Universidad Zaragoza

Proyecto Fin de Carrera

Evaluación del rendimiento de aplicaciones intensivas en datos con Apache Spark

Autor/es

Andrés Malo Ascaso

Director/es

José Ignacio Requeno Jarabo
José Javier Merseguer Hernáiz

Escuela de Ingeniería y Arquitectura / Universidad de Zaragoza

2017

Resumen

El objetivo del proyecto consiste en estimar los recursos y tiempos que consumirá una aplicación intensiva en datos desarrollada en la tecnología Apache Spark. Para ello, diseñaremos e implementaremos una herramienta informática que permita obtener automáticamente predicciones del rendimiento de una aplicación Spark a partir de la definición de su comportamiento mediante un modelo en UML. Los pasos a seguir se dividen en 4 fases.

Primeramente, vamos a realizar una serie de pruebas exhaustivas sobre la plataforma Apache Spark. Estas pruebas nos permitirán conocer en detalle la tecnología y sus características. En particular nos interesa conocer en detalle los parámetros más importantes de la tecnología que influyen en el rendimiento de las aplicaciones intensivas en datos. Toda esta información sera utilizada para poder aplicar el siguiente apartado.

En segundo lugar, vamos a desarrollar un lenguaje de dominio específico (DSML, Domain Specific Modeling Language) para Apache Spark, aprovechando lo aprendido en el paso anterior. En concreto, este lenguaje es una extensión del lenguaje de modelado unificado (UML), aplicando la técnica de perfiles (profiles). Los perfiles UML nos permiten particularizar los conceptos de la tecnología sin tener que re definir el resto del lenguaje. Adicionalmente, modelaremos diferentes aplicaciones intensivas en datos utilizando el lenguaje propuesto.

En tercer lugar, propondremos una transformación de dichos diseños UML en un modelo formal para la evaluación del rendimiento de una aplicación desarrollada en Apache Spark; en particular, en redes de Petri. Con ello procederé a automatizar la creación de redes de Petri, mediante el lenguaje QVTo (Operational Query/View/Transformation). Estas redes de Petri se analizarán para obtener resultados de prestaciones.

En cuarto lugar, realizaremos una validación de los resultados obtenidos con las redes de Petri. En el caso de que las predicciones de la red de Petri no se aproximen con los resultados obtenidos por la ejecución real de la aplicación Spark, actualizaremos el perfil de UML y/o la transformación a redes de Petri, repitiendo el proceso hasta aproximar lo máximo posible los resultados a los valores reales.

Índice

Glosario

- Calculadora: Programa que permite conocer resultados de una simulación.
- Framework: Conjunto de conceptos, prácticas y criterios para enfocar un tipo de problemática particular.
- Grafo acíclico dirigido (DAG): Grafo finito dirigido sin ciclos
- Lenguaje de dominio específico (DSML, Domain Specific Modeling Language): Lenguaje dedicado a resolver un problema en particular.
- Lenguaje QVTo (Operational Query/View/Transformation): Estándar para realizar transformaciones de modelo a modelo.
- Perfil UML: Mecanismo de UML para extender las sintaxis y semántica propias de UML para expresar conceptos específicos de un dominio particular.
- Redes de Petri: Representación matemática o gráfica de un sistema a eventos discretos en el cual se puede describir la topología de un sistema distribuido, paralelo o concurrente.
- Transiciones temporizadas: Tipo de transición que se dispara después de un tiempo de habilitación aleatorio exponencialmente distribuido.
- UML: Lenguaje de modelado gráfico para visualizar, especificar, construir y documentar un sistema.

Introducción y objetivos

A día de hoy, casi cualquier sistema industrial lidia con ingentes cantidades de datos; y la tendencia es a tener que tratar aun más. Es por ello que las tecnologías preparadas para poder trabajar con grandes cantidades de información de manera eficiente, conocidas como Big Data, están en auge y muchos sistemas dependen del correcto funcionamiento de estas. Apache Spark es una tecnología que ha destacado últimamente por ofrecer buenas prestaciones en el manejo de grandes volúmenes de datos. Además, incluye múltiples librerías y herramientas auxiliares que le aportan mayor versatilidad y facilidad con respecto a otras tecnologías existentes (p.ej. Apache Hadoop MapReduce). Es por ello que este proyecto consiste en analizar dicha tecnología.

Durante la fase de análisis y diseño de una nueva aplicación software, los requisitos funcionales se suelen anteponer a los requisitos temporales y de prestaciones. Sin embargo, las decisiones de diseño tomadas en este punto tienen un gran impacto en el rendimiento final del sistema. Por ello, es importante predecir el comportamiento de la aplicación software en las fases iniciales del proyecto; antes de las fases de implementación y despliegue. Con este proyecto se pretende ofrecer la posibilidad de estimar el comportamiento de un sistema hipotético bajo ciertas condiciones para ayudar a hallar condiciones óptimas sin tener que usar recursos reales en averiguarlo. Con ello sería posible hallar y prevenir prematuramente ciertos riesgos como cuellos de botella en el rendimiento o minimizar el desperdicio de recursos en la medida de lo posible. Incluso es posible conocer cómo puede llegar a escalar respecto a los recursos y volumen de datos tratados.

El objetivo del proyecto consiste en asesorar al desarrollador/arquitecto software en cuestiones relativas al rendimiento de aplicaciones implementadas con la tecnología Apache Spark. En particular, queremos estimar los recursos y tiempos que consumirá una aplicación intensiva en datos desarrollada en dicha tecnología. Para ello, diseñaremos e implementaremos una herramienta informática que permita obtener automáticamente predicciones del rendimiento de una aplicación Spark a partir de la definición de su comportamiento mediante un modelo en UML. Los pasos a seguir se dividen en 4 fases.

Primeramente, vamos a realizar una serie de pruebas exhaustivas sobre la plataforma Apache Spark. Estas pruebas nos permitirán conocer en detalle la tecnología y sus características. En particular nos interesa conocer en detalle los parámetros más importantes de la tecnología que influyen en el rendimiento de las aplicaciones intensivas en datos. Toda esta información sera utilizada para poder aplicar el siguiente apartado.

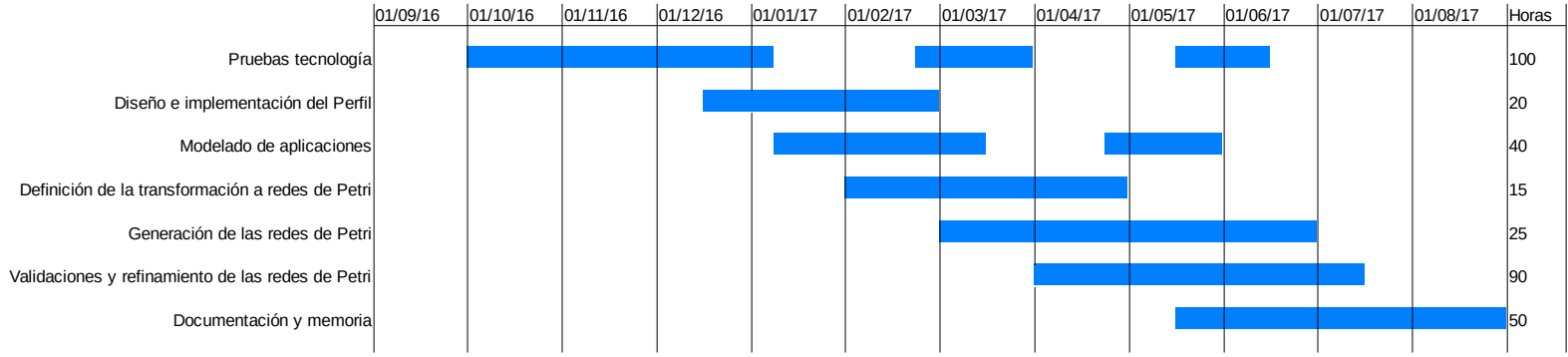
En segundo lugar, vamos a desarrollar un lenguaje de dominio específico (DSML, Domain Specific Modeling Language) para Apache Spark, aprovechando lo aprendido en el paso anterior. En concreto, este lenguaje es una extensión del lenguaje de modelado unificado (UML), aplicando la técnica de perfiles (profiles). Los perfiles UML nos permiten particularizar los conceptos de la tecnología sin tener que re definir el resto del lenguaje. Adicionalmente, modelaremos diferentes aplicaciones intensivas en datos utilizando el lenguaje propuesto.

En tercer lugar, propondremos una transformación de dichos diseños UML en un modelo formal para la evaluación del rendimiento de una aplicación desarrollada en Apache Spark; en particular, en redes de Petri. Con ello procederé a automatizar la creación de redes de Petri, mediante el lenguaje QVTo (Operational Query/View/Transformation). Estas redes de Petri se analizarán para obtener resultados de prestaciones.

En cuarto lugar, realizaremos una validación de los resultados obtenidos con las redes de Petri. En el caso de que las predicciones de la red de Petri no se aproximen con los resultados obtenidos por la ejecución real de la aplicación Spark, actualizaremos el perfil de UML y/o la transformación a redes de Petri, repitiendo el proceso hasta aproximar lo máximo posible los resultados a los valores reales.

Este documento consta de 5 apartados que me permitirán explicar el trabajo desarrollado así como todos las diferentes fases realizadas a lo largo del proyecto. En el primer apartado explico DICE Simulator, la herramienta sobre la que se basa este proyecto. En esta herramienta integro el perfil de Spark para UML, la transformación automática de modelos UML perfilados en redes de Petri, y las calculadoras para la obtención de las métricas de prestaciones. En el siguiente apartado explico la tecnología Spark y un resumen de dicho análisis. En el tercer apartado, presento el perfil de Spark que he diseñado para UML y una guía de modelado de las aplicaciones de Spark utilizando este lenguaje. Esto lleva al siguiente apartado, que consiste en la definición y la implementación de la transformación automática de los modelos UML en redes de Petri. En el último apartado explicaré cómo comprobamos los resultados obtenidos y comparándolos con datos reales, los utilizamos para optimizar las simulaciones.

Fases del proyecto



En este diagrama se puede observar la evolución de las diversas fase a lo largo del proyecto. Si bien el proyecto se ha prolongado a lo largo de los meses, este proyecto se llevó a cabo a la vez que estaba trabajando, por lo que la disponibilidad diaria era bastante limitada. Entre semana, la disponibilidad se limitaba a unas 4 horas diarias.

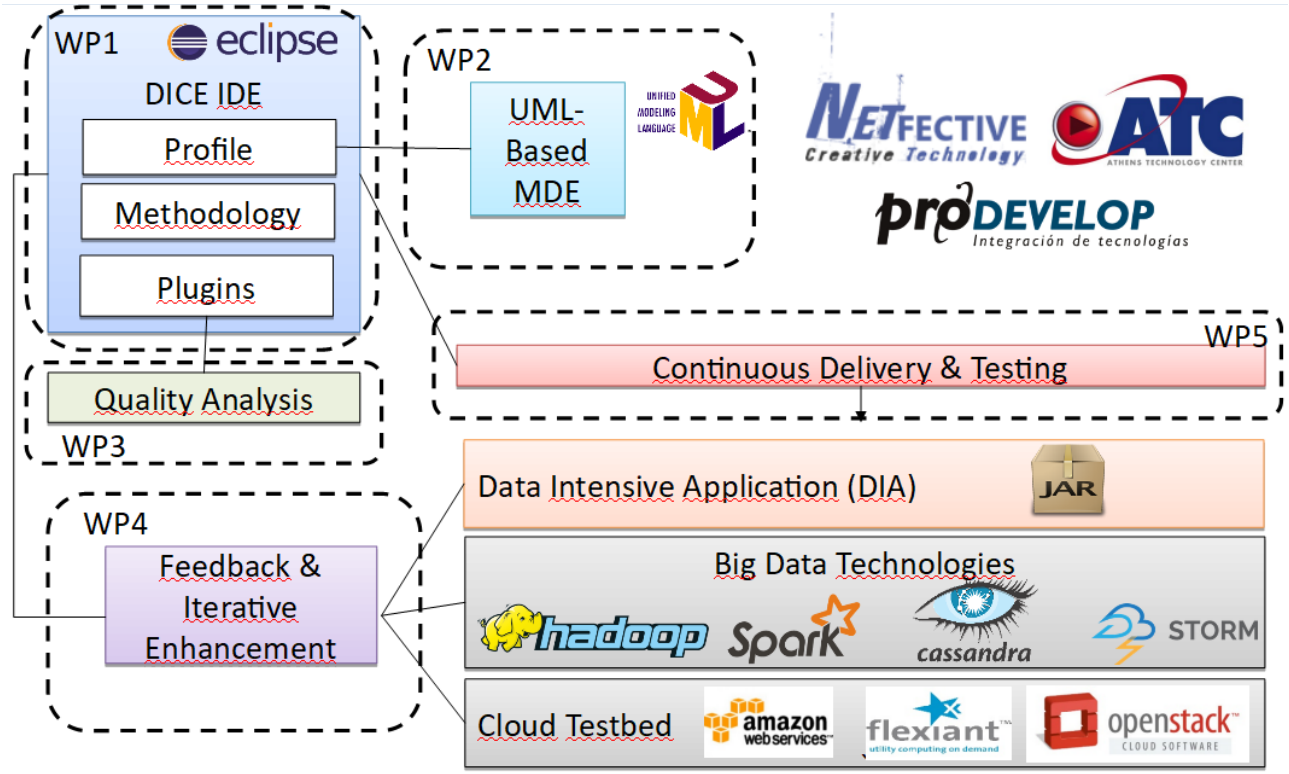
En este diagrama hay que destacar la cantidad de tareas que aparecen en paralelo, y la longitud de las tareas. Las tareas aparecen muchas veces en paralelo debido a sus interdependencias, y que muchas veces era necesario algún punto de una tarea para continuar en otras fases. La longitud de las tareas se debe a que las tareas no se cerraban completamente, si no que normalmente requerían pequeños cambios conforme se iba progresando en el desarrollo del proyecto.

A lo largo del proyecto, ha habido varias tareas que resultaron más complicadas y completarlas llevó más tiempo del previsto. El uso de la herramienta DICE Simulator al principio resultó complicada, debido a la cantidad de detalles ínfimos, pero fundamentales para el correcto funcionamiento de esta. Por otro lado, trabajar con los logs de Spark pertenecientes a múltiples simulaciones resultó tedioso y complicado por tener que ir buscando la información específica necesaria a lo largo de los logs. Por fortuna, al ir avanzando en esta tarea se pudo encontrar ciertos patrones que permitieron agilizar esta tarea. Además, el equipo $DISCO_5$ colaboró plenamente en todos estos problemas y me ayudó a poder trabajar con la herramienta DICE Simulator.

Otro punto que resultó especialmente complicado fue la fase de validación de las transformaciones. Al encontrar que los valores simulados no eran demasiado acertados, encontrar la solución a estas desviaciones requirió realizar diferentes aproximaciones. En este punto fue necesario profundizar en el funcionamiento de la herramienta para poder hallar los detalles necesarios. De nuevo, la colaboración con el equipo fue fundamental.

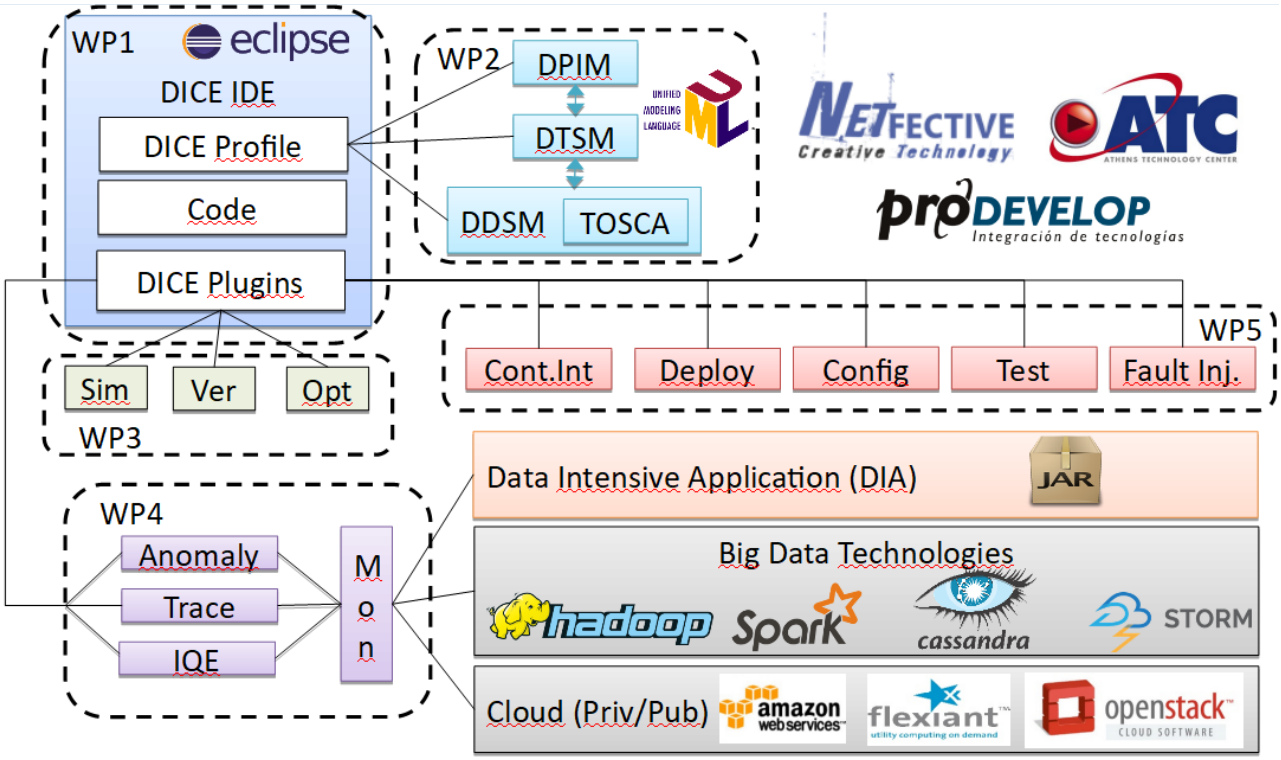
Presentación de DICE

DICE[2] es un proyecto de investigación que está llevándose a cabo desde febrero de 2015. Varios equipos de toda Europa están participando en este proyecto como parte del Horizon 2020. Por parte de la universidad de Zaragoza, el equipo DISCO está colaborando en dicho proyecto.

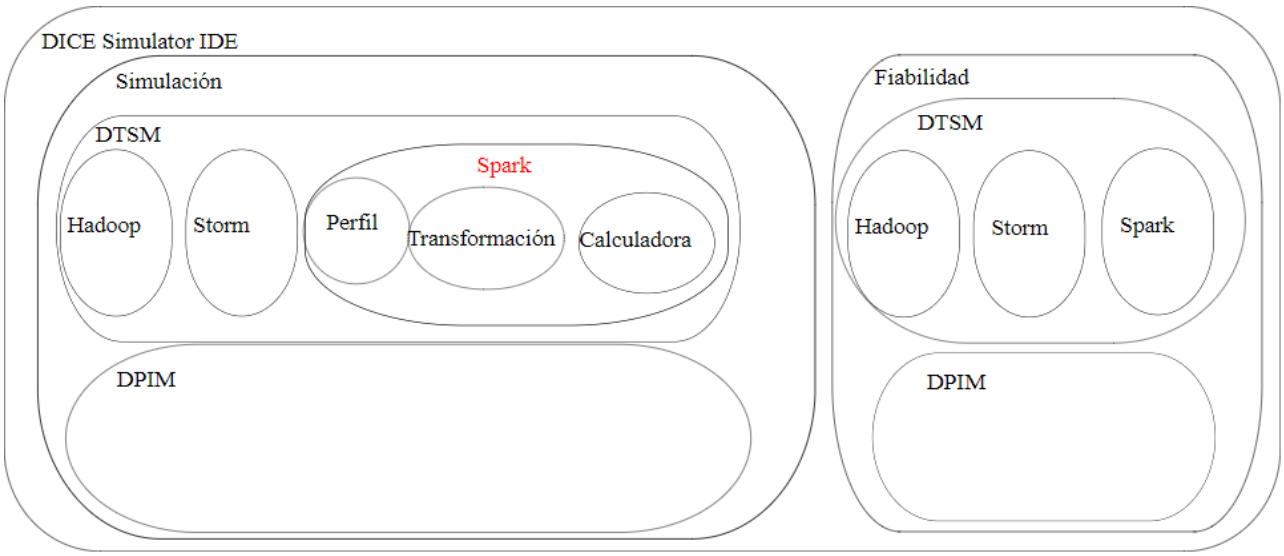


El objetivo de DICE es ofrecer perfiles UML[15] y herramientas que ayudaran a los diseñadores de software a tratar con la fiabilidad, seguridad y eficiencia de las aplicaciones intensivas en datos. La aplicación permite realizar simulación de prestaciones y simulación de fiabilidad/tolerancia a fallos de un sistema. Además, esta herramienta permite que los modelos de las aplicaciones puedan ser usados en ambos casos de manera que sean independientes de la tecnología o asociados a una tecnología concreta. Mas detalles de esta herramienta pueden ser encontrados en los siguientes links.

http://www.dice-h2020.eu/
https://github.com/dice-project/DICE-Simulation
https://github.com/dice-project/DICE-Profiles



Mi proyecto fin de carrera se enmarca como una colaboración dentro del equipo DISCO de la universidad de Zaragoza. Mi trabajo realizado se incluye dentro de la herramienta llamada DICE Simulator (véase WP3), que permite simular aplicaciones de tecnologías Big Data. Esta herramienta ya permitía simular otras tecnologías previamente, como Apache Hadoop o Apache Storm. El propósito del proyecto consiste en agregar la tecnología Apache Spark a la herramienta DICE Simulator. En la figura podemos observar los principales componentes de la herramienta. La parte coloreada muestra las extensiones desarrolladas durante mi proyecto.



La herramienta se divide en dos, dependiendo de las funcionalidades que ofrece cada apartado. Por una parte, la herramienta es capaz de hallar la fiabilidad de un sistema (por ejemplo, tiempo de respuesta entre fallos, probabilidad de caídas, etc,..). La otra funcionalidad consiste en simular

aplicaciones. Dentro de ambos apartados, la herramienta permite trabajar a dos niveles de abstracción diferentes. Por un lado, se puede trabajar con tecnologías específicas (nivel DTSM, acrónimo de DICE Platform And Technology Specific Model). La otra posibilidad es trabajar con independencia de una tecnología específica (nivel DPIM, acrónimo de DICE Platform Independent Model).

Dentro del nivel DTSM es donde he desarrollado el proyecto, incluyendo extensiones para Apache Spark. Dichas extensiones incluyen 1) el diseño de un perfil específico de Spark para UML, 2) la definición e implementación de una transformación automática de modelos perfilados en UML hacia modelos adecuados para la evaluación de prestaciones (en especial, redes de Petri); y 3) la implementación de una serie de calculadoras que extraen métricas de rendimiento a partir de la evaluación de los modelos de prestaciones.

Al iniciar este proyecto, buscamos trabajos previos similares que sirvieran de base para el proyecto. Hemos buscado perfiles UML y simuladores específicos de Spark que permitieran predecir el comportamiento de esta tecnología. Sin embargo, los perfiles y herramientas actuales son demasiado genéricos (enfocado a Big Data en general). Por tanto, la aproximación presentadas en este proyecto es completamente genuina y novedosa.

Presentación de Spark

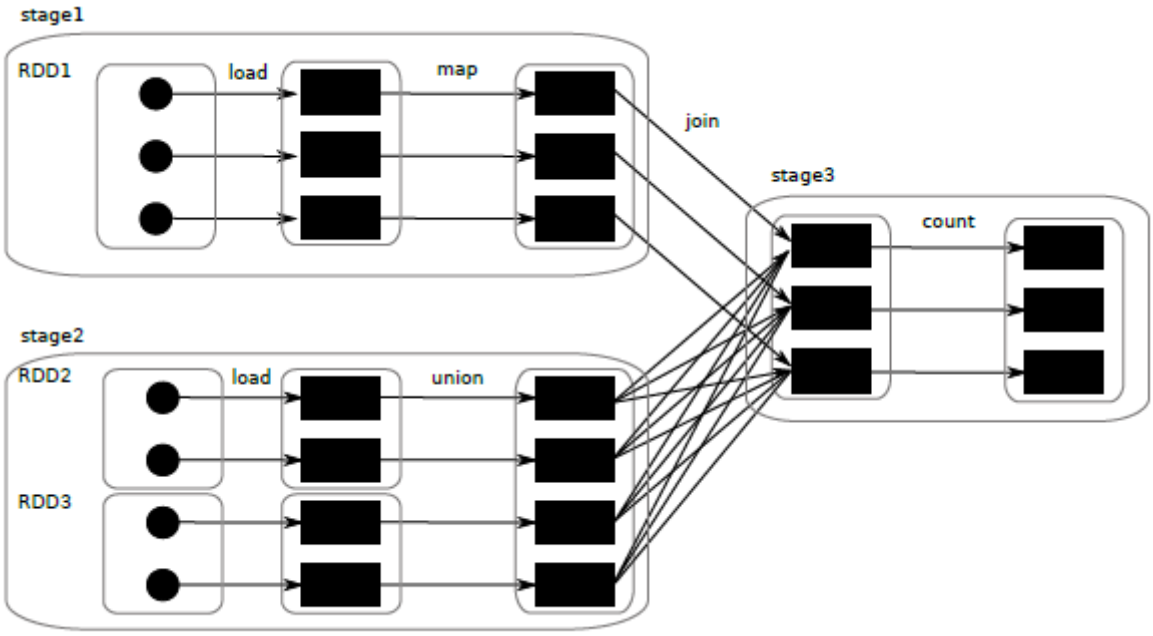
Apache Spark[1] es un tecnología en auge, utilizada por multitud de compañías, tales como Netflix, Yahoo!, Alibaba, Goldman Sachs o Toyota. Es capaz de procesar grandes cantidades de datos, así como transformar ficheros planos en estructuras de datos que pueden ser utilizadas para tratamiento de datos. Spark integra diferentes librerías que permiten el procesamiento datos estructurados (Spark SQL + DataFrames), análisis en tiempo real (Spark Streaming), aprendizaje automático (MLlib) o computación de grafos (GraphX). Por ello, las aplicaciones desarrolladas utilizando esta tecnología pueden tener unos requisitos elevados respecto al rendimiento y, al ser personalizables, ciertos parámetros tienen un gran impacto en este.

Apache Spark es un framework de computación distribuido tolerante a fallos que permite procesar grandes cantidades de información en entornos distribuidos. Está basado en Apache Hadoop MapReduce, que a su vez se basa en el paradigma de programación MapReduce. Spark soluciona o minimiza las principales limitaciones que han aparecido en tecnologías previas. Programar en Spark es mucho más sencillo e intuitivo que trabajar con Hadoop. Mientras que Hadoop se basa en maps y reduces planos, Spark introduce operaciones más complejas como join, filter y groupBy. Además Spark tiende a obtener mejor rendimiento que Hadoop MapReduce para aplicaciones similares. Spark es capaz de obtener mejores prestaciones al mantener en memoria los resultados intermedios de los cálculos mientras sea posible, en vez de guardar esta información periódicamente en disco como Hadoop MapReduce.

La unidad básica con la que trabaja Spark son los resilient distribute datasets (RDD), estructuras de datos distribuidas tolerantes a fallos que permiten repartir y almacenar el conjunto de datos original entre diversos equipos de trabajo. Una aplicación Spark consiste en una sucesión de operaciones sobre porciones del RDD. Tanto la información como la computación son distribuidas entre los recursos disponibles de un cluster. El framework se encarga de la distribución automática y coordinación entre todos los flujos de trabajo, que resulta transparente para el usuario final.

Spark ofrece dos tipos de operaciones: *transformaciones* y *acciones*. Las transformaciones son operaciones que crean un nuevo RDD a partir de otro RDD previo. Este tipo de operaciones son perezosas, lo que significa que la única forma de garantizar que se ejecutarán todos los fragmentos de una operación de este tipo es invocar a otra operación de tipo acción. Las transformaciones pueden ser de dos tipos, *narrow* o *wide*. Las operaciones de tipo narrow son aquellas que cada fragmento ya contiene toda la información necesaria para completar la operación, como por ejemplo operaciones de tipo map o filter. Por otro lado, las operaciones de tipo wide son aquellas que sí requieren de información de varias particiones previas para calcular el resultado final, como por ejemplo, operaciones groupByKey o reduceByKey. El otro tipo de operaciones son las acciones. Son operaciones que devuelven un resultado a partir de un RDD, pero no crean un nuevo RDD. Ejemplos de estas operaciones son first, take, collect o count.

Spark organiza el trabajo en múltiples fases, que se separan cada vez que hay una redistribución de datos o cuando se invoca a una operación de tipo acción. Cada fase se compone de un número de tareas que realizan transformaciones narrow. Esta fase finaliza cuando se redistribuyen los datos. Abstrayéndose, la ejecución de un workflow de una aplicación Spark puede ser descrita como un grafo acíclico dirigido (DAG)[17] mostrando las modificaciones aplicadas sobre los RDD.



Ambas transformaciones y acciones están divididas internamente en varias subtareas que se ejecutan en paralelo dentro del framework. Cada subtarea ejecuta el mismo método definido en Spark, pero sobre un pequeño fragmento de datos del dataset (RDD). Por defecto, el número de tareas en las que se divide cada operación está determinado por el número de particiones en las que el RDD de entrada de la operación ha sido dividido previamente. Por ello, el paralelismo en la operación tiene una relación directa con el número de particiones de el RDD procesado. El número de tareas paralelas ejecutadas por Spark en una operación sobre un RDD sera igual al número de particiones. Sin embargo, este número de tareas no es siempre el mismo para todas las operaciones. Este número puede variar a lo largo de la ejecución, dependiendo de varios factores. El número de particiones puede ser definido explícitamente por el usuario en algunos casos, o ser ajustado automáticamente por el contexto dependiendo del tamaño del conjunto de datos y de la configuración del cluster. Como ejemplo, las operaciones de unión, intersección o el producto cartesiano dan como resultado un RDD con un número de particiones igual a la suma, el máximo o el producto del número de particiones de los RDD originales.

Por último, el planificador de Spark distribuye las tareas a los nodos disponibles del cluster. En Spark, hay 2 tipos de planificaciones, uno externo a la ejecución de un programa (inter-aplicación), que distribuye los recursos entre todas las aplicaciones de Spark que se está ejecutando en el cluster, y otro a nivel interno del programa(intra-aplicación), que se encarga de distribuir las N tareas dentro de un programa.

Spark cuenta con tres planificadores a nivel de inter-aplicación: YARN, Mesos y standalone. Varían en la gestión de los recursos del cluster. Proporcionan una repartición de los recursos estática entre las aplicaciones previamente a la ejecución (YARN, standalone), o una compartición de los cores del CPU en ejecución, en el caso de Mesos.

A nivel de intra-aplicación, hay 3 posibles planificaciones: fair, fifo o ninguna. En caso de necesidad, se puede crear una planificación más compleja que permita ordenar dependiendo de varios factores, para crear una distribución optima.

En la siguiente tabla se resumen los principales conceptos de Spark introducidos en este capitulo.

#	Concepto	Significado
1	RDD	Estructura de datos distribuidos tolerante a fallos
2	Partición	Fragmento de información dentro de un RDD
3	Operación Spark	Función que transforma o actúa sobre RDDs
4	Transformación	Tipo de operación que crea un nuevo RDD
5	Acción	Tipo de operación que no produce RDD como respuesta
6	Paralelismo	Número de tareas concurrentes por operación Spark
7	Fase	Agrupación de tareas previas a una repartición
8	Planificación	Repartición de tareas

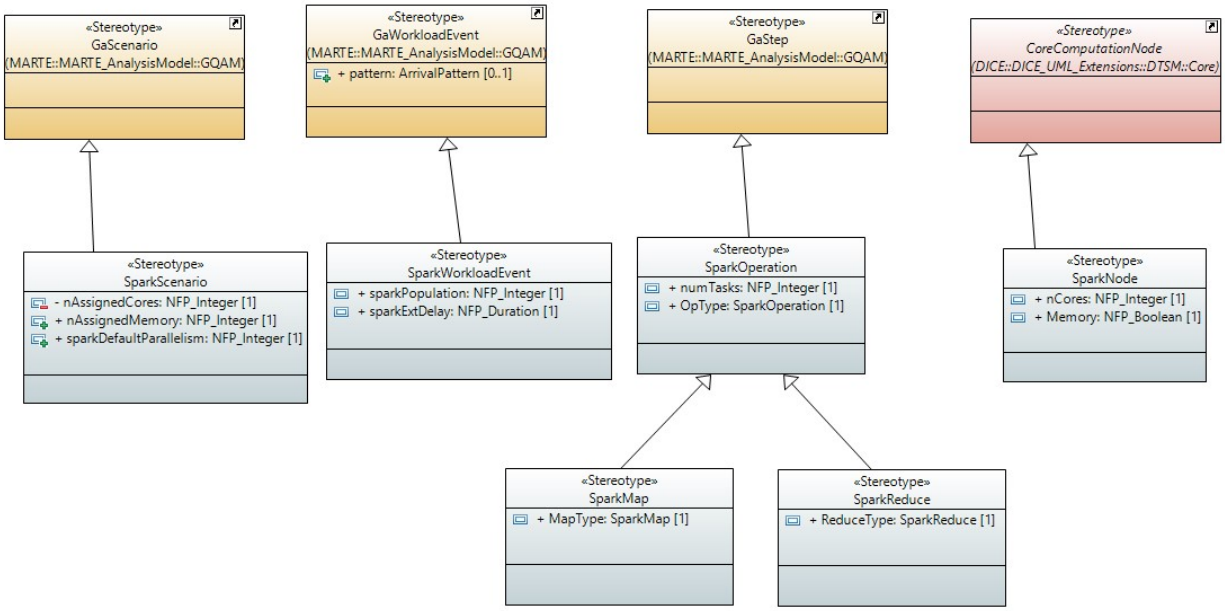
Modelado de aplicaciones Spark en UML

En este apartado, presento mi aproximación al modelado de aplicaciones Spark para evaluar rendimientos utilizando diagramas UML. Es necesario representar las características de Spark y los parámetros identificados en el apartado anterior. Concretamente, se trabaja con diagramas de actividad complementados con diagramas de despliegue, que serán los diagramas que se transformarán en modelos para el análisis de prestaciones.

Perfil de Spark

Los perfiles de UML proveen un mecanismo de extensión genérico para construir modelos UML en dominios particulares. Están basados en estereotipos y valores etiquetados adicionales que se aplican a acciones, atributos y más. Un perfil es una colección de tales extensiones que describen conjuntamente las características de algún entorno en particular y facilitan el modelado de un sistema en ese dominio.

Cada tecnología tiene diferentes detalles específicos, y los perfiles nos permiten capturar estos detalles para utilizarlos posteriormente en otros modelos UML. Por tanto, en este proyecto es necesario definir un nuevo perfil de Spark. Esta implementación del perfil se integra junto al resto de perfiles que la herramienta DICE soporta (DICE profiles[13]).



En la figura podemos ver el resultado del perfil de Spark. En este perfil observamos todos los parámetros que tienen un impacto directo en el rendimiento de una ejecución y todos los tipos de operaciones posibles que puede invocar un programa.

El perfil y los modelos UML que veremos posteriormente han sido creados utilizando la herramienta Eclipse Papyrus[5]. Para el proyecto, también utilizamos MARTE[8], una especificación de un perfil UML que añade a UML la capacidad de modelar sistemas en tiempo real o sistemas

embebidos. Basar el perfil en MARTE nos permite utilizar ciertos atributos de prestaciones, que utilizaremos posteriormente. Por ello, la mayor parte de los objetos creados en el perfil heredan de las estructuras y atributos definidas por el estándar MARTE. Además, MARTE nos ofrece la posibilidad de usar los perfiles NFP y VSL. NFP sirve para describir las propiedades no funcionales de un sistema. VSL ofrece un lenguaje para especificar los valores de las métricas, constantes, propiedades y parámetros. En este proyecto ambos NFP y VSL los hemos utilizado para especificar las métricas y calcular el rendimiento, respectivamente.

Concepto Spark	Estereotipo	Aplicado a	Variable	Tipo
Operación Spark genérica	Spark-Operation	Nodo Acción/Actividad		
			OpType	SparkOperation
			numTasks	NFP_Integer
			hostDemand	NFP_Duration
Transformación	SparkMap	Nodo Acción/Actividad		
			MapType	SparkMap
Acción	SparkReduce	Nodo Acción/Actividad		
			ReduceType	SparkReduce
Planificador	SparkScenario	Diagrama de actividad		
			nAssignedCores	NFP_Integer
			nAssignedMemory	NFP_Integer
			sparkDefaultParallelism	NFP_Integer
Inicialización RDD	Spark-Workload-Event	Nodo inicial		
			sparkPopulation	NFP_Integer
			sparkExtDelay	NFP_Duration
Nodo computacional	SparkNode	Nodo/Dispositivo		
			nCores	NFP_Integer
			Memory	NFP_Boolean

SparkScenario representa una invocación a un programa, y refleja la información propia del sistema que realizara la ejecución. Los parámetros mostrados, nAssignedCores y nAssignedMemory, hacen referencia al conjunto de cores y memoria asignados específicamente a dicha ejecución. Aunque las máquinas tengan varios cores, no todos ellos estarán disponibles, y este parámetro reserva esa cantidad de recursos para que realmente están asignados a una ejecución. Hay que destacar la configuración de defaultParalellism, que refleja el número de particiones que tomará una operación por defecto.

SparkWorkloadEvent hace referencia a la creación de un RDD de cero, incluyendo la inicialización de sus datos y el tiempo de distribución de los fragmentos entre los nodos. SparkPopulation hace referencia a la cantidad de datos de los que dispone dicho RDD. Por otro lado, sparkExtDelay refleja el tiempo que tarda en movilizarse la estructura de datos y transmitir la información a los nodos correspondientes del cluster.

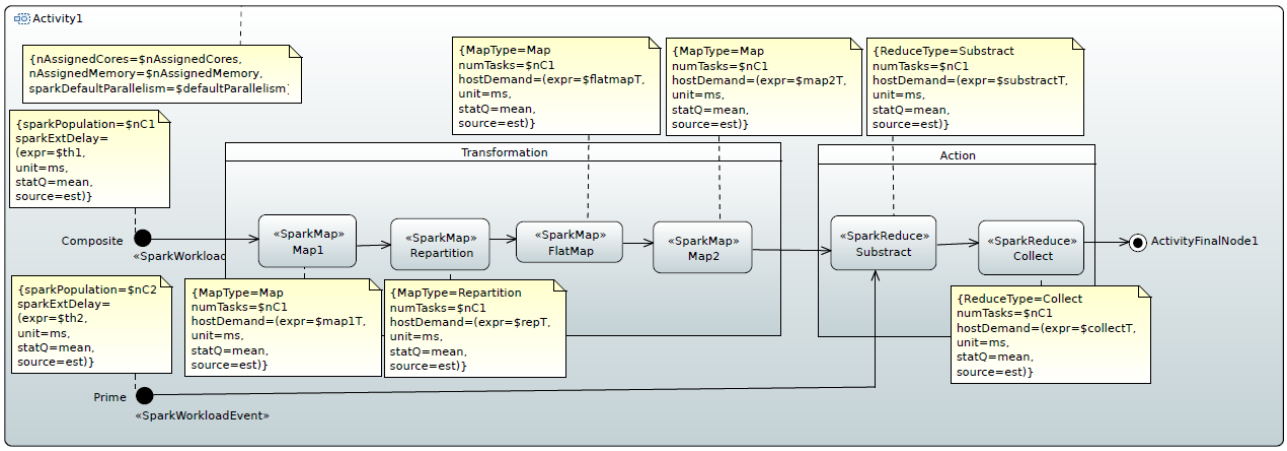
SparkOperation representa las operaciones disponibles. numTask hace referencia al número de subtareas en que se divide cada operación en el caso de tomar un valor distinto al ntask definido por Spark. OpType permite diferenciar si la operación es una transformación o una acción. SparkMap recoge las transformaciones, mientras que SparkReduce representa las acciones. En ambos casos la operación específica se guarda en los campos enumerables MapType o ReduceType, respectivamente. Para las transformaciones, la operación puede ser un map, filter, repartition u operaciones entre RDD como unión, intersección,… En el caso de las acciones se puede seleccionar entre reduce, collect, count o foreach.

Por último, SparkNode representa las máquinas reales de las que se dispone para la ejecución, así como las características propias de dichas máquinas, como la memoria o el número de cores disponibles.

Aplicaciones en Spark

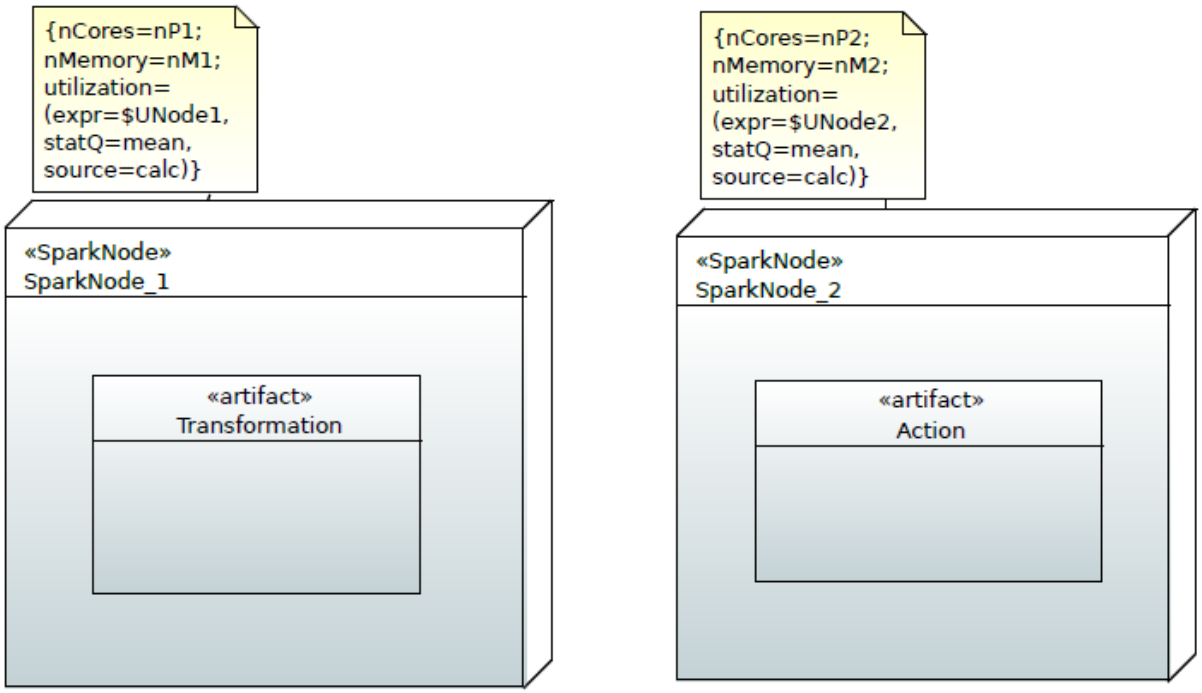
En este punto, se presenta una guía sobre cómo se modelan en UML las aplicaciones de Spark usando el perfil definido para dicha tecnología.

Modelar programas de Spark resulta relativamente sencillo, dado que la correspondencia es para cada operación de Spark una operación correspondiente en el modelo. Estos se agruparan en una sucesión de operaciones que finalizaran siempre con una operación de tipo acción, o con una transformación de los datos. Dada la naturaleza *perezosa* de las operaciones de tipo transformación, es difícil conocer el tiempo exacto de una operación de este tipo. En su lugar, podemos conocer el tiempo que tarda una operación de tipo acción con todas las operaciones de tipo transformación que le preceden. Por ello, agruparemos cada operación acción con las operaciones de tipo transformación que le preceden.



En este ejemplo, podemos ver uno de los programas de ejemplo utilizado durante el proyecto. El programa calcula todos los primos existentes entre 0 y N, eliminando todos los números que no lo son. El proceso comienza creando en el Map1 un conjunto de datos, que contiene el par (a, lista[2, N/a]) para todos los valores de a en el rango [1, N/2]. Estos valores se redistribuyen para que todos los nodos del cluster tengan un fragmento de RDD (lista) de tamaño equivalente (Repartition). Las dos siguientes operaciones multiplican los valores de cada par por a (Flatmap y Map2). En la siguiente operación (Substract) se necesitan 2 RDD de entrada. Uno es el que acabamos de calcular. El otro es un nuevo RDD que contiene todos los números en el rango [0,N]. La operación elimina los valores que se encuentran en el primer RDD del segundo RDD. Por último, recopilamos los resultados obtenidos en el Collect.

Este ejemplo es una variante del modelado de UML estándar, ya que tiene dos nodos de inicio. Por defecto, el UML estándar sólo permite un nodo inicial y un nodo final en el diagrama de actividades. Dado que presentamos el DAG de una aplicación Apache Spark mediante el diagrama de actividades, es necesario que en los modelos admitan varios nodos iniciales, e igualmente sea posible tener varios nodos finales. Cada nodo inicial representa la creación de un nuevo RDD con el que poder realizar operaciones.



El diagrama de actividad se complementa con el diagrama de despliegue. Las tareas de Spark se agrupan lógicamente en particiones en el diagrama de actividad, que se mapean como nodos de ejecución físicos en el diagrama de despliegue. Esta adicción restringe la concurrencia lógica, el número de tareas que la aplicación Spark puede ejecutar en paralelo, al número de cores disponible.

Para más detalles, en los anexos se incluyen más ejemplos de diagramas UML perfilados de otros programas Spark utilizados a lo largo del proyecto. Asimismo, otro anexo incluye información acerca de la notación utilizada en el diagrama.

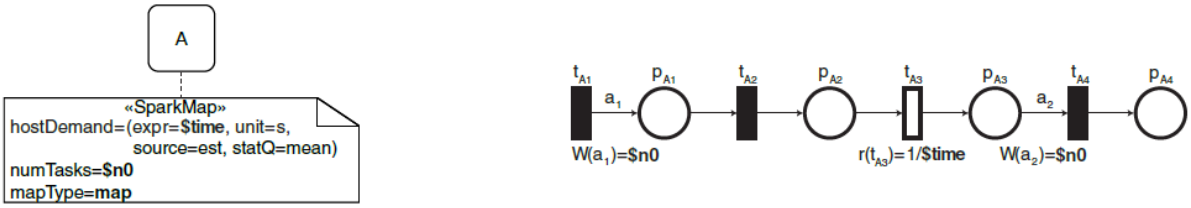
Transformación de un modelo perfilado en UML en redes de Petri

Los modelos de UML perfilados para la tecnología Spark recogen toda la información relevante del diseño junto con las características y configuraciones de la tecnología para esa aplicación en particular. Sin embargo, para obtener métricas de rendimiento necesitamos transformar esos modelos UML en modelos formales adecuados para el análisis de prestaciones. Por ello, en esta sección definimos una transformación de modelos UML perfilados a dichos modelos formales para su análisis. El modelo elegido para tal función son las Generalized Stochastic Petri Net[16] (GSPN). En esta sección proponemos los patrones de transformación. Cada patrón toma como entrada parte del diseño de un modelo UML perfilado en Spark, lo que produce una subred GSPN.

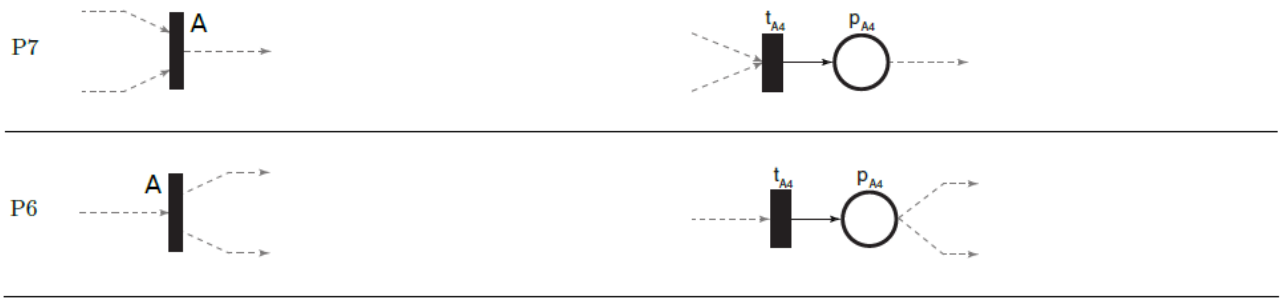
P1

P2

P3



P4



En esta tabla, se ven los patrones de transformación para las porciones de los diagramas UML que hemos visto en anteriores apartados. La primera columna de la tabla muestra el patrón que se transforma, y la segunda columna muestra el correspondiente resultado, como una red de Petri. La notación muestra en negrita los elementos y atributos importantes del modelo UML que tienen asociación directa con la red de Petri. En punteado aparecen otras subredes de Petri transformadas en otros patrones. Los detalles de cada transformación pueden encontrarse en un anexo al final de esta memoria.

Calculadoras

Con estas transformaciones podemos obtener una red de Petri que nos permite simular una aplicación, pero necesitamos saber que información podemos extraer de la simulación y como. Por ello, necesitamos extraer de los resultados de simulación en bruto aquellos datos que puedan darnos información acerca del rendimiento del sistema. Por defecto, la simulación numérica de una red de Petri nos devuelve 1) el ritmo de disparo de las transiciones; y 2) el número medio de marcas en cada lugar.

El análisis de la simulación de la red de Petri nos permite extraer tres métricas sobre el rendimiento de la aplicación modelada: el tiempo de respuesta, la productividad de los resultados y el porcentaje de uso de los recursos.

La productividad mide la cantidad de ejecuciones de la aplicación Spark finalizadas por unidad de tiempo. Para calcularlo, tomamos el throughput de la transición que marca el final de la aplicación (en la transformación P1, $t_{Activity2}$). Dado que una ejecución completa supondrá que una marca pase por dicha transición, el throughput de dicha transición es el resultado que necesitamos.

El tiempo de respuesta es el tiempo medio que tarda en finalizar una ejecución de la aplicación. Lo calculamos a partir del throughput hallado en el párrafo anterior. El tiempo de respuesta es el inverso del throughput.

$$Tiempo\,respuesta = 1/Throughput$$

El porcentaje de uso de recursos hardware nos indica la media de recursos que usará la aplicación a lo largo de toda la ejecución. El porcentaje se calcula respecto al total de cores físicos que existen en el conjunto de nodos del cluster. Para ello, utilizaremos el número medio de marcas en el lugar (patrón P12 en los anexos, lugar p_r) que indica el número de recursos disponibles. El número total de recursos menos dicha media, dividido por el número total de recursos nos devuelve el resultado deseado.

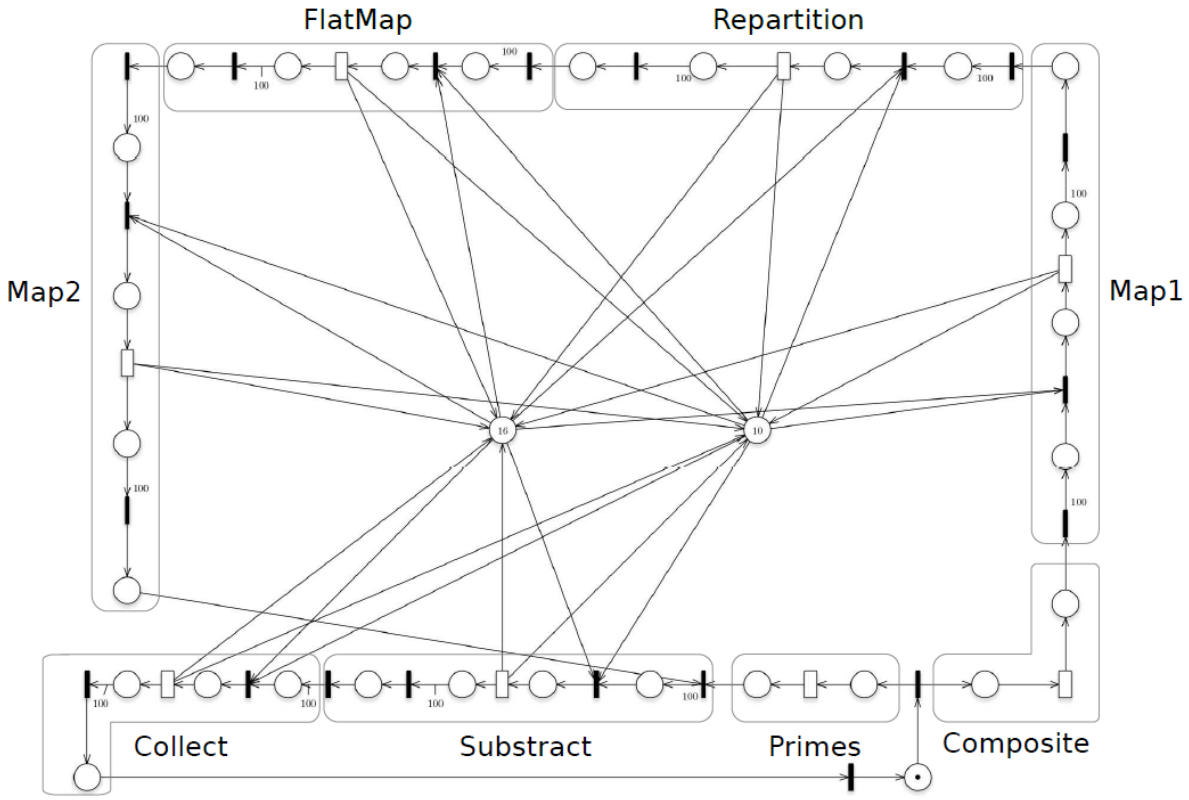
$$Uso\ recursos = (Marcado\ inicial\ Pr - Media\ marcas\ Pr) / Marcado\ inicial\ Pr$$

Todos estos cálculos están integrados en la herramienta DICE Simulator.

Automatización de la transformación en redes de Petri

La transformación de modelos de UML en redes de Petri, así como la evaluación de métricas de rendimiento, están completamente automatizadas y son transparentes para el usuario final. Primero, la transformación utiliza QVT-MOF 2.0 para obtener un fichero Petri Net Markup Language (PNML)[10], un estándar ISO basado en XML que guarda redes de Petri. Durante la transformación también se genera un fichero de trazas, que relaciona los elementos de la red de Petri con los elementos originales del modelo de UML. Esto ayuda a identificar los elementos en la red de Petri que la herramienta necesita identificar durante el análisis de rendimiento. Después, Acceleo ha sido usado para implementar la transformación de PNML en un formato específico que utiliza la herramienta GreatSPN tool, herramienta que se utiliza para realizar las simulaciones.

Tomando estas transformaciones y aplicándolas al ejemplo de primos presentado en la sección anterior, la red de Petri generada como resultado se puede ver en la siguiente imagen



El nodo que vemos abajo con una marca inicial representa el inicio de la aplicación. En este punto se divide en dos, que corresponde a los dos nodos iniciales representados en el diagrama de actividad del apartado anterior. Cada uno de los dos caminos recorre las operaciones, hasta unirse en el substract, a partir del cual se realizan las operaciones finales en el mismo camino. Hay que destacar los nodos centrales que interactúan gran parte de las transacciones. Dichos nodos

representan los recursos hardware disponibles en el cluster de Spark y sirven para calcular los porcentajes de utilización del sistema.

Validación de los resultados

Ejecución vs Simulación

Una vez definidas las métricas de rendimiento a considerar y su forma de cálculo, el siguiente paso del proyecto es verificar si los resultados obtenidos para una aplicación simulada con la red de Petri se aproximan a los resultados reales de la aplicación Spark.

Para la fase de validación de la transformación, tomamos varios programas y paquetes, disponibles públicamente, de diversas fuentes. De la propia release del código de Spark, tomamos un programa de ejemplo que permite calcular el valor aproximado de pi, llamado sparkPi. También tomamos una batería de programas, llamada Databricks[20], que permite probar diversas operaciones de Spark y medir sus resultados bajo diferentes parámetros. Por último, tomamos el programa primes[21], programa que se ha usado como ejemplo en esta memoria, que permite calcular los números primos de un rango.

Estas aplicaciones las sometemos a una serie de pruebas. Por un lado, las ejecutamos en un entorno de pruebas, variando las configuraciones con las que se ejecutaban, y se guardaban los logs resultantes. El entorno de ejecución donde se han ejecutado las pruebas consiste en un equipo coordinador, o Spark Master, y 6 estaciones de trabajo para realizar los cómputos. Las estaciones de trabajo son máquinas virtuales desplegadas en el cluster de Flexiant[13]. Cada equipo está caracterizado por una CPU virtual de 4x2.60GHz con 4GBytes de RAM, una conexión Gigabit ethernet y un sistema operativo Ubuntu Linux (v 14.04).

Por otro lado, se crearon los modelos UML (y sus transformaciones a las redes de Petri) que representaban estas aplicaciones y se simularon dichas aplicaciones. Cada aplicación se simulaba varias veces, variando los parámetros utilizados. Los parámetros se seleccionaban de manera que las ejecuciones se parecieran a las ejecuciones reales.

Tras ambos pasos, se procede a comparar los resultados obtenidos. Se comparan los tiempos de ejecución, al ser un valor que se podía obtener directamente desde los logs de las ejecuciones reales.

Núm. cores	numTasks	T. aplic (s)	T. simul (s)	Error (%)
6	100	49,820	59,970	20,373
6	200	85,610	54,270	36,608
6	400	78,160	75,380	3,557
6	800	141,130	139,120	1,424
12	100	27,990	39,520	41,193
12	200	43,270	51,580	19,205
12	400	55,500	57,580	3,748
12	800	104,360	105,700	1,284
18	100	36,510	36,020	1,342
18	200	45,220	59,790	32,220
18	400	56,170	59,980	6,783
18	800	99,970	102,900	2,931
24	100	30,210	40,960	35,584
24	200	46,690	43,350	7,154
24	400	58,210	63,770	9,552
24	800	106,070	113,090	6,618

Los parámetros que hemos variado en la configuración son el número de cores asignados a la aplicación y el número de particiones del RDD. Otros parámetros como por ejemplo, el tiempo de operación de cada tarea Map y Reduce, se han mantenido constantes entre configuraciones. Estos últimos valores se han extraído de la monitorización de los logs.

Discusión

Con el objetivo de intentar reducir el porcentaje de error de los tiempos de respuesta calculados por la simulación de la red de Petri, nos hemos fijado en las transiciones temporizadas generadas mediante la transformación del modelo UML en una red de Petri. Las transiciones temporizadas de la red de Petri aplican una determinada distribución estadística para simular el comportamiento de la ejecución, y creemos que esta es la fuente de las desviaciones.

Por defecto, las transiciones temporizadas de la red de Petri siguen una distribución estadística exponencial[18] con lambda igual al tiempo asignado en esa transición (parámetro $time de P3). Esto significa que, en media, una transición temporizada modelada con esta distribución disparará una marca (procesará una tarea) cada 1/lambda unidades de tiempo.

Hemos estudiado la forma en la que se lanzan las subtareas de una transformación/acción en el cluster de Spark: normalmente, las tareas se lanzan en bloques de máximo K tareas con K igual al número de cores reservados en el cluster por la aplicación. Por ello, hemos concluido que la distribución estadística Erlang es la que mejor se ajusta a nuestras necesidades.

La distribución estadística Erlang[19] es una distribución que generaliza la distribución exponencial con un parámetro K entero. A grandes rasgos, una distribución Erlang es equivalente a la suma de K distribuciones exponenciales con parámetro lambda. Esto significa que una transición temporizada modelada con una distribución Erlang en la red de Petri es capaz de procesar K marcas (tareas) en paralelo en un tiempo lambda, de forma similar a como se lanzan las tareas en el cluster de Spark.

Núm. cores	numTasks	T. aplic (s)	T. simul Exp (s)	T. simul Erlang (s)	Error Exp (%)	Error Erlang (%)
6	100	49,820	59,970	37,076	20,373	25,582
6	200	85,610	54,270	35,842	36,608	58,410
12	100	27,990	39,520	32,256	41,193	15,229
12	200	43,270	51,580	35,605	19,205	17,713
18	100	36,510	36,020	31,031	1,342	15,010
18	200	45,220	59,790	44,005	32,220	2,696
24	100	30,210	40,960	35,507	35,584	17,526
24	200	46,690	43,350	35,664	7,154	23,608

En esta tabla de tiempos podemos observar los tiempos de respuesta obtenidos por el DICE Simulator al cambiar la configuración de las transiciones temporizadas exponenciales en la red de Petri (ver imagen en página 20) por transiciones temporizadas Erlang. Se han incluido solo los tiempos con 100 y 200 particiones, donde encontramos resultados positivos. Se puede observar que una distribución Erlang ofrece un error menor que la distribución exponencial.

El valor de configuración K de la distribución Erlang lo hemos inferido de los logs de monitorización de la aplicación SparkPrimos en el clúster de Spark. Para ello, hemos extraído los tiempos de ejecución medios y las desviaciones típicas de las tareas de cada operación

transformación/acción. Por defecto, este valor K lo hemos aproximado como el paralelismo por defecto en el cluster (SparkScenario → defaultParallelism)

De acuerdo a los resultados experimentales mostrados en las tablas comparativas de tiempos, la distribución Erlang se ajusta mejor para las experimentaciones en las que el RDD se particiona en pocas tareas (numTasks < 200) con respecto al número de cores del cluster. La distribución exponencial, pese a que se ajusta peor en condiciones de bajo paralelismo y número de cores reservados, presenta un mejor comportamiento para situaciones de alta paralelización (numTasks > 400).

Como conclusión, podemos asumir que la elección de una correcta distribución estadística resulta relevante. Para configuraciones de un RDD con poco paralelismo (p.ej., un RDD con pocas particiones), la distribución Erlang es la que mejor se aproxima a los resultados experimentales del cluster de Spark. El parámetro K de la distribución se ajusta a la forma en la que el planificador lanza las tareas en bloques de K elementos (K <= número de cores reservados en el cluster). A su vez, para RDDs con alto paralelismo (p.ej., un RDD con muchas particiones), la distribución exponencial es la que mejor se aproxima a los resultados experimentales del cluster de Spark porque la forma en la que actúa el planificador de tareas se difumina para grandes poblaciones de tareas.

Conclusión

En esta memoria se han presentado las principales contribuciones realizadas a lo largo de las diversas fases del proyecto. En particular, he presentado una aproximación novedosa para el modelado y análisis de prestaciones de aplicaciones desarrolladas con Apache Spark. El objetivo de este trabajo ha sido crear un entorno para guiar a los ingenieros de software durante la fase de diseño, de modo que aumenten que la calidad de sus aplicaciones. Por ejemplo, siguiendo esta aproximación, se puede evaluar el impacto de alojar una aplicación Spark en un servidor o cluster específicos (por ejemplo, según el número de cores, su velocidad, etc); y predecir sus tiempos de respuesta o la utilización de los recursos hardware.

Mi aportación es única en varios aspectos. En primer lugar, he introducido una nueva interpretación del diagrama de actividad en UML para modelar el DAG. También he introducido un nuevo perfil para Spark, que refleja los conceptos necesarios para la evaluación de prestaciones, en particular redes de Petri. He propuesto patrones de transformación de fragmentos de UML para obtener un modelo formal para la evaluación de prestaciones. Estos patrones han sido validados comparando las prestaciones estimadas mediante las redes de Petri contra los resultados obtenidos al utilizar las aplicaciones de Spark reales.

Se seleccionaron aquellos resultados cuyo error era demasiado grande y se investigo las razones de la diferencia. Se propusieron alternativas para esos casos, diseñándolas, implementándolas y verificándolas de nuevo contra los resultados reales. Por último, he implementado el perfil Spark, la transformación y calculadoras utilizando el entorno de modelado Papyrus en la plataforma Eclipse y el lenguaje QVT. Todo esto está integrado en el DICE Simulator, un plugin para Eclipse disponible gratuitamente.

He de destacar que a lo largo del proyecto he ganado conocimiento y experiencia en múltiples ámbitos en los cuales mi experiencia era muy limitada hasta la fecha. Principalmente, he adquirido una gran experiencia al trabajar con meta-modelos y meta-lenguajes. No solamente como herramientas de diseño, si no que además he aprendido a usar los propios modelos con finalidades de desarrollo. Por ejemplo, para evaluar propiedades no funcionales en tiempo de modelado. He descubierto la utilidad del modelado y diseño de aplicaciones al comienzo del ciclo de vida de una aplicación software. Dichas tareas, comúnmente subestimadas, permiten agilizar mucho cualquier desarrollo posterior. Por otro lado, he aprendido a manejar Apache Spark, una tecnología con un gran potencial.

Mejoras futuras

A lo largo de estos meses, he trabajado con el DICE Simulator, y he llegado a conocer en profundidad dicha herramienta. Si bien ofrece una gran cantidad de opciones y es funcional, a lo largo del desarrollo he encontrado que hay puntos que o bien se podría mejorar o incluso incluir una nueva funcionalidad ayudaría en gran medida al uso de dicha herramienta. Aquí presento las posibles mejoras que se podrían aplicar a la herramienta en el futuro.

Extender las funcionalidades de Spark en el perfil
A lo largo de este proyecto el análisis se focalizó en el core de Spark y el procesamiento en batch. Sin embargo, Spark posee más módulos y extensiones que permiten añadir más funcionalidades a la tecnología. Un buen ejemplo sería Spark Streaming, que permite procesar transmisiones de datos en tiempo real. Añadir dichas funcionalidades a las ya creadas en el perfil de Spark permitiría ampliar el modelado a nuevas aplicaciones.

Descripciones en los parámetros de entrada a la simulación
Al realizar cada simulación, se realiza un paso en el que asignan los valores numéricos de cada variable. Dichas variables aparecen con los nombres utilizados en el perfil. A pesar de que hemos utilizado nombres relativamente descriptivos, aun así hay variables que pueden llevar a confusión. Añadir un breve campo de descripción que se muestre en cada variable, describiendo de manera breve el valore esperado, ayudaría a evitar confusión y simulaciones erróneas. Asimismo, añadir un manual o algunas indicaciones a los parámetros heredados desde MARTE y cómo anotarlos podría ser muy provechoso y facilitaría el uso de la herramienta.

Datos agregados en simulaciones
La herramienta DICE Simulator permite configurar la ejecución de varias simulaciones de la red de Petri desde una única ventana de usuario. En dicha ventana, se introducen todos lo valores posibles para las variables de entrada del modelo UML. A continuación, se muestran todas las permutaciones posibles de dichos valores. Como ultimo paso, seleccionas las permutaciones que quieres ejecutar.

Sobre el papel, funciona bien, pero a la hora de lanzar varias combinaciones la ejecución no siempre es tan sencilla. En todos nuestros ejemplos, ciertos valores de unas variables iban directamente ligados con un valor concreto de otras variables, con lo que mostrar todas las combinaciones posibles de valores de cada variable con muchos valores no servía. Permitir asociar valores entre variables agilizaría mucho las ejecuciones de múltiples combinaciones.

Diferentes transformaciones y formalismos
Todos los casos que conozco en el sistema DICE Simulator han trabajado con transformaciones de UML en redes de Petri. Pero dichas redes no sirven para todos los casos; o es más sencillo modelar un problema con otro formalismo. Por ejemplo, se puede considerar redes de colas como modelos para simular el comportamiento de aplicaciones Spark. Ello conllevaría repetir el ciclo presentado en la última parte de este proyecto: definición de la transformación, implementación, especificación de las calculadoras y validación.

Anexos

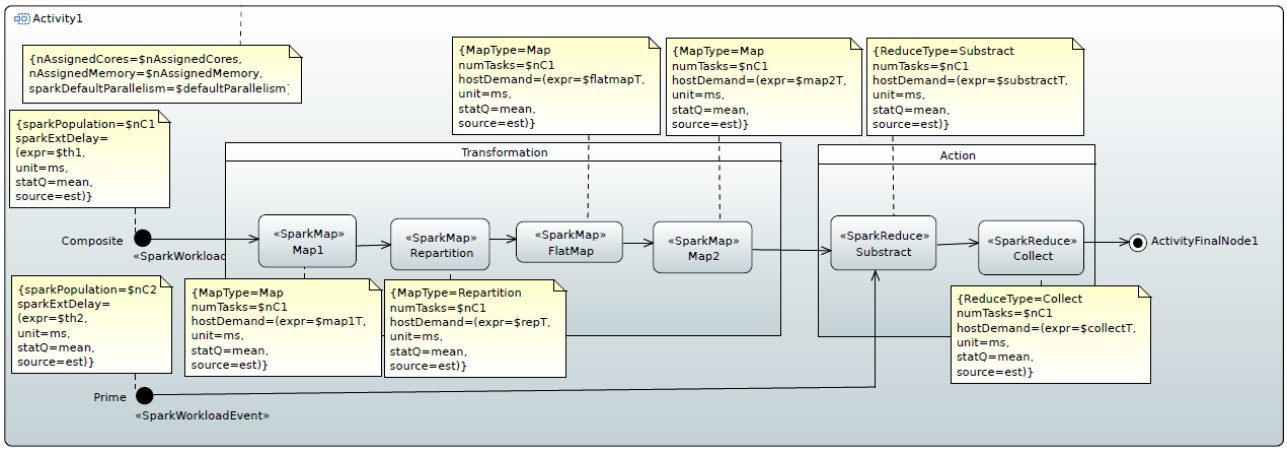
Referencias/Bibliografía

1. Apache Spark Website - https://spark.apache.org/
2. The DICE Consortium. Proyecto DICE Website - http://www.dice-h2020.eu/
3. The DICE Consortium. DICE Simulator - https://github.com/dice-project/DICE-Simulation
4. The DICE Consortium. Spark profile - https://github.com/dice-project/DICE-Profiles
5. Equipo DISCO Website - http://webdiis.unizar.es/DISCO/
6. Eclipse Website - https://eclipse.org/
7. Papyrus Website - https://eclipse.org/papyrus/
8. Perfil UML de MARTE, Documentación - http://www.omg.org/spec/MARTE/1.1/
9. Lenguaje QVTo, Website - https://wiki.eclipse.org/QVTo
10. Petri Net Markup Language (PNML) - http://www.pnml.org/
11. GreatSPN Website - http://www.di.unito.it/~greatspn/index.html
12. Acceleo Website - https://eclipse.org/acceleo/
13. Flexiant Cloud Orchestator Website - https://www.flexiant.com/
14. Estándar UML. Articulo en Wikipedia - https://es.wikipedia.org/wiki/Lenguaje_unificado_de_modelado
15. Perfil UML - http://wikis.uca.es/wikiPLII/index.php/UML_Profiles
16. Stochastic Petri net - https://en.wikipedia.org/wiki/Stochastic_Petri_net
17. Grafo acíclico dirigido (DAG) - https://es.wikipedia.org/wiki/Grafo_ac%C3%ADclico_dirigido
18. Distribución exponencial. Articulo en Wikipedia - https://en.wikipedia.org/wiki/Exponential_distribution
19. Distribución Erlang. Articulo en Wikipedia - https://en.wikipedia.org/wiki/Erlang_distribution
20. Databricks spark-perf Website - https://github.com/databricks/spark-perf
21. Shawn B, articulo. *Improving Spark Performance With Partitioning* - http://dev.sortable.com/spark-repartition/

Editar un modelo UML perfilado con Spark

UML nos permite modelar el comportamiento de prácticamente cualquier aplicación de Spark utilizando el perfil y un nivel de abstracción adecuado durante el modelado. Este apartado explica los detalles para crear dicho modelo.

Partiendo del ejemplo explicado en la memoria, veremos fácilmente cómo crear el modelo de cualquier otra aplicación.

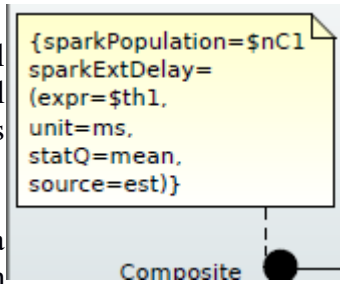


La mejor forma de diseñar cualquier modelo es tomar como referencia el flujo de los datos de un RDD dentro de la aplicación reflejado como un DAG. Toda aplicación comienza con ciertos datos iniciales/parámetros de entrada, representados como el nodo inicial en UML. Esta información es sometida a ciertas operaciones para obtener nuevos datos, lo que se representa con un nodo de actividad en UML que toma datos de entrada y devuelve un resultado. Todas las operaciones se representarían del mismo modo, dado que todas las operaciones de la tecnología tienen el mismo comportamiento: toman los datos de entrada y devuelven un resultado tras un determinado tiempo. El nodo final representaría el hecho de que la aplicación ha hallado el resultado deseado y es devuelto por dicha aplicación.

La representación de flujos de trabajo mediante un DAG permite tener varios nodos iniciales/finales en el modelo UML, así como varios caminos de ejecución en el diagrama de actividades. El ejemplo más claro se ve en el substract en la imagen. Las operaciones como substract o union necesitan más de un dato de entrada, y esto queda representado como dos flechas de entrada en la operación.

Por último, podemos observar varias particiones UML que sirven para enlazar las operaciones del diagrama de actividad con el diagrama de despliegue y los recursos físicos. El efecto práctico sería el mismo si englobamos todas las operaciones bajo la misma partición, pero hemos ido englobándolos en bloques de operaciones que acaban en una acción (por ejemplo, reduce). De esta forma, mostramos de manera explícita el agrupamiento de operaciones en fases que hace Spark durante la ejecución.

Anotar propiedades no funcionales mediante el lenguaje VSL



Las expresiones VSL son usadas en los modelos basados en el perfil de Spark en dos casos: 1) para especificar las variables NFP en el modelo (parámetros de entrada) y 2) para especificar las métricas que sera calculadas por el modelo (especificar resultados de salida)

Como se puede ver en el ejemplo, la notación consiste en la clásica asignación de programación. A la derecha del igual tendremos un valor o variable que se convertirá en un valor, y a la izquierda se encuentra la variable en la que se guardara dicho valor. Se pueden utilizar constantes, variables (introducidas con un símbolo de dólar $) o expresiones.

(expr=$th1, unit=ms, statQ=mean, source=est)

Este ejemplo es utilizado para dar valor a una variable NFP (caso 1). Este ejemplo especifica que a la variable del objeto que tenga asignado este ejemplo, se le asignara th1 (expr) milisegundos (unit), que es un valor medio (statQ) obtenido de una estimación en un sistema real (source).

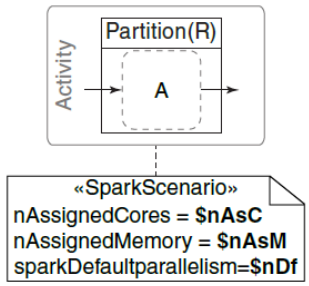
(expr=$UNode1, statQ=mean, source=calc)

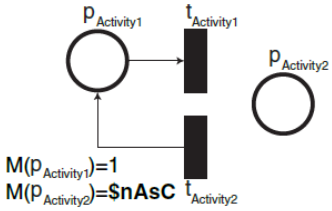
Este segundo ejemplo se usaría para anotar el atributo de “utilización” de los Spark Nodes (dispositivos hardware), siendo el segundo caso descrito (especificación de métricas de salida). La fórmula indica que queremos calcular (source) un valor medio (statQ) en la variable UNode1 (expr), como porcentaje (ausencia de unit).

Patrones de transformación de UML

En la memoria se presentaron brevemente las transformaciones que se aplican para transformar un modelo UML que representa una aplicación Spark en una red de Petri. En esta sección voy a explicar con mayor detalle en que consiste cada transformación y cual es su cometido. En todos los casos mostrados, cualquier componente que se muestre con una linea discontinua, representa interfaces con otros patrones.

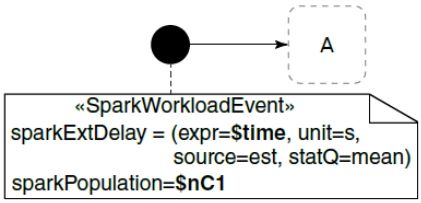
P1

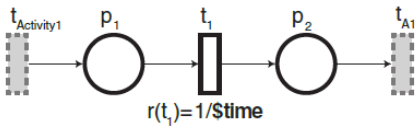




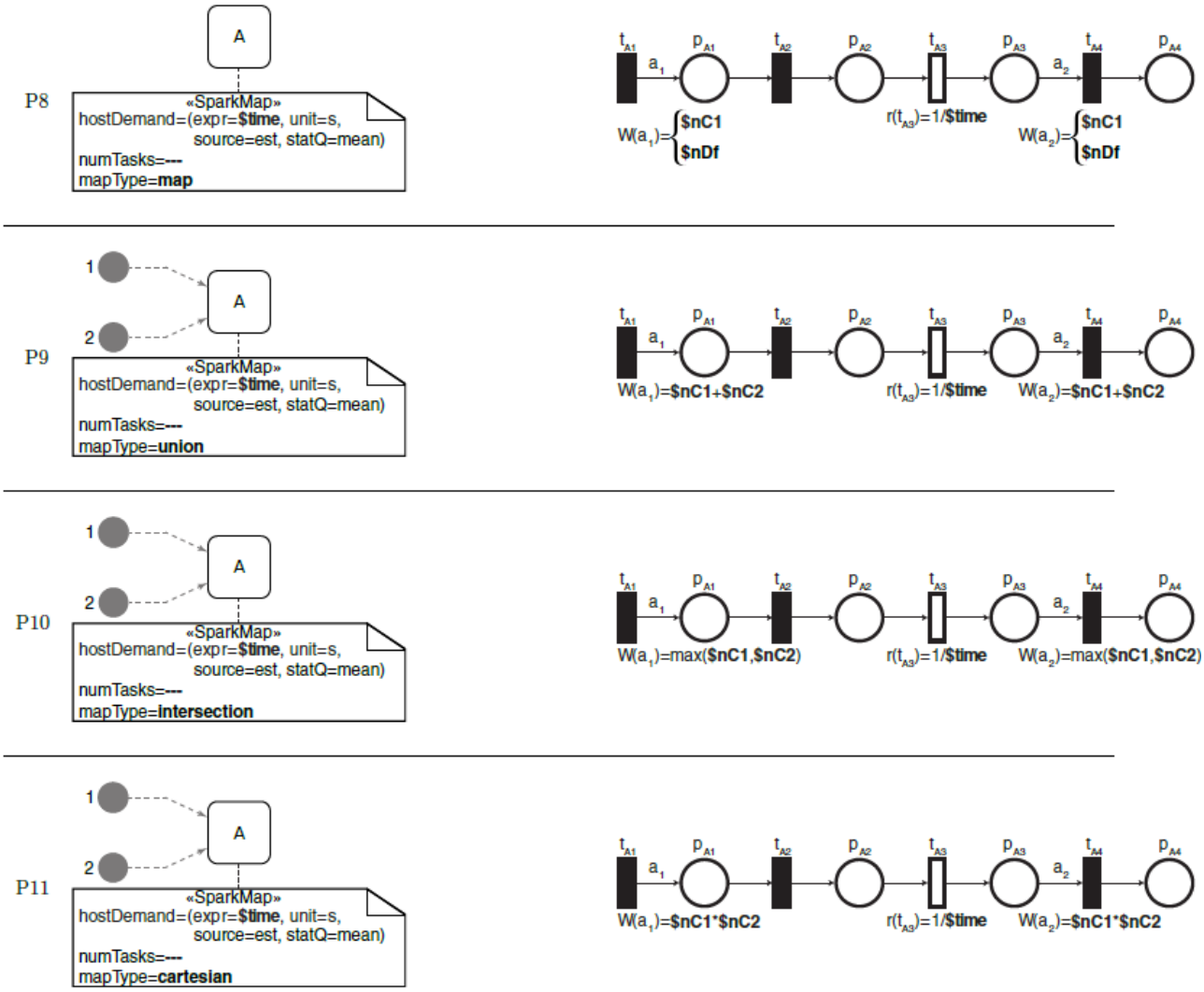
Esta transformación define el lugar y transición inicial de la red de Petri, y la transición final. Mediante esta transformación, la composición de patrones da lugar a una red de Petri cerrada. Esta transformación también incluye la inicialización de los nodos de recursos. SparkDefaultParallelism puede ser utilizando en P8 (ver más adelante). nAssignedMemory fue añadido en la implementación del perfil, pero la versión actual de la transformación no utiliza este dato de momento.

P2



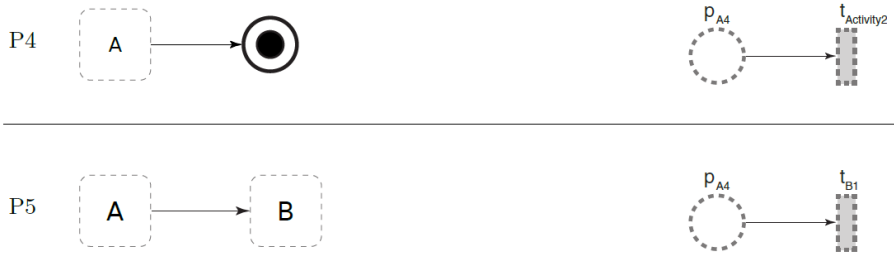


Este caso representa la inicialización de un RDD de la aplicación. Representa el tiempo que se tarda desde que iniciamos la aplicación hasta que la primera operación real se lanza, y refleja tiempos de inicialización de datos o de aplicaciones. SparkPopulation se utiliza para iniciar el número de fragmentos que tendrá por defecto el RDD.



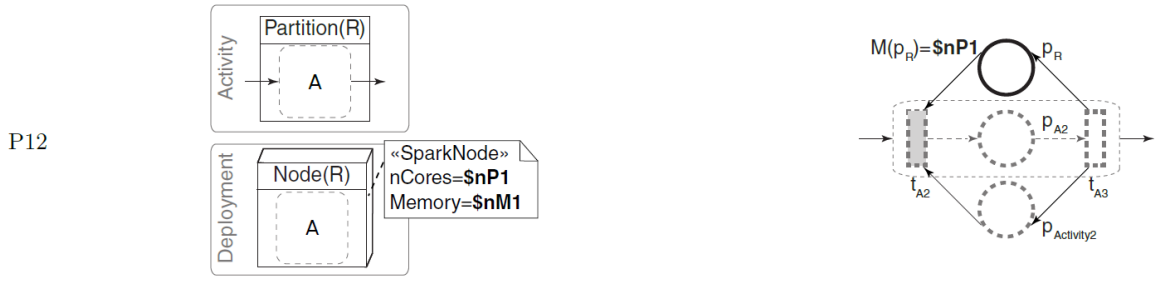
Estas transformaciones representan operaciones básicas de Spark. Dependiendo del caso se utilizará cada uno de los 4 casos presentados. Una vez comenzada una operación, esta se dividirá en numTasks subtareas. Normalmente, numTasks será el número de particiones configurado en Spark (sparkPopulation en P2). En ciertos casos, cuando numTasks no está definido explícitamente por el usuario, la división se hará utilizando defaultParalellism de P1 o la población definida por sparkPopulation. La creación de las distintas subtareas de la operación se muestra con el peso $W(a_1)$ del arco a_1. Cada una de las subtareas se lanzará (t_{A2}) cuando se encuentre algún recurso disponible. La operación tardará un tiempo en ejecutarse (t_{A3}). Una vez todas las tareas hallan finalizado (a_2) se dará por concluida la operación (t_{A4}).

Las 3 variantes (P9, P10 y P11) representan operaciones cuyo número de tareas asociado es diferente. nCX es el valor de SparkPopulation de cada uno de los interfaces previos. En mapType, se observa el tipo de operación específica que requiere de cada transformación.



Estos casos representan uniones entre componentes, pero no tienen impacto ya que solo indican transiciones entre dos apartados de la aplicación. P4 indica el final de la aplicación, con lo que unirá el final de A con el inicio de la aplicación mostrado en P1, cerrando la red de Petri. El patrón P5 es similar a P4, pero unirá el final de la interfaz A con el inicio de la interfaz B.

Estas transformaciones se mantienen por compatibilidad con el estándar UML. P6 permite la división de una aplicación en dos subtareas; mientras que P7 marca la sincronización.



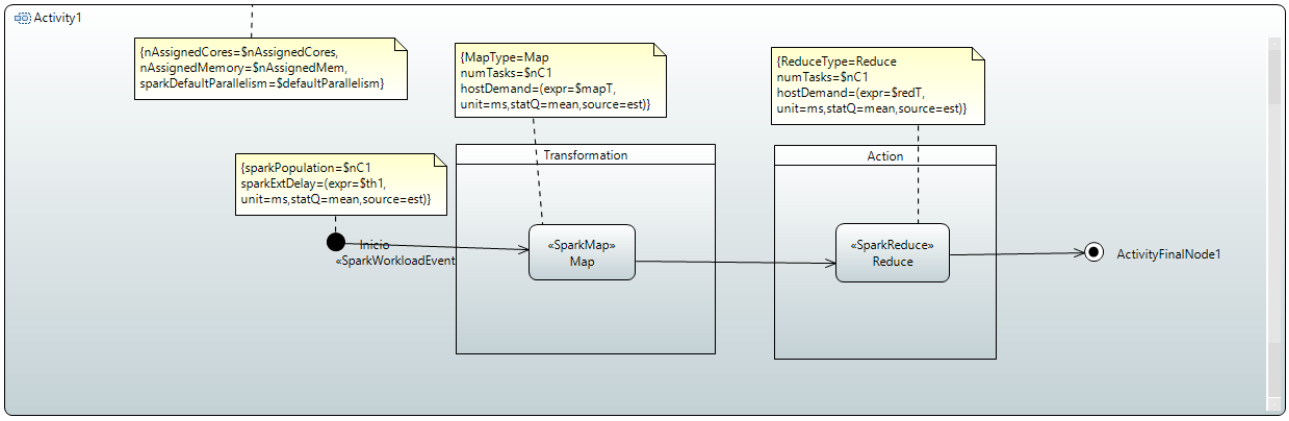
Por último, queda el tema de los recursos. Ya hemos comentado que cada subtarea de una operación se inicia cuando hay un recurso (core) disponible en el cluster. Pero esta característica no quedaba reflejada en dicha transformación, sino que es un cambio que se aplica en este patrón P12. A cada operación que consume recursos se le aplicará esta transformación. Esto representa que para un sistema con $nP1 recursos disponibles, en un momento dado no puede haber ejecutándose a la vez más de $nP1 subtareas, por lo que la siguiente tarea ha de esperar que haya algún recurso disponible.

Programas de pruebas

A lo largo de todo el proyecto se han utilizado varios programas para poner a prueba la herramienta, el perfil y las transformaciones. En este apartado se presentan cada programa modelado y los tiempos obtenido para dicha programas.

El primer programa que presento viene incluido con la propia release de Apache Spark. Es uno de los ejemplos que Spark utiliza en el tutorial para aprender cómo lanzar programas desde Spark. Dado que este caso fue utilizado varias veces durante las pruebas para poder comprobar el funcionamiento del cluster, se decidió incluirlo en la batería de pruebas.

1. SparkPi

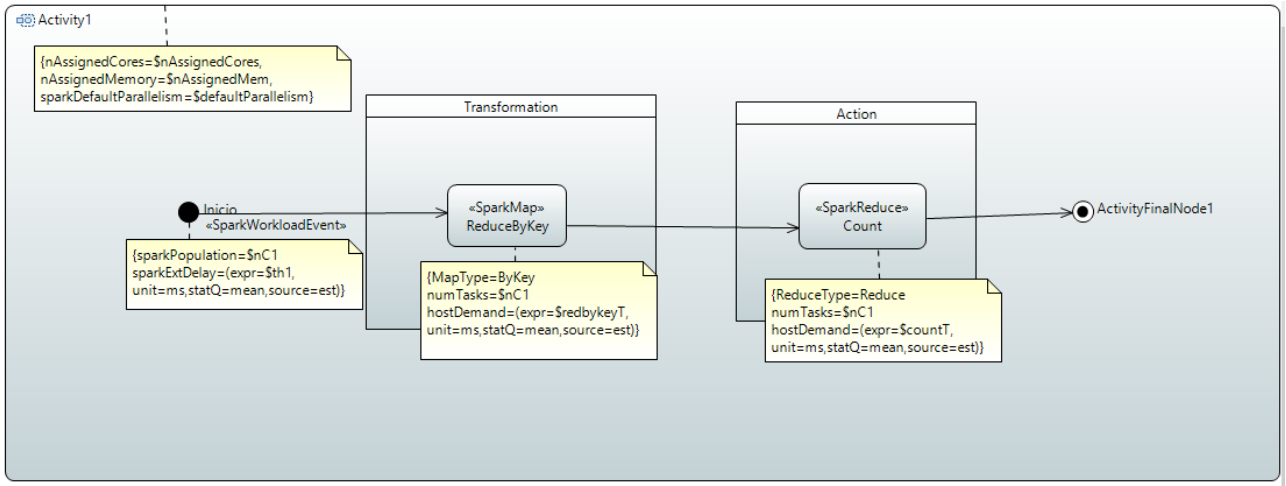


Núm. cores	numTasks	T. aplic (s)	T. simul (s)	Error (%)
6	100	14,542	22,702	56,116
6	200	16,579	9,528	42,530
6	400	20,795	14,019	32,586
6	800	26,304	20,709	21,271
12	100	14,376	8,204	42,931
12	200	15,810	9,422	40,408
12	400	18,092	11,330	37,377
12	800	24,128	17,778	26,320
18	100	14,652	9,698	33,808
18	200	16,845	10,660	36,718
18	400	16,130	11,121	31,053
18	800	22,927	17,500	23,669
24	100	15,397	11,443	25,681
24	200	16,220	11,584	28,583
24	400	19,784	13,241	33,073
24	800	26,128	19,280	26,208

El resto de programas utilizados pertenecen a Databricks$_{20}$, una batería de programas que permite probar diversas operaciones de Spark y medir sus prestaciones bajo diferentes condiciones. Contiene varios programas que incluyen las diversas operaciones básicas que ofrece Spark. Databricks ofrece facilidades para poder cambiar rápidamente las variables más relevantes respecto al rendimiento. Realiza tareas de inicialización de RDD, evitando que estos influyan en los tiempos de las aplicaciones, y permite realizar ejecuciones consecutivas de cada aplicación para poder obtener medias. Además, ya lleva implementado un sistema de logs para obtener unos resultados más precisos.

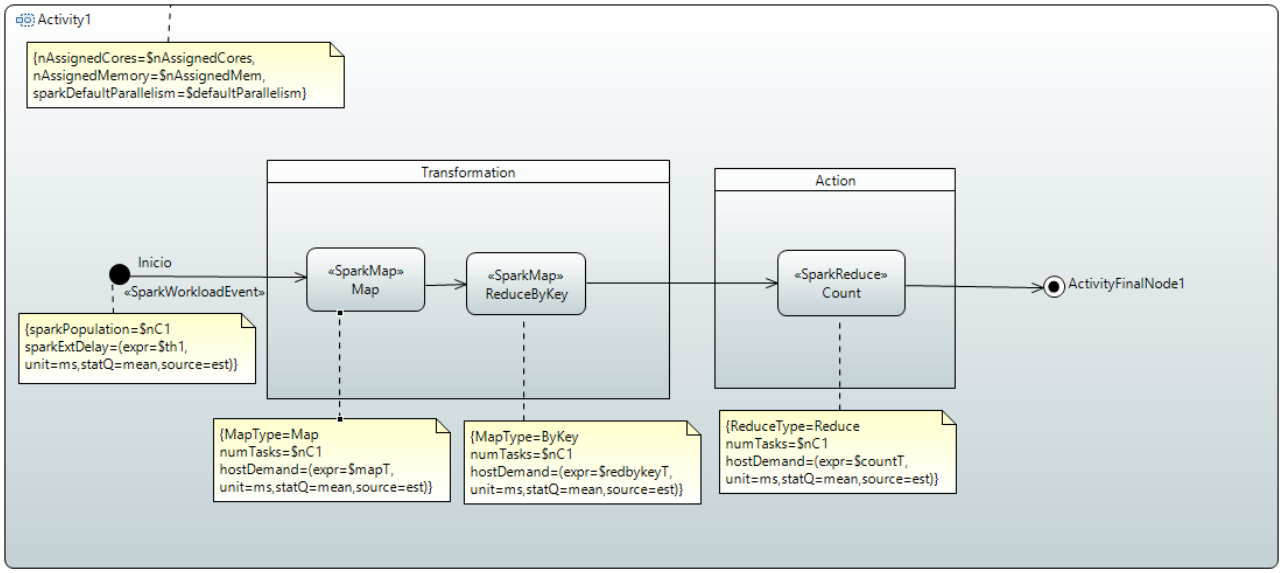
La mayor parte de los programas de esta batería están centradas en obtener información acerca del rendimiento de una operación específica, por lo cual estos programas suelen incluir dicha operación y las operaciones necesarias para que dicha operación pueda ejecutarse. Por ello, estas aplicaciones son bastante sencillas

2. aggregateByKeyInt



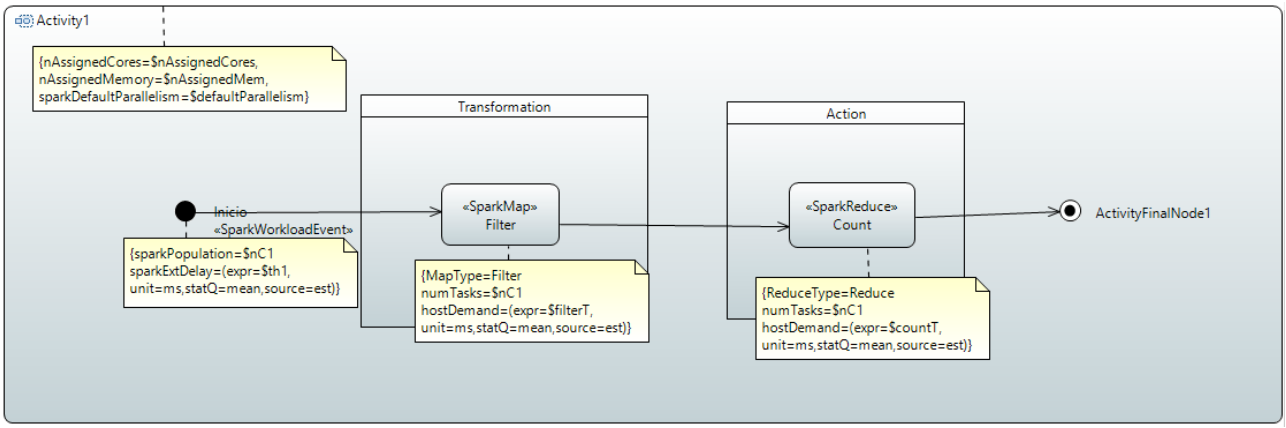
Núm. cores	numTasks	T. aplic (s)	T. simul (s)	Error (%)
6	100	31,328	31,440	0,356
6	200	53,694	49,218	8,336
6	400	99,476	89,336	10,193
6	800	224,742	205,159	8,713
12	100	19,450	19,051	2,050
12	200	25,866	28,876	11,638
12	400	55,497	56,762	2,279
12	800	137,315	138,165	0,619
18	100	13,695	18,065	31,912
18	200	26,414	19,521	26,094
18	400	41,884	45,172	7,850
18	800	106,097	106,083	0,013
24	100	21,403	15,123	29,341
24	200	28,021	23,436	16,362
24	400	36,726	43,006	17,099
24	800	98,417	101,828	3,466

3. aggregateByKey



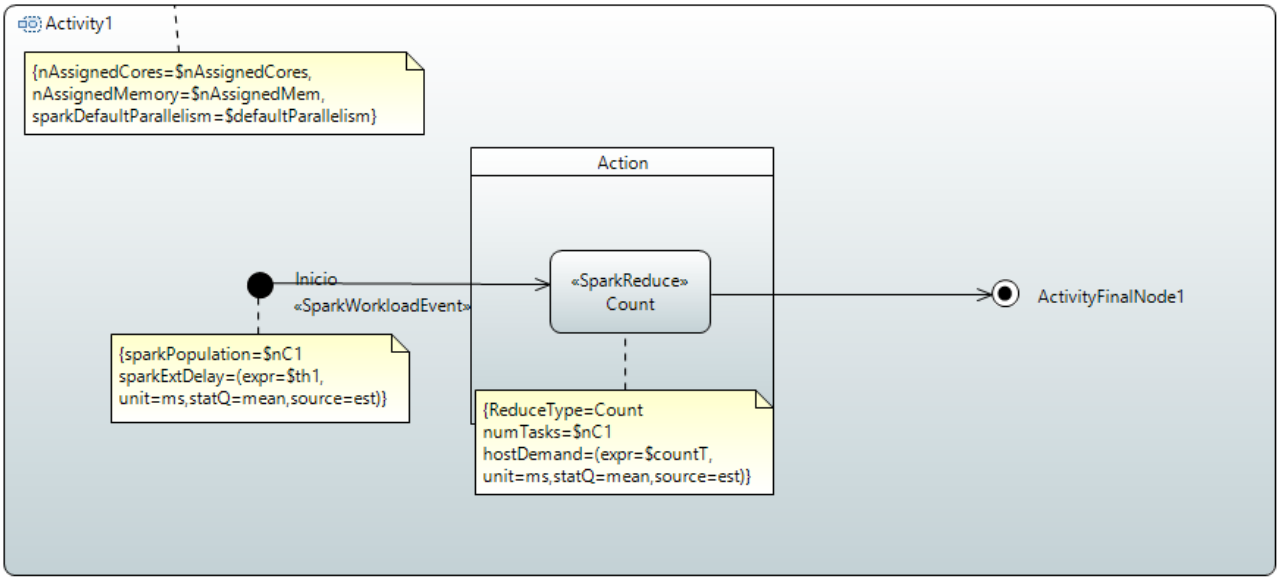
Núm. cores	numTasks	T. aplic (s)	T. simul (s)	Error (%)
6	100	16,603	23,430	41,116
6	200	24,535	31,127	26,866
6	400	42,103	50,884	20,857
6	800	81,009	99,641	23,000
12	100	10,509	15,681	49,215
12	200	13,955	19,882	42,474
12	400	26,218	33,199	26,626
12	800	49,859	60,256	20,852
18	100	12,751	12,533	1,708
18	200	13,908	16,833	21,028
18	400	25,006	21,018	15,950
18	800	35,266	46,111	30,752
24	100	7,588	14,091	85,695
24	200	16,408	15,258	7,008
24	400	30,574	17,623	42,361
24	800	39,628	31,992	19,268

4. countWithFilter



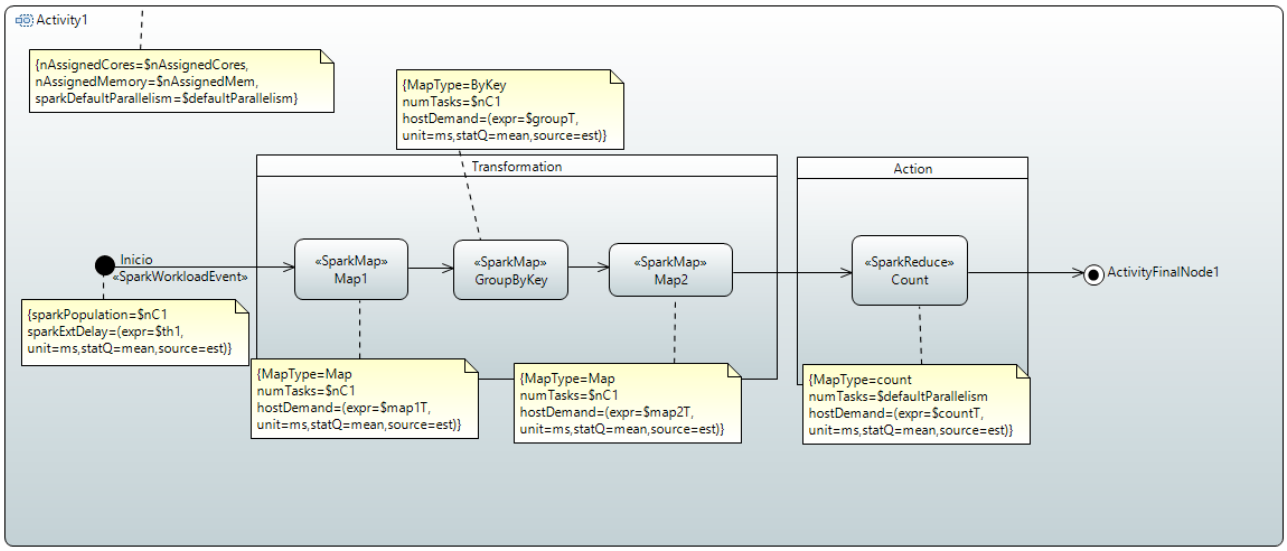
Núm. cores	numTasks	T. aplic (s)	T. simul (s)	Error (%)
6	100	7,331	4,690	36,021
6	200	9,019	10,735	19,029
6	400	10,989	13,996	27,362
6	800	16,375	19,852	21,234
12	100	3,983	6,109	53,387
12	200	5,361	6,790	26,652
12	400	6,772	8,250	21,822
12	800	8,794	11,543	31,255
18	100	3,419	5,008	46,487
18	200	3,262	5,205	59,555
18	400	4,992	6,412	28,436
18	800	6,869	8,693	26,548
24	100	3,165	5,163	63,135
24	200	2,738	4,834	76,566
24	400	3,731	5,166	38,467
24	800	5,188	6,815	31,356

5. count



Núm. cores	numTasks	T. aplic (s)	T. simul (s)	Error (%)
6	100	7,331	8,312	13,383
6	200	9,019	9,697	7,515
6	400	10,989	11,909	8,370
6	800	16,375	16,236	0,849
12	100	3,983	5,484	37,687
12	200	5,361	5,974	11,440
12	400	6,772	6,928	2,303
12	800	8,794	9,521	8,265
18	100	3,419	4,959	45,032
18	200	3,262	5,187	59,017
18	400	4,992	5,724	14,673
18	800	6,869	7,037	2,440
24	100	3,165	4,672	47,606
24	200	2,738	4,690	71,303
24	400	3,731	4,982	33,521
24	800	5,188	5,950	14,686

6. naive

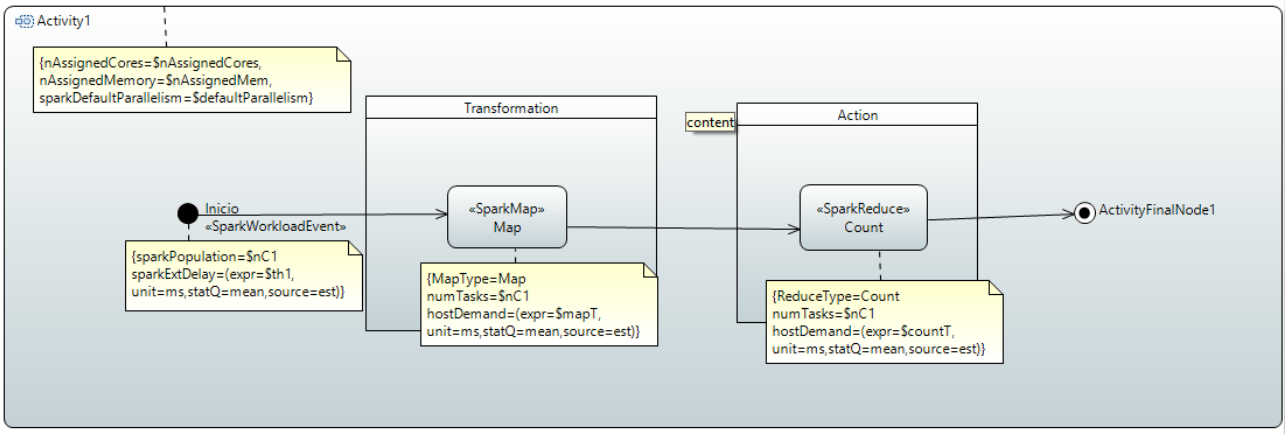


Núm. cores	numTasks	T. aplic (s)	T. simul Exp (s)	T. simul Erlang (s)	Error Exp (%)	Error Erlang (%)
6	100	28,309	23,948	22,322	15,405	21,148
6	200	22,500	32,630	29,690	45,021	31,957
6	400	23,367	33,610	29,898	43,834	27,951
6	800	32,468	41,774	35,783	28,663	10,209
12	100	14,214	18,786	17,049	32,167	19,947
12	200	15,966	26,728	25,579	67,405	60,212
12	400	17,548	27,478	25,420	56,585	44,861
12	800	19,564	31,608	27,764	61,563	41,912
18	100	14,675	21,019	20,154	43,227	37,334
18	200	14,766	20,564	19,090	39,269	29,281
18	400	14,908	25,589	23,627	71,646	58,487
18	800	16,524	28,292	25,500	71,215	54,323
24	100	9,925	21,938	19,707	121,037	98,560
24	200	13,871	24,224	22,762	74,638	64,098
24	400	13,492	23,188	21,261	71,862	57,581
24	800	12,658	22,702	20,133	79,352	59,050

De los programas usados en la batería de pruebas, este es el más complejo y el que mejor nos ha permitido comprobar las desviaciones en las simulaciones. En este ejemplo, el count final utiliza como número de particiones defaultParalellism, que en las pruebas contenía un valor menor de particiones que numTasks. En esos casos es cuando la distribución exponencial no se ajusta correctamente, y se probó la distribución Erlang. Ninguno de los otros programas contiene esta situación, por ello no se realizaron baterías separadas entre las distintas distribuciones. La distribución Erlang mejora los porcentajes de error en la mayoría de los casos.

7. scheduler_throughput

Este programa es uno de los más básicos de la batería y no estaba configurado para variar el número de particiones. A pesar de ello, lo incluimos en las pruebas y estos son los resultados obtenidos.



Núm. cores	numTasks	T. aplic (s)	T. simul (s)	Error (%)
6	100	23,380	16,464	29,583
12	100	12,259	8,419	31,323
18	100	8,740	6,869	21,412
24	100	6,610	5,706	13,669