro comportamiento, el cual nos permite ejecutar comportamientos secuencialmente, el mismo hereda de la clase **SequentialBehaviour**.

Por lo tanto para la **lectura**, se ejecuta una instancia de **TransferenciaBehaviour** con los parámetros: Container origen, container destino, path origen y path destino.

Mientras que para la **escritura**, se ejecuta una instancia de **TransferenciaBehaviour** con los parámetros invertidos, es decir, como container origen se indica el container destino y cómo container destino se indica el container origen. Esto permitirá copiar el archivo del container origen al container destino.

➡ El código fuente se encuentra en *ejercicios/ejercicio-3/src*.
➡ Para ejecutar, seguir las instrucciones del archivo *ejercicios/ejercicio-3/README.md.*

Inciso B

Implementación

La implementación es la combinación de los ejercicios anteriores, es decir, como primer comportamiento se ejecuta una instancia de **TransferenciaBehaviour** con los parámetros: Container origen, container destino, path origen y path destino. Esto servirá para traernos el archivo del container destino al container origen.
Luego, para la implementación de la copia se agrega un segundo comportamiento que es instancia de **TransferenciaBehaviour** con los parámetros invertidos, es decir, como container

origen se indica el container destino y cómo container destino se indica el container origen. Esto permitirá copiar el archivo del container origen al container destino.

- ➡ El código fuente se encuentra en *ejercicios/ejercicio-3/src*.
- ➡ Para ejecutar, seguir las instrucciones del archivo *ejercicios/ejercicio-3/README.md.*

Comparación Cliente/Servidor

Al igual que en el ejercicio 2, esta solución cambia mucho por el paradigma, ya que en JADE el agente se va moviendo por los diferentes containers para leer/escribir los archivos manteniendo el estado del programa a través de la red. Mientras que en el cliente/servidor de las prácticas pasadas se tienen que comunicar datos para armar el contenido del archivo.
Por el lado de JADE se simplifican las cosas ya que el agente puede moverse al filesystem destinatario y trabajar más cómodamente, pero por el lado de C/S se simplifica la forma de comunicación ya que no se tiene que estar pendiente de la migración del agente de container en container.