Universidad Zaragoza

Trabajo Fin de Grado

Evaluación del rendimiento de aplicaciones intensivas en datos con Apache Tez

Performance evaluation of data-intensive applications with Apache Tez

Autor

Iñigo Gascón Royo

Directores

José Ignacio Requeno Jarabo

José Javier Merseguer Hernáiz

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2017

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Iñigo Gascón Royo,

con nº de DNI 78759000P en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Evaluación del rendimiento de aplicaciones intensivas en datos con Apache Tez,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 18 de Abril de 2017

Fdo: I. GASCÓN

Evaluación del rendimiento de aplicaciones intensivas en datos con Apache Tez

RESUMEN

Durante los últimos años, el *Big Data* ha sufrido un gran auge en un rango muy variado de sectores del mercado. Analizar todos los datos disponibles para conocer la demanda de productos por adelantado, o incluso para crear dichas demandas, se ha vuelto una práctica común para muchas empresas. El procesamiento de estas grandes cantidades de datos no es algo trivial y ha de realizarse con tecnologías específicamente diseñadas para ese propósito. Es por ello que cada vez surgen más propuestas en esa área de la informática, cada una con sus ventajas e inconvenientes. En todas ellas, no obstante, hay una característica de vital importancia: el rendimiento.

El trabajo que aquí se presenta se enmarca dentro del proyecto DICE, situado en el marco del plan Horizon2020 de la Comisión Europea, y en el cual participa el Grupo de I+D en Computación Distribuida (DiSCo) de la Universidad de Zaragoza. El objetivo final de DICE consiste en ofrecer un perfil UML original, un conjunto de herramientas software y una metodología que ayudarán a los diseñadores de aplicaciones intensivas en datos a plantear la seguridad, eficiencia y fiabilidad de dichas aplicaciones. Entre dichas herramientas software se encuentra un simulador capaz de evaluar el rendimiento de aplicaciones basadas en diferentes tecnologías (DICE-Simulator), y es a este simulador al cual se le va a aportar una nueva tecnología mediante la realización de este trabajo.

Para poder realizar dicha aportación, integrando la tecnología de Apache Tez en el simulador, se han realizado una serie procesos para analizar y modelar la tecnología. En primer lugar, se ha investigado a fondo el funcionamiento de Tez, comprobando su rendimiento en aplicaciones reales y extrayendo tantos los conceptos básicos de la tecnología como aquellos relativos al rendimiento de la misma, vitales para poder simular de forma fiel su comportamiento.

A continuación, habiendo extraído toda la información necesaria resultante de la investigación, se ha creado un perfil para UML que captura de forma precisa todos los conceptos relativos al funcionamiento de Tez y a su rendimiento en aplicaciones reales.

Una vez completado el modelo UML, ha sido necesario transformar dicho modelo a otro distinto que permitiese analizar y evaluar el rendimiento, basado en los conceptos recogido por el primero. Para ello, se ha propuesto un conjunto de patrones de transformación del modelo UML a un modelo de redes de Petri, concretamente a redes de Petri estocásticas, las cuales permiten analizar el rendimiento en aplicaciones informáticas.

Por último, se han realizado una serie de experimentos para verificar que las transformaciones propuestas son correctas y reflejan de forma precisa el funcionamiento de una aplicación Tez real en términos de rendimiento.

Índice

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción y objetivos

El *Big Data* no es un concepto nuevo en el mundo de la informática. Sin embargo, durante los últimos años, ha tomado cada vez más relevancia en numerosas empresas de sectores distintos y muy variados. Estas empresas han comenzado a emplear la información resultante del procesamiento de estas grandes cantidades de datos para comprender las necesidades de sus clientes, conviertíendolas así en una oportunidad de negocio.

A pesar de que ya existían aplicaciones para analizar volúmenes grandes de datos, éstas se centraban principalmente en datos muy concretos y poco variables. Ahora, con nuevas tecnologías emergiendo en el mercado, encontramos aplicaciones capaces de procesar mayores volúmenes de datos, con una mayor variedad y, además, a mayor velocidad, pudiendo incluso procesar información en tiempo real. El rendimiento de estas aplicaciones es, entre otras características, una de las propiedades diferenciadoras más importantes y más críticas.

1.1. Contexto

Con la intención de mejorar la calidad en el desarrollo de aplicaciones intensivas en datos, en Febrero de 2015 nació DICE [1], un proyecto de investigación y desarrollo que se sitúa en el marco del plan Horizon2020 de la Comisión Europea. En dicho proyecto participan el Grupo de I+D en Computación Distribuida (DiSCo) de la Universidad de Zaragoza, el Imperial College London, el Politecnico di Milano, el IeAT y un conjunto de empresas a nivel internacional. Es en este marco en el que se sitúa el trabajo que aquí se presenta, resultado de la colaboración con el Grupo DiSCo.

El objetivo final del proyecto consiste en ofrecer un perfil UML original [12], un conjunto de herramientas software y una metodología que ayudarán a los diseñadores de aplicaciones intensivas en datos a plantear la seguridad, eficiencia y fiabilidad de dichas aplicaciones.

Todos estos objetivos (los perfiles, las herramientas, etc.) y las contribuciones que está realizando el proyecto DICE son las que pueden observarse en la Figura 1.1.

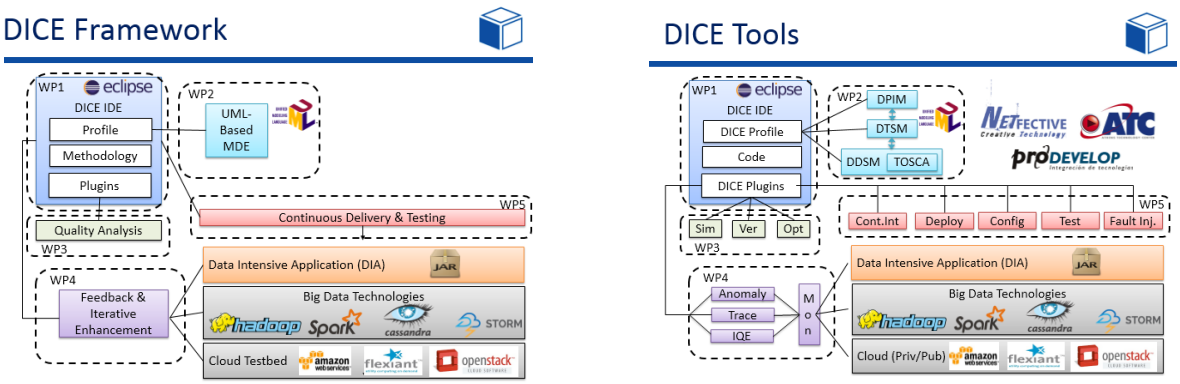


Figura 1.1: Objetivos y contribuciones del proyecto DICE.

1.2. Aportaciones

Una de las herramientas software que ofrece el proyecto DICE consiste en un simulador [11] capaz de evaluar el rendimiento de aplicaciones basadas en diferentes tecnologías (ver Figura 1.1, paquete WP3), de forma que pueda asesorar al desarrollador de cara a mejorar su diseño e implementación. Dentro de dicha evaluación, el simulador también permite extraer otras métricas de interés como pueden ser la utilización y el tiempo de respuesta de una aplicación.

La evaluación del rendimiento se realiza mediante la transformación de modelos semi formales (UML) a modelos que permitan la evaluación de prestaciones (redes de Petri). Para ello (la transformación), es necesario capturar los parámetros de configuración más importantes de la tecnología y que impactarán en el rendimiento de la futura aplicación. Esos parámetros servirán para afinar la transformación del modelo en una red de Petri correspondiente con la que obtener mejores predicciones.

Por tanto, en este trabajo se creará un lenguaje de dominio específico (DSML, Domain Specific Modeling Language) sobre UML aplicando la técnica de perfiles. Este lenguaje recoge los parámetros de prestaciones más relevantes de Apache Tez, la nueva tecnología que se va a incluir en el proyecto DICE. A partir de este lenguaje, se ha propuesto una transformación de diseños de Apache Tez a redes de Petri, las cuales, una vez validadas, se incluirán en el software simulador mencionado.

En la Figura 1.2 se puede apreciar de forma gráfica la contribución de este trabajo al proyecto DICE. Toda la investigación y el desarrollo realizados se incluirán en el paquete DTSM(*DICE Platform and Technology Specific Model*) de la herramienta de simulación, lo cual incluye el perfil creado para la tecnología Tez, la transformación automática a redes de Petri y el cálculo de las métricas de rendimiento.

Como ha podido verse en la Figura 1.1, el perfil que ofrece DICE como objetivo final del proyecto se divide en tres sub-paquetes distintos. El que se ha creado para Tez se incluye

en DTSM, debido a que es el paquete que recoge los perfiles dependientes de la tecnología. Como puede observarse, ya existen perfiles para otras tecnologías distintas. Por otro lado, el DPIM(*DICE Platform Independent Model*) incluye todo el trabajo que supone una abstracción de la tecnología, al cual no se ha contribuido en este proyecto. Finalmente, la herramienta de Fiabilidad, que también incluye perfiles de varias tecnologías, tampoco se ha tocado durante el trabajo.

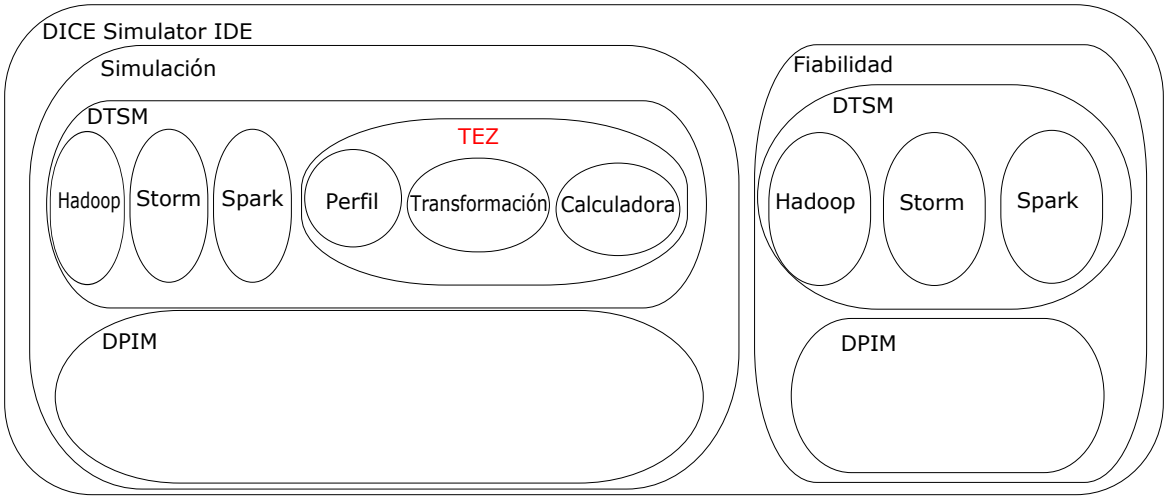


Figura 1.2: Esquema de las contribuciones del proyecto DICE.

Por último, cabe destacar que la investigación y el desarrollo realizados son originales y no existían antes de realizarlos en este proyecto. El perfil creado para Apache Tez es totalmente nuevo y no se había realizado antes para esta tecnología.

1.3. Herramientas

A lo largo del desarrollo del trabajo se han empleado un conjunto de herramientas para las distintas fases del mismo. A continuación se enumeran las más relevantes con una breve explicación de las mismas:

- Hadoop: framework sobre el que se ha montado Apache Tez.
- IntelliJ: desarrollo de aplicaciones basadas en Apache Tez.
- Papyrus: entorno para realizar modelos UML en Eclipse.
- Eclipse: base para la instalación y uso del plugin Papyrus, y para la ejecución del DICE Simulator.
- GreatSPN: modelado y simulación de redes de Petri.
- Git: control de versiones.

1.4. Metodología

A lo largo del trabajo se ha seguido una metodología bien definida en cinco fases, que se han ido realizando tanto de forma secuencial como concurrente, en función de las tareas de cada una de ellas.

En primer lugar se ha llevado a cabo una fase exhaustiva de investigación de la tecnología Apache Tez, de forma que se ha aprendido acerca de su funcionamiento, sus distintas configuraciones y sus características principales, muy importantes de cara al modelado de su perfil en la siguiente fase. Durante el desarrollo de esta fase también se han implementado algunas aplicaciones de ejemplo con la tecnología, de forma que se alcanzase una comprensión no sólo teórica sino también práctica de su funcionamiento. No obstante, la implementación de las aplicaciones finales se han realizado en la siguiente fase.

En segundo lugar se han implementado dos ejemplos distintos de aplicaciones basadas en Apache Tez, una más simple que mostraba el funcionamiento principal de una aplicación desarrollada con esta tecnología, y una más compleja que incluyese todas las configuraciones posibles que se ofrecen. Tras su implementación, se han realizado numerosas ejecuciones con distintos datos de entrada para obtener datos empíricos del rendimiento de dichas aplicaciones, tanto en ejecuciones en un sistema local como en un clúster con mayores recursos disponibles.

En tercer lugar se ha llevado a cabo el modelado del nuevo lenguaje de dominio específico mediante la técnica de perfiles. A lo largo de esta fase, por tanto, se ha desarrollado un perfil que representase las características principales de Apache Tez y que permitirán más adelante una transformación a un modelo que pueda evaluarse.

En cuarto lugar, se han transformado los diseños propuestos a redes de Petri, de forma que se pudiesen realizar simulaciones cuyos resultados deberían de coincidir con los obtenidos en la fase anterior. A lo largo de esta fase se han propuesto y modificado numerosas transformaciones de estas redes hasta que se ha conseguido representar y reproducir de la forma más fiel posible el comportamiento y la estructura de Tez.

Finalmente, para verificar que las transformaciones propuestas son correctas y efectivamente reproducen de forma fiel el comportamiento de Tez, se han realizado una serie de experimentos ejecutando las redes creadas a partir de fichas transformaciones. En función de los resultados obtenidos en cada experimento, se han modificado (o no) las transformaciones propuestas.

1.5. Organización de esta memoria

Esta memoria contiene cinco capítulos estructurados de la siguiente forma: el Capítulo 2 introduce la tecnología de Apache Tez, en la cual se basa este trabajo, y los conceptos básí-

cos de ésta relativos al rendimiento; el Capítulo 3 presenta los modelos UML creados para la tecnología de Tez, en los cuales se verán representados los conceptos introducidos en el capítulo anterior; el Capítulo 4 expone los patrones de transformación propuestos de los modelos UML a las redes de Petri, empleadas para evaluar el rendimiento; el Capítulo 5 presenta la validación de los patrones de transformación anteriormente presentados y, finalmente, en el Capítulo 6 se encuentran las conclusiones del proyecto.

Se aprovechan los anexos para explicar más a fondo algunos de los detalles sobre la realización de este trabajo. En el Anexo A, se puede ver cómo se ha invertido el tiempo durante el proyecto, incluyendo un diagrama temporal y un desglose por horas. En el Anexo B, se incluye información adicional sobre el funcionamiento y uso de las GSPNs.

Capítulo 2

Apache Tez

En este capítulo se introduce la tecnología Apache Tez, sobre la cual se ha realizado la evaluación del rendimiento en este trabajo. Además, una vez presentados los conceptos básicos de esta tecnología, se profundizará en las características principales de Tez que se han empleado durante dicha evaluación y que más adelante se modelarán mediante la técnica de perfiles. La identificación de las principales características y parámetros de configuración de Tez permitirá crear un perfil tecnológico adecuado, el cual se presentará en el capítulo siguiente.

2.1. ¿Qué es Apache Tez?

Apache Tez es un *framework* de ejecución distribuido enfocado a aplicaciones intensivas en datos basadas en YARN y ejecutadas sobre Hadoop. Apareció por primera vez en 2013 y se presenta como una evolución de MapReduce y de otras tecnologías similares que han aparecido a lo largo de los últimos años.

Hadoop se ha empleado durante mucho tiempo como plataforma para el procesamiento de grandes cantidades de datos en lotes (*batch-processing*). En los últimos años han surgido nuevos casos de uso con necesidades distintas, en los cuales el procesamiento en tiempo real era necesario, como por ejemplo en aplicaciones de *machine learning*.

Tez aparece a raíz de estas necesidades, sustituyendo en muchos casos a MapReduce como base del procesamiento de datos en aplicaciones basadas en Apache Hive o Apache Pig. Tez proporciona una API [3] para desarrolladores que permite modelar de forma muy precisa el movimiento de datos en una aplicación, y además con muy poco código necesario para ello. Cabe destacar que Tez no está pensado para el uso directo de usuarios finales, sino para desarrolladores de productos y aplicaciones que sí estarán dirigidas a usuarios finales y que contarán con mejoras destacables en rendimiento y flexibilidad.

Empleando esta API, los desarrolladores pueden definir el movimiento de datos y, en general, el flujo de ejecución de una aplicación, mediante un DAG (Grafo Acíclico Dirigido). Esto permite que la construcción de cualquier aplicación sea muy gráfica y sencilla, pudiendo

visualizar fácilmente lo que se está construyendo. La ventaja de estos DAG frente a los que puede ofrecer MapReduce se basa en una característica particular de Tez: mientras que en MapReduce hay que definir todos los trabajos que lleva a cabo el DAG como un conjunto de Map y Reduce, Tez permite emplear el patrón MRR, el cual consiste en realizar un solo Map al comienzo de su ejecución (o varios, si se están ejecutando tareas en paralelo que siguen distintos hilos) y a continuación varias fases de Reduce. Esto permite que Tez pueda realizar el paso de datos entre un nodo de procesamiento y otro sin necesidad de escribir los datos en el sistema de ficheros de Hadoop, consiguiendo así una gran mejora en el rendimiento, como ya se ha mencionado. En la Figura 2.1 puede verse un ejemplo de la comparación entre MapReduce (izquierda) y Tez (derecha).

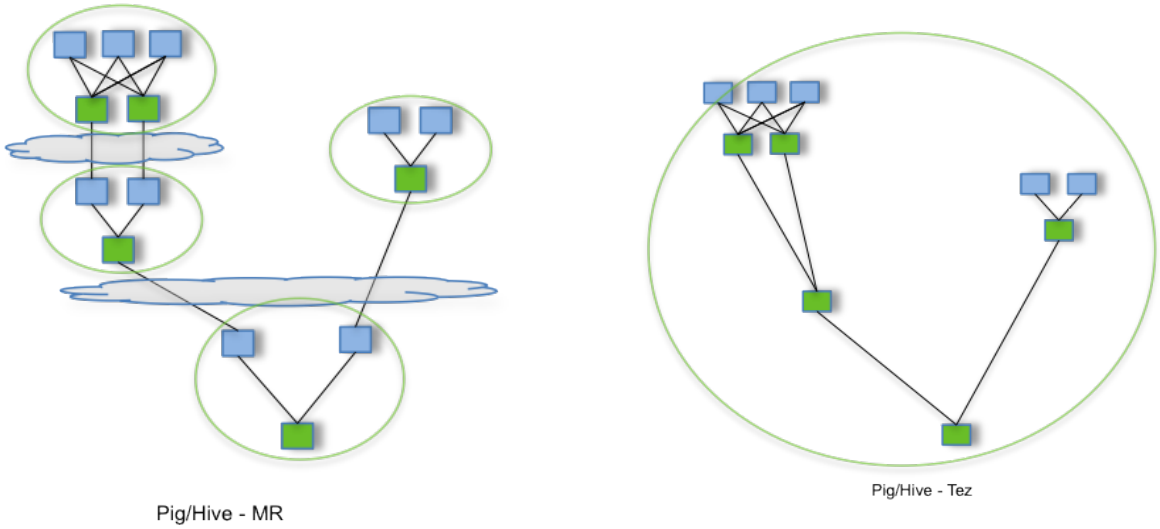


Figura 2.1: Comparación entre la ejecución de un DAG con Hive o Pig sobre MapReduce y Tez.

Acerca de esta API, y de los conceptos incluidos en ella, como los DAG que ya se han mencionado, o los *Vertices* y los *Edges* que conforman dichos DAGs, se va a hablar más en profundidad en las siguientes secciones de este documento.

2.2. DAG API

El tipo de aplicaciones de procesamiento de datos en los que se centra Apache Tez se pueden representar de forma natural como DAGs. En estos DAGs, los datos que se van a procesar provienen de fuentes de datos (E.g. ficheros), los cuales se van transformando a lo largo de un número definido de procesos y finalmente se escriben en ficheros que habitualmente se encuentran en un sistema de ficheros distribuido de Hadoop.

Tez no es la primera tecnología en emplear los DAG como forma de modelar la computación de una aplicación. Concretamente, en Tez se emplean como herramienta principal para modelar el movimiento de los datos en todas sus aplicaciones. Para ello, se emplean los con-

ceptos ya conocidos de *Vertices* y *Edges*, los cuales se van a explicar en profundidad y en el contexto de Apache Tez a continuación. [10]

2.2.1. *Vertices*

Un *Vertex* en Tez representa un nodo de procesamiento de datos, es decir, contiene parte de la lógica de la aplicación. En ellos, los datos se transforman y se procesan, pasando de un *Vertex* a otro hasta escribir finalmente la información resultante en el HDFS (*Hadoop Distributed File System*).

A la hora de implementar una aplicación, el desarrollador tendrá que asignarle a cada uno de los *Vertex* una clase *processor*, la cual define la lógica de ese *Vertex* y será lo que se ejecute en cada una de las tareas del mismo.

Por lo general, un *Vertex* de un DAG ejecutará de forma paralela un número (habitualmente alto) de tareas. Estas tareas implementarán cada una de ellas la clase *processor* que se le ha asignado al *Vertex*, de forma que cada una de ella realiza una parte del trabajo total de éste. Cada tarea consumirá una cantidad de entradas de datos del total que le llegan al *Vertex*, habitualmente divididas de forma equitativa entre todas las tareas, contando así con una carga de trabajo similar en todas ellas.

El número de tareas en las que se ejecute un *Vertex* se verá fijado en la propia definición del *Vertex*. En el caso de ser el uno de los primeros *Vertices* del DAG, los cuales consumen los datos desde una fuente de datos y no desde otro *Vertex*, este paralelismo les vendrá definido por la propia fuente de datos. No obstante, a pesar de que puede definirse de forma estática, Tez también es capaz de definirlo en tiempo de ejecución, lo cual lo convierte en una herramienta muy potente a la hora de ahorrar recursos y de ejecutar el trabajo asignado de la forma más rápida y eficiente posible.

Se profundizará más en las características de un *Vertex* en la siguiente sección de este documento, donde se centrará la atención en los conceptos más relevantes de Tez de cara al rendimiento.

2.2.2. *Edges*

Un *Edge* en Tez define el movimiento de datos entre un *Vertex* y otro dentro de un mismo DAG. Durante la implementación de una aplicación, el desarrollador ha de definir una serie de *Edges*, cada uno con las propiedades correspondientes para comunicar los *Vertices* origen y destino. Una vez definidos, tendrá que especificar qué *Vertices* se conectan a través de qué *Edges*, definiendo así el DAG para la aplicación. Las propiedades que pueden configurarse para cada *Edge* son las siguientes:

En primer lugar, el *scheduling*. Esta propiedad permite especificar cuándo se van a planificar (*schedule*) las tareas del *Vertex* destino. Existen dos opciones para esta propiedad:

– *Sequential*. Indica que las tareas del *Vertex* destino se programarán cuando las del *Vertex* origen haya finalizado su ejecución.

– *Concurrent*. Indica que las tareas del *Vertex* destino se irán programando de forma concurrente a la ejecución de las tareas origen.

No obstante, en la versión de Apache Tez empleada durante el desarrollo del proyecto (0.8.5), la opción *concurrent* aún no se encuentra disponible, por lo que Tez obliga a emplear el tipo *sequential* para todos los *Edges*.

En segundo lugar, la fuente de datos. Esta propiedad es la que especifica cuánto tiempo estarán disponibles para el *Vertex* de destino los datos que el *Vertex* origen ha generado y le ha enviado a través del *Edge*. Existen tres posibilidades en Apache Tez:

– *Persisted*. Indica que la información de salida de cada una de las tareas del *Vertex* origen estará disponible para el *Vertex* destino incluso después de que dichas tareas hayan finalizado. No obstante, esta información puede desaparecer más adelante, de forma que habría que volver a ejecutar el *Vertex* si se necesitase la información.

– *Persisted-reliable*. Indica que todos los datos de un *Vertex* origen se escribirán y guardarán de forma segura en el sistema de ficheros, de manera que siempre estarán disponibles tras su ejecución. Obviamente, emplear esta opción supondría perder parte de las ventajas que ofrece Apache Tez frente a MapReduce en cuanto a no escribir la información entre nodos en disco, perjudicando así al rendimiento final de la aplicación.

– *Ephemeral*. Indica que la información del *Vertex* origen sólo estará disponible durante la ejecución de las tareas de dicho *Vertex*.

Para emplear la opción *ephemeral*, es un requerimiento indispensable fijar la opción *concurrent* en la propiedad de *scheduling* del *Edge*. No obstante, como ya se ha indicado, esta opción no está disponible en la versión actual de Apache Tez, por lo que la opción de la fuente de datos de tipo *ephemeral* tampoco lo está. Además, el tipo de fuente de datos *persisted-reliable* tampoco se encuentra disponible, por lo que actualmente Tez te obliga a emplear el tipo *persisted*.

Y por último, se encuentra la propiedad del movimiento de datos. Esta es, sin duda alguna, la propiedad más importante que ofrece Apache Tez y que lo diferencia en gran medida de otras tecnologías similares. Lo que especifica esta propiedad es cómo se van a enviar los datos desde las tareas del *Vertex* origen a las del *Vertex* destino. En Tez, los datos que se envían son, por lo general, parejas clave-valor, una a una. De nuevo, Tez ofrece tres opciones de configuración distintas dentro de esta propiedad:

– *One-to-one*. Indica que toda la información que genere la enésima tarea del *Vertex* origen, irá a parar a la enésima tarea del *Vertex* destino. Por este motivo, el número de tareas de ambos *Vertices* deberá de ser igual.

– *Broadcast*. Indica que cada salida de cada una de las tareas del *Vertex* origen, se enviará a todas las tareas del *Vertex* destino. Es decir, cada mensaje de salida se copia a todas las tareas del destino.

– *Scatter-gather*. Indica que la enésima salida de cada una de las tareas del *Vertex* origen, se enviarán a la enésima tarea del *Vertex* destino. Por ejemplo, si tenemos un *Vertex* origen con 3 tareas, y un *Vertex* destino con 2 tareas, la 1ª salida de las 3 tareas del origen, irá a la 1ª tarea del destino, recibiendo así 3 mensajes.

En la Figura 2.2 puede encontrarse un ejemplo ilustrando los tres tipos de movimiento de datos.

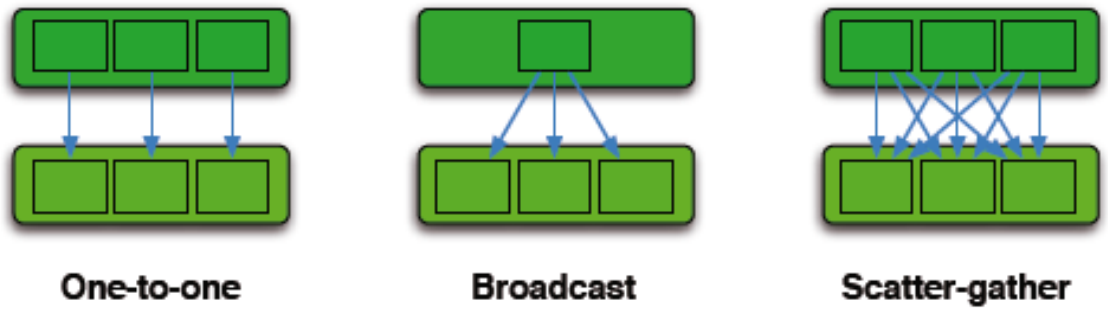


Figura 2.2: Ejemplo de los tres tipos de movimientos de datos configurables en un *Edge*

2.2.3. Fuentes de entrada y salida de datos

Con los dos conceptos que se acaban de definir (los *Vertices* y los *Edges*), puede construirse un DAG completo que defina el flujo de datos y de ejecución de una aplicación. Sin embargo, dichos datos, los que procesa el DAG, han de provenir de algún punto al comienzo del DAG, y del mismo modo han de escribirse en algún otro lugar cuando se hayan procesado, ya sea en un punto intermedio del DAG o al final del mismo.

En Tez, esto se ha modelado como las Fuentes de entrada y salida de datos (*Data sources and sinks*). Cada uno de los *Vertices* que dan comienzo al procesamiento en un DAG han de tener asignada una fuente de datos.[1] Por tanto, hay que tener en cuenta que la clase *processor* asignada a esos *Vertices* va a tener como tarea, al menos, la lectura de los datos de la fuente de entrada, que realmente no se diferencia en nada de leerlos de otro *Vertex*, salvo

[1]Tez puede tener tantos *Vertices* de inicio como caminos distintos se vayan a ejecutar en un mismo DAG. Por defecto, sólo admite una fuente de datos para cada *Vertex*. Sin embargo, al ser muy personalizable, pueden crearse clases propias que hereden de otras predefinidas y que admitan más entradas.

que llevará más tiempo de procesamiento. De igual manera, tanto en los *Vertices* finales de un DAG como en cualquier otro *Vertex* intermedio que quiera realizar una escritura en disco, habrá que asignarles una salida de datos. En estas salidas se escribe igual que si se van a pasar los datos a otro *Vertex* a través de un *Edge*, y por lo general los *processors* que se les asignan a estos *Vertex* extienden de otra clase que vacía los *buffers* de escritura al final del procesamiento del *Vertex* para que se escriba todo en disco. A diferencia de las fuentes de entrada de datos, un *Vertex* puede tener asignadas tantas fuentes de salida como se deseen, mezclándolas también con escrituras en *Edges* para enviar los datos a otros *Vertex*.

Estas fuentes de entrada y salida de datos, aunque tienen algunas propiedades configurables en las que no se va a entrar en detalle en este documento, son principalmente rutas a ficheros en un sistema de ficheros, habitualmente el sistema de ficheros distribuido de Hadoop.

Para todos los demás *Vertices* del DAG que no sean los primeros, los últimos o que no vayan a escribir en disco, no es necesario definir ningún tipo de entrada o salida de datos, dado que al conectar los unos con los otros mediante *Edges* ya se están definiendo todas las entradas y salidas de cada uno de ellos.

2.2.4. Ejemplo práctico

Finalmente, una vez expuestos todos los conceptos básicos de la DAG API de Apache Tez, se va a presentar a continuación un breve ejemplo práctico de su uso para definir un DAG. Dicho ejemplo puede verse en el Ejemplo 2.1 y va a explicarse a continuación.

En este caso práctico, se define un DAG para un programa simple de conteo de palabras en un fichero. Para ello, primero se crea el DAG, que inicialmente está vacío. A continuación, se crean los dos *Vertices* que van a formar parte de dicho DAG, uno que va a leer del fichero de entrada y va a separar cada una de las palabras, al cual se le ha asignado una fuente de entrada de datos; y otro que va a juntar los resultados que le llegan del primer vértice y que, finalmente, escribirá el conteo total en la salida de datos que se le ha asignado. Para que ese flujo de datos tenga lugar, se define un *EdgeProperty*. Esto no es un *Edge*, sino un objeto que contiene todas las propiedades para ese *Edge* (las propiedades que se han introducido en la Sección 2.2.2). Finalmente, se añaden al DAG los dos *Vertices* creados y, ahora sí, se crea el *Edge* que los comunica, empleando las propiedades definidas en la línea anterior. Como puede verse, aunque se trate de un ejemplo simple, es muy sencillo y visual crear un DAG que represente el flujo de datos de una aplicación.

```
1   DAG dag = DAG.create("WordCount");
2
3   Vertex tokenizerVertex = Vertex.create("Tokenizer", TokenProcessor.class)
4       .addDataSource("Input", HdfsInitializer.class);
5
6   Vertex summationVertex = Vertex.create("Summation, SumProcessor.class)
7       .addDataSink("Output", HdfsCommitter.class);
8
9   EdgeProperty edgeProperty = EdgeProperty.create(scatter_gather,
10      KeyValueShuffleWriter.class, KeyValueShuffleReader.class);
11
12  dag.addVertex(tokenizerVertex).addVertex(summationVertex)
13      .addEdge(Edge.create(tokenizerVertex, summationVertex, edgeProperty));
```

Ejemplo 2.1: Ejemplo práctico del uso de la DAG API para definir un DAG.

2.3. Rendimiento en Tez

A lo largo de este capítulo se han introducido los conceptos principales de Tez para comprender la herramienta y para saber cómo se construyen los DAG en Tez, en los cuales está basado todo el flujo de datos de cada aplicación Tez.

En esta última sección, se van a poner en conjunto algunos conceptos que no se habían mencionado aún, junto con otros que ya se han introducido. Estos conceptos tienen especial relevancia en el rendimiento de las aplicaciones Tez, y son por tanto parte fundamental de este trabajo de cara a modelar el rendimiento de éstas.

En la Tabla 2.1 pueden encontrarse listados todos los conceptos relacionados más estrechamente con el rendimiento que vamos a tocar en esta sección y que van a ser los que se modelen en el perfil. A continuación se van a describir cada uno de estos conceptos de forma

#	**Concepto**	**Significado**
1.	*Vertex*	Procesamiento y transformación de datos
2.	*Edge*	Paso de datos entre un *Vertex* y otro
3.	*Parallelism*	Número de tareas que se ejecutan en cada *Vertex*
4.	*MinSrcFraction*	Número de tareas que tienen que finalizar para empezar a programar las siguientes
5.	*MaxScrFraction*	Número de tareas que tienen que finalizar para programar todas las siguientes
6.	*DataMovementType* (dmt)	Tipo de política de movimiento de datos en un *Edge*
7.	*VirtualCores*	Número de núcleos virtuales asignados a cada tarea de un *Vertex*
8.	*Memory*	Tamaño de memoria asignado a cada tarea de un *Vertex*

Tabla 2.1: Conceptos de Tez con impacto en el rendimiento

un poco más detallada de lo que se muestra en la tabla. Los dos primeros, *Vertex* y *Edge*, al igual que el número 6, *DataMovementType*, ya se han expuesto y explicado en la Sección 2.2 de este mismo capítulo, por tanto, se va a pasar a explicar los demás conceptos de la tabla.

En primer lugar, en el índice 3 de la tabla, se encuentra el concepto de paralelismo (*parallelism*), el cual, en cierto modo, ya se ha mencionado en secciones anteriores. Esta propiedad de las aplicaciones Tez hace referencia al número de tareas que se van a ejecutar en paralelo en un mismo *Vertex*. Es decir, cada una de las tareas de ese *Vertex* implementará el mismo *processor* y se ejecutará en paralelo con el resto de tareas de ese *Vertex*, procesando un número determinado de entradas cada una. Claro está, no se pueden ejecutar un número de tareas infinito en paralelo; el número de tareas que podrán ejecutarse en paralelo vendrá definido por otros conceptos que se verán más adelante en esta sección.

A continuación, en el índice 4 de la tabla, se encuentra el concepto *MinSrcFraction*. Esta propiedad, aunque se cuenta como una propiedad más para los *Vertices* en este trabajo, sólo es configurable para aquellos *Vertices* que reciban información de otro *Vertex* a través de un *Edge* con política *scatter-gather*. Lo que especifica esta característica es lo siguiente: las tareas del *Vertex* de destino, en una conexión entre dos *Vertices*, se empezarán a planificar (*schedule*) cuando un porcentaje igual al definido en *MinSrcFraction* de las tareas del *Vertex* origen se haya completado. Por ejemplo, si hemos fijado un 0.3 como valor para la propiedad *MinSrcFraction* del *Vertex* destino, sus tareas empezarán a programarse cuando el 30 % de las tareas del *Vertex* origen se hayan completado.

El siguiente concepto en la tabla, *MaxSrcFraction*, se explica casi por sí solo tras la descripción del concepto anterior. En este caso, el porcentaje asignado a esta característica indica cuánto el *Vertex* destino podrá programar todas sus tareas. Obviamente, *MaxSrcFraction* siempre habrá de ser mayor que *MinSrcFraction*. Estos dos conceptos se emplean en Tez para conseguir un "inicio perezoso" de las tareas de los próximos *Vertices* que se van a ejecutar. Este "inicio perezoso" le resulta especialmente útil a Tez cuando se lo ha programado para ajustar automáticamente el paralelismo del próximo *Vertex*, de forma que cuanto más tiempo tiene, con más información cuenta para realizar dicho ajuste. Con la información que consiga reunir en un periodo de tiempo determinado, Tez ejecutará ciertos algoritmos que le permitirán determinar cuál es el paralelismo óptimo para el *Vertex* destino en función de los datos que le estén llegando del *Vertex* origen.

Finalmente, en la tabla pueden encontrarse dos conceptos relacionados con los recursos asignados a un *Vertex* en los índices 7 y 8. Más concretamente, estos dos recursos, núcleos virtuales (*virtualCores*) y memoria (*memory*), se asignan a cada una de las tareas de un *Vertex*, y no al *Vertex* propiamente dicho. Estas dos características pueden asignarse de dos formas distintas: durante la creación de un *Vertex*, pudiendo así personalizar los recursos de cada *Vertex* de un DAG; o durante la configuración del entorno, fijando los mismos recursos para todos los *Vertices* del DAG. Por supuesto, estos estarán limitados a la cantidad de recursos que se le hayan asignado al entorno donde se va a ejecutar, los cuales también se pueden personalizar. Es mediante estos dos conceptos, especialmente el de núcleos virtuales,

mediante los cuales se verá limitada la característica del paralelismo en un *Vertex*, dado que al final cualquier aplicación se ve limitada por los recursos físicos de los que dispone.

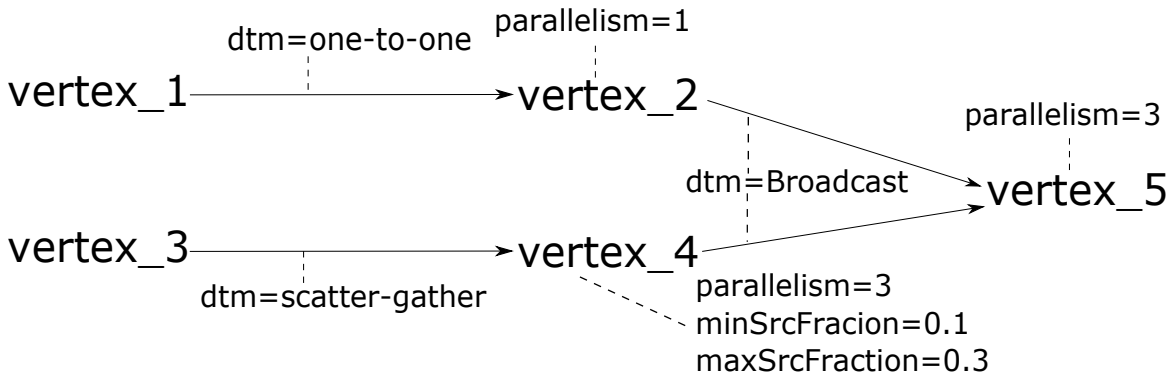


Figura 2.3: Ejemplo de aplicación Tez

Para finalizar con este capítulo de introducción de la tecnología Apache Tez, se presenta en la Figura 2.3 un ejemplo de una aplicación Tez sencilla. Concretamente, se trata de una aplicación que implementa un contador de palabras. Dado que se hará referencia a este ejemplo más adelante en el documento, se va a explicar brevemente para afianzar todos los conceptos que se han ido exponiendo en este capítulo.

Comenzando por los dos primeros *Vertices*, "vertex_1" y "vertex_3", son los encargados de leer los ficheros de los cuales se va a hacer el conteo de palabras. Además, se encargan de dividir todo lo que se lee del fichero en palabras y enviarlas una a una a sus siguientes *Vertices*. Como se puede observar, en la figura no se representan ni las fuentes de entrada ni las de salida, por lo que se asume que cada *Vertex* del comienzo del DAG tiene una fuente de entrada de datos y el *Vertex* final tiene una salida. El fichero puede ser el mismo para los dos *Vertices* de comienzo, simplemente se leerá dos veces.

Continuando con el "vertex_2", éste está conectado al "vertex_1" mediante un *Edge* con política *one-to-one*, lo cual lo obliga a tener un paralelismo 1, pues el paralelismo del "vertex_-1" también es de 1.

Pasando al otro camino de ejecución, se encuentra el "vertex_4", el cuál se encuentra conectado con el "vertex_3" mediante un *Edge* con política *scatter-gather*. En este caso, se puede apreciar que el *Vertex* cuenta con un paralelismo de 3, y se han fijado las propiedades de *MinSrcFraction* y *MaxSrcFraction* a 0.1 y 0.3, respectivamente. Dado que es el *Vertex* destino de una conexión *scatter-gather*, es posible fijar estas propiedades.

Por último, los *Vertices* 2 y 4 convergen en el "vertex_5", conectándose ambos mediante dos *Edges* con política *broadcast*, lo cual implica que cada mensaje se copiará a cada una de las tareas del "vertex_5". En este caso, se ha fijado también con valor 3 el paralelismo de este último *Vertex*, por tanto, cada uno de los mensajes de los dos *Edges* de entrada se procesarán

3 veces.

Ahora que ya se han introducido y ejemplificado todos los conceptos básicos de Tez, así como los más relevantes para el rendimiento de sus aplicaciones, en el siguiente capítulo se va a exponer el modelado de dichos conceptos a un modelo UML.

Capítulo 3

Modelado UML de aplicaciones Tez

Tras analizar la tecnología Tez y presentar los conceptos y parámetros más relevantes para analizar su rendimiento, es necesario modelar de alguna forma esta tecnología con dichos parámetros. Este modelo servirá más adelante para realizar una transformación automática a otros modelos que permitan el análisis de rendimiento, que en este caso serán redes de Petri.

Para realizar dicho modelo se ha creado un lenguaje de dominio específico (DSML, Domain Specific Language) aplicando la técnica de perfiles para la tecnología de Tez. El modelo de Tez incluye la topología de la tecnología, es decir, los DAG de Tez; los parámetros que ya se han identificado en el Capítulo 2 y el despliegue. Por tanto, en este capítulo se va a presentar el modelado en UML de Tez con diagramas de actividad, los cuales se complementarán con diagramas de despliegue. Serán estos diagramas los que se transformarán más adelante a otros modelos para el análisis de su rendimiento.

3.1. Diagramas UML para Tez

La Figura 3.1 representa el diagrama de actividad UML correspondiente al ejemplo introducido en la Figura 2.3. Tal como se muestra en el ejemplo, un diagrama de actividad de una aplicación Tez siempre comenzará con un conjunto de nodos iniciales que estarán conectados a los primeros *Vertices* de dicho diagrama. Esto se debe, como ya se ha explicado en el Capítulo 2, a que los primeros *Vertices* son los encargados de leer los datos que se van a procesar en el DAG, y estos nodos iniciales representan las fuentes de información que se van a leer. De igual manera, existe un nodo al final del diagrama que representa la salida de datos al finalizar el procesamiento.

En el enfoque que aquí se presenta, un diagrama de actividad UML se interpreta como un DAG de una topología particular de Tez. En el caso de las acciones, la semántica que se representa es la misma que la del UML estándar: cada acción es una tarea que finaliza su ejecución cuando ha procesado todos los datos de entrada. No obstante, cabe recordar que al configurar un Edge como *persisted* (ver Sección 2.2.2), la información puede perderse y ser necesario volver a ejecutar un nodo.

Por otra parte, en el caso de los arcos que conectan las actividades, no representan una sucesión lógica de acciones, como en el estándar UML, sino que representan canales de comunicación entre dos tareas.

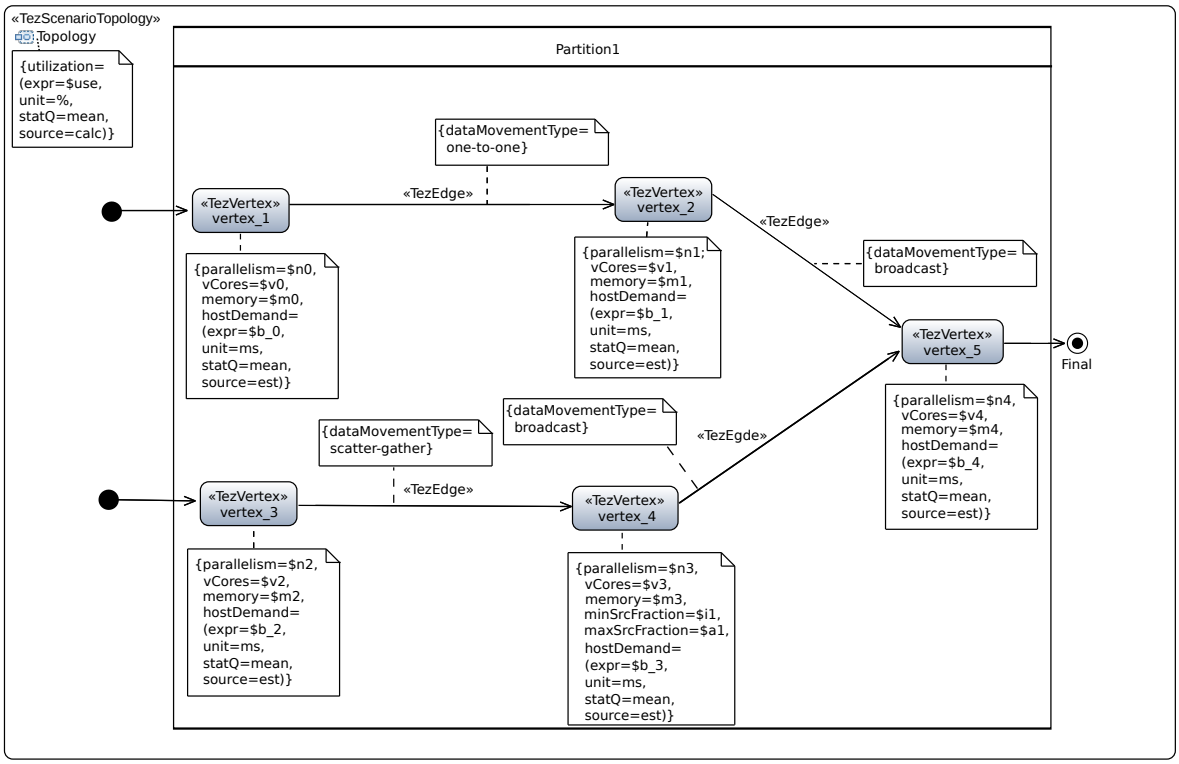


Figura 3.1: Ejemplo de diagrama de actividad de una aplicación Tez con anotaciones del perfil

Una vez expuesto el diagrama de actividad para aplicaciones Tez, se presenta a continuación el diagrama de despliegue correspondiente al ejemplo presentado (ver Figura 3.2).

Como puede observarse, todos los nodos del diagrama de actividades se encuentran agrupados dentro de una partición. Las particiones representan un recurso de computación en los diagramas de actividad de Tez; por tanto, en la Figura 3.2 se muestran todos los nodos dentro de un mismo recurso de computación.

3.2. Perfil UML para Tez

Como puede observarse en la Figura 3.1, el diagrama no sólo cuenta con los elementos habituales de un diagrama de actividad, sino que incluye numerosas notas asociadas a cada elemento. Estas notas representan las propiedades principales de Tez relativas al rendimiento y que ya se han recogido en la Tabla 2.1.

Dado que estas propiedades no podían representarse de ninguna forma en un diagrama de actividad, se decidió convertirlas en estereotipos y etiquetas. Ambos dos son mecanismos de extensión que ofrece el estándar de UML. Basados en esto, se creó un perfil UML para

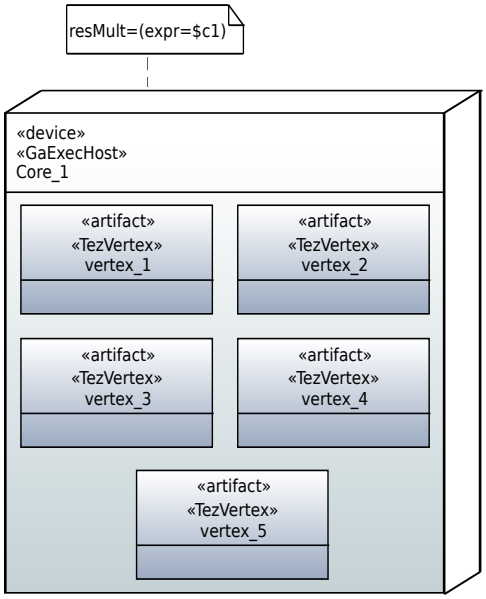


Figura 3.2: Ejemplo de diagrama de despliegue de una aplicación Tez con anotaciones del perfil

Tez.

Un perfil de UML es un conjunto de estereotipos definidos que pueden aplicarse a elementos de un modelo UML para extender su semántica. En este caso, se han empleado para extender el UML con los conceptos capturados de Tez.

El perfil creado para Tez hereda en gran parte del perfil estándar de MARTE [7]. Esto se debe, principalmente, a que MARTE ofrece el sub-perfil GQAM, un *framework* para análisis cuantitativo especializado en análisis de rendimiento. Dado que lo que se busca retratar en el perfil de Tez son conceptos de rendimiento que más adelante se emplearán en la transformación a un modelo de análisis (las redes de Petri), este perfil es perfecto para lo que se quiere conseguir.

Además del sub-perfil GQAM, MARTE ofrece otros dos sub-perfiles que serán de gran utilidad también para la definición del perfil de Tez:

- NFP. Este sub-perfil tiene como objetivo describir propiedades no funcionales en un sistema. En este caso, la propiedad no funcional que se busca describir es el rendimiento.
- VSL. Este sub-perfil provee un lenguaje textual que permite especificar valores de métricas, restricciones, propiedades y parámetros relacionados, en este caso, con el rendimiento.

En los modelos UML basados en el perfil de Tez, las expresiones VSL se emplean con dos objetivos principales: (i) especificar los valores de las propiedades NFP en el modelo, es decir, los parámetros de entrada; y (ii) para especificar las métricas que se calcularán para el modelo, es decir, los resultados finales de la ejecución.

Un ejemplo del uso de VLS para especificar parámetros es la etiqueta `parallelism`, la cual puede encontrarse en todos los nodos del diagrama de la Figura 3.1 y se corresponde con un valor de tipo `NFP_Integer`.

Por otro lado, dos ejemplos interesantes del uso de VLS para especificar métricas resultantes de la ejecución son los siguientes:

```
(expr=$b_0, unit=ms, statQ=mean, source=est)
    (1)        (2)        (3)        (4)
```

Esta primera expresión especifica que el **vertex_1** de la Figura 3.1 requiere `$b_0` (1) *milisegundos* (2) de tiempo de ejecución, cuyo valor promedio (3) se obtendrá de una estimación en el sistema real (4). Esta expresión se corresponde con la etiqueta `hostDemand`, que puede encontrarse en todos los nodos de la Figura 3.1, y es de tipo `NFP_Duration`.

```
(expr=$use, unit=%, statQ=mean, source=calc)
    (1)        (2)        (3)        (4)
```

En cuanto a la segunda expresión, especifica la métrica de rendimiento que se va a calcular, que en la Figura 3.1 se corresponde con la etiqueta `utilization`. Concretamente, especifica que se quiere calcular (4) la utilización como porcentaje de tiempo (2), de todo el sistema o de un recurso en particular, cuyo valor promedio (3) se obtendrá en la variable `$use` (1). Dicho valor es, claramente, el resultado de la simulación del modelo de rendimiento.

Una vez introducido el concepto de perfil y aquellos perfiles existentes en los que se basa el que se ha creado para Tez, se presentan a continuación en la Tabla 3.1 los estereotipos y etiquetas creados para el perfil de Tez.

Concepto Tez	**Estereotipo**	**Aplicado a**	**Etiqueta**	**Tipo**
Vertex	≪TezVertex≫	Acción y Nodo de actividad		
Parallelism			parallelism	NFP_Integer
MinSrcFraction			minScrFraction	NFP_Real
MaxSrcFraction			maxScrFraction	NFP_Real
VirtualCores			virtualCores	NFP_Integer
memory			memory	NFP_Integer
			hostDemand	NFP_Duration
Edge	≪TezEdge≫	Control del flujo		
DataMovementType			dataMovementType	{one-to-one, scatter-gather, broadcast}

Tabla 3.1: Estereotipos y etiquetas del perfil de Tez

En primer lugar se encuentra el estereotipo creado para los *Vertices* de Tez. Éste hereda del estereotipo MARTE::GQAM::GaStep (Ver Figura 3.3), el cual representa un paso computacional y le proporciona la propiedad `hostDemand`, reflejada en la tabla como una etiqueta con el mismo nombre. El resto de etiquetas que acompañan al estereotipo del *Vertex* se corresponden con los conceptos de Tez ya explicados en la Sección 2.3, Tabla 2.1.

En segundo lugar, se encuentra el estereotipo creado para los *Edges* de Tez. Éste también hereda del estereotipo MARTE::GQAM::GaStep (Ver Figura 3.3), dado que también representa un paso computacional, que en este caso se corresponde con el movimiento de datos entre dos nodos. Al igual que el estereotipo del *Vertex*, cuenta con etiquetas asociadas a él. En este caso, la única etiqueta asociada es la que representa el tipo de movimiento de datos, ya explicado en la sección Sección 2.2.

Por otro lado, también se ha empleado otro estereotipo de MARTE para representar los recursos de computación, como puede observarse en el diagrama de despliegue de la Figura 3.2 ya expuesto. Dicho estereotipo es el MARTE::GQAM::GaExecHost, el cual permite definir los recursos disponibles, es decir, el número de núcleos, representado con la etiqueta `restMult`.

El perfil de Tez se ha implementado empleando el entorno de modelado Papyrus para Eclipse.

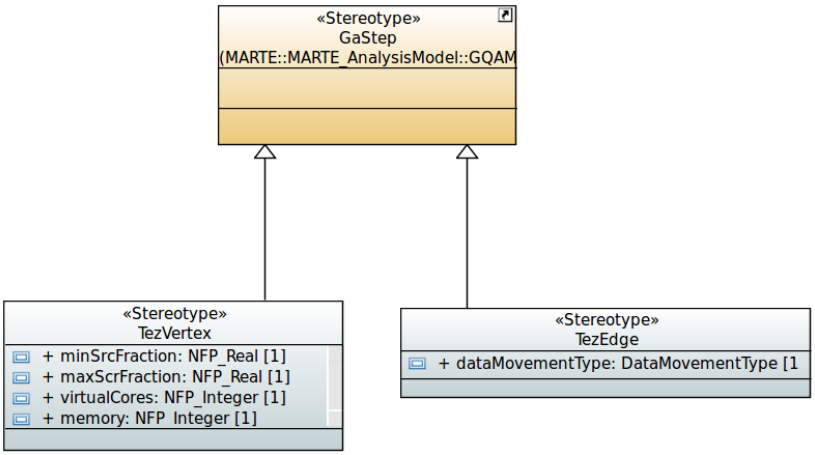


Figura 3.3: Implementación del perfil de Tez

3.3. Topologías complejas de Tez

Empleando la propuesta que se presenta en este trabajo, pueden modelarse topologías más complejas de las que se han expuesto en este documento para aplicaciones Tez. Sin embargo, la topología no es suficiente cuando se quieren representar comportamientos más complejos de las tareas que se ejecutan (los *Vertices*). Hasta este punto del documento, se ha asumido que las tareas eran cajas negras y, por tanto, se han modelado como acciones en un diagrama de actividad.

No obstante, UML cuenta con funciones de anidamiento de modelos en los nodos de actividad; es decir, permite modelar lo que ocurre dentro de una actividad con otro diagrama de actividad UML. Gracias a esta característica, la propuesta presentada permite al diseñador que la emplee para modelar con detalle las tareas que más interés tengan en el rendimiento de una aplicación Tez. Al ser simplemente programas Java, el modelado de estas tareas puede realizarse con la interpretación habitual de un diagrama de actividad, empleando sub-actividades, sincronizaciones, etc.. Por último, la especificación en detalle de estos *Vertices* de Tez soporta anotaciones de MARTE, por lo que pueden utilizarse otras especificaciones o anotaciones de rendimiento.

Capítulo 4

Transformación del modelo UML a redes de Petri

Ahora que ya se ha creado un perfil que recoge los conceptos principales de Tez, una técnica de modelado en UML, es necesario transformar esos modelos a otro modelo que permita evaluar el rendimiento de una aplicación a partir de las métricas ya mencionadas como utilización, tiempo de respuesta y producción.

En este caso, como ya se ha mencionado a lo largo del documento, el modelo de evaluación de rendimiento que se va a emplear son las Redes de Petri Estocásticas Generalizadas o GSPN (*Generalized Stochastic Petri Nets*) [5]. (El Anexo B recoge una pequeña introducción de estas redes de Petri.)

Por consiguiente, en este capítulo se van a presentar un conjunto de patrones de transformación. Dichos patrones toman como entrada una parte del diseño de Tez en UML presentado en el Capítulo 3 y generan una sub-red GSPN. Los patrones presentados son transformaciones que pueden automatizarse (mediante lenguajes como QVT) para obtener el modelo completo de GSPN. Durante el desarrollo de este trabajo, las transformaciones y el modelo final se han realizado manualmente.

4.1. Transformación del diagrama de actividad

En las Tablas 4.1 y 4.2 se presentan los patrones de transformación para los diagramas de actividad y despliegue. En cada una de las tablas, la columna de la izquierda contiene el patrón de entrada, es decir, los elementos UML (en algunos de ellos con los estereotipos definidos en la Sección 3.2); y, por otro lado, la columna de la derecha contiene la sub-red GSPN correspondiente con los elementos UML. Para una comprensión más sencilla de estas transformaciones, se han representado con líneas discontinuas y en color gris aquellos elementos que no pertenecen a la transformación, pero sí están relacionados con ella.

Comenzando con la Tabla 4.1, en primer lugar se encuentra el patrón *P1*. Éste se corresponde con el comienzo y final de una actividad, que da lugar a una red de Petri cerrada

para ejecuciones consecutivas de la misma. El lugar se corresponde con el del patrón *P2* y la transición con la última del patrón *P6*. Como se puede observar, cuenta con un marcaje inicial en el lugar de comienzo, permitiendo así que la transición inmediata se dispare y de comienzo a todos los flujos de ejecución definidos.

A continuación, el patrón *P2* representa el comienzo de un flujo de ejecución en un red y, por consiguiente, la generación de los datos (*tokens*) que se van a procesar. El primer lugar y transición, presentados con líneas discontinuas y en gris, son los definidos en el patrón *P1*.

En la fila siguiente de la tabla se encuentra el patrón *P3*, correspondiente a la transformación de un *Vertex* de Tez. La sub-red generada consta de dos lugares, una transición con tiempo y un conjunto de arcos. Los lugares p_{A1} y p_{A2} representan los estados de espera y de procesamiento de los mensajes entrantes, respectivamente. El lugar p_{A1} contará con tantos *tokens* como indique el valor de la etiqueta *parallelism* ($n0). Por otro lado, el tiempo de las transiciones temporales en las GSPN se representa mediante una tasa (la tasa de generación de *tokens*)[1]. Dado que se quiere representar el tiempo de ejecución medio de una tarea de un *Vertex*, el tiempo en estas transiciones temporales se representa mediante la inversa del tiempo de ejecución medio de dichas tareas (1/$time).

A continuación se encuentran los patrones *P4* y *P5*, los cuales representan la recepción de un mensaje en un *Vertex*. En *P3* se puede observar el caso más sencillo, en el cual un *Vertex* destino recibe mensajes de un *Vertex* de origen, mientras que en *P5* se representa la recepción de mensajes de dos o más *Vertices* de origen. En ambos patrones, la transición temporal t_A hace referencia a la transición del mismo nombre en el patrón *P3*; no obstante, al no formar parte íntegra del patrón de transformación, se representa con líneas discontinuas y en gris, como ya se ha mencionado anteriormente. De igual manera, los lugares p_{B1} y p_{B2} (p_{C1} y p_{C2} en el patrón *P5*) representan los lugares del patrón *P3* del *Vertex* de destino.

Como puede observarse, existen un lugar y un transición inmediata "extra" que no serían necesarios para que los mensajes llegasen de un *Vertex* a otro. No obstante, estas dos adiciones cumplen un propósito muy importante a la hora de crear una red de Petri de una aplicación Tez. Tal y como aparece representado en la red, los arcos que entran y salen de la primera transición inmediata cuentan con un peso en cada uno, los cuales se corresponden con el paralelismo de *Vertex* de origen en el caso del arco de entrada y del paralelismo del *Vertex* destino en el caso del arco de salida. Este consumo y generación de *tokens* permite representar la división de mensajes entre las distintas tareas de un *Vertex* en función de su paralelismo, y de esta forma poder simular correctamente el consumo de recursos de la aplicación.

Por otro lado, en el caso del patrón *P5*, no existe ningún tipo de sincronización entre los mensajes entrantes de los *Vertices* origen, más allá de que la transición inmediata que

[1]En las GSPNs de este trabajo, los *tokens* tienen tres posibles significados: cada *token* puede representar un conjunto de mensajes entre un *Vertex* y otro, el paralelismo de un *Vertex*, o los recursos disponibles de un sistema.

consumirá los mensajes del *Vertex* destino habrá de esperar a que haya tantos *tokens* como la suma de los paralelismos de los *Vertices* origen, asegurándose así de que han finalizado, como se verá más adelante.

Para finalizar con la Tabla 4.1, se encuentra el patrón *P6* (el patrón *P7* se detalla en la Sección 4.2). Este patrón representa el final de un flujo de procesamiento de datos, que finalizará escribiendo los resultados en un sistema de ficheros. Los últimos lugares y transición inmediata dan lugar a una red cerrada para poder volver a empezar la ejecución. Aunque se representan dos salidas, pueden ser una o más salidas las que se den en una red.

Centrando la atención ahora en la Tabla 4.2, se pueden observar tres patrones distintos de movimiento de datos entre dos *Vertices*. Estos tres patrones se corresponden con los tres tipos de movimiento de datos presentados en la Sección 2.2.2 y modelados más adelante en la Sección 3.2.

En primer lugar se encuentra el patrón *P8*, correspondiente al tipo de movimiento de datos *broadcast*. Tal y como se presenta, el patrón no se diferencia en nada del movimiento de datos básico presentado en la Tabla 4.1. Esto se debe a que la única particularidad de este tipo de movimiento de datos es que envía una copia de cada mensaje a cada una de las tareas del *Vertex* destino. Dado que en la transición temporal del *Vertex* ya se contempla el tiempo de ejecución medio de las tareas de ese *Vertex* (es decir, contando con todas las copias de mensajes que le llegan), representar la multiplicación de mensajes en la red de Petri sólo daría lugar a multiplicar el tiempo de ejecución de forma incorrecta.

A continuación se encuentra el patrón *P9*, que se corresponde con el tipo de movimiento de datos *scatter-gather*. En este caso, la diferencia con el patrón básico es apreciable. La segunda transición inmediata pasa de ser una única a dividirse en n transiciones inmediatas, donde n es el paralelismo del *Vertex* destino (B). Cada una de esas transiciones cuenta con una probabilidad equiprobable de dispararse. Las transiciones temporales de cada uno de los caminos pueden configurarse con tiempos de ejecución distintos. De esta forma, la red de Petri trata de representar la distribución de los mensajes siguiendo el tipo *scatter-gather*, el cual ya se ha explicado en la Sección 2.2.2.

Para finalizar, la Tabla 4.2 contiene el patrón *P10*, correspondiente al tipo de movimiento de datos *one-to-one*. Como se puede observar, el resultado es muy similar al patrón *P8*, con la diferencia de que cada uno de los caminos contiene un lugar más. Cada uno de estos lugares extra en cada uno de los caminos contendrán un *token* inicial, de forma que, aunque existan recursos suficientes para seguir procesando mensajes, no será posible si este *token* no se encuentra disponible. Mediante estos lugares, se trata de representar en la red de Petri el concepto del emparejamiento de tareas del *Vertex* origen y el destino, como también se ha explicado en la Sección 2.2.2. Al igual que en el patrón *scatter-gather*, las transiciones temporales pueden configurarse con un tiempo de ejecución distinto cada una.

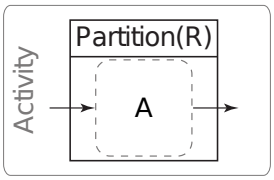
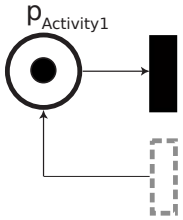
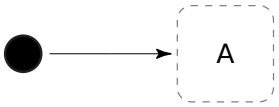
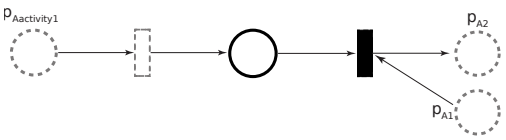
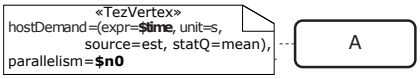
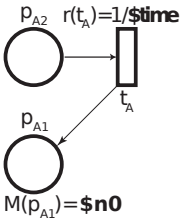
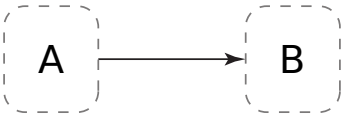
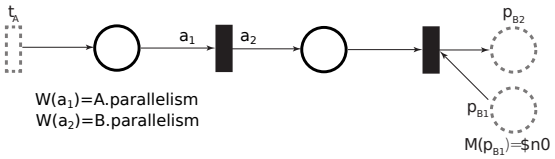
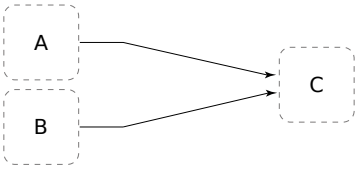
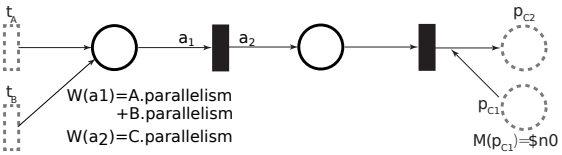
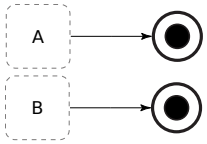
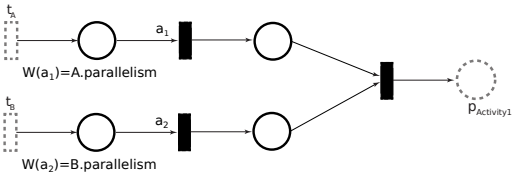
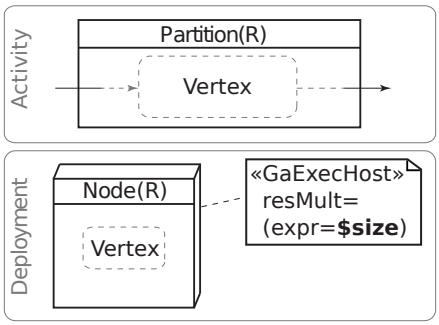
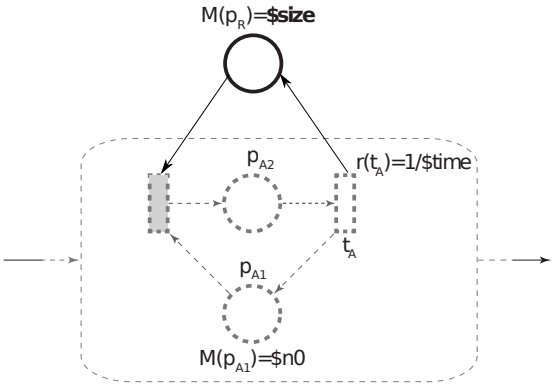
	Patrón UML	Patrón de red de Petri
P1		
P2		
P3		
P4		
P5		
P6		
P7		

Tabla 4.1: Patrones de transformación para diagramas de actividad de Tez perfilados

4.2. Transformación del diagrama de despliegue

Al igual que los elementos que forman los diagramas de actividad, también existe un patrón para uno de los conceptos que se encuentra representando en el diagrama de despliegue. Se trata del concepto de los recursos disponibles, del número de núcleos disponibles para que se ejecute la aplicación. Este concepto queda retratado mediante la etiqueta *resMul*, como puede observarse en el patrón *P7* de la Tabla 4.1.

El patrón de transformación queda mapeado en la GSPN como un lugar más en la red, del cual tendrán que consumir *tokens* las transiciones inmediatas previas a la ejecución de un *Vertex*, como se ha visto en el patrón *P4*. El lugar que se añade comenzará con un número de *tokens* igual al número de cores disponibles en la máquina donde se vaya a ejecutar la aplicación. De esta forma, la adición de este lugar permite representar las restricciones físicas que se imponen a la concurrencia lógica de la aplicación, es decir, el número de tareas en cada *Vertex* de Tez que pueden realmente ejecutarse de forma paralela.

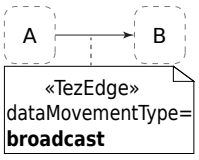
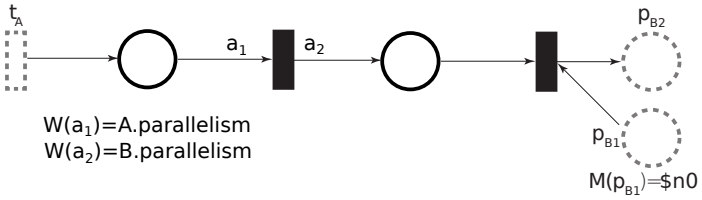
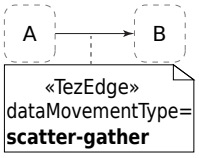
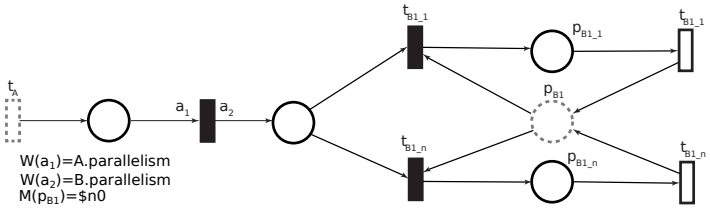
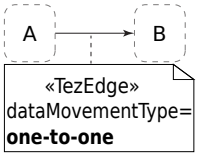
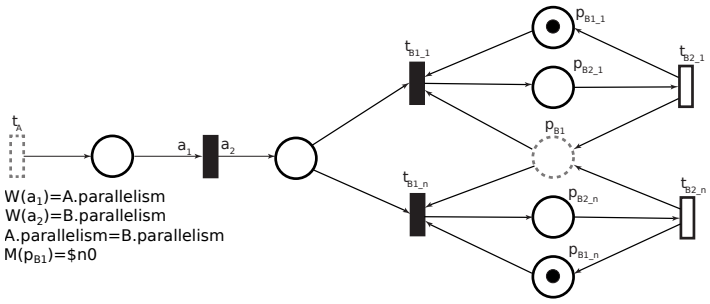
	Patrón UML	Patrón de red de Petri
P8		
P9		
P10		

Tabla 4.2: Patrones de transformación para diagramas de actividad de Tez perfilados II

4.3. Modelo de rendimiento e implementación

Una vez presentados todos los patrones de transformación necesarios para poder implementar una aplicación Tez cualquiera mediante las redes de Petri, se presenta en la Figura 4.1

el modelo final del ejemplo presentado en las Figuras 3.1 y 3.2. Dicho modelo se ha obtenido aplicando y combinando los patrones propuestos, y se ha simplificado para que resulte más sencillo de leer y comprender.

Como se puede observar, los *Vertices* 1, 2, 3, 4 y 5 tiene paralelismo $n0, $n1, $n2, $n3 y $n4, respectivamente. Los parámetros $a0 y $a2, que representan los *tokens* que se deberán consumir al finalizar los *Vertices* 1 y 3, tendrán valores iguales a los parámetros $n0 y $n2. Por otro lado, los parámetros $a1 y $a3 contarán con los mismos valores que $n1 y $n3, representando los *tokens* que se han de generar para la ejecución de los *Vertices* 2 y 4. Finalmente, el parámetro $a4 contará con el valor correspondiente a la suma de los parámetros $n1 y $n3 (o $a1 y $a3), y el $a5 con el mismo valor que el $n4.

Además, tal y como queda representado, los *Vertices* 1 y 2 se encuentran conectados mediante un tipo de movimiento de datos *one-to-one*, los *Vertices* 3 y 4 mediante un *scatter-gather* y, finalmente, los *Vertices* 2 y 4 convergen en el *Vertex* 5 mediante un tipo *broadcast*.

En el caso de este trabajo, el modelo mostrado en la Figura 4.1 se ha creado manualmente mediante la combinación de los patrones ya detallados. Como se mencionará más en detalle en el posible trabajo futuro en base a este proyecto, la generación de las redes de Petri podrá automatizarse y será transparente para el usuario final.

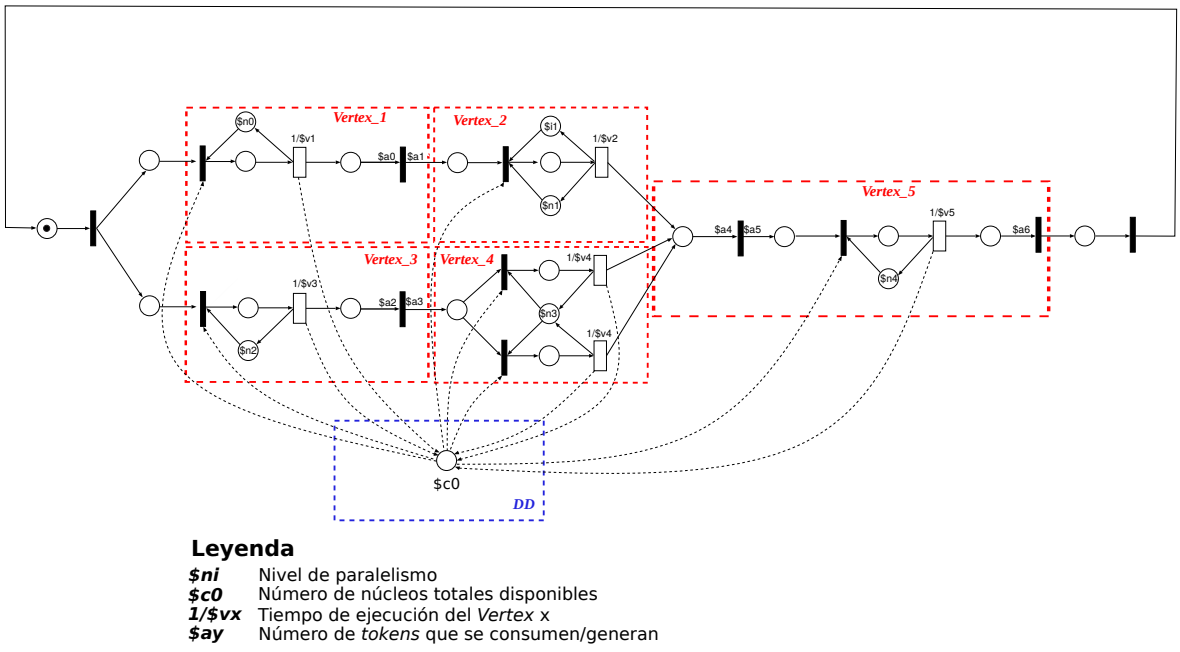


Figura 4.1: GSPN del modelo de las Figuras 3.1 y 3.2

Capítulo 5

Validación del modelo de rendimiento

Una vez presentados los patrones de transformación del modelo UML al modelo de análisis de rendimiento con redes de Petri, es necesaria una validación de dichas transformaciones para verificar su correcto funcionamiento. Es decir, se quiere comprobar que los patrones que representan distintos conceptos de la tecnología de Tez los están representando correctamente.

Para ello, se ha realizado un conjunto de experimentos sobre la GSPN de la Figura 4.1, cuyos resultados se presentan en la Tabla 5.1. Dichos resultados se han comparado con los obtenidos de las ejecuciones en un *cluster* de Tez correspondientes de la aplicación Tez que se representa en la GSPN, también presentados en la Tabla 5.1.

La ejecución de las simulaciones con las GSPNs se ha realizado mediante la herramienta GreatSPN [6], un simulador dirigido por eventos, en la cual se ha fijado unos parámetros de confianza del 99 % y una precisión de 3 % para todas las pruebas. La herramienta incluye, además, extensiones para el análisis estructural con las que extraer propiedades intrínsecas de la red de Peri. Mediante la interpretación de los resultados de la simulación guiada por eventos en la GSPN, se pueden calcular las siguientes métricas:

- Utilización del hardware (en el único lugar del patrón *P7* de la Tabla 4.1), calculada como:

$$\frac{\text{Número total de núcleros disponibles} - \text{Número medio de núcleos no utilizados}}{\text{Número total de núcleros disponibles}} \tag{5.1}$$

- Productividad, calculada como: Tiempo medio de la transición inmediata final de la GSPN (la última transición inmediata del patrón *P6*)

- Tiempo de respuesta de la aplicación, calculada como:

$$\frac{1}{\text{Tiempo medio de la transición inmediata final de la GSPN}} \tag{5.2}$$

Por otro lado, la ejecución de la aplicación Tez para obtener los tiempos de respuesta reales se han realizado sobre una máquina con 8 núcleos Intel Core i7-6700 a 3,4GHz, 31GB de memoria RAM, conexión Gigabit ethernet y un sistema operativo Ubuntu Linux en su versión 16.10.

Debido a ciertas limitaciones impuestas por la tecnología de Tez, ha resultado imposible determinar con certeza el entrelazado de las tareas de cada uno de los *Vertices* durante la ejecución de una aplicación, es decir, el orden y el momento en el que se ejecutan; por consiguiente, no ha sido posible replicar con precisión los resultados en aplicaciones con *Vertices* con paralelismo mayor que uno. Debido a este contratiempo, las pruebas realizadas cuyos resultados se presentan en la Tabla 5.1 se han ejecutado con paralelismo igual a uno en sus 5 *Vertices*.

De igual manera, el entrelazado entre los propios *Vertices* también ha sido imposible de determinar, motivo por el cual los tiempos de respuesta presentados en esta sección cuentan con un error relativo de entre el 10 y el 24 %, dado que la GSPN ejecutada cuenta con dos caminos paralelos.

Centrando finalmente la atención en la Tabla 5.1, con los resultados de los experimentos, se pueden encontrar las siguientes columnas:

- Entrada 1 y Entrada 2. En estas dos columnas se especifica el tamaño de los ficheros de entrada para cada una de las ejecuciones. Al contar con dos flujos que se ejecutan en concurrente, se ha introducido un fichero distinto a cada uno. Dado que ambos flujos no cuentan con el mismo tiempo de ejecución, los distintos ficheros se han ido permutando entre las dos entradas.

- Tiempo de respuesta de Tez. En esta columna se indica el tiempo de respuesta en mili segundos de la ejecución de la aplicación Tez con las entradas indicadas.

- Tiempo de respuesta del simulador. En esta columna se indica el tiempo de respuesta en mili segundos correspondiente a la ejecución de la red de Petri en el simulador con las entradas indicadas.

- Error Relativo. En esta columna se indica el error relativo existente entre los dos tiempos de respuesta obtenidos, el real y el simulado.

- Utilización. En esta columna se indica el porcentaje de utilización de los recursos (núcleos) de la aplicación completa durante su simulación en GreatSPN.

Como se puede observar en los resultados, dado que contamos con *Vertices* de paralelismo igual a uno, la utilización resultante es muy baja, dado que del total de 8 núcleos disponibles, apenas se usan 2 de forma continua a lo largo de toda la ejecución.

Entrada 1	Entrada 2	T. Respuesta Tez (ms)	T. Respuesta Simulador (ms)	Error Relativo (%)	Utilización (%)
700KB	50KB	5990	4711	21,35	15,87
6MB	50KB	6580	4996	24,07	16,43
6MB	700KB	8360	6466	22,65	16,14
50KB	60MB	11700	10324	11,76	14,08
60MB	700KB	11830	9416	20,4	15,7
60MB	6MB	13710	10508	23,35	16,29
50KB	130MB	13490	12050	10,67	13,93
700KB	130MB	14150	12230	13,56	14,3
130MB	6MB	15420	12026	22	16,26
130MB	60MB	18290	13974	23,6	16,49

Tabla 5.1: Resultados de los expermientos

Por otro lado, como ya se ha mencionado, el error relativo de los tiempos de respuesta obtenidos oscila entre el 10 y el 24 %, en función de los ficheros de entrada, principalmente. Dado que no se conoce el entrelazado de los *Vertices*, la red de Petri ejecuta en paralelo ambos hilos, por lo que el tiempo final obtenido de la simulación siempre es más bajo que el obtenido por Tez. En los casos con ficheros de tamaños muy distintos, como por ejemplo 130MB y 50KB, el camino asociado al fichero de 50KB apenas afecta a la ejecución del otro hilo, dado que tiene que esperar a que finalice antes de entrar en el *Vertex* final, por lo que el tiempo es mucho más ajustado.

No obstante, los tiempos de respuesta parciales obtenidos por la simulación de la red de Petri parcial para cada uno de los *Vertices* son muy similares a los correspondientes en la ejecución real, por lo que una vez se consiga con precisión el entrelazado de los *Vertices*, se podrá ajustar la red de Petri para obtener resultados mucho más exactos. Los resultados individuales tanto de tiempo de ejecución como de utilización de los *Vertices* de forma individual no se han incluido debido al gran tamaño de estos y a la poca información que aportaban respecto a la Tabla 5.1.

Debido a los contratiempos encontrados con respecto a la tecnología de Tez, los resultados no son tan positivos como se esperaba al no poder caracterizar el entrelazado en ejecuciones con paralelismo mayor que uno. Sin embargo, es posible que tras una investigación todavía más profunda de la herramienta, o si ésta avanza durante los próximos años en cuanto a las estadísticas que se ofrecen, se puedan obtener estos datos y, de esta forma, adaptar la red de Petri para obtener los resultados que se esperan. El cómo adaptar estas redes de Petri se comentará en la sección de trabajo futuro en las conclusiones de este documento.

Capítulo 6

Conclusiones

En este trabajo se ha presentado un nuevo enfoque para el modelado y el análisis del rendimiento para aplicaciones desarrolladas con Apache Tez. El objetivo de la investigación y el desarrollo que se han llevado a cabo concuerdan con los del proyecto europeo DICE en el que se encuentra enmarcado: guiar a ingenieros del software a mejorar la calidad de los sistemas desarrollados durante la fase de diseño. Para este fin, se ofrece una herramienta capaz de simular y predecir el comportamiento de una aplicación desarrollada en Apache Tez. En particular, la aproximación presentada en este trabajo nos permite calcular métricas de rendimiento como: los tiempos de respuesta, la productividad y la utilización de una determinada aplicación en una máquina o un *cluster*.

El mencionado enfoque que se presenta es sin duda único en muchos aspectos: se ha creado un nuevo perfil de UML para la tecnología Tez, el cual captura los conceptos necesarios para la evaluación del rendimiento; se han propuesto patrones de transformación para obtener un modelo de rendimiento (red de Petri) a partir de UML y, finalmente, se han empleado GSPNs para el análisis de prestaciones.

6.1. Trabajo futuro

Como ya se ha expuesto en el Capítulo 5, la validación de los patrones de transformación a GSPNs propuestos se han realizado fijando el paralelismo de todos los *Vertices* a uno. Esto, como se ha explicado, se debe a la imposibilidad que ha supuesto extraer los tiempos de ejecución de las tareas paralelismo mayor que uno, y del entrelazamiento entre los propios *Vertices* en una aplicación con varios caminos de ejecución paralelos.

Por este motivo, el trabajo futuro debería centrarse en primer lugar en hallar una forma de obtener con más precisión el entrelazamiento de las tareas de un *Vertex* para, así, poder ajustar la GSPN con los resultados obtenidos y conseguir tiempos de respuesta correctos en ejemplos con paralelismos mayor que uno. Si se consiguen hallar dichos entrelazamientos, el siguiente paso sería adaptar la red de Petri aplicándole a las transiciones temporales una distribución probabilística que permitiese representarlos correctamente.

Por defecto, una GSPN aplica una distribución exponencial, la cual, debido a los porcentajes de error mostrados en la Tabla 5.1, en el caso que se ha planteado aquí es insuficiente y resulta inexacta para los resultados que se quieren obtener cuando existe paralelismo mayor que uno. Por tanto, se plantea la posibilidad de aplicar en su lugar otro tipo de distribuciones, como podría ser la distribución Erlang.

Una vez se haya conseguido discernir qué distribución es la correcta y se consigan los resultados esperados, los dos siguientes y últimos pasos a realizar serían: (i) la automatización de la transformación del modelo UML al de redes de Petri, que únicamente consistiría en introducir los patrones de transformación en el plugin QVT para Eclipse, donde existe la integración para redes de Petri de otras tecnologías; y (ii) la integración de este simulador en la herramienta de simulación del proyecto DICE, donde ya se encuentran simuladores para otras tecnologías como Storm, Spark o Map-Reduce.

Finalmente, podrían plantearse otros acercamientos en lo referente al modelo de análisis de rendimiento para el simulador. En este caso, se han empleado las redes de Petri; sin embargo, existen otros modelos que tal vez podrían servir al mismo propósito, como podría ser un modelo de redes de colas u otros similares.

6.2. Conclusiones personales

La realización de este proyecto ha supuesto no sólo un gran esfuerzo sino también un fantástico proceso de aprendizaje y evolución personal. Como ya se menciona en la introducción del documento, el *Big Data* se encuentra en auge y es cada vez más importante en gran cantidad de sectores empresariales. Haber tenido la oportunidad de participar en un proyecto internacional como este, investigando una de las herramientas que se utiliza y se utilizará todavía más en unos años, ha sido una gran experiencia y me ha permitido introducirme en esta área de la informática que no había tocado a lo largo de la carrera. Además, es una gran satisfacción saber que todo este trabajo se empleará más adelante y podrán utilizarlo otros colegas que puedan necesitarlo para su trabajo.

Por otra parte, realizar un proyecto más grande e importante que cualquier otro realizado en el grado, y hacerlo solo (contando, por supuesto, con el apoyo de mis directores), ha supuesto una evolución personal que me será tremendamente útil en el futuro. Un proyecto así obliga a uno mismo a continuar, a seguir avanzando y a llegar hasta el final, porque sabes que si no lo haces tú, nadie más lo va a hacer por ti. Y esto, al fin y al cabo, es una realidad que se va a cumplir en el mundo laboral.

Finalmente, quisiera agradecer a mis dos directores, Nacho y José, por toda su inestimable ayuda y paciencia a lo largo de la realización de este trabajo y por guiarme hasta buen puerto.

Capítulo 7

Bibliografía

[1] DICE H2020 Website. http://www.dice-h2020.eu/.

[2] Apache Tez Website. https://tez.apache.org/, .

[3] Apache Tez API. https://tez.apache.org/releases/0.8.4/tez-api-javadocs/, .

[4] G. Balbo and M. Silva. *Performance Models for Discrete Event Systems with Synchronisations: Formalisms and Analysis Techniques*, volume 1. MATCH, first edition, 1998.

[5] G. Chiola, M. A. Marsan, G. Balbo, and G. Conte. Generalized Stochastic Petri nets: A Definition at the Net Level and its Implications. *IEEE Transactions on Software Engineering*, 19(2):89–107, 1993.

[6] Dipartamento di informatica, Universitá di Torino. GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets, Dec., 2015. URL: http://www.di.unito.it/~greatspn/index.html.

[7] OMG. UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems, Version 1.1. URL: http://www.omg.org/spec/MARTE/1.1/, Juny 2011.

[8] J. I. Requeno, J. Merseguer, and A. Malo. Performance Analysis of Apache Spark Applications using Stochastic Petri Nets. Internal report Unizar. Not yet published.

[9] J. I. Requeno, J. Merseguer, and S. Bernardi. Performance analysis of apache storm applications using stochastic petri nets. *Proceedings of the 5th IEEE International Workshop on Formal Methods Integration*, 2017.

[10] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, pages 1357–1369. ACM, 2015.

[11] Universidad de Zaragoza. DICE Simulation, August, 2017. URL: https://github.com/dice-project/DICE-Simulation.

[12] Universidad de Zaragoza. Tez profile, August, 2017. URL: https://github.com/dice-project/DICE-Profiles/tree/dtsm-tez.

Anexos

Anexo A

Planificación

En este anexo se expone en detalle la planificación seguida en el desarrollo de este trabajo. Como puede observarse en la Figura A.1, el proyecto dio comienzo en noviembre de 2016, en el cual se comenzó a analizar la herramienta mediante un primer acercamiento a la misma.

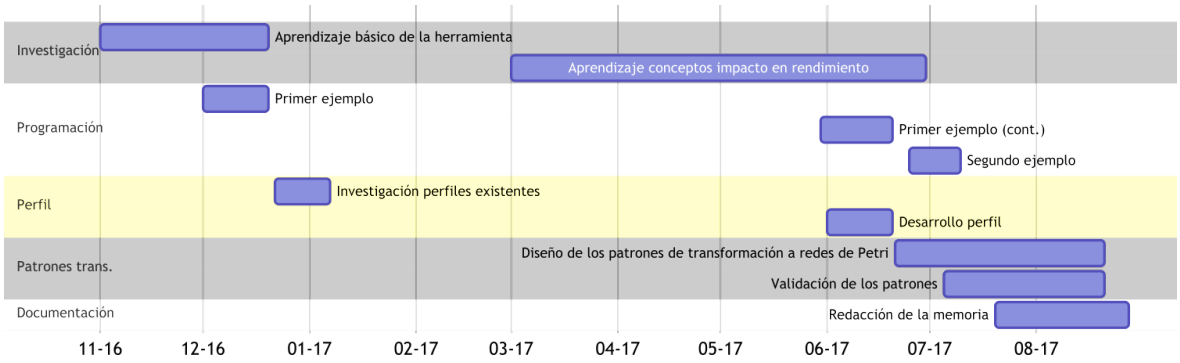


Figura A.1: Diagrama de Gantt del proyecto

Tal y como queda reflejado en el diagrama, el proyecto ha avanzado más lentamente a lo largo del curso lectivo y el impulso final se ha realizado durante los tres últimos meses de la duración total del mismo. No obstante, se puede observar tanto en el diagrama de Gantt como en la Tabla A.1, que muestra un desglose en horas del proyecto, que la tarea que más horas ha requerido y que se ha realizado a lo largo del curso ha sido la de la investigación de la herramienta. Dicha investigación era necesaria para proseguir con el resto de tareas del proyecto, razón por la cual se han pospuesto a los últimos meses.

Apartado	Tarea	Horas
Investigación	Aprendizaje básico de la herramienta	38
	Investigación conceptos rendimiento	81
Programación	Primer ejemplo de aplicación Tez	55
	Segundo ejemplo de aplicación Tez	17
Diseño perfil	Investigación perfiles existentes	9
	Diseño y desarrollo perfil Tez	17
Diseño patrones transformación	Diseño patrones de transformación a RdP	27
	Validación de los patrones	20
Documentación	Redacción de la memoria	60
	Total	324

Tabla A.1: Desglose por horas invertidas en cada tarea del proyecto

Anexo B

Redes de Petri Estocásticas Generalizadas

Una GSPN (*Generalized Stochastic Petri Net*) es una Red de Petri con una interpretación estocástica del tiempo, por lo que resulta idónea para realizar análisis de rendimiento.

Un modelo de GSPN se representa como un grafo bipartito con dos tipos de vértices: los lugares y las transiciones. Los lugares se representan en el grafo como círculos, los cuales pueden contener *tokens*. La distribución de *tokens* en los distintos lugares de un grafo representa el estado del sistema modelado en un momento determinado.

El movimiento de dichos *tokens* entre los lugares se realiza mediante las transiciones, las cuales cuentan con reglas de activación y disparo y donde los lugares anteriores y posteriores a una transición representan pre y post condiciones. Concretamente, cuando una transición se dispara, consume (genera) tantos *tokens* del lugar de origen (destino) como se indica en el peso de los arcos que une los lugares con la transición. Las transiciones, además, pueden ser de dos tipos: inmediatas, las cuales se disparan en tiempo cero; y temporales, las cuales se disparan tras un tiempo establecido.

En el modelo de rendimiento de Tez presentado en este trabajo, los lugares en las GSPNs representan pasos intermedios en el flujo de procesamiento de datos. Por su parte, las transiciones representan la ejecución de las tareas de tez (los *Vertices*) y se ejecutan cuando se cumplen ciertas condiciones o cuando ha pasado el tiempo establecido. Finalmente, los *tokens* representan conjuntos de mensajes que siguen el flujo de ejecución, el paralelismo de un *Vertex* o los recursos disponibles en el sistema donde se está ejecutando la aplicación.