Wonderfunc

Arquitectura Funcional Distribuida basada en Funciones Lambda

Alumno: Víctor Ceballos Espinosa

Tutor: José Juan Hernández Cabrera

Tutor: José Évora Gómez

Introducción 1

Estado del arte 2

Arquitecturas funcionales 2

Motores de abstracción 4

Caso de uso 5

Guía de usuario 8

Para empezar con Wonderfunc 8

Añadir nodos map y filter 9

Añadir nodo filter 9

Añadir nodo map 9

Mezclar nodos filter y map 10

Cambiar los contenedores de nodos 10

Uso de repositorio de funciones 11

Ejecución del Pipeline 12

Desarrollo del framework 14

Métodos 14

Test Driven Development 14

Gitflow. Gestión de versiones 15

Gestión de la configuración del software 16

Fases 16

Identificación del problema y de posibles competidores 16

Definición de objetivos 17

Definición de la API y del álgebra para la API de Wonderfunc 17

Implementación del Stream, Nodos (interfaces) y Pipeline 17

Implementación de los nodos locales 17

Implementación de NodeContainer y cambios de NodeContainer 18

Implementación de los FunctionRepositories 18

Herramientas 18

GIT 18

Maven 18

Java 18

Tests 19

Guía de referencia 20

Nodos 20

ExpressionExecutor 23

Mensajes 25

Contenedores de funciones 26

Repositorios de funciones 27

Conclusiones **29**
Punto de vista personal 29
Punto de vista de contribución 29
Posibles continuaciones de Wonderfunc 30
Bibliografía **3**2
Apéndice. Justificación de las competencias específicas cubiertas **3**3

Glosario

Framework: Puede ser traducido como marco de trabajo. También se entiende como un entorno pensado para hacer más sencilla la programación de cualquier aplicación o herramienta actual.

Arquitectura funcional: Se trata de una arquitectura en la que los módulos son funciones. Se basa en la programación funcional, en la que la que los módulos ni tienen un estado interno ni comparten un estado con otros módulos. Esto se traduce en que las dependencias entre módulos no introduzcan restricciones a la hora de reorganizar dichos módulos.

Ejecución distribuida: Forma de llevar a cabo una ejecución de código en la que dicho código no se ejecuta en una única máquina si no en varias.

Motor de abstracción: Permiten a los desarrolladores abstraerse de los detalles de implementación con el objetivo de definir un pipeline funcional de un sistema. Normalmente, permiten desplegar el pipeline funcional en un única *plataforma de computación en la nube* como Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform o Algorithmia. La definición del pipeline se suele realizar con un lenguaje propio del motor de abstracción o lenguajes genéricos como XML o JSON.

Pipeline: Dentro de una arquitectura funcional, un pipeline es una organización específica de los módulos que la forman.

Plataforma de computación en la nube: Plataforma de computación que ofrece servicios a través de internet.

Java: Lenguaje de programación de alto nivel, de propósito general, concurrente, orientado a objetos y basado en clases. Se trata de un lenguaje de programación de tipado fuerte y estático.

Expresión lambda: También reciben el nombre de funciones lambda. Se trata de funciones simples y anónimas que no pueden ser recursivas. De forma general, estas funciones lambda pueden ser sustituidas por rutinas nombradas. Por esto, su utilidad recae en la conveniencia a la hora de organizar y escribir código más limpio y de mejor calidad por lo tanto.

Stream: Flujo de datos que fluye por un pipeline.

Introducción

Wonderfunc es un framework de código abierto orientado al desarrollo de proyectos de Big Data. El objetivo de este framework es apoyar el desarrollo de aplicaciones escalables facilitando la creación de arquitecturas funcionales que pueden ser ejecutadas de modo distribuido.

Es un *motor de abstracción* que permite describir y desplegar *pipelines de funciones*, primordiales en este tipo de arquitecturas. El proyecto está disponible en GitHub [1] y se distribuye con licencia Apache 2.0.

Genéricamente, los motores de abstracción permiten a los desarrolladores abstraerse de los detalles de implementación con el objetivo de definir un pipeline funcional de un sistema. Normalmente, los motores de abstracción permiten desplegar el pipeline funcional en un única *plataforma de computación en la nube* como Amazon Web Services (AWS) [2], Microsoft Azure [3], Google Cloud Platform [4] o Algorithmia [5]. La definición del pipeline se suele realizar con un lenguaje propio del motor de abstracción o lenguajes genéricos como XML o JSON.

En concreto, Wonderfunc permite que esta definición se realice directamente como código Java, a la vez que facilita la integración entre varias plataformas de computación para realizar ejecuciones distribuidas.

Con Wonderfunc la defincón del pipeline es muy expresiva, es decir, pocas líneas de código permiten expresar toda la complejidad de la arquitectura funcional sin perder facilidad de comprensión. En este sentido, para mejorar la expresividad de la definición, los *pipelines* incluyen funciones lambda [6].

Esta expresividad contribuye a una mejora en la flexibilidad a la hora de implementar arquitecturas funcionales distribuidas. Se trata de poder modificar fácilmente la manera en la que se ejecuta una arquitectura funcional, sin que ello afecte a otras partes del código.

Este documento está dividido en 5 secciones. En la sección *Antecedentes y estado del arte* se describen brevemente las técnicas previas que dan lugar a la necesidad de wonderfunc. Más adelante, en la sección *Utilidad. Caso de uso*, se expone la utilidad del framework Wonderfunc así como un posible caso de uso. Para continuar, en la *Guía de usuario* se explica cómo usar Wonderfunc para los principales casos de uso de una forma introductoria. En la sección *Desarrollo del framework* se muestran las herramientas y métodos usados para implementar Wonderfunc así como las fases en las que se ha dividido. Por último, en la sección *Conclusiones*, se dan a conocer las conclusiones que se han obtenido desde un punto de vista personal y desde el punto de vista de la contribución.

Estado del arte

Wonderfunc se inspira en la programación funcional, haciendo uso de expresiones lambda para creación arquitecturas funcionales que pueden ser distribuibles en diferentes plataformas de computación en la nube así como en local. Todo esto se lleva a cabo mediante la descripción de un *pipeline* por el que fluye un conjunto de datos (*stream)*.

Arquitecturas funcionales

Las arquitecturas funcionales son arquitecturas modulares que están basadas en la programación funcional. La programación funcional está basada en el hecho de que no existe un estado interno dentro de las funciones. En lugar de esto, se parte de un estado inicial y mediante el uso de funciones, se va cambiando dicho estado, de tal forma que el estado de salida de una función es el estado de entrada de la siguiente. En estas arquitecturas funcionales, los módulos son funciones donde no se guarda ningún estado. Tampoco se comparten estados entre funciones, de tal manera que el acoplamiento entre módulos, no introduce restricciones a la hora de distribuirlos. Esto tiene como ventaja, que estas funciones pueden estar distribuidos en diferentes máquinas permitiendo por ende, una ejecución distribuida. Adicionalmente, permite reorganizar y agrupar estos módulos de diferentes maneras, aportando una gran flexibilidad. En Wonderfunc a las diferentes organizaciones de los módulos, se les ha llamado Pipeline.

En Wonderfunc para definir estos *pipelines* manteniendo una gran expresividad, se utilizan expresiones lambda. Estas expresiones reciben también el nombre de funciones lambda. Se trata de funciones simples y anónimas que no pueden ser recursivas. De forma general, estas funciones lambda pueden ser sustituidas por rutinas nombradas. Por esto, su utilidad recae en la conveniencia a la hora de organizar y escribir código más limpio y de mejor calidad por lo tanto. La anatomía de una expresión lambda se puede observar en las siguientes imágenes.

```
a -> a * 2
```

```
(a, b) -> b > a
```

Figura 1. Anatomía de una expresión lambda

Por un lado, a la izquierda de la flecha, se declaran los parámetros de la expresión. En caso de que sea un único parámetro, se puede prescindir de los paréntesis. Por otro lado, a la derecha de la flecha, se escribe la expresión que se ejecutará.

La forma de especificar la función a ejecutar en cada nodo del *pipeline* es haciendo uso de estas expresiones lambda en Wonderfunc.

Una arquitectura funcional permite llevar a cabo una ejecución distribuida debido a la ausencia de restricciones a la hora de organizar y agrupar los módulos. Una ejecución distribuida consiste en la ejecución de los módulos que conforman la arquitectura en diferentes máquinas.

Esto se permite en Wonderfunc cambiando el contenedor de funciones. De esta forma, se podría ejecutar parte del *pipeline* en una plataforma de computación en la nube como Amazon Web Services (AWS) y otra parte en local u otra plataforma.

Motores de abstracción

Por la naturaleza de Wonderfunc, este framework se clasifica como un motor de abstracción en el ámbito de las tecnologías utilizadas en Big Data. Se considera un motor de abstracción ya que, mediante su API, abstrae la creación de pipelines distribuibles en diferentes plataformas de computación funcional en la nube y la integración entre ellas. Dentro de esta categoría, actualmente las tecnologías existentes más utilizadas son Apache Pig [7] y Cascading [8].

Tal y como se comentó en la sección de arquitecturas funcionales, dentro del contexto de Wonderfunc, un *pipeline* es una organización específica de los diferentes módulos por la que fluirán unos datos. En este caso, estos módulos serán nodos en los que se encuentran las diferentes funciones lambda que irán modificando el flujo de datos (stream) que atraviesa dicho *pipeline*.

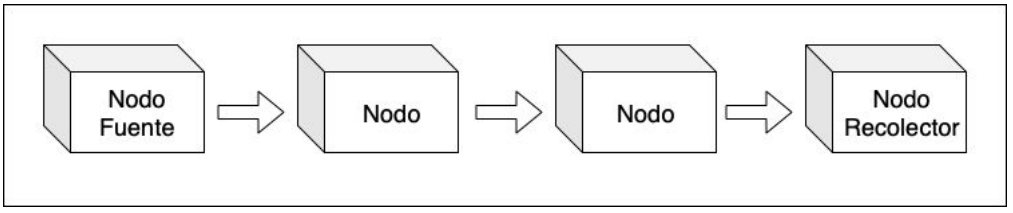

Figura 2. Pipeline

En el caso de Wonderfunc, cuando se hace referencia al Stream, se está hablando de los datos que atraviesan el *pipeline* siendo transformados por las diferentes funciones que lo conforman. Algún ejemplo de *stream* de datos podría ser una lista de números o una colección de imágenes.

Al clasificar Wonderfunc como un motor de abstracción, aparecen unos competidores que también son clasificados como motores de abstracción para la definición de *pipelines* funcionales:

- **Apache Pig:** *"Pig es una plataforma de alto nivel para crear programas MapReduce utilizados en Hadoop. El lenguaje de esta plataforma es llamado Pig Latin. Pig Latin abstrae la programación desde el lenguaje Java MapReduce en una notación que*

hace de MapReduce programación de alto nivel, similar a la de SQL para sistemas RDBMS."

- **Cascading:** *"El Ecosistema Cascading es una colección de aplicaciones, lenguajes y APIs para el desarrollo de aplicación con un uso intensivo de datos.*

  En el núcleo del ecosistema se encuentra Cascading, una API en Java [9] *para definir flujos de datos complejos y su integración con sistemas de back-ends y un planificador de consultas para mapear y ejecutar flujos lógicos hacia una plataforma de computación*"

Comparando estos competidores con Wonderfunc, se puede observar que Apache Pig se asemeja bastante a Wonderfunc en cuanto al uso de la abstracción para facilitar la creación de arquitecturas funcionales Sin embargo, está centrado en abstraer la ejecución de programas MapReduce. Wonderfunc por otro lado, está centrado en la ejecución de un Pipeline de funciones lambda en diferentes plataformas entre los que podría estar Hadoop.

Por otro lado, Cascading es mucho más cercano a Wonderfunc. Como se puede ver en la cita anterior, Cascading está abierto a conectar diferentes sistemas de *back-end* para hacer pasar por ellos un flujo de datos. En Wonderfunc se ha puesto mucho trabajo en simplificar la API lo máximo posible aumentando la expresividad y aportando una gran flexibilidad. Esta es una de las mayores mejoras frente a Cascading que presenta Wonderfunc.

Caso de uso

Wonderfunc es un framework orienteado a facilitar la construcción de *pipelines* de funciones lambda que pueden ser ejecutados de forma distribuida en diferentes plataformas de computación en la nube o en su defecto en local. Estas plataformas de computación en la nube pueden ser utilizadas para diseñar software que sea escalable. Esto hace que Wonderfunc pueda ser utilizado para hacer grandes procesamientos de datos en paralelo además de facilitar la escalabilidad. Para hacer uso del framework de Wonderfunc se ha diseñado una API sencilla, expresiva, flexible e intuitiva. Esto hace que crear un *pipeline* para procesar datos usando Wonderfunc sea extremadamente simple.

Wonderfunc puede ser utilizado en innumerables casos de uso. Sin embargo, se ha pensado en uno específicamente con el objetivo de ilustrar lo anteriormente comentado acerca de la conveniencia de Wonderfunc.

Pensemos en una empresa que vende sus productos (por ejemplo en Amazon) y quiere hacer un análisis de esos productos en base a los comentarios que realizan los consumidores sobre estos. Este análisis pretende obtener de todos los comentarios, aquellos que son clasificados como negativos para poder centrarse en esos detalles con el objetivo de mejorar el producto. En el código que se muestra a continuación, se omite la fase de obtención de esos comentarios y se salta directamente a la fase de análisis usando Wonderfunc para procesarlos.

Para simular la fase de obtención de los comentarios, se ha implementado un método que devuelve una lista de *Strings*, de tal manera que cada uno de estos, representa un comentario sobre el producto.

```
private static List<String> list() {

    return Arrays.asList("I hate it!",
            "I really like the final design. It is very ergonomic",
            "This product doesn't work at all...",
            "I enjoy using this product most of the time",
            "I wish I had known this before",
            "This is awesome!",
            "I will never stop using this",
            "This is pointless",
            "I don't like the way this works...");
}
```

Figura 3. Método para obtener los comentarios

Una vez ha sido aclarado esto, se explicará cómo se ha usado Wonderfunc para conseguir el objetivo enunciado en la parte superior con el siguiente código:

```java
private static void useCase() throws InterruptedException {

    List<Boolean> output = new ArrayList<>();

    FunctionRepository algorithmia = new Algorithmia("sim4SmnjN9o5CRPEKS4QxTJBWLg1");
    Function<String, String> sentimentAnalysis = algorithmia.function("nlp/SentimentAnalysis/1.0.5");

    Pipeline<Boolean> pipeline = new Stream<>(list())
            .on(new LocalNodeContainer())
                .map(s -> sentimentAnalysis.apply(s))
                    .with(new RemoteExpressionExecutor(new CommentMarshalling()))
                .map(d -> d <= 0)
            .collectTo(output);

    Thread pipelineThread = pipeline.execute();

    pipelineThread.join();

    for (int i = 0; i < output.size(); i++) if (output.get(i)) System.out.println(list().get(i));
}
```

Figura 4. Método para el caso de uso

En primer lugar como se puede observar, se crea una lista donde se guardará la salida del Pipeline mediante el método *collectTo*. A continuación, se crea un repositorio de funciones para Algorithmia. Es importante notar que para crear este repositorio de funciones, se le tiene que pasar por constructor a Algorithmia la *API_KEY* del usuario de Algorithmia.

Más adelante, se crea un *Stream* pasándole la lista de comentarios que se crea al llamar al método *list*.

Con todos estos preparativos, se puede empezar a construir el *Pipeline*. En primer lugar se llama al método *on,* pasando por parámetros un *LocalNodeContainer* para especificar que los nodos que se añadan a continuación, serán de tipo local. Estos nodos tendrán la función lambda a ejecutar cuando reciban un dato proveniente del nodo anterior. A continuación, se llama al método *map* para crear un nodo de tipo *map*. En la expresión lambda que se ejecutará en ese nodo de tipo map, es importante notar que se llama al método *apply* de una función escogida haciendo uso del repositorio de funciones de Algorithmia. Al querer ejecutar una función que se encuentra en un repositorio de funciones, es necesario especificar que se quiere utilizar un *RemoteExpressionExecutor* haciendo uso del método *with*. Esto es así porque por defecto se aplica el *LocalExpressionExecutor*, ejecutándose dicha función de forma síncrona. Sin embargo, para ejecutar una función de un repositorio de funciones, es necesario que sea ejecutado de forma asíncrona. A este *RemoteExressionExecutor* se le pasa por contructor un *CommentMarshalling.* Este objeto tiene la responsabilidad de transformar los diferentes datos que lleguen al nodo para poder mandarlos a ejecutar la función en Algorithmia. Adicionalmente, tienen la responsabilidad de

transformar los resultados provenientes de Algorithmia a mensajes a enviar al siguiente nodo del *pipeline.*

La función de Algorithmia en cuestión, para cada comentario devuelve un número entre -1 y 1, de tal manera que un -1 significa que el comentario es muy negativo y un 1 significa que el comentario es muy positivo.

Pues bien, tras añadir el nodo de tipo *map* para hacer uso de la función de Algorithmia, se añade otro nodo de tipo *map*, cuya funcionalidad es mapear los números menores o iguales a cero a *True* y los mayores de cero a falso. Esto genera una lista de valores booleanos de la siguiente forma:

[true, false, false, true, ...]

Esta salida es la que se guarda en la lista creada al principio llamada *output* mediante el método *collectTo*.

Una vez se ha creado el *pipeline* que se quiere ejecutar, se ejecuta llamando al método *execute.* Este método como se puede comprobar, devuelve un objeto de la clase *Thread.* A continuación, se espera a que se termine de ejecutar dicho hilo.

A partir de aquí, lo único que queda por hacer es recorrer dicha lista y en caso de que el valor sea *true*, se imprime el comentario inicial. En caso de que no lo sea, no se imprime el comentario.

Desde este punto se podría continuar con el análisis usando los comentarios que han sido categorizados como negativos.

Guía de usuario

Esta guía de usuario pretende ayudar al usuario de Wonderfunc a utilizar el framework para los principales casos de uso. Para casos más específicos, se tendrá que ir a la guía de referencia.

Para empezar con Wonderfunc

Una vez se han incluido las dependencias, para poder utilizar Wonderfunc, bastará con instanciar un objeto de la clase Stream pasándole una lista de objetos. Estos objetos serán los que el nodo fuente comenzará a transmitir por el *pipeline*.

```
public class Main() {
    public static void main(String[] args) {
        Stream<Integer> stream = new Stream<>(list());
    }

    private static List<String> list() {
        return Arrays.asList("Hello this is a prove to check if everything went correctly".split(" "));
    }
}
```

Figura 5. Creación de Stream

Sin embargo, con el código anterior no vamos a conseguir hacer mucho. Antes de continuar, es importante recalcar la diferencia entre un *stream* y un *pipeline*. Un *stream* no es más que el flujo de datos. Un *pipeline* es un conjunto de nodos, incluidos un nodo fuente y un nodo recolector al final del mismo, por el que fluye un conjunto de datos. Pues bien, el nodo inicial se añade automáticamente al crear un objeto de la clase Stream, pero el nodo recolector será necesario añadirlo manualmente ejecutando el método *collectTo* y pasándole una lista donde se guarde el resultado de la ejecución:

```
public class Main() {
    public static void main(String[] args) {
        List<Integer> outputList = new ArrayList<>();
        Pipeline<Integer> pipeline = new Stream<>(list()).collectTo(outputList);
    }

    private static List<String> list() {
        return Arrays.asList("Hello this is a prove to check if everything went correctly".split(" "));
    }
}
```

Figura 6. Creación de Pipeline básico

Ya hemos conseguido con el código anterior que el conjunto de datos fluya por todo el *pipeline* hasta llegar al último nodo (nodo recolector), añadido al usar el método *collectTo*. Es importante notar que al añadir el nodo recolector, ya se pasa a tener un objeto de tipo Pipeline por lo explicado anteriormente. Desafortunadamente, este código no hace ningún

tipo de cambio en los datos. Para filtrar o mapear los datos, será necesario añadir nodos de tipo *map* para mapear o de tipo *filter* para filtrar.

Añadir nodos *map* y *filter*

Tras crear el Stream de datos tal y como se vió en el código mostrado anteriormente, se pueden añadir nodos al *pipeline* de tipo *map* o tipo *filter.*

Añadir nodo *filter*

Para añadir un nodo *filter*, bastará con usar el método *filter*, pasando por parámetro una función lambda que cumpla con la interfaz funcional *Predicate*.

```
public class Main() {
    public static void main(String[] args) {
        List<String> outputList = new ArrayList<>();
        Pipeline<String> pipeline = new Stream<>(list())
                        .filter(e -> e.contains("e"))
                        .collectTo(outputList);
    }

    private static List<String> list() {
        return Arrays.asList("Hello this is a prove to check if everything went correctly".split(" "));
    }
}
```

Figura 7. Creación de Pipeline con un nodo de tipo *filter*

Añadir nodo *map*

Para añadir un nodo *map*, se tiene que usar el método *map*, pasando por parámetro una función lambda que cumpla con la interfaz funcional *Function*.

```
public class Main() {
    public static void main(String[] args) {

        List<Integer> output = new ArrayList<>();

        Pipeline pipeline = new Stream<>(list())
                        .map(String::length)
                        .collectTo(output);
    }

    private static List<String> list() {
        return Arrays.asList("Hello this is a prove to check if everything went correctly".split(" "));
    }
}
```

Figura 8. Creación de Pipeline con un nodo de tipo *map*

Mezclar nodos *filter* y *map*

Es posible usar en el mismo *pipeline* nodos *filter* y nodos *map* intercalados de cualquier forma que se desee.

```
public class Main() {
    public static void main(String[] args) {

        List<Integer> output = new ArrayList<>();

        Pipeline<Integer> pipeline = new Stream<>(list())
                        .map(String::length)
                        .filter(l -> l > 5)
                        .collectTo(output);
    }

    private static List<String> list() {
        return Arrays.asList("Hello this is a prove to check if everything went correctly".split(" "));
    }
}
```

Figura 9. Creación de Pipeline mezclando nodos *filter* y *map*

Cambiar los contenedores de nodos

Hasta ahora, todo el *pipeline* se ejecuta en local. Sin embargo, Wonderfunc permite la ejecución distribuida del *pipeline* en diferentes plataformas de computación funcional en la nube. Para conseguir cambiar los contenedores de nodos donde se van a ejecutar dichas funciones, bastará con tras haber creado el Stream, usar el método *on* pasándole por parámetro un objeto que implemente la interfaz creada llamada *NodeContainer*. Inicialmente se ha implementado una clase llamada *LocalNodeContainer* para ejecutar en local el *pipeline*. Sin embargo, Wonderfunc es una framework de código abierto, lo que quiere decir que cualquiera puede implementar y añadir nuevas clases que implementen la interfaz *NodeContainer* aportando nueva funcionalidad.

Por defecto se aplica el *LocalNodeContainer*. A modo de ejemplo, se muestra a continuación cómo añadirlo de forma explícita.

```
public class Main() {
    public static void main(String[] args) {

        List<Integer> output = new ArrayList<>();

        Pipeline<Integer> pipeline = new Stream<>(list())
                        .on(new LocalNodeContainer())
                            .map(String::length)
                            .filter(l -> l > 5)
                        .collectTo(output);
    }

    private static List<String> list() {
        return Arrays.asList("Hello this is a prove to check if everything went correctly".split(" "));
    }
}
```

Figura 10. Creación de Pipeline especificando el NodeContainer

A continuación se muestra un ejemplo de *pipeline* distribuido entre la máquina en local y AWS. En este ejemplo se asume que la clase que implementa *NodeContainer* para AWS está implementada.

```
public class Main() {
    public static void main(String[] args) {

        List<Integer> output = new ArrayList<>();

        Pipeline<Integer> pipeline = new Stream<>(list())
                        .on(new LocalNodeContainer())
                            .map(String::length)
                        .on(new AWS("aws_access_key_id", "aws_secret_access_key"))
                            .filter(l -> l > 5)
                        .collectTo(output);
    }

    private static List<String> list() {
        return Arrays.asList("Hello this is a prove to check if everything went correctly".split(" "));
    }
}
```

Figura 11. Creación de Pipeline cambiando dos veces el NodeContainer

Uso de repositorio de funciones

Wonderfunc permite, a parte de ejecutar funciones lambda creadas en tiempo ejecución en diferentes contenedores de nodos, usar funciones ya creadas y almacenadas en lo que se ha llamado un *FunctionRepository*. Este concepto es explicado en profundidad en la guía de referencia. Para usar funciones que se encuentren alojadas en un repositorio de funciones, habrá que crear un objeto de la clase FunctionRepository y escoger la función a utilizar en el nodo de tipo *map:*

```
public class Main() {
    public static void main(String[] args) {

        List<Boolean> output = new ArrayList<>();

        FunctionRepository algorithmia = new Algorithmia("YOUR_API_KEY");
        Function<String, String> sentimentAnalysis = algorithmia.function("nlp/SentimentAnalysis/1.0.5");

        Pipeline<Boolean> pipeline = new Stream<>(list())
                        .on(new LocalNodeContainer())
                            .map(s -> sentimentAnalysis.apply(s))
                                .with(new RemoteExpressionExecutor(new CommentMarshalling()))
                            .map(d -> d <= 0)
                        .collectTo(output);
    }

    private static List<String> list() {
        return Arrays.asList("Hello this is a prove to check if everything went correctly".split(" "));
    }
}
```

Figura 12. Creación de Pipeline usando un FunctionRepository

Algorithmia es una plataforma que contiene una gran cantidad de funciones lambda que implementan diferentes algoritmos incluyendo algoritmos de *Machine Learning.*

Para ejecutar funciones que se encuentren en estos repositorios de funciones, es necesario especificar que se quiere utilizar un *RemoteExpressionExecutor* para ejecutar asíncronamente dicha función. Esto está explicado en profundidad en la guía de referencia.

Ejecución del Pipeline

Para ejecutar el Pipeline, tras haberlo completado con el método *collectTo*, se tendrá que llamar al método *execute.*

```
public class Main {
    public static void main(String[] args) throws InterruptedException {

        List<Integer> output = new ArrayList<>();
        NodeContainer local = new LocalNodeContainer();

        Thread pipelineThread = new Stream<>(list())
                .on(local)
                    .filter(s -> s.contains("y"))
                    .map(String::length)
                .collectTo(output)
                .execute();
    }

    public static List<String> list() {
        return Arrays.asList("Hello this is a test to check if everything went correctly".split(" "));
    }
}
```

Figura 13. Ejecución de un Pipeline

Cómo se puede observar, al llamar al método *execute()*, se devuelve un hilo. Esto se ha hecho así para que el desarrollador que esté usando Wonderfunc tenga total libertad para decidir si la ejecución la quiere hacer en paralelo o si de lo contrario prefiere esperar a que se termine de ejecutar el Pipeline.

Desarrollo del framework

Para llevar a cabo la implementación de Wonderfunc, se ha hecho uso de una serie de métodos y herramientas. Adicionalmente, dicha implementación se ha dividido en una serie de fases que han facilitado llevar a cabo un método ágil de trabajo basada en iteraciones similar a SCRUM.

Métodos

Test Driven Development

TDD [10] son las siglas de *Test Driven Development* (Desarrollo guiado por pruebas). Se trata de un método en el que el desarrollo avanza gracias a la construcción de tests unitarios. En esta ocasión, se han ido implementando tests unitarios para ir implementando y probando las nuevas funcionalidad. TDD está dividido en tres etapas claramente diferenciadas:

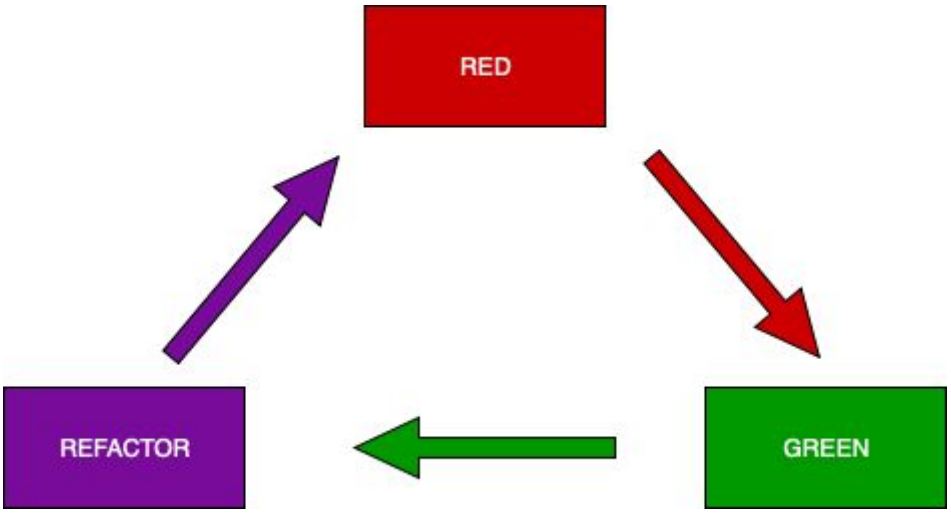

Figura 14. TDD

- **Red:** Se escribe un test que falla. Este test será escrito en función del requisito que se quiera implementar. Si esta prueba no falla, significa que el requisito ya está implementado o que la prueba es errónea.

- **Green:** Se escribe la menor cantidad de código posible para implementar el requisito y pasar la prueba.

- **Refactor:** Principalmente se utiliza para eliminar duplicidades en el código. Consiste por lo tanto en “limpiar” el código. Para llevar a cabo esta limpieza, se van haciendo

pequeños cambios y pasando las pruebas para comprobar que no se ha introducido algún error.

A lo largo de la implementación de Wonderfunc, ha habido una gran cantidad de reuniones. Pues bien, al inicio de las reuniones, se ha hecho una refactorización del código con el objetivo de limpiarlo y hacer revisiones de código junto con los tutores. Más adelante se decidía los nuevos requisitos a implementar y por último, fuera de las reuniones, se ha escrito las pruebas y el código mínimo necesario para pasarlas. Este ciclo se ha seguido hasta completar la implementación.

Gitflow. Gestión de versiones

Para gestionar las versiones y trabajar en equipo entre los tutores y el alumno, se ha utilizado Git [11]. Más precisamente, se ha utilizado el método Gitflow [12]. Gitflow consiste en organizar las ramas de un repositorio de tal forma que cada rama tiene unas características determinadas, en las que sólo pueden existir *commits* con ciertas condiciones. Principalmente existen dos ramas:

- **Master:** Los *commits* que se encuentren en esta rama, deben estar preparados para pasar a producción.

- **Develop:** En esta rama, se encuentran los *commit* que conformarán la siguiente funcionalidad en pasar a producción.

Cada vez que se incorpora código a la rama *master*, se tiene una nueva versión.

Además de estas ramas principales, existen tres tipos de ramas secundarias:

- **Feature:** Esta rama sale siempre de la rama *develop*. Adicionalmente, se incorpora únicamente con *develop*. En esta rama se incluye código de funcionalidades que una vez terminadas, serán añadidas a la rama *develop*.

- **Release:** Se origina a partir de develop. Sin embargo, se incorpora siempre a *master* y *develop*. Esta rama existe como paso previo a añadir código a la rama *master*. En ella se hacen los últimos ajustes y se corrigen los últimos errores.

- **Hotfix:** Esta rama se origina a partir de *master* y se incorpora tanto a *master* como a *develop*. Se utiliza para hacer pequeñas correcciones en el código en producción. La diferencia principal con la rama *release* es que estos correcciones no se planifican.

Para implementar Wonderfunc, la forma en la que se ha utilizado Gitflow ha sido la siguiente. Tras dejar claros los nuevos requisitos en las reuniones, al empezar a trabajar en cualquiera de ellos, se creaba una nueva rama *feature* llamada con el nombre resumido del requisito a implementar. Se trabajaba en dicha rama y una vez terminada la implementación, se incorporaba a *develop*. Cuando todos los requisitos de la iteración estaban implementados y sus respectivas ramas *feature* fueron incorporadas en la rama *develop*, en la siguiente reunión se abría una rama *release* a partir de *develop*. En esta rama, junto con la ayuda de los tutores, se refactorizaba y limpiaba el código. Al finalizar esas sesiones de revisión de código, se incorporaba la rama *release* en las ramas *master* y *develop*.

Por último, hubo casos en los que se encontraron errores mientras se implementaban nuevas funcionalidades. Para solventarlos, se abrieron ramas de tipo *hotfix* y se incorporaron tanto a *master* como a *develop* tal y como indica el método Gitflow.

Gestión de la configuración del software

La Gestión de la Configuración del Software (GCS/SCM) es un conjunto de actividades diseñadas para identificar y definir los elementos en el sistema que probablemente cambien, controlando el cambio de estos elementos a lo largo de su ciclo de vida, estableciendo relaciones entre ellos, definiendo mecanismos para gestionar distintas versiones de estos elementos, y auditando e informando de los cambios realizados.

Para conseguir gestionar la configuración del software, se ha hecho uso de Maven.

Fases

A lo largo de la implementación de Wonderfunc, se ha pasado por una serie de fases:

- En primer lugar, se tuvo que identificar el problema. En esta ocasión, el problema identificado consiste en la falta de una herramienta que permitiese desde Java, crear un *pipeline* de funciones lambda con la posibilidad de conectarse a diferentes plataformas de computación en la nube para el procesamiento de grandes cantidades de datos mediante una API sencilla, expresiva e intuitiva. Una vez el problema estaba identificado, se estudiaron los posibles competidores de Wonderfunc. Para poder reconocer los competidores de Wonderfunc, se tuvo que categorizar Wonderfunc. Sin duda alguna cae dentro de la categoría de motores de abstracción. Dentro de esta categoría, se ha encontrado Apache Pig y Cascading. Una vez conocidos los posibles competidores, se estudió cómo se podrían mejorar, obteniendo como resultado Wonderfunc.

- Teniendo claro el problema a resolver, se establecieron unos objetivos iniciales. Cabe destacar que estos objetivos fueron cambiando durante la implementación. Los objetivos iniciales y principales fueron:

    1. Crear una API sencilla, legible, expresiva, flexible e intuitiva.
    2. Crear una herramienta para la creación de Pipelines para el procesamiento de datos altamente escalable.

    Tal y como se ha comentado, estos macro objetivos fueron cambiando a lo largo del tiempo, entre los que se destaca el cambio de implementación de un *pipeline* en el que todos sus nodos se ejecutarán en la misma plataforma de computación en la nube, a un *pipeline* en el cual sus nodos podían ser ejecutados en diferentes plataformas de computación en la nube o incluso en local.

- Lo primero que hicimos tras establecer los objetivos, fue diseñar el álgebra que queríamos que Wonderfunc tuviese así como la API. Uno de los mayores objetivos, consistía en crear una API maximizando la legibilidad del código y la flexibilidad del mismo mediante ese álgebra.

- A partir de la API, se comenzó con las iteraciones de reuniones e implementaciones de requisitos. En primer lugar se comenzó con la clase Stream. Esto es así porque es dicha clase la que permite añadir los diferentes nodos a lo que será el *pipeline* al llamar al método collectoTo. De forma paralela, se implementaron las interfaces relacionadas con los nodos. Como es lógico pensar por lo anteriormente comentado, lo siguiente fue implementar la clase Pipeline.

- Con el objetivo de poder ejecutar un Pipeline para hacer diferentes pruebas y aportar una serie de nodos por defecto, se implementaron todos los tipos de nodos locales. Para más detalle sobre esto, se puede ir al manual de referencia al apartado de Nodos.

- El concepto de NodeContainer apareció cuando surgió la idea de ejecutar el *pipeline* en diferentes plataformas de computación en la nube. Más información sobre la interfaz NodeContainer se puede encontrar en el apartado de Manual de referencia.

- La interfaz FunctionRepository se decidió implementar cuando apareció la idea de que se pudiera usar funciones no sólo creadas directamente en Java, si no también en otros repositorios de funciones. El ejemplo que se ha decido usar ha sido Algorithmia.

Herramientas

GIT

Git es un sistema de control de versiones distribuido que puede ser utilizado para gestionar las versiones de proyectos de diferentes tamaños. En Wonderfunc, se ha utilizado Git para mantener un histórico de los cambios realizados en el código a lo largo de toda la implementación e iteraciones. Esto nos ha permito hacer numerosos cambios, permitiéndonos probar diferentes arquitecturas y deshacer los cambios en aquellos momentos en los que la solución no era buena. Conjuntamente con Git, se ha utilizado el método Gitflow explicado anteriormente en la sección de métodos.

Maven

En lo relacionado con la gestión de la configuración, se ha utilizado Maven.

Apache Maven es un software de gestión de proyectos. En esta ocasión, hemos utilizado Apache Maven para incluir las dependencias de Wonderfunc haciendo uso de un fichero POM.xml. Esto ha permitido controlar no sólo las dependencias, si no configurar las diferentes versiones necesarias de dichas dependencias desde un lugar centralizado.

Adicionalmente, al ser Wonderfunc un proyecto de código abierto, se ha decidido subir al repositorio central del que dispone Wonderfunc. De esta manera, al igual que se ha hecho uso de dependencias como puede ser JUnit, se podrá usar Wonderfunc en otros proyectos Java.

Java

Java es un lenguaje de programación de propósito general, concurrente, orientado a objetos. La decisión de usar Java como lenguaje de programación para Wonderfunc, está basada en la gran experiencia de los tutores y del alumno en dicho lenguaje. Además, al querer que se trate de un proyecto de código abierto, es beneficioso la gran comunidad que se encuentra detrás de este lenguaje. Además, Java es uno de los lenguajes más utilizados en el mundo de la ingeniería del software, lo que permite que Wonderfunc llegue a más desarrolladores y que pueda ser usado en más proyectos.

Tests

Como ya se ha comentado, para implementar Wonderfunc, se ha utilizado el método TDD, de tal forma que los tests han guiado dicha implementación. Los tests que se han creado cubren las siguientes especificaciones:

- Se devuelve una lista vacía cuando la lista de entrada (*stream*) está vacía y no hay nodos en el *pipeline.*

- Se devuelve una lista vacía cuando la lista de entrada (*stream)* está vacía y hay un nodo de tipo *filter* en el *pipeline*.

- Se devuelve una lista vacía cuando la lista de entrada (*stream)* está vacía y hay un nodo de tipo *map* en el *pipeline*.

- Se devuelve una lista vacía cuando la lista de entrada (*stream)* está vacía y hay un nodo de tipo *filter* y un nodo de tipo *map* en el *pipeline*.

- Se devuelve una lista vacía cuando la lista de entrada (*stream*) está vacía y no hay nodos en el *pipeline* especificando el contenedor de nodos a usar*.*

- Se devuelve una lista vacía cuando la lista de entrada (*stream)* está vacía y hay un nodo de tipo *filter* en el *pipeline* especificando el contenedor de nodos a usar.

- Se devuelve una lista vacía cuando la lista de entrada (*stream)* está vacía y hay un nodo de tipo *map* en el *pipeline* especificando el contenedor de nodos a usar.

  Se devuelve una lista vacía cuando la lista de entrada (*stream)* está vacía y hay un nodo de tipo *filter* y un nodo de tipo *map* en el *pipeline* especificando el contenedor de nodos a usar.

- Se devuelve una lista con los mismos datos de entrada cuando no se añaden nodos en el *pipeline.*

- Se devuelve una lista filtrada cuando se añade un nodo de tipo *filter* en el *pipeline.*

- Se devuelve una lista mapeada cuando se añade un nodo de tipo *map* en el *pipeline.*

- Se devuelve una lista de Doubles al mapear usando un FunctionRepository.

Guía de referencia

Como se ha explicado, Wonderfunc es un framework que permite la creación de *pipelines* funcionales de alto nivel mediante la encadenación de diferentes tipos de nodos en los que se ejecutan funciones definidas por expresiones lambda.

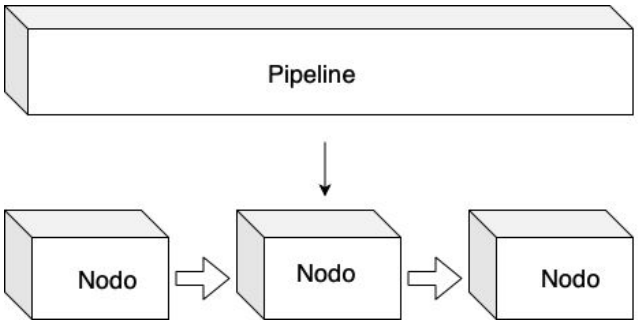

Figura 15. Disección del Pipeline

Nodos

En Wonderfunc se distinguen principalmente dos tipos de nodos, nodos de tipo *map* y nodos de tipo *filter.* Independientemente del tipo de un nodo, todos ellos implementan tres interfaces:

- Node
- Target
- Relay

Esta diferencia de interfaces, ha sido creada siguiendo el principio de Segregación de Interfaces de SOLID. Con esto conseguimos que un nodo se comporte en ciertas ocasiones como un *Node*, en otras ocasiones como *Target* y en otras como *Relay*.

La responsabilidad de un *Node*, no es otra que guardar la referencia del siguiente nodo del Pipeline. Por esto, en la interfaz *Node*, se puede encontrar el método *next*. Este método espera recibir un *Target.*

Por otro lado, la responsabilidad de un *Target* consiste en recibir un mensaje cuando el nodo anterior a este, decida mandarlo, de tal forma que en dicha interfaz encontraremos el método *push*, que recibe el mensaje mandado por el anterior nodo.

Por último, la responsabilidad de un *Relay*, consiste en mandar un mensaje al siguiente nodo del Pipeline. Para conseguir esto, en la interfaz *Relay*, se encuentre el método *relay*.

A modo de resumen, a la hora de ir incorporando los diferentes nodos al *pipeline*, serán tratados como *Node*. Al mandar un mensaje al siguiente nodo, se comportan como *Relay* y por último, al recibir un mensaje, se comportan como *Target*.

En el caso de los nodos de tipo *map*, cabe mencionar que tienen una diferencia importante con los nodos de tipo *filter*. Los nodos de tipo *map* pueden ser ejecutados de forma síncrona o de forma asíncrona. Para conseguir esto, dentro de cada nodo de tipo *map* se puede encontrar un objeto de la clase ExpressionExecutor. Este ExpressionExecutor será el encargado de ejecutar la función que se aloje en el nodo, de tal forma que dependiendo del ExpressionExecutor que el nodo tenga, será ejecutada la función síncrona o asíncronamente. Este concepto de ExpressionExecutor es explicado con más detalle más adelante. Gráficamente, un nodo de tipo *map* tendría la siguiente anatomía:

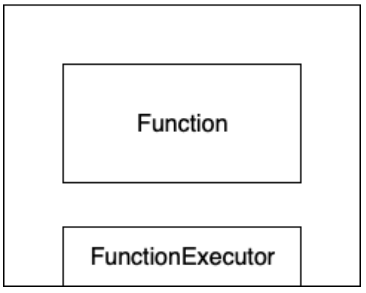

Figura 16. Anatomía del MapNode

Existen dos tipos de nodos adicionales. Estos nodos son el nodo fuente y el nodo recolector. Por la funcionalidad requerida, estos nodos no implementan las tres interfaces comentadas anteriormente.

En el caso del nodo fuente, tan solo implementa la interfaz de *Node* y la interfaz *Relay*. Esto es así porque este nodo no va a recibir mensajes de un nodo anterior en el *pipeline*, por lo que no es necesario que implemente la interfaz *Target*.

Por otro lado, el nodo recolector, tampoco implementa todas las interfaces. Como cabe esperar, el nodo recolector tan solo implementa la interfaz *Target*, ya que la única responsabilidad que tendrá es recibir todos los mensajes provenientes del *pipeline*.

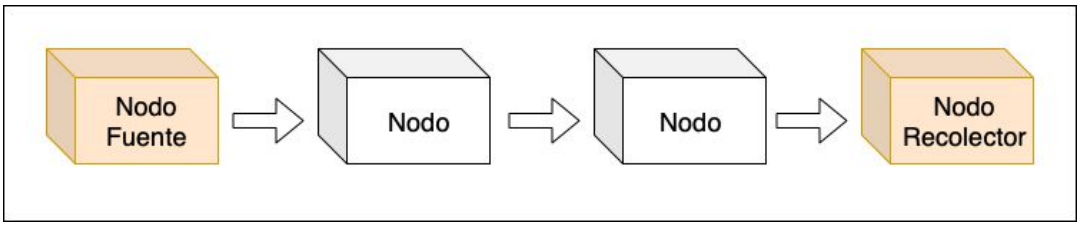

Figura 17. Anatomía del Pipeline

Se puede observar la gran ventaja de hacer esa segregación de interfaces descrita en SOLID. Al haber separado las responsabilidades en diferentes interfaces, no es necesario forzar a una clase a implementar métodos no necesarios.

A continuación se muestra el UML que recoge la arquitectura que se ha implementado para los nodos.

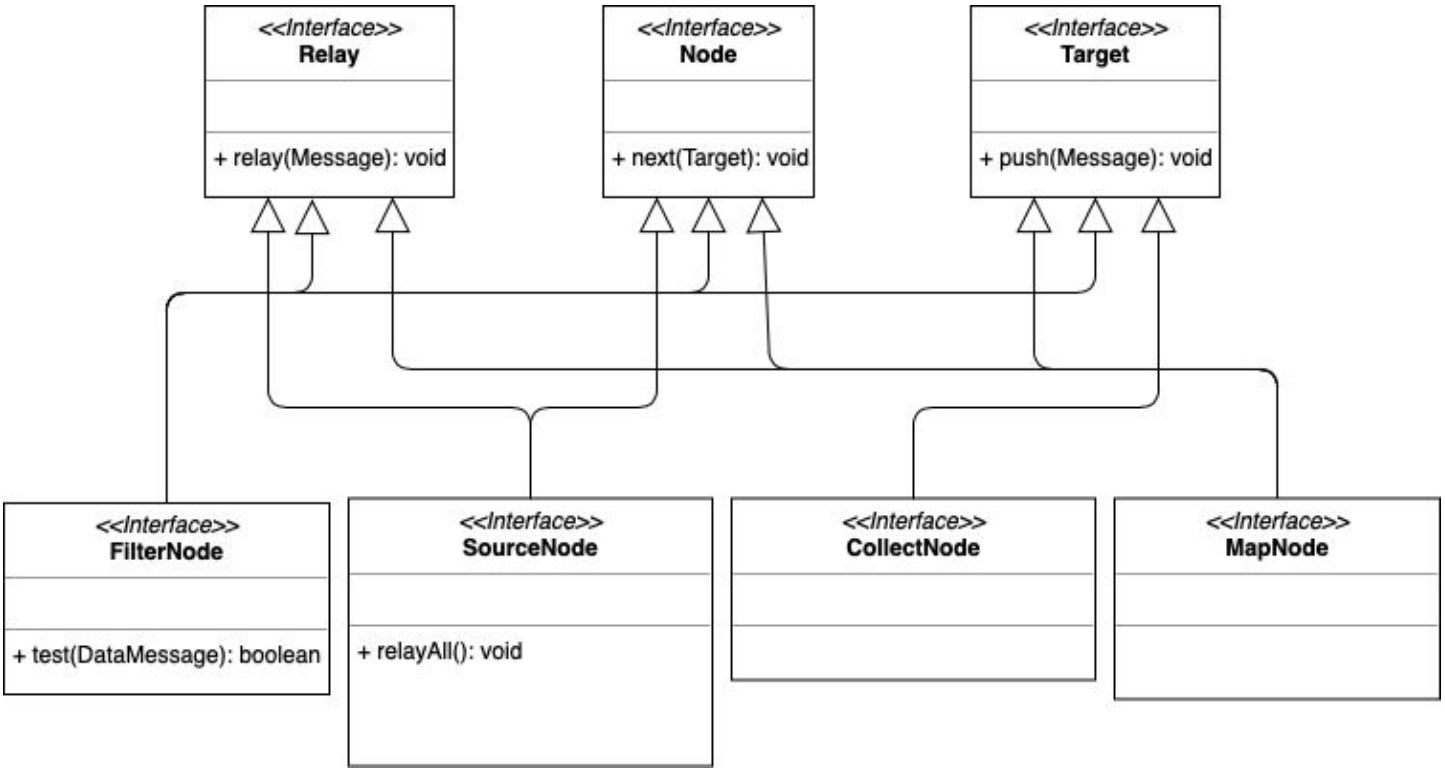

Figura 18. UML de los nodos

Como se puede observar, se trata de interfaces. Esto es así para permitir la implementación de nuevos tipos de nodos dependiendo de las necesidades.

A toda la explicación de los diferentes tipos de nodo, es importante añadir una explicación adicional. Estos nodos, tal y como se ha comentado anteriormente, pueden ser ejecutados en diferentes plataformas de computación funcional en la nube como pueden ser AWS, Azure o Google Cloud Platform. A parte de poder ser ejecutados en dichas plataformas, también pueden ser ejecutados en local. De hecho, esta opción es la que viene por defecto y es la que se aplica también por defecto, de tal manera que se han implementado todos los tipos de nodos en su versión local. Sin embargo, una rama de continuación para Wonderfunc podría ser la implementación de los nodos para las ejecuciones en AWS, Azure o Hadoop junto con los respectivos contenedores de funciones que serán explicados más adelante.

ExpressionExecutor

En Wonderfunc, el ExpressionExecutor que se encuentra dentro de los nodos de tipo *map* es el que dicta si la función que en ellos se encuentra, se va a ejecutar de forma síncrona o asíncronamente. Por defecto, los nodos de tipo *map* tienen un LocalExpressionExecutor. Es decir, se van a ejecutar de forma síncrona. Para cambiar de un tipo de ExpressionExecutor

a otro, será necesario hacer uso del método *with* de la clase Stream pasando por parámetros el ExpressionExecutor. A continuación se muestra un ejemplo:

```
Pipeline<Integer> pipeline = stream
            .on(new LocalNodeContainer())
                .map(s -> sentimentAnalysis.apply(s))
                    .with(new RemoteExpressionExecutor(new CommentMarshalling()))
                .map(d -> d <= 0)
            .collectTo(output);
```

Figura 19. Uso del ExpressionExecutor

En caso de que un nodo de tipo *map* tenga un LocalExpressionExecutor simplemente hará un *mapping* de la entrada, haciendo uso de la función lambda que posea, a la salida tan pronto como reciba la entrada.

Por otro lado, en caso de querer ejecutar una función que se encuentre en un repositorio de funciones, tiene sentido querer ejecutarla de forma asíncrona por varias razones. Una razón puede ser que dicha ejecución lleve mucho tiempo y no se quiera esperar. Otra razón puede ser querer reducir los costes de llamada a esas funciones en los repositorios de funciones. Para conseguir esto, se ha implementado un RemoteExpressionExecutor que es capaz de mandar a ejecutar trabajos en lote y que se ejecute de forma asíncrona.

La forma en la que estos RemoteExpressionExecutor funcionan es que irá almacenando los mensajes entrantes para posteriormente ejecutar la función que se encuentre en el nodo en modo lotes. Evidentemente, todo esto como su propio nombre indica, será ejecutado de forma asíncrona. Es decir, al entrar un mensaje a un nodo de este tipo, no se producirá directamente un mensaje de salida. Al contrario, se almacenará hasta que se llene un caché de la que disponen. Al llenarse dicha caché, se ejecutará la función mandando como entrada el lote de mensajes almacenados. Mientras que lleva a cabo esa ejecución en el repositorio de funciones, se seguirá recibiendo mensajes provenientes del *pipeline* al nodo. Al recibir la respuesta de la función que se encuentre en el repositorio de funciones se volverá a transformar a un formato de *stream* de mensajes individuales.

Como se puede observar, por constructor a los RemoteExpressionExecutor se les tiene que pasar un objeto que tiene que implementar la interfaz Marshalling.

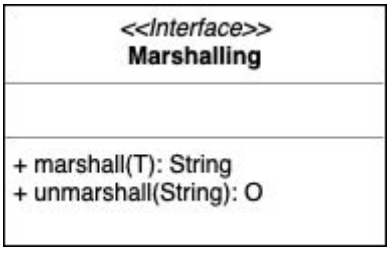

Figura 20. UML del Marshalling

La transmisión de datos entre nodos se hace mediante una interfaz llamada Message que será explicada más adelante. Sin embargo, las funciones que se encuentran en los repositorios de funciones, normalmente reciben los datos en forma de JSON. Pues bien, en Wonderfunc, son los objetos que implementan esta interfaz Marshalling los que tienen esa responsabilidad. Es por eso que el RemoteExpressionExecutor que se la pasa al método *with*, tiene que tener un objeto que implemente la interfaz Marshalling, de tal forma que cada vez que se recibe un mensaje, se hace uso de del método *marshall* del Marshalling para transformar de un Message al JSON con el formato requerido por la función en el repositorio de funciones. Al llenarse la caché, se juntaran esos mensajes transformados y se mandarán a la función en el repositorio de funciones. Por último, cuando se reciba la respuesta de la función, se volverán a transformar a objetos de la clase Message haciendo uso del método *unmarshall* del Marshalling.

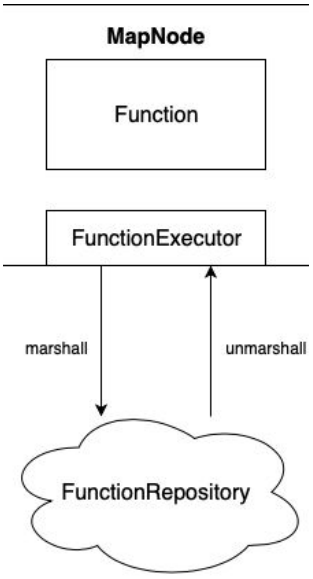

Figura 21. Comunicación con el FunctionRepository

Mensajes

Tal y como se ha explicado, a través del *pipeline* fluye un flujo de datos que inicialmente es inyectado mediante una lista al mismo. Posteriormente, el nodo fuente al ejecutar el *pipeline*, comenzará a mandar cada uno de los datos que se encuentran en esa lista, comenzando así el flujo de datos. Este nodo fuente, no manda los datos en forma bruta por el *pipeline*. Dichos datos son encapsulados en objetos de una clase llamada *DataMessage*, de tal manera que toda la comunicación y traspaso de mensajes, se hace a través de la interfaz *Message*, que es implementada por *DataMessage*.

De alguna manera, el *pipeline* tiene que saber cuándo ha terminado de mandar mensajes. En otras palabras, cuándo se ha mandado el último mensaje. El objetivo de esto es poder

tener la certeza de que se puede cerrar las conexiones con esas plataformas de computación funcional en la nube. Para implementar esto, se decidió crear una nueva clase que implemente la interfaz *Message* llamada *EndOfStreamMessage*. De esta manera, los nodos al recibir un mensaje de tipo *EndOfStreamMessage*, simplemente lo dejan pasar. Al llegar ese mensaje de final del *pipeline*, se cerrará el mismo.

En la implementación en local de la que ya se ha hablado, esto no se hace así. Esto es porque no se necesita abrir una conexión entre el último nodo antes del nodo recolector y el nodo recolector. El comportamiento implementado consiste en que si a un nodo del *pipeline* le llega un mensaje de tipo *EndOfStreamMessage*, lo dejará pasar. Al llegar los diferentes mensajes al nodo recolector, únicamente si son de tipo *DataMessage*, serán añadidos a la lista de salida pasada al Stream mediante el método *collectTo*.

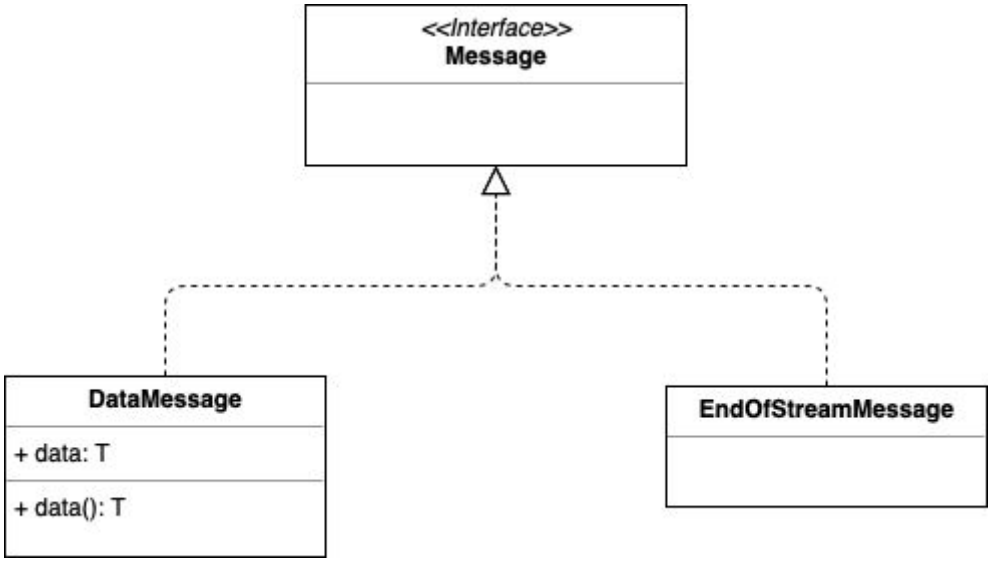

Figura 22. UML de Message

Contenedores de funciones

Como se ha comentado anteriormente, Wonderfunc permite la creación de un *pipeline* de alto nivel añadiendo nodos. Para añadir estos nodos, se usan los métodos *map* y *filter*. Sin embargo, no se ha comentado el lugar donde se van a ejecutar dichos nodos. Este entorno de ejecución depende del contenedor que esté aplicado en cada momento. Para cambiar el contenedor y por lo tanto el entorno de ejecución, bastará con hacer uso del método *on* pasando un objeto que implemente la interfaz NodeContainer por parámetros. Esto permite una ejecución distribuida en diferentes plataformas, pudiéndose ejecutar ciertas funciones en AWS, otras en Azure o incluso en local. Con Wonderfunc se ha decidido implementar un NodeContainer local que ha recibido el nombre de LocalNodeContainer. Esto quiere decir que por defecto, los nodos serán ejecutados en local. Sin embargo, se ha desacoplado el código mediante el uso de interfaces, de tal manera que se permite la fácil implementación y uso de nuevos NodeContainers.

Una vez esto ha quedado claro, cabe destacar que en realidad, cuando se hace uso de los métodos *map* o *filter* para añadir un nuevo nodo al *pipeline*, es responsabilidad del NodeContainer aplicado en ese momento, crear el nodo que se va a añadir. Es por esto que la interfaz NodeContainer tiene los siguientes métodos:

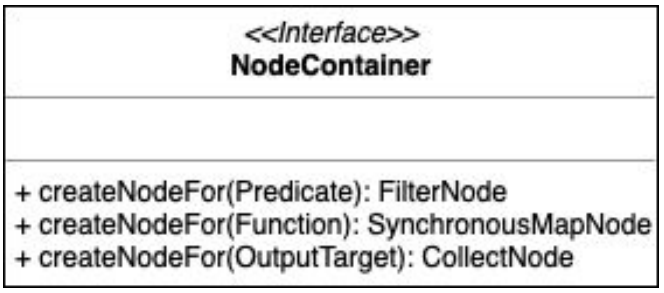

Figura 23. UML de NodeContainer

Tal y como se puede observar, en la interfaz *NodeContainer* existen métodos para la creación de todos los tipos de nodos que existen a excepción del nodo fuente. Esto es porque el nodo fuente es añadido por defecto al instanciar un objeto de tipo *Stream*.

Se ha seguido un diseño por contrato, de tal forma que en un futuro, cuando se quiera añadir un nuevo tipo de *NodeContainer*, se tendrá que implementar dicha interfaz y por lo tanto implementar los métodos que en ella se encuentran.

Repositorios de funciones

Hasta ahora, se ha indicado que a los nodos de tipo *map*, se les permite pasar una función que cumpla con la interfaz *Function* definidas a la hora de crear el Pipeline. Sin embargo, en Wonderfunc se ha pensado en la posibilidad de usar funciones lambda que están ya implementadas en otros repositorios. Estos repositorios han recibido el nombre de de *FunctionRepository*. Un ejemplo de repositorio puede ser Algorithmia. Algorithmia es una plataforma que cuenta con una gran cantidad y variedad de funciones que implementan algoritmos complejos de Procesamiento de Lenguaje Natural entre otros o incluso funciones que usan *Machine Learning*. Gracias a los objetos de la clase *FunctionRepository*, la utilidad de Wonderfunc se extiende mucho más permitiendo realizar operaciones de mapeos más complejas usando estos algoritmos.

En la implementación base de Wonderfunc, se ha añadido un *FunctionRepository* que permite hacer uso de las funciones que se encuentran el Algorithmia. Para añadir al Pipeline un nodo de tipo *map* que contenga una función que provenga de un *FunctionRepository*, bastará con implementar un *FunctionRepository* y llamar al método *function*. Este método devolverá una función que podrá ser utilizada en la expresión lambda al añadir un nodo de tipo *map* haciendo uso del método *map* de la clase Stream.

Como se explicó anteriormente, las funciones y filtros son ejecutados en el entorno de ejecución que se haya establecido en un momento dado mediante el uso del método *on.* Esto permite que una función proveniente de un repositorio, pueda ser ejecutada por ejemplo en local o en AWS.

Conclusiones

Logros personales

Llevar a cabo Wonderfunc, ha supuesto una serie de resultados que desde un punto de vista personal, me han enriquecido. Por un lado, haber realizado un proyecto por completo, me ha permitido ver en primera persona todas las fases del desarrollo de software, desde la identificación del problema, pasando por la idea, hasta llegar a la implementación de la solución. Por otro lado, Wonderfunc no hubiese sido posible llevarlo a cabo si no hubiese sido al trabajo en equipo entre los tutores y yo. Por esto, resalto el gran aprendizaje que me ha supuesto trabajar en equipo a parte de los conocimientos que he recibido por parte de ellos. Adicionalmente, he tenido la oportunidad de utilizar las herramientas y métodos nombrados en esta memoria fuera del entorno de prácticas de asignaturas, pudiendo experimentar problemas que ocurren a la hora de llevar a cabo un proyecto real. Otro punto a destacar, ha sido la experiencia de iniciar un proyecto de código abierto. Además, he podido indagar mucho más y entender mejor conceptos relacionados con la ingeniería del software como pueden ser las arquitecturas distribuidas o las arquitecturas funcionales. Por último, un gran resultado ha sido el desarrollo de mis habilidades de comunicación y pensamiento lateral. Esto es así porque a lo largo de la implementación de Wonderfunc, en ningún momento nos quedamos con la primera solución que aparecía. Al contrario, siempre se pensó en diferentes soluciones intentando encontrar la mejor de ellas.

Contribuciones

Desde el punto de vista de la contribución realizada, cabe resaltar las ventajas que tiene Wonderfunc frente a lo que se ha considerado la competencia del mismo. Los principales competidores de Wonderfunc tal y como se ha explicado anteriormente son Apache Pig y Cascading. La mayor contribución de Wonderfunc, es que se trata de una mezcla de ambas tecnologías en una sola mejorando drásticamente el nivel de expresividad del código y la flexibilidad, de tal forma que con unas pocas líneas, se da la posibilidad de generar una arquitectura funcional distribuible. Además, se trata de un proyecto que se está utilizando en algunos proyectos internos en la empresa The Agile Monkeys donde me encuentro trabajando en el momento en el que escribo este informe para el procesamiento de datos. Esto ha sido así gracias a que con Wonderfunc se ha resuelto el problema existente de la interconección entre las diferentes plataformas de computación en la nube. Por último, es importante comentar que en el curso académico 2019-2020, me dispongo a cursar un máster en ciencia de datos, de tal forma que tengo intenciones de seguir usando Wonderfunc para las numerosas tareas de procesamiento de datos que se tendrán que llevar a cabo explotando las ventajas de Wonderfunc así como siguiendo dándolo a conocer entre mis compañeros, creando una comunidad detrás de este proyecto que crezca con el tiempo.

Posibles continuaciones de Wonderfunc

Al tratarse Wonderfunc de un proyecto de código abierto, puede evolucionar a partir del punto en el que está.

Implementación de nuevos contenedores de funciones

En la implementación actual de Wonderfunc, se ha implementado el contenedor de funciones lambda en local. Sin embargo, se podría implementar numerosos contenedores de funciones nuevos. Algunos ejemplos son: AWS, Azure, Hadoop, etc.

La ventaja de la forma en la que se ha implementado Wonderfunc, es que está abierto a extensiones, de tal forma que cualquier contenedor que quiera ser implementado, podrá ser implementado sin cambiar el código. Únicamente habrá que añadir clases nuevas. Esto no tiene que restringirse a los ejemplos enumerados en el párrafo anterior.

Despliegue

En caso de que se quiera hacer uso de contenedores de funciones en proveedores de servidores en la nube por ejemplo, será necesario hacer previo a la ejecución del Pipeline, un despliegue de las funciones a ejecutar en dichos proveedores de servicios en la nube. A modo de ejemplo, si se quisiera añadir un nodo al Pipeline que ejecute una función en AWS, habrá que implementar el despliegue de dicha función en AWS Lambda.

Implementación de nuevos repositorios de funciones

De forma similar a lo que ocurre con los contenedores de funciones, en la implementación actual de Wonderfunc, se ha implementado el repositorio de funciones de Algorithmia únicamente. Sin embargo, esto no tiene por qué acabar aquí. Se podría implementar nuevos repositorios de funciones sin ningún problema. Esto es así debido a que Wonderfunc está diseñado para ser altamente extensible.

Módulo para generar un repositorio

Otra funcionalidad que se ha pensado que podría llegar a ser interesante, consiste en la creación de un módulo de Wonderfunc que permita la creación de un repositorio de funciones en un servidor propio o incluso la posibilidad de añadir funciones a repositorios de funciones existentes como puede ser el de Algorithmia.

Fork

Existen ocasiones en las que para una misma entrada, resulta interesante ejecutar dos funciones. A modo de ejemplo, imaginemos que la entrada es un documento PDF y que por un lado, estamos interesados en clasificar si en dicho documento aparece alguna imagen y

no. Adicionalmente, queremos saber el tema principal del texto de dicho PDF. Para solventar esta situación, se ha pensado en que en un futuro se podría implementar la funcionalidad que ha recibido el nombre de *fork*. Con esto, se permitiría que a la hora de definir un *pipeline*, se pueda tener la funcionalidad explicada anteriormente.

Split automático

Otra funcionalidad que se ha pensado en implementar en un futuro es que cuando se esté ejecutando diferentes funciones en proveedores que no escalen de forma automática (como por ejemplo el caso de ejecutar las funciones en local), si se detecta un caso de cuello de botella en el que una función no es capaz de asimilar el flujo de datos de entrada, dar la posibilidad de que automáticamente se lancen nuevos nodos con la misma función con el objetivo de repartir dicho flujo de entrada. Con esto se conseguiría optimizar más el tiempo de ejecución del *pipeline*.

Bibliografía

[1] https://github.com/wonderfunc/wonderfunc

[2] Amazon Web Services (AWS). https://aws.amazon.com/es/

[3] Microsoft Azure. https://azure.microsoft.com/es-es/

[4] Google Cloud Platform. https://cloud.google.com/?hl=es

[5] Algorithmia. https://algorithmia.com

[6] Expresiones lambda. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

[7] Apache Pig. https://pig.apache.org

[8] Cascading. https://www.cascading.org

[9] Java. https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf

[10] TDD. Beck, K., Test Driven Development - by Example. Boston: Addison Wesley, 2003.

[11] Git. https://git-scm.com

[12] Gitflow. Vincent Driessen. A successful Git branching model. https://nvie.com/posts/a-successful-git-branching-model/

[13] Draw.io (Creación de imágenes y UMLs). https://www.draw.io

Apéndice. Justificación de las competencias específicas cubiertas

CII08: Para construir Wonderfunc se ha tenido que analizar a la competencia (Apache Pig y Cascading), diseñando y construyendo una solución que mejora el estado del arte así como resolviendo problemas existentes en la actualidad como puede ser la integración entre plataformas de computación en la nube. Así mismo, la aplicación es robusta puesto que ha sido cubierto con tests unitarios y de integración. La elección del lenguaje ha sido clave para mantener el desarrollo de la aplicación vivo en el futuro puesto que Java es uno de los lenguajes más usado en la actualidad.

CII011: Wonderfunc es un framework que permite construir pipelines funcionales y distribuidos. En este aspecto, se ha implementado una solución que permite la distribución del pipeline definido entre diferentes plataformas de computación en la nube.

CII014: Tal y como ya se ha comentado a lo largo de este documento, con Wonderfunc se ha creado una solución que permite la ejecución del pipeline definido de forma concurrente así como distribuida en diferentes máquinas.

CII016: Durante todo el desarrollo de Wonderfunc se ha seguido el método *Test Driven Development.*