ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

[DΣIM]

Técnicas avanzadas de programación

Prácticas 1 y 2

Curs 23/24

Estudiants: Miguel Robledo Kusz
Víctor Fosch Tena

ÍNDEX

Introducción

Function as a Service, o sus siglas FaaS, es un modelo de computación en la nube dentro del paradigma “serverless”, que permite a los usuarios registrar, listar, invocar y eliminar acciones, e invocar dichas acciones mediante su identificador con los parámetros de entrada que se deseen. Con este modelo, los usuarios pueden ejecutar funciones en la nube y sin tener que preocuparse por la gestión de esta ni de los recursos.

El objetivo de esta práctica es modelar un sistema FaaS completo con Java, permitiendo al usuario ejecutar funciones, adaptando el sistema a los recursos que haya disponibles en cada momento. En nuestro caso nos basaremos en la arquitectura de OpenWhisk de IBM.

En la segunda parte, el objetivo es el desarrollo de un marco de trabajo en Scala para poder interactuar con la primera parte del trabajo. Además también hemos implementado en Scala algunas funcionalidades como los métodos WordCount i CountWords.

Para poner en contexto, hablaremos brevemente sobre conceptos clave en este trabajo como qué son y para qué sirven el *Controller* y el *Invoker*.

En un sistema FaaS, el controller es la clase maestra, la que se encarga, como su nombre indica, de controlar y gestionar todo el sistema. En nuestro caso, la clase Controller lleva a cabo, entre otras,las tareas de:

- Registrar las acciones, además de asegurarse de que no haya identificadores repetidos.
- Gestionar la memoria disponible en el sistema.
- Gestionar los invokers y la asignación de acciones a estos en función de la política que esté vigente, en función de la memoria disponible en cada invoker.
- Gestionar las políticas que se aplican en el sistema en todo momento.

La clase invoker es la encargada de ejecutar las acciones y está comandada por el controller. Un *Invoker* es la entidad que se encarga de iniciar la ejecución de una función, siempre que ésta esté dentro de sus capacidades de memoria disponible.

Componentes del sistema

Controller

Tiene una lista de invokers, un Map donde se almacenan las acciones, un entero para el conteo de la memoria disponible, un PolicyManager que veremos más tarde, ExecutorService para los threads y una lista de métricas que se utilizan con el patrón Observer (más tarde lo veremos).

Atributos

```
private List<Invoker> invokers;
private Map<String, Action> actions;
private int availableMemory;
private PolicyManager policyManager;
private ExecutorService executorService;
private List<Metric> metrics;
```

Constructor

Si el número de invokers o la memoria es nula o negativa lanzamos una excepción.

```
public Controller(int invokerCount, int memoryPerInvoker) {
    if (invokerCount ≤ 0 || memoryPerInvoker ≤ 0) {
        throw new RuntimeException(message:"Invalid invokerCount or memoryPerInvoker");
    }
    actions = new HashMap<>();
    invokers = new ArrayList<>();
    availableMemory = invokerCount * memoryPerInvoker;
    policyManager = new PolicyManager(new RoundRobin()); // Round Robin by default
    executorService = Executors.newFixedThreadPool(invokerCount * 32);  // Initializes the
    metrics = new ArrayList<>();
    for (int i = 0; i < invokerCount; i++) {
        Invoker invoker = new Invoker(memoryPerInvoker, i);
        invoker.setObserver(this);
        invokers.add(invoker);
    }
}
```

Registrar acción

Primero comprueba si la acción ya está registrada, si no lo está, la registra.

```
public <T, R> void registerAction(String actionName, Function<T, R> function, int memorySize) {
    if (isActionRegistered(actionName)) return;
    Action action = new Action(function, memorySize, actionName);
    actions.put(actionName, action);
}
```

Invocación única síncrona

Si la acción no está registrada o no hay suficientes recursos lanza una excepción.
Después reserva memoria, elige el invoker con la política determinada, ejecuta la acción en el invoker elegido y por último libera la memoria.

```
public <T, R> R invoke(String actionName, T args) {
    Action action = actions.get(actionName);
    if (action == null) {
        throw new RuntimeException("Action not registered: " + actionName);
    }

    if (action.getMemorySize() > availableMemory) {
        throw new RuntimeException("Not enough memory to execute action: " + actionName);
    }

    availableMemory -= action.getMemorySize(); // Reduces the available memory for the action.

    Invoker invoker = policyManager.selectInvoker(invokers, action);
    R result = invoker.execute(action, args);

    availableMemory += action.getMemorySize(); // Restores the available memory after the action.

    return result;
}
```

Invocación grupal síncrona

Hace lo mismo que antes pero para un grupo de parámetros.

```
public <T, R> List<R> invoke(String actionName, List<T> listOfParams) {
    Action action = actions.get(actionName);
    if (action == null) {
        throw new RuntimeException("Action not registered: " + actionName);
    }

    if (availableMemory < action.getMemorySize() * listOfParams.size()) {
        throw new RuntimeException("Not enough memory to execute group action: " + actionName);
    }

    // Reserves memory for this group invocation
    availableMemory -= action.getMemorySize() * listOfParams.size();

    policyManager.getPolicy().reset(); // Resets the invoker selection policy.

    List<R> results = new ArrayList<>();
    for (T params : listOfParams) {
        Invoker invoker = policyManager.selectInvoker(invokers, action);
        results.add(invoker.execute(action, params));
        availableMemory += action.getMemorySize(); // Restores the available memory after the action.
    }
    return results;
}
```

Invocación única asíncrona

Hace lo mismo que la síncrona pero en un nuevo hilo de ejecución y procurando que el acceso a la variable compartida de recursos disponibles está sincronizado y la elección del invoker también, para así evitar condiciones de carrera.

```
public <T, R> Future<R> invoke_async(String actionName, T args) {
    return executorService.submit(() -> {
        Action action = actions.get(actionName);
        if (action == null) {
            throw new RuntimeException("Action not registered: " + actionName);
        }

        synchronized (this) {
            if (action.getMemorySize() > availableMemory) {
                throw new RuntimeException("Not enough memory to execute action: " + actionName);
            }
            availableMemory -= action.getMemorySize(); // Reserves memory.
        }

        Invoker invoker;
        synchronized (policyManager) {
            try{
                invoker = policyManager.selectInvoker(invokers, action);
            } catch (Exception e) { // Important in case invoker selection fails, to release memory.
                synchronized (this) {
                    availableMemory += action.getMemorySize(); // Releases memory.
                }
                throw new RuntimeException(e.getMessage());
            }
        }

        try {
            return invoker.execute(action, args);       // Executes the action on the selected invoker.
        } finally {
            synchronized (this) {
                availableMemory += action.getMemorySize(); // Releases memory.
            }
        }
    });
}
```

Invocación grupal asíncrona

Llama a invoke_async único para cada uno de los parámetros de la lista de entrada. Después combina todos los CompletableFuture de la lista resultante en un único CompletableFuture que devolverá la lista con los resultados.
En esta implementación también se procura un acceso sincronizado a la variable de recursos disponibles.

```java
public <T, R> CompletableFuture<List<R>> invoke_async(String actionName, List<T> listOfParams) {
    Action action = actions.get(actionName);
    if (action == null) {
        throw new RuntimeException("Action not registered: " + actionName);
    }

    // Checks if there is enough memory for all actions
    synchronized (this) {
        if (action.getMemorySize() * listOfParams.size() > availableMemory) {
            throw new RuntimeException("Not enough memory to execute group action: " + actionName);
        }
    }

    List<CompletableFuture<R>> futureResults = listOfParams.stream()
        .map(params -> CompletableFuture.supplyAsync(() -> {
            try {
                Future<R> future = invoke_async(actionName, params);
                return future.get();
            } catch (Exception e) {
                throw new RuntimeException(e.getCause().getMessage());
            }
        }, executorService))
        .collect(Collectors.toList());

    // Combines all futures into a single CompletableFuture containing a list of results
    return CompletableFuture.allOf(futureResults.toArray(new CompletableFuture[0]))
        .thenApply(v -> futureResults.stream()
                                     .map(CompletableFuture::join)
                                     .collect(Collectors.toList()));
}
```

Invoker

Tiene una variable entera que almacena la memoria disponible del invoker. También tiene un Lock que utilizamos para el acceso sincronizado a variables compartidas, un observer (que veremos más adelante) y un id (se asigna durante la creación de cada invoker en el constructor del Controller).

Atributos

```
private int availableMemory; // Attribute for available memory.
private final Lock lock = new ReentrantLock();
private Observer observer; // Reference to the Controller or another observer.
private int id;
```

Constructor

```
public Invoker(int memorySize, int id) {
    this.availableMemory = memorySize; // Attribute for available memory.
    this.id = id;
}
```

Execute

Procuramos un acceso sincronizado a la variable availableMemory mediante el ReentrantLock.
Obtenemos la función a ejecutar de la acción que nos pasan por parámetro y la ejecutamos con el argumento especificado. Antes de acabar liberamos la memoria procurando un acceso sincronizado con el lock. También medimos el tiempo de ejecución de la acción en el invoker, cosa que tiene que ver con el patrón Observer que veremos después.

```
public <T, R> R execute(Action action, T args) {
    long startTime = System.nanoTime(); // Starts the timer

    lock.lock();
    try {
        if (action.getMemorySize() > availableMemory) {
            throw new RuntimeException(message:"Invoker does not have enough memory to execute the action");
        }
        availableMemory -= action.getMemorySize(); // Reduces the available memory for the action.
    } finally {
        lock.unlock();
    }

    R result;
    @SuppressWarnings("unchecked")
    Function<T, R> function = (Function<T, R>) action.getFunction();

    try {
        result = function.apply(args); // Executes the action.
    } finally {
        lock.lock();
        try {
            availableMemory += action.getMemorySize(); // Restores the available memory after the action.
        } finally {
            lock.unlock();
            long endTime = System.nanoTime(); // Stops the timer

            if (observer != null) {
                Metric metric = new Metric(action.getActionName(), endTime - startTime, this.id, action.getMemorySize());
                observer.update(metric); // Notifies the observer
            }
        }
    }
    return result;
}
```

Action

Tiene una función con parámetros desconocidos, un tamaño de memoria y un nombre.

Atributos

```
public Action(Function<?, ?> function, int memorySize, String actionName) {
    this.function = function;
    this.memorySize = memorySize;
    this.actionName = actionName;
}
```

Se utiliza básicamente para poder saber cuanta memoria ocupa una acción y su función, así el invoker puede reservar esa memoria y ejecutar esa función. El nombre se utiliza para el patrón observer, el cual calcula unas métricas para cada una de las acciones que se han ejecutado en el sistema durante el ciclo de vida del framework.

Patrones de diseño

A continuación, explicaremos los diferentes patrones de diseño que se han utilizado para implementar diversas funcionalidades en la primera práctica.

Strategy

El patrón Strategy se utiliza para poder implementar diversas técnicas de elección de invokers para la ejecución de acciones.
Se implementa mediante una variable de tipo PolicyManager en el Controller. El PolicyManager tiene un atributo de tipo ResourcePolicy.

```
private ResourcePolicy resourcePolicy;

/**
 * Constructs a PolicyManager with the specified resource policy.
 *
 * @param resourcePolicy the resource policy to be managed
 */
public PolicyManager(ResourcePolicy resourcePolicy) {
    this.resourcePolicy = resourcePolicy;
}
```

Entonces, el ResourcePolicy es una interface de Java.

```
public interface ResourcePolicy {
    /**
     * Selects an invoker from the provided list of invokers to execute the given action.
     *
     * @param invokers the list of invokers
     * @param action   the action to be executed
     * @return the selected invoker
     */
    Invoker selectInvoker(List<Invoker> invokers, Action action);

    /**
     * Resets the state of the resource policy.
     */
    void reset();
}
```

La cual es implementada por las diversas estrategias concretas, por ejemplo, RoundRobin.

```java
public class RoundRobin implements ResourcePolicy {
    private int currentIndex = 0;

    /**
     * Selects an invoker from the provided list of invokers to execute the given action in a round-robin fashion.
     *
     * @param invokers the list of available invokers
     * @param action   the action to be executed
     * @return the selected invoker
     * @throws RuntimeException if no invoker has enough memory to execute the action
     */
    @Override
    public Invoker selectInvoker(List<Invoker> invokers, Action action) {

        for (int i = currentIndex; i < invokers.size(); i++) {
            if (invokers.get(i).getAvailableMemory() ≥ action.getMemorySize()) {
                System.out.println("Invoker selected: " + i);
                currentIndex = (i + 1) % invokers.size();
                return invokers.get(i);
            }
        }

        throw new RuntimeException(message:"No invoker has enough memory to execute the action");
    }

    /**
     * Resets the state of the resource policy.
     */
    @Override
    public void reset() {
        currentIndex = 0;
    }
}
```

De esta manera, gracias al método del Controller siguiente:

```java
public void setPolicy(ResourcePolicy resourcePolicy) {
    policyManager.setPolicy(resourcePolicy);
}
```

Podemos asignar cualquier clase que implemente el contrato ResourcePolicy como política de elección de invokers. Por ejemplo:

```java
Controller controller = new Controller(invokerCount:4, memoryPerInvoker:2048);
controller.setPolicy(new GreedyGroup());
```

Observer

La idea del observer es poder recolectar diferentes métricas durante la ejecución de las acciones. En este caso, el observador es el Controller, el cual quiere estar al tanto de todas las métricas de las acciones que se ejecutan y los invokers son los que notifican de estas métricas.

Para empezar, el Controller implementa la interface Observer que está definida de la siguiente forma:

```
public interface Observer {
    /**
     * Updates the observer with a metric.
     *
     * @param metric the metric to update the observer with
     */
    void update(Metric metric);
}
```

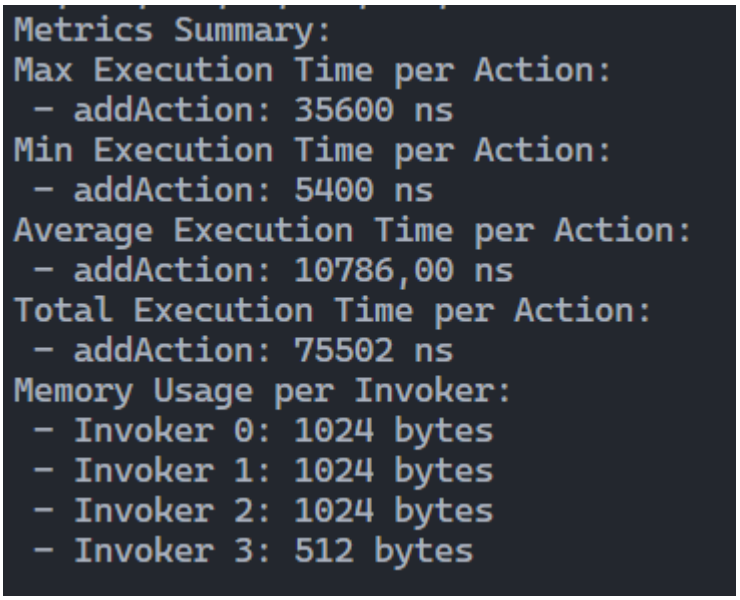
Como hemos visto antes en el execute del Invoker, cada vez que se ejecuta una acción se crea un objeto Metric con las estadísticas de la ejecución de esa acción y se le pasa en la llamada al método controller.update, el cual está implementado de la siguiente forma:

```
@Override
public void update(Metric metric) {
    synchronized (metrics) {
        metrics.add(metric);
    }
}
```

Básicamente se añade la métrica a la lista de métricas del Controller de manera sincronizada, para evitar condiciones de carrera ya que pueden haber diferentes hilos, en caso de invocaciones asíncronas, tratando de añadir una métrica.

Cuando queramos consultar las métricas, llamamos al método del Controller calculateMetricsSummary el cual utiliza streams y collections de Java para crear el resumen de las métricas que se pedían en la memoria.

Al final, el resultado se verá así, para el TestObserver.java:



Decorator

Utilizamos el patrón Decorator para implementar un cronómetro para cada acción que se ejecuta y memoization, es decir, ahorrarnos la ejecución de una acción si ya tenemos el resultado para una acción que se ha ejecutado antes con los mismo parámetros.

La implementación del Decorator, como bien sabemos, utiliza una relación de herencia y otra de clientela (composición) con la clase que se quiere decorar, en este caso, Function<T,R>.

Cronómetro

```java
public class TimerFunctionDecorator<T, R> implements Function<T, R> {
    private final Function<T, R> function;

    /**
     * Constructs a TimerFunctionDecorator with the specified function.
     *
     * @param function the function to be decorated
     */
    public TimerFunctionDecorator(Function<T, R> function) {
        this.function = function;
    }

    /**
     * Applies this function to the given argument and measures the execution time.
     *
     * @param args the function argument
     * @return the result of applying the function
     */
    @Override
    public R apply(T args) {
        long startTime = System.nanoTime();
        R result = function.apply(args);
        long endTime = System.nanoTime();

        System.out.println("Tiempo de ejecución: " + (endTime - startTime) + " ns");
        return result;
    }
}
```

Memoization

```java
public class MemoizationDecorator<T, R> implements Function<T, R> {
    private final Function<T, R> function;
    private final Map<T, R> cache;

    /**
     * Constructs a new MemoizationDecorator with the specified function.
     *
     * @param function The function to be memoized.
     */
    public MemoizationDecorator(Function<T, R> function) {
        this.function = function;
        this.cache = new HashMap<>();
    }


    /**
     * Applies this function to the given arguments, with memoization.
     *
     * @param args The function arguments.
     * @return The function result, either retrieved from cache or computed.
     */
    @Override
    public R apply(T args) {
        if (cache.containsKey(args)) {
            System.out.println(x:"Cache hit!");
            return cache.get(args);
        }

        R result = function.apply(args);
        cache.put(args, result);
        return result;
    }
}
```

La decoración de una función se haría y registraría en el controller de la siguiente forma:

```java
Controller controller = new Controller(invokerCount:4, memoryPerInvoker:1024);
Function<Map<String, Integer>, Integer> f = x -> x.get(key:"x") + x.get(key:"y");

Function<Map<String, Integer>, Integer> memoizationFunction = new MemoizationDecorator<>(f);
Function<Map<String, Integer>, Integer> memoizationTimerFunction = new TimerFunctionDecorator<>(memoizationFunction);
controller.registerAction(actionName:"addAction", memoizationTimerFunction, memorySize:512);
```

La salida, será la siguiente para el TestDecorator.java:

```
--------Testing Decorator with RoundRobin policy--------
Invoker selected: 0
Invoker selected: 1
Invoker selected: 2
Invoker selected: 3
Invoker selected: 0
Invoker selected: 1
Cache hit!
Invoker selected: 2
Cache hit!
Tiempo de ejecución: 5900 ns
Tiempo de ejecución: 151800 ns
Tiempo de ejecución: 111501 ns
Tiempo de ejecución: 26400 ns
Tiempo de ejecución: 3600 ns
Tiempo de ejecución: 4901 ns
Tiempo de ejecución: 38299 ns
[5, 10, 16, 7, 5, 23, 23]
```

Reflection

Utilizamos Reflection para poder utilizar el Controller de una manera orientada a objetos. Es decir, no tendremos que crear y registrar las acciones manualmente en el Controller sino que se hará mediante el patrón de diseño mencionado.

Primero, tenemos que llamar al método del Controller createProxy, para así obtener un ActionProxy:

```
public IActionProxy createProxy() {
    IActionProxy proxy = (IActionProxy) Proxy.newProxyInstance(
        IActionProxy.class.getClassLoader(),
        new Class<?>[]{IActionProxy.class},
        new DynamicProxy(this, new ActionProxyImpl())
    );
    return proxy;
}
```

Sobre este ActionProxy ya podremos realizar invocaciones. Pero el funcionamiento del patrón es el siguiente.

IActionProxy define una interface con los métodos (acciones) que se registrarán y ejecutarán por los invokers del Controller.

```
public interface IActionProxy {
    /**
     * Calculates the factorial of the given integer.
     *
     * @param a the integer to calculate the factorial of
     * @return the factorial of the given integer
     */
    Integer calculateFactorial(Integer a);

    /**
     * Sums two integers.
     *
     * @param a the first integer
     * @param b the second integer
     * @return the sum of the two integers
     */
    Integer sumNumbers(Integer a, Integer b);
}
```

ActionProxyImpl implementa los métodos de la interface.

```java
public class ActionProxyImpl implements IActionProxy {


    /**
     * Calculates the factorial of the given integer.
     *
     * @param a the integer to calculate the factorial of
     * @return the factorial of the given integer
     */
    @Override
    public Integer calculateFactorial(Integer a) {
        int factorial = 1;
        for (int i = 1; i ≤ a; i++) {
            factorial *= i;
        }

        return factorial;
    }

    /**
     * Sums two integers.
     *
     * @param a the first integer
     * @param b the second integer
     * @return the sum of the two integers
     */
    @Override
    public Integer sumNumbers(Integer a, Integer b) {
        return a+b;
    }
}
```

DynamicProxy implementa a la interface InvocationHandler.

```java
public class DynamicProxy implements InvocationHandler {
    private final Controller controller;
    private final Object target;

    /**
     * Constructs a DynamicProxy object with the specified controller and target.
     *
     * @param controller the controller used for managing actions
     * @param target     the target object to proxy
     */
    public DynamicProxy(Controller controller, Object target) {
        this.controller = controller;
        this.target = target;
    }

    /**
     * Invokes the specified method on the target object.
     *
     * @param proxy  the proxy object
     * @param method the method to invoke
     * @param args   the arguments to pass to the method
     * @return the result of the method invocation
     * @throws Throwable if an error occurs during method invocation
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Function<Object[], Object> function = x -> {
            try {
                return method.invoke(target, x);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        };

        controller.registerAction(method.getName(), function, memorySize:32);

        return controller.invoke(method.getName(), args);
    }
}
```

De esta manera, cada vez que invoquemos un método de la interfaz IActionProxy, por ejemplo:

```java
IActionProxy proxy = controller.createProxy();

Integer result = proxy.sumNumbers(a:2,b:3);
```

Esta llamada la interceptará el InvocationHandler que hemos implementado, de manera que creara una función donde se llama al método sumNumber, la registrará en el controller y la invocará. Así conseguimos invocar acciones en nuestro sistema mediante orientación a objetos.

APIs contratos

En este apartado explicaremos la API que proporciona nuestro framework de ejecución de funciones, esto es, los acuerdos formales que especifican cómo deben interactuar los componentes del sistema, así como la definición de los endpoints disponibles y cómo utilizarlos.

Para poder utilizar el framework, el desarrollador solamente debe conocer los detalles del componente controlador Controller, los demás componentes, como el Invoker o el PolicyManager van a cargo del mencionado controlador.

Los métodos del Controller que debe conocer el desarrollador son los siguientes:

- constructor
- registerAction
- invoke
- invoke_async
- shutdown

Ahora procederemos a explicar los mencionados métodos.

constructor

Es el constructor de la clase, al cual se le deben pasar dos argumentos. El primero hace alusión al número de invokers y el segundo a la cantidad de memoria que tendrá asignada cada invoker.

registerAction

Este método se encarga de almacenar una nueva acción en el controlador.
Los parámetros necesarios son tres, el primero es el nombre que queremos asignar a la acción, el cual es importante ya que después lo utilizaremos para realizar las invocaciones. El segundo parámetro es la función, la cual admite dos parámetros genéricos T y R. Por último, el tercer parámetro es el tamaño de memoria que utilizará la acción para ejecutarse.

invoke

Este método es el que se encarga de ejecutar la acción indicada por parámetro. En total, admite dos parámetros. El primero es el nombre de la acción que queremos ejecutar, la cual

tendrá que estar previamente registrada con el método mencionado antes. El segundo es el parámetro que queremos pasarle a la función para que lo utilice durante su ejecución, es de tipo genérico T.

Un dato importante sobre este método es que se utiliza polimorfismo ad-hoc en la clase Controller para su implementación, lo que quiere decir que también se puede llamar a este método pasándole una lista de parámetros en lugar de uno solo, lo que llevará al controlador a ejecutar una invocación grupal, donde se ejecutará la acción mencionada para cada uno de los parámetros de la lista, pero antes comprobando que existan suficientes recursos entre todos los invokers como para ejecutar esa llamada grupal.

Es importante saber que tanto este método, como el invoke_async, pueden lanzar una excepción en tiempo de ejecución, de manera que tendremos que utilizar un bloque try catch durante su llamada.

invoke_async

Este método hace básicamente lo mismo que el anterior pero sin bloquear el hilo de ejecución principal, de manera que podemos realizar diversas llamadas a invoke_async en nuestro código sin tener que esperar a que finalice cada una de ellas.
Para recoger el resultado de esta llamada utilizamos un objeto de la clase Future de java, lo que nos permite designar un placeholder para el resultado de la ejecución que se nos devolverá en algún momento del futuro próximo.

El controlador utiliza la clase ExecutorService de java para crear un pool de threads i así poder implementar esta concurrencia.

shutdown

Es importante que el desarrollador conozca este método ya que es el encargado de cerrar el ExecutorService una vez hayamos llegado al final de nuestro código, así el programa podra finalizar correctamente.
Antes debemos estar seguros de que ya se han ejecutado todas nuestras invocaciones asíncronas.

Puntos de extensión y ciclo de vida

Los puntos de extensión más importantes que presenta el framework son dos, uno es el que se implementa mediante el patrón Strategy y el otro se implementa mediante Decorator.

Esto es debido a que, para el patrón Strategy, se define una interface de Java llamada ResourcePolicy, la cual se puede implementar por parte del desarrollador de la manera que crea conveniente y después se le puede asignar al controlador instanciado en nuestro programa mediante el método setPolicy.

Por ejemplo, podemos crear una nueva estrategia de elección de invokers llamada BestFit, la cual implementa la interface ResourcePolicy y después se la podemos asignar al controlador:

```
controller.setPolicy(new BestFit())
```

El otro punto de extensión se consigue utilizando el patrón Decorator. Podemos crear cualquier clase que implemente Function<T,R> y a su vez tenga una relación de clientela (composición) con Function<T,R>. De esta manera, tenemos que definir un constructor donde le pasaremos la referencia de la función que queremos decorar y también se debe implementar el método apply con @Override.

Una vez hecho esto, podemos decorar una función y registrarla en el controlador.

Realmente este no es un punto de extensión propiamente dicho ya que no se está implementando una nueva funcionalidad en el sistema de ejecución de funciones, sino que se está creando una función con un decorador. Se podría considerar como una extensión en el caso por ejemplo de la Memoization, ya que con este decorador no se ejecutará la función en caso de que se encuentre el resultado en la caché.

El ciclo de vida del framework inicia con la instanciación del controlador, al cual le podemos registrar y invocar acciones tanto de manera síncrona como asíncrona tal y como hemos visto antes. Para el caso de las invocaciones asíncronas, el ciclo de vida acabará con la llamada al método shutdown(), ya que a partir de ese momento el ExecutorService estará cerrado y no se permitirá la ejecución de más hilos de manera concurrente. Para el caso de las invocaciones síncronas, aún después de la llamada a shutdown() se podrán seguir registrando y invocando acciones en el controlador.

Validación y testeo de servicios

Para realizar la validación y el testeo de los servicios proporcionados por el framework hemos implementado una serie de clases con métodos main y también hemos utilizado JUnit para realizar un testing individual de cada uno de los componentes del sistema. Todos estos tests se encuentran en el proyecto de la primera parte.

Invocaciones únicas asíncronas (src.main.Tests.test_async.TestUniqueAsync.java)

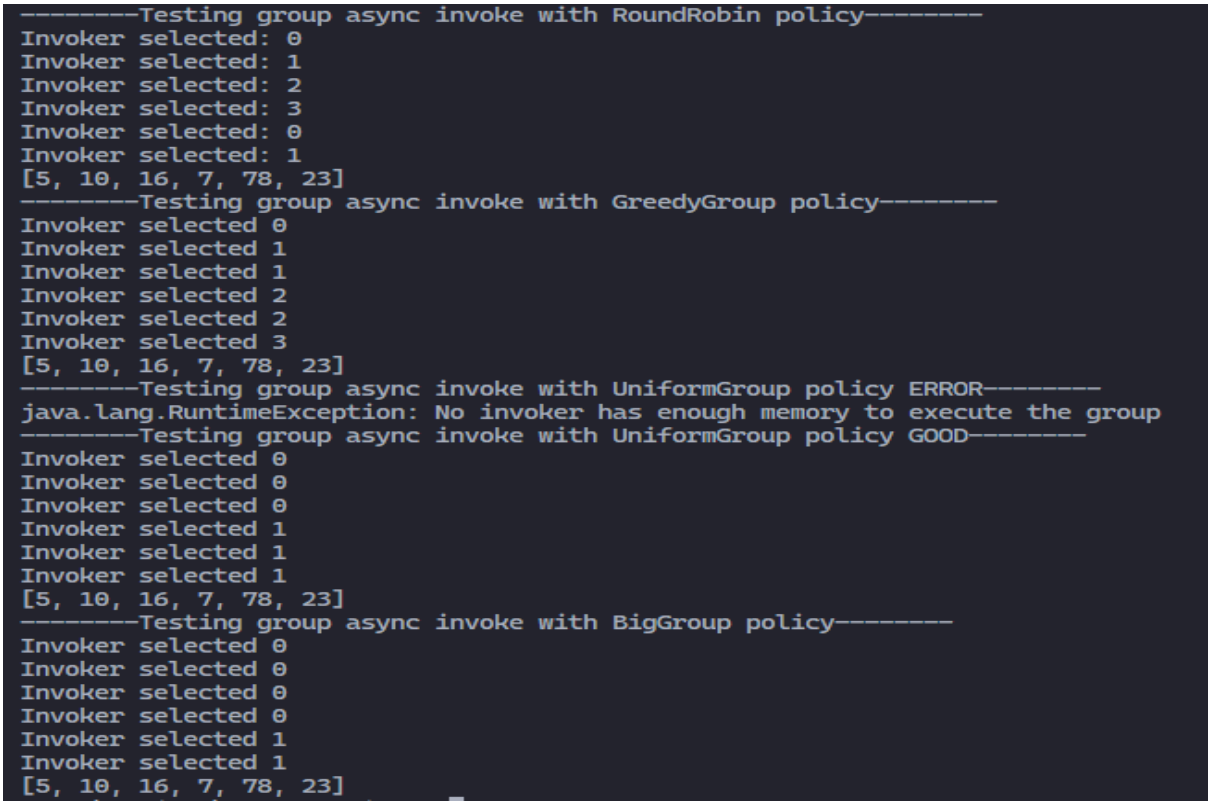
En este caso, registramos una acción de 512 MB en un controlador con 4 invokers de 1024 MB cada uno. Para el RoundRobin vemos que cada invocación se ejecuta en un invoker distinto. Para el GreedyGroup se ejecutan 2 acciones (lo máximo posible) por cada invoker. El UniformGroup se ha intentado ejecutar con un tamaño de grupo de 4, lo que es imposible ya que cada invoker solamente permite la ejecución de 2 acciones de 512 MB, por lo tanto salta un error en tiempo de ejecución avisando del problema. Por último, se ejecuta BigGroup con un tamaño de 2, lo que ejecutará 2 acciones por invoker.

```
--------Testing unique async invoke with RoundRobin policy--------
Invoker selected: 0
Invoker selected: 1
Invoker selected: 2
Invoker selected: 3
Invoker selected: 0
Invoker selected: 1
48
--------Testing unique async invoke with GreedyGroup policy--------
Invoker selected 0
Invoker selected 0
Invoker selected 1
Invoker selected 1
Invoker selected 2
Invoker selected 2
Invoker selected 3
Invoker selected 0
Invoker selected 0
72
--------Testing unique async invoke with UniformGroup policy--------
No invoker has enough memory to execute the group
--------Testing unique async invoke with BigGroup policy--------
Invoker selected 0
Invoker selected 0
Invoker selected 1
Invoker selected 1
Invoker selected 2
Invoker selected 2
Invoker selected 3
Invoker selected 3
64
```

Invocaciones grupales asíncronas (src.main.Tests.test_async.TestGroupAsync.java)

Realmente el invoke_async para invocaciones grupales es más que nada syntactic sugar, ya que hace lo mismo que el invoke_async único pero en un bucle para cada uno de los parámetros de la lista.

De esta manera, el funcionamiento es el mismo. Un aspecto a destacar de este test es que hemos vuelto a hacer que el UniformGroup falle, dándole un tamaño de grupo de 3 cuando cada invoker (de los 4) tiene 1024 y la acción a ejecutar ocupa 512 MB, de manera que no puede. Para el siguiente UnifromGroup lo que hemos hecho ha sido registrar la función otra vez pero con menos ocupación de memoria, en concreto 256 MB, de manera que ahora si que se puede asignar ese tamaño de grupo. Por último se ha ejecutado el BigGroup con un tamaño de grupo de 2, como se pueden empaquetar 2 grupos en un invoker ya que en total son 1024 MB, es lo que ha hecho el BigGroup.

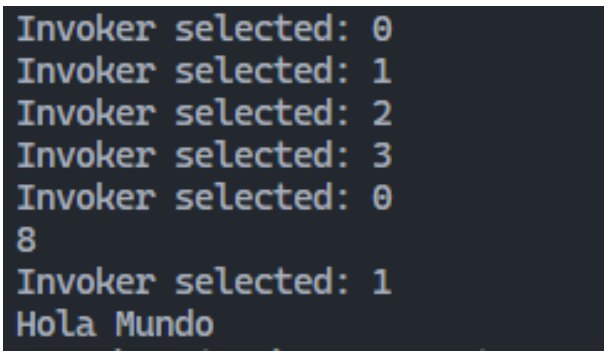


Invocaciones únicas síncronas (src.main.Tests.test_invoke.TestUniqueInvoke.java)

Para este caso, hemos ejecutado cinco invocaciones únicas síncronas. Para las invocaciones síncronas siempre tendremos memoria disponible a no ser que el tamaño en memoria de la acción que queramos ejecutar sea superior a la memoria de cualquiera de los invokers.

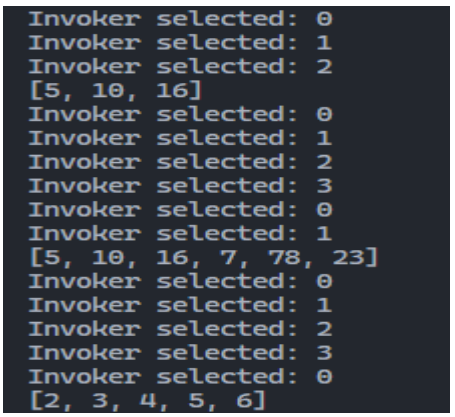
En este caso, hemos ejecutado una acción de 256 MB sobre 4 invokers de 1024 MB, como vemos, tenemos memoria suficiente.

Para la última invocación hemos ejecutado una acción que concatena Strings, así podemos probar la versatilidad que ofrece el controlador al permitir el registro de acciones que aceptan y devuelven distintos tipos de datos.



Invocaciones grupales síncronas (src.main.Tests.test_invoke.TestGroupInvoke.java)

En este caso ejecutamos una acción de 256 MB sobre 4 invokers de 1024 MB cada uno utilizando la política RoundRobin. La invocación grupal que más memoria requiere es la segunda, la cual realizará la invocación de 6 veces la misma acción para diferentes parámetros, lo que consumirá un total de 256 * 6 MB, como el controlador tiene suficientes recursos para ejecutar la llamada grupal, no habrá ningún problema.



Junit

Para la validación de los componentes del sistema, tanto el registro como invocaciones y control de memoria hemos utilizado unit testing implementado con la librería de junit.

Todo esto se puede ver en el proyecto de la primera parte, en el path src/test.

Un test bastante importante es el que comprueba si la concurrencia funciona correctamente, para comprobarlo hemos implementado una función que detiene la ejecución del programa durante 5 ms, y la hemos llamado 3 veces seguidas tanto para la versión síncrona como asíncrona. En la versión síncrona el tiempo de ejecución es mayor o igual a 15 y en la versión asíncrona es menor de 15, como podemos ver en la siguiente captura (los tests dan bien).

```java
// En este Test comprobamos que el tiempo de ejecución es mayor o igual a 15 ms
@Test
public void testInvokeSeq(){
    // Comprobar que el tiempo de ejecución es 15 y no 5 como sería de manera concurrente
    long start = System.currentTimeMillis();
    controller.invoke(actionName:"sleepAction", args:5);
    controller.invoke(actionName:"sleepAction", args:5);
    controller.invoke(actionName:"sleepAction", args:5);
    long end = System.currentTimeMillis();
    assertTrue(end - start ≥ 15);
}


// En este Test comprobamos que el tiempo de ejecución es menor a 15 ms (concurrencia)
@Test
// Test invoke_async
public void testInvokeAsync() {
    // Comprobar que el tiempo de ejecución es 5 y no 15 como sería de manera secuencial
    long start = System.currentTimeMillis();
    Future<Integer> fut1 = controller.invoke_async(actionName:"sleepAction", args:5);
    Future<Integer> fut2 = controller.invoke_async(actionName:"sleepAction", args:5);
    Future<Integer> fut3 = controller.invoke_async(actionName:"sleepAction", args:5);

    try {
        fut1.get();
        fut2.get();
        fut3.get();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }

    long end = System.currentTimeMillis();
    assertTrue(end - start < 15);    // Comprobamos que tarda menos de 15 ms (concurrencia)
}
```

Instalación y configuración

Para la primera parte del proyecto se requiere tener instalado JDK, para así poder hacer uso del código en Java. Una buena opción es utilizar VSCode con extensiones de Java para una mejor experiencia.

Para la segunda parte del proyecto también es necesario tener JDK instalado, además de las librerías de scala. En este caso, es mejor optar por el IDE IntelliJ, el cual gestiona de una manera muy amena las dependencias para poder trabajar con scala, definiendo un archivo de configuración .sbt y demás ajustes.

Para las dos partes es de vital importancia importar la clase Controller que es la encargada de gestionar toda la ejecución de acciones. No es necesario asignar una política de selección de invokers manualmente, ya que el controlador utiliza RoundRobin de manera predeterminada. En caso de querer usar otra estrategia, el desarrollador puede crear una (como se ha explicado antes) o se tienen que importar las otras estrategias, las cuales no vamos a explicar ya que no forman parte del sistema de ejecución de funciones como tal.

Una vez seguidos todos estos pasos, ya se puede pasar con la instanciación del controlador y el registro y invocación de funciones.

Ejemplos funciones y código Map Reduce

En este apartado veremos ejemplos de algunas funciones que se pueden registrar y invocar en nuestro framework Faas y al final veremos un caso de uso bastante interesante con el MapReduce.

Funciones

Gracias a la implementación de los métodos del controlador de manera genérica, podemos crear y registrar funciones con cualquier tipo de datos de entrada y salida. Veamos algunos ejemplos.

```
controller = new Controller(invokerCount:4, memoryPerInvoker:1024);

Function<Map<String, Integer>, Integer> add = x → x.get(key:"x") + x.get(key:"y");
Function<Map<String, String>, String> concat = x → x.get(key:"x") + x.get(key:"y");
Function<Integer, Integer> increment = x → x + 1;
Function<String, String> toUpperCase = x → x.toUpperCase();
Function<Integer, Integer> factorial = x → {
    int result = 1;
    for (int i = 1; i ≤ x; i++) {
        result *= i;
    }
    return result;
};

controller.registerAction(actionName:"addAction", add, memorySize:256);
controller.registerAction(actionName:"concatAction", concat, memorySize:256);
controller.registerAction(actionName:"incrementAction", increment, memorySize:256);
controller.registerAction(actionName:"toUpperCaseAction", toUpperCase, memorySize:256);
controller.registerAction(actionName:"factorialAction", factorial, memorySize:256);
```

Ejecución add

```
@Test
public void testInvokeActionSum() {
    int result = controller.invoke(actionName:"addAction", Map.of(k1:"x", v1:6, k2:"y", v2:2));
    assertEquals(8, result);
}
```

Ejecución concat

```
@Test
public void testInvokeActionConcat() {
    String result = controller.invoke(actionName:"concatAction", Map.of(k1:"x", v1:"Miguel ", k2:"y", v2:"Robledo"));
    assertEquals("Miguel Robledo", result);
}
```

Ejecución increment

```
@Test
public void testInvokeActionIncrement() {
    int result = controller.invoke(actionName:"incrementAction", args:5);
    assertEquals(6, result);
}
```

Ejecución touppercase

```
@Test
public void testInvokeActionUpperCase() {
    String result = controller.invoke(actionName:"toUpperCaseAction", args:"Miguel Robledo");
    assertEquals("MIGUEL ROBLEDO", result);
}
```

Ejecución factorial, para este caso hemos hecho una invocación grupal

```
@Test
public void testInvokeListIntegersFactorial() {
    List<Integer> input = Arrays.asList( ...a:1,2,3,4,5,6);
    List<Integer> result = controller.invoke(actionName:"factorialAction", input);
    for (int i = 0; i < result.size(); i++) {
        assertEquals(factorial(input.get(i)), result.get(i));
    }
}
```

MapReduce

Veremos el código tanto en Java como Scala. Necesitamos principalmente 3 clases:

- ➢ WordCount
- ➢ CountWords
- ➢ Reduce

WordCount java

```java
public class WordCount {

    /**
     * Counts the occurrences of words in the given list of strings.
     *
     * @param text the list of strings
     * @return a map containing the word counts
     */
    public static Map<String,Integer> wordCount(List<String> text) {
        Map<String,Integer> wordCountMap = new HashMap<>();

        for(String word : text) {
            word = word.toLowerCase().replaceAll(regex:"[^a-zA-Z]", replacement:"");

            // If it was a word, it is added to the dictionary
            if (!word.isEmpty()) {
                wordCountMap.merge(word, value:1, Integer::sum);
            }
        }

        return wordCountMap;
    }
}
```

WordCount scala

```scala
class WordCount {
  Miguel Robledo
  def wordCount(text: List[String]): Map[String, Int] = {
    text.flatMap(_.toLowerCase.replaceAll("[^a-zA-Z]", "").split(" "))
      .filter(_.nonEmpty)
      .groupBy(identity)
      .view.mapValues(_.length)
      .toMap
  }
}
```

CountWords java

```java
public class CountWords {

    /**
     * Counts the number of words in the given list of strings.
     *
     * @param text the list of strings
     * @return the number of words
     */
    public static int countWords(List<String> text) {
        int nWords = 0;

        for(String word : text) {
            word = word.toLowerCase().replaceAll(regex:"[^a-zA-Z]", replacement:"");

            // If it was a word, it is added to the dictionary
            if (!word.isEmpty()) {
                nWords++;
            }
        }

        return nWords;
    }
}
```

CountWords scala

```scala
class CountWords {
  Miguel Robledo
  def countWords(text: List[String]): Int = {
    text.flatMap(_.toLowerCase.replaceAll("[^a-zA-Z]", "").split(" "))
      .count(_.nonEmpty)
  }
}
```

Reduce java

```java
public class Reduce {

    /**
     * Reduces a list of word count maps into a single word count map.
     *
     * @param wordCountList the list of word count maps
     * @return a map containing the reduced word counts
     */
    public static Map<String, Integer> reduceWordCount(List<Map<String, Integer>> wordCountList) {
        Map<String, Integer> resultMap = new HashMap<>();

        for (Map<String, Integer> wordCount : wordCountList) {
            for (Map.Entry<String, Integer> entry : wordCount.entrySet()) {
                String word = entry.getKey();
                Integer count = entry.getValue();
                resultMap.put(word, resultMap.getOrDefault(word, defaultValue:0) + count);
            }
        }

        return resultMap;
    }

    /**
     * Reduces a list of word counts into a total count.
     *
     * @param countWordsList the list of word counts
     * @return the total count of words
     */
    public static int reduceCountWords(List<Integer> countWordsList){
        int total = 0;
        for (int count : countWordsList) {
            total += count;
        }
        return total;
    }
}
```

Reduce scala

```scala
class Reduce {
  Miguel Robledo
  def reduceWordCount(wordCountList: List[mutable.Map[String, Int]]): Map[String, Int] = {
    wordCountList.flatten              : List[(String, Int)]
      .groupBy(_._1)                   : Map[String, List[(String, Int)]]
      .view.mapValues(_.map(_._2).sum) : MapView[String, Int]
      .toMap                           : Map[String, Int]
  }

  Miguel Robledo
  def reduceCountWords(countWordsList: List[Int]): Int = {
    countWordsList.sum
  }
}
```

Para proceder con el MapReduce, es necesario leer los ficheros, procesando cada fichero a una lista de strings y guardando cada una de esas listas en otra lista que tendrá 10 elementos (los 10 ficheros leídos).

Este código implementado en la clase FilesReader no se ha hecho en Scala, sino que se ha utilizado la implementación en Java para la primera práctica y para la segunda.

```java
public class FilesReader {

    public static List<List<String>> readFiles() {
        String pathFiles = "src/main/Files";
        List<List<String>> textCollections = new ArrayList<>();

        try {
            Path folderPath = Paths.get(pathFiles);
            DirectoryStream<Path> fileStream = Files.newDirectoryStream(folderPath);

            // Load each text into a position in the list
            for (Path path : fileStream) {
                String filePath = path.toString();
                List<String> text = new ArrayList<>();
                File file = new File(filePath);

                // Each text is loaded into a list of strings
                try (Scanner scanner = new Scanner(file)) {
                    String word;
                    while (scanner.hasNext()) {
                        word = scanner.next();
                        text.add(word);
                    }
                } catch (FileNotFoundException e) {
                    System.out.println("IOException: " + e.getMessage());
                }

                textCollections.add(text);
            }

            fileStream.close();
        } catch (IOException e) {
            System.out.println(x:"There was a problem loading the texts");
        }

        return textCollections;
    }

}
```

Ahora, teniendo todo esto, podemos ver el MapReduce.

MapReduce java

```java
Controller controller = new Controller(invokerCount:10, memoryPerInvoker:1024);
Function<List<String>, Map<String, Integer>> func = list -> WordCount.wordCount(list);
controller.registerAction(actionName:"wordCount", func, memorySize:512);

Function<List<String>, Integer> func2 = list -> CountWords.countWords(list);
controller.registerAction(actionName:"countWords", func2, memorySize:512);

List<List<String>> textCollections = FilesReader.readFiles();

Future<List<Map<String, Integer>>> wordCountResults = controller.invoke_async(actionName:"wordCount", textCollections);
Future<List<Integer>> countWordsResults = controller.invoke_async(actionName:"countWords", textCollections);

try {
    Map<String, Integer> finalWordCountResult = Reduce.reduceWordCount(wordCountResults.get());
    int finalCountWordsResult = Reduce.reduceCountWords(countWordsResults.get());
    System.out.println(finalCountWordsResult);
    saveWordCountsToFile(finalWordCountResult, fileName:"src/main/Tests/test_mapreduce/results/mapReduceResults.txt");
} catch (Exception e) {
    System.out.println(e.getMessage());
}

controller.shutdown();
```

MapReduce scala

```scala
object TestMapReduce extends App {
  val controller = new Controller(10, 1024)

  val wordCount = new WordCount()
  val countWords = new CountWords()
  val reduce = new Reduce()

  val wordCountFunc: JavaList[String] => JavaMap[String, Integer] =
    (list: JavaList[String]) => {
      val scalaList = list.asScala.toList
      val wordCountResult = wordCount.wordCount(scalaList)
      wordCountResult.map { case (word, count) => (word, count: Integer) }.asJava
    }

  controller.registerAction("wordCount", wordCountFunc.asJava, 512)

  val countWordsFunc: JavaList[String] => Integer =
    (list: JavaList[String]) => {
      val scalaList = list.asScala.toList
      val countWordsResult = countWords.countWords(scalaList)
      countWordsResult: Integer
    }


  controller.registerAction("countWords", countWordsFunc.asJava, 512)

  val textCollections = FilesReader.readFiles()

  val wordCountResultsFuture: CompletableFuture[JavaList[JavaMap[String, Int]]] =
    controller.invoke_async("wordCount", textCollections)

  val countWordsResultsFuture: CompletableFuture[JavaList[Int]] =
    controller.invoke_async("countWords", textCollections)

  Try {
    val wordCountResults: JavaList[JavaMap[String, Int]] = wordCountResultsFuture.get()
    val countWordsResults: JavaList[Int] = countWordsResultsFuture.get()

    val scalaWordCountResults: List[mutable.Map[String, Int]] = wordCountResults.asScala.map(_.asScala).toList
    val scalaCountWordsResults: List[Int] = countWordsResults.asScala.toList



    val finalWordCountResult = reduce.reduceWordCount(scalaWordCountResults)
    val finalCountWordsResult = reduce.reduceCountWords(scalaCountWordsResults)

    val javaWordCountResults: JavaMap[String, Integer] = finalWordCountResult.map { case (k, v) => (k, v: Integer) }.asJava

    println(finalCountWordsResult)
    FilesReader.saveWordCountsToFile(javaWordCountResults, "src/main/scala/Tests/test_mapreduce/results/mapReduceResults.txt")
  } match {
    case Success(_) =>
    case Failure(exception) => println(exception.getMessage)
  }

  controller.shutdown()
}
```

En Scala la cosa se complica ya que debemos realizar conversiones entre los tipos de Java y Scala , ya que el WordCount, CountWords y Reduce están implementados en Scala pero el controlador está implementado en Java.

De esta manera, cuando definimos las funciones WordCount y CountWords vemos que tienen tipos de datos de Java, tanto de entrada como de salida (JavaList y JavaMap, los cuales se utilizan mediante el import java.util.{List => JavaList, Map => JavaMap), pero antes de llamar a wordCount sobre la instancia de WordCount, pasamos la lista a scala con la función asScala.toList.

Otro momento donde es importante realizar las conversiones es en el bloque Try.
Cuanto obtenemos los resultados de las llamadas asíncronas con get() tenemos la información con tipos de datos de colecciones Java, JavaList (java.util.List) para ser más precisos.

Tenemos que pasar esa información a tipos de datos de Scala. Para el caso de wordCount tenemos que hacer un .asScala.map(_.asScala).toList para pasar no solamente la JavaList a List de Scala sino que también cada uno de los JavaMap de las posiciones de la lista a Map de Scala. Para el countWords hacemos .asScala.toList y lo tenemos, ya que es una lista de enteros.

Una vez tenemos los resultados con tipos de datos de Scala podemos llamar a la función reduceWordCount y reduceCountWords de la instancia de la clase Reduce, ya que esta vez Reduce lo hemos implementado en Scala y utiliza tipos de datos de Scala.
El resultado de reduceWordCount es un tipo de datos de Scala, el cual lo tendremos que pasar a Java y así poder llamar al método saveWordCountsToFile el cual almacenará el resultado final del WordCount con Map y Reduce en un fichero. El resultado del conteo de palabras, CountWords, se mostrará por la salida estándar.

Ejemplos del marco en Java y Scala

Vamos a ver algunos ejemplos de utilización del framework tanto en Java como en Scala.

Java

Realizamos una invocación grupal asíncrona

```
Controller controller = new Controller(invokerCount:4, memoryPerInvoker:1024);
Function<Map<String, Integer>, Integer> f = x -> x.get(key:"x") + x.get(key:"y");
controller.registerAction(actionName:"addAction", f, memorySize:512);

List<Map<String, Integer>> input = Arrays.asList(
        Map.of(k1:"x", v1:2, k2:"y", v2:3),
        Map.of(k1:"x", v1:9, k2:"y", v2:1),
        Map.of(k1:"x", v1:8, k2:"y", v2:8),
        Map.of(k1:"x", v1:5, k2:"y", v2:2),
        Map.of(k1:"x", v1:12, k2:"y", v2:66),
        Map.of(k1:"x", v1:15, k2:"y", v2:8));



System.out.println(x:"————Testing group async invoke with RoundRobin policy————");
Future<List<Integer>> results = controller.invoke_async(actionName:"addAction", input);

try {
    System.out.println(results.get().toString());
} catch (Exception e) {
    System.out.println(e.getMessage());
}

controller.shutdown();
```

Realizamos unas cuantas invocaciones únicas asíncronas

```
Controller controller = new Controller(invokerCount:4, memoryPerInvoker:1024);
Function<Map<String, Integer>, Integer> f = x -> x.get(key:"x") + x.get(key:"y");
controller.registerAction(actionName:"addAction", f, memorySize:512);

// Se ejecutan 1 acción por invoker
System.out.println(x:"————Testing unique async invoke with RoundRobin policy————");
Future<Integer> fut1 = controller.invoke_async(actionName:"addAction", Map.of(k1:"x", v1:6, k2:"y", v2:2));
Future<Integer> fut2 = controller.invoke_async(actionName:"addAction", Map.of(k1:"x", v1:6, k2:"y", v2:2));
Future<Integer> fut3 = controller.invoke_async(actionName:"addAction", Map.of(k1:"x", v1:6, k2:"y", v2:2));
Future<Integer> fut10 = controller.invoke_async(actionName:"addAction", Map.of(k1:"x", v1:6, k2:"y", v2:2));
Future<Integer> fut11 = controller.invoke_async(actionName:"addAction", Map.of(k1:"x", v1:6, k2:"y", v2:2));
Future<Integer> fut12 = controller.invoke_async(actionName:"addAction", Map.of(k1:"x", v1:6, k2:"y", v2:2));

try {
    System.out.println(fut1.get() + fut2.get() + fut3.get() + fut10.get() + fut11.get() + fut12.get());
}catch (Exception e) {
    System.out.println(e.getCause().getMessage());
}

controller.shutdown();
```

Scala

Realizamos 6 invocaciones únicas asíncronas

```scala
val controller = new Controller(4, 1024)

val addAction: Function[JavaMap[String, Integer], Integer] = (t: JavaMap[String, Integer]) => {
  t.getOrDefault("x", 0) + t.getOrDefault("y", 0)
}

controller.registerAction("addAction", addAction, 256)


// Ejemplo para RoundRobin policy
val roundRobinResults = (1 ≤ to ≤ 6).map(_ => controller.invoke_async("addAction", Map("x" -> 6, "y" -> 2).asJava))
roundRobinResults.foreach(f => println(f.get()))

controller.shutdown()
```

Realizamos una invocación grupal asíncrona

```scala
val controller = new Controller(4, 1024)
val addAction: Function[JavaMap[String, Integer], Integer] = (x: JavaMap[String, Integer]) => {
  x.getOrDefault("x", 0) + x.getOrDefault("y", 0)
}
controller.registerAction("addAction", addAction, 512)
val inputScala = List(
  Map("x" -> 2, "y" -> 3),
  Map("x" -> 9, "y" -> 1),
  Map("x" -> 8, "y" -> 8),
  Map("x" -> 5, "y" -> 2),
  Map("x" -> 12, "y" -> 66),
  Map("x" -> 15, "y" -> 8)
)
val inputJava: JavaList[JavaMap[String, Int]] = inputScala.map(_.asJava).asJava
println("--------Testing group async invoke with RoundRobin policy--------")
var results = controller.invoke_async("addAction", inputJava)
try {
  println(results.get().toString)
} catch {
  case e: Exception => println(e.getMessage)
}
controller.shutdown()
```