

Trabajo Fin de Grado

Área de arquitectura de computadores y  
sistemas operativos

# sparkanalyzer

## Instrumentación de Apache Spark

*Estudiante: José Manuel García Sánchez*  
*Consultor: Francesc Guim*

## Tabla de contenido

1	Definición de proyecto.....	3
1.1	Objetivos y motivaciones.....	3
1.2	Resultados esperados y entregables.....	4
1.3	Ciclo de vida.....	4
1.4	Infraestructura para proyecto.....	5
2	Estructura del proyecto.....	5
2.1	Actividades.....	5
2.1.1	Planificación TFG (PEC1).....	6
2.1.2	Estudio Apache Spark, desarrollo de Pintool para profiling (PEC2).....	6
2.1.3	Desarrollo de sistema de visualización (PEC3).....	8
2.1.4	Memoria y Presentación TFG.....	8
3	Planificación.....	8
3.1	Calendario.....	8
3.2	Planificación.....	8
3.3	Esfuerzo estimado.....	9
4	Desarrollo del proyecto.....	10
4.1	Apache Spark.....	10
4.1.1	Estudio y funcionamiento y primeros pasos.....	10
4.1.2	Despliegue e instalación. Compilación.....	14
4.1.3	Integración con PIN Tool.....	15
4.2	PIN Tool.....	19
4.2.1	Diseño.....	20
4.2.2	Codificación.....	22
4.2.3	Output de la instrumentación.....	30
5	Pruebas y validación.....	33
5.1	Pruebas de integración.....	35
6	Conclusiones y trabajos futuros.....	39
6.1	Justificación del proyecto.....	39
6.2	Conclusiones.....	40
7	Anexos.....	41
7.1	Fichero de configuración pinSpark.conf.....	41
7.2	Codificación en C/C++ de la PIN Tool. PinSpark.cpp.....	42
7.3	Codificación en Scala del fichero para Apache Spark. WorkerCommandBuilder.scala.....	67
8	Glosario de términos.....	69
9	Bibliografía.....	70
10	Referencias.....	71

# 1 Definición de proyecto

## 1.1 Objetivos y motivaciones

El presente documento recoge el plan de trabajo, la planificación el resultado para el trabajo fin de Grado en Ingeniería Informática de la UOC<sup>i</sup>. En este caso el área de conocimiento elegida corresponde a arquitectura de computadores y sistemas operativos. En el documento por tanto se indican los objetivos del proyecto, la planificación del mismo y los resultados obtenidos.

El área elegida en este caso corresponde a mi línea principal de investigación a lo largo de los diferentes estudios que he cursado en la UOC. Tanto en el trabajo fin de carrera en la Ingeniería Técnica en Informática de sistemas como en el Master en Ingeniería Informática orienté los diferentes trabajos en este área de conocimiento. Dado que mi orientación profesional está centrada en el mundo de los sistemas y las comunicaciones en entornos empresariales, me resulta especialmente grato poder expresarme y colaborar al conocimiento general en la Ingeniería Informática desarrollando y trabajando en proyectos que aporten valor mas allá del puro tramite necesario para lograr la titulación.

En anteriores proyectos hemos tenido una relación importante con el mundo de la instrumentación (especialmente con las PIN Tools<sup>ii</sup>) y por ello en primera instancia cuando contacte con el consultor Francesc Guim para discutir posibilidades de proyecto le planteé seguir con la misma línea de trabajo. Su propuesta enlaza con la línea de trabajo de la instrumentación y por tanto podemos seguir una línea de investigación común también en este proyecto.

El proyecto a implementar se basa en el diseño de una solución que permita instrumentar el comportamiento de Apache Spark<sup>iii</sup>, obteniendo información sobre el comportamiento a bajo nivel de los sistemas durante la ejecución de cargas de trabajo. La instrumentación se realizará implementando una pintool que se encargue de extraer la información en tiempo real.

Como segunda parte del TFG establecemos los trabajos correspondientes a la representación en alto nivel (vía Web) de esta información de tal forma que podamos consultar de forma sencilla esa información en tiempo real.

Así, por tanto, para la elaboración del trabajo fin de grado necesitaremos:

- Realizar un estudio en profundidad del funcionamiento de Apache Spark
- Diseñar el modelo de la información que sería interesante conocer en una ejecución
- Implementar una pintool<sup>iv</sup> que use dicho modelo a efecto de instrumentar ejecuciones reales
- Implementar una pasarela a base de datos o almacenamiento persistente para almacenar la información en tiempo real a representar
- Como segunda parte, implementar el mecanismo de representación en alto nivel (Web) usando el almacenamiento persistente como fuente

Me gustaría remarcar que el proyecto en si mismo tiene una importante complejidad, dado que el diseño del modelo y la programación de la pintool van a consumir una gran parte de los recursos. El planteamiento de la segunda parte con la presentación a alto nivel lo vamos a considerar opcional y dependerá del avance de los trabajos de la primera parte. No obstante, lo vamos a asignar a la PEC3 dentro de la planificación. En caso de que no se pueda llegar a esta parte se reflejará en los documentos siguientes y se realizará una replanificación.

Para el desarrollo del proyecto necesitaremos disponer de conocimientos en las diferentes tecnologías implicadas:

- Apache Spark y su modelo de trabajo distribuido.
- Modelos de rendimiento hardware sobre arquitectura x86
- Todo el desarrollo e realizarán sobre un sistema Linux. El lenguaje escogido para el desarrollo es C++, ya que es el idioma de la herramienta Pintool. Será necesario también conocimientos de Java (Apache Spark tiene gran parte desarrollada en Java) y posiblemente de un lenguaje de representación Web (Apache) para la segunda parte del proyecto.

## ***1.2 Resultados esperados y entregables***

Los entregables para el presente trabajo fin de grado son los siguientes:

- El plan de trabajo, que recoge la información inicial y la planificación de las diferentes tareas a ejecutar.
- El producto final, que consiste en el diseño del modelo y de la pintool. Como segunda parte, se incluye la representación de alto nivel (Web)
- La memoria, documento final que recogerá todos los trabajos realizados a lo largo del proyecto y demostrará el cumplimiento de los objetivos. La memoria deberá incluir todos los estudios y pruebas realizadas, así como el detalle de la solución elaborada.

## ***1.3 Ciclo de vida***

Para el control y adecuado seguimiento del proyecto, he optado por aplicar un ciclo de vida en cascada o clásico. Considero que este ciclo de vida es adecuado para este proyecto por los siguientes motivos:

- Tenemos una definición clara desde el principio de los objetivos del mismo, y no se contempla que varíen a lo largo de la vida del proyecto.
- El proyecto está muy acotado y tiene un tamaño mas que adecuado para que el uso del ciclo de vida en cascada no plantee inconvenientes.

## 1.4 Infraestructura para proyecto

A efecto del desarrollo de las diferentes tareas, es necesario disponer de una infraestructura donde realizar el desarrollo y las simulaciones. La siguiente tabla indica el resumen del equipamiento hardware y software

Equipo portátil para desarrollo	Asus Eee PC 1201, Intel Atom 1.6Ghz, 3 GB de RAM, Ubuntu 15.04
Pinkit	Herramienta de generación de trazas
g++	Versión 4.9.2
LibreOffice	Versión 4.4.2

Todo el trabajo de documentación se realiza sobre LibreOffice versión 4.0. Para el control de la planificación del proyecto, usamos la herramienta de software libre Planner.

Adicionalmente dado que estamos analizando un sistema de Bigdata necesitamos probar la interacción de la Pintool cuando se presentan varios nodos de calculo. Para estas pruebas usaremos una infraestructura alojada en un proveedor IAAS<sup>v</sup>, en este caso en particular, Amazon AWS. La compilación y el desarrollo lo realizaremos también en la infraestructura desplegada en AWS EC2.

## 2 Estructura del proyecto

### 2.1 Actividades

La siguiente tabla recoge las diferentes actividades a ejecutar para el desarrollo del proyecto. Los niveles indican la anidación de las tareas, desde el nivel 1, el más general, hasta el nivel 4, el mas detallado.

Código	Nivel 1	Nivel 2	Nivel 3	Nivel 4
01	Planificación TFG (PEC1)			
01.01		Diseño plan de trabajo		
01.02		Entrega plan de trabajo (PEC1)		
02	Estudio Spark+Pintool (PEC2)			
02.01		Análisis		

		funcionamiento Apache Spark		
02.02		Diseño modelo de rendimiento HW		
02.03		Desarrollo herramienta		
02.03.01			Desplegar cluster Spark	
02.03.02			Diseño pintool	
02.03.03			Codificación	
02.03.03.01				Codificación C++ pintool
02.03.03.02				Pruebas de funcionamiento
02.04		Revisión final y documentación. Entrega PEC2		
03	Desarrollo de sistema de visualización (PEC3)			
03.01		Modificación de PIN tool para exportar a BBDD		
03.02		Crear sistema de representación Web		
03.03		Revisión final y documentación. Entrega PEC3		
04	Memoria y presentación TFC (FINAL)			
04.01		Elaboración Memoria proyecto		
04.02		Elaboración Presentación proyecto		
04.03		Entrega Memoria y Presentación		

### 2.1.1 Planificación TFG (PEC1)

El presente documento es la ejecución de esta tarea. Mediante la ejecución de la misma, hemos definido el alcance del proyecto, se han determinado las tareas necesarias y se ha establecido una planificación para la ejecución de los diferentes trabajos.

### 2.1.2 Estudio Apache Spark, desarrollo de Pintool para profiling (PEC2)

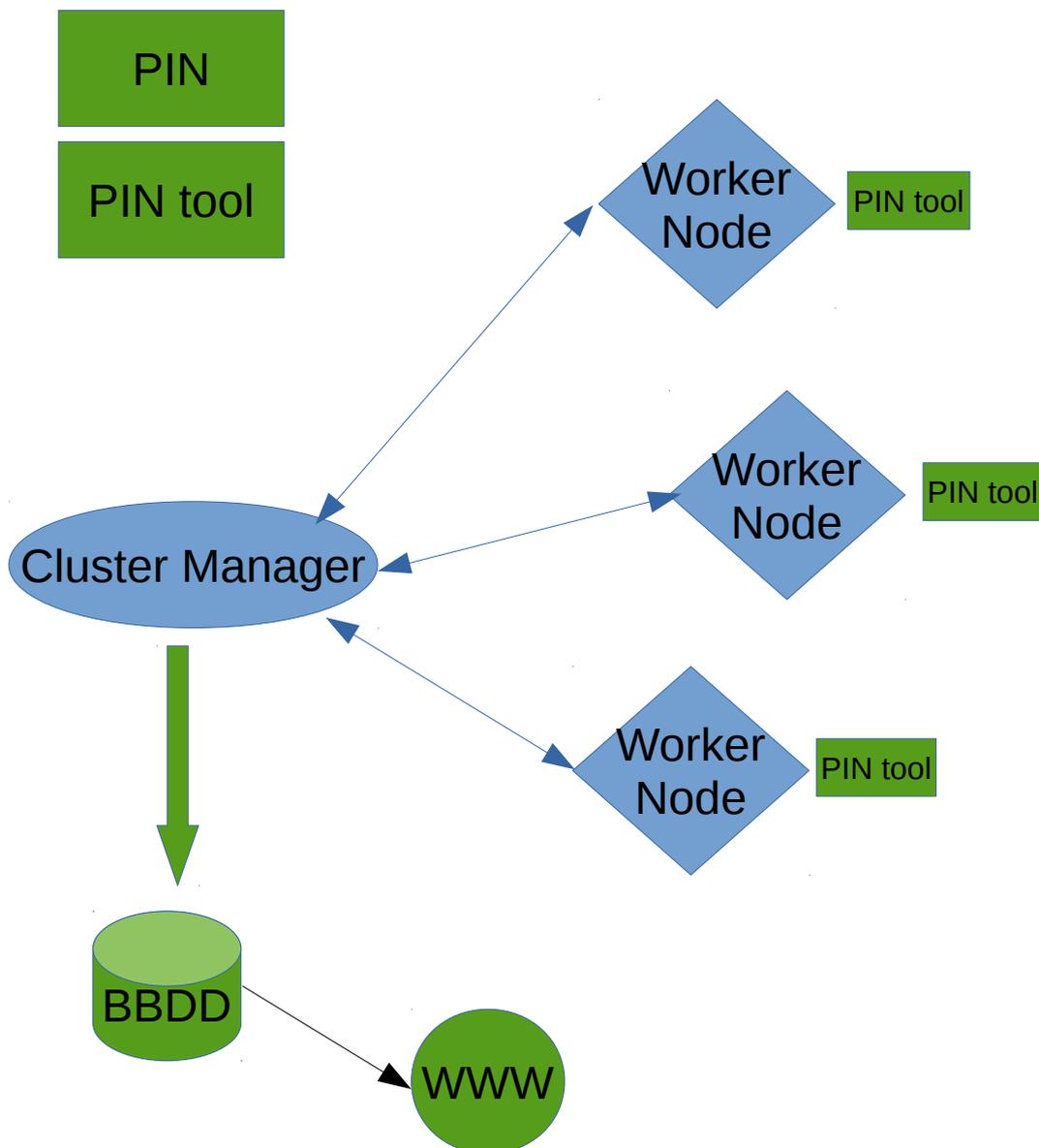
Esta tarea es realmente el núcleo del proyecto. En ella definimos las siguientes actividades:

- Análisis del funcionamiento de Apache Spark
- Análisis del funcionamiento de Pintool y capacidades

- Diseño del modelo para obtener el rendimiento de los trabajos Spark ejecutados en el cluster.
- Construcción de un modelo de Pintool para obtener información en tiempo real sobre las tareas Spark
- Codificación en C++ de pintool
- Pruebas de integración

La arquitectura resultante que tenemos que construir corresponde a este diagrama

*Diagrama 1. Arquitectura*



### **2.1.3 Desarrollo de sistema de visualización (PEC3)**

Como fase final del proyecto, nos quedaría aplicar los diseñado a un entorno de ejecución real. Las actividades a realizar serían

- Diseño de una solución para representación visual en tiempo real de la información extraída por PIN.
- Modificación de PIN Tool para que la información resultante se almacene en un medio persistente a efecto de su representación
- Revisión final de aplicación y documentación

### **2.1.4 Memoria y Presentación TFG**

Esta tarea incluye la realización del documento final de memoria del TFG y la realización de una presentación del TFG. Con la finalización de esta tarea, se daría por finalizado el proyecto al cumplir todos los objetivos del mismo.

## **3 Planificación**

### **3.1 Calendario**

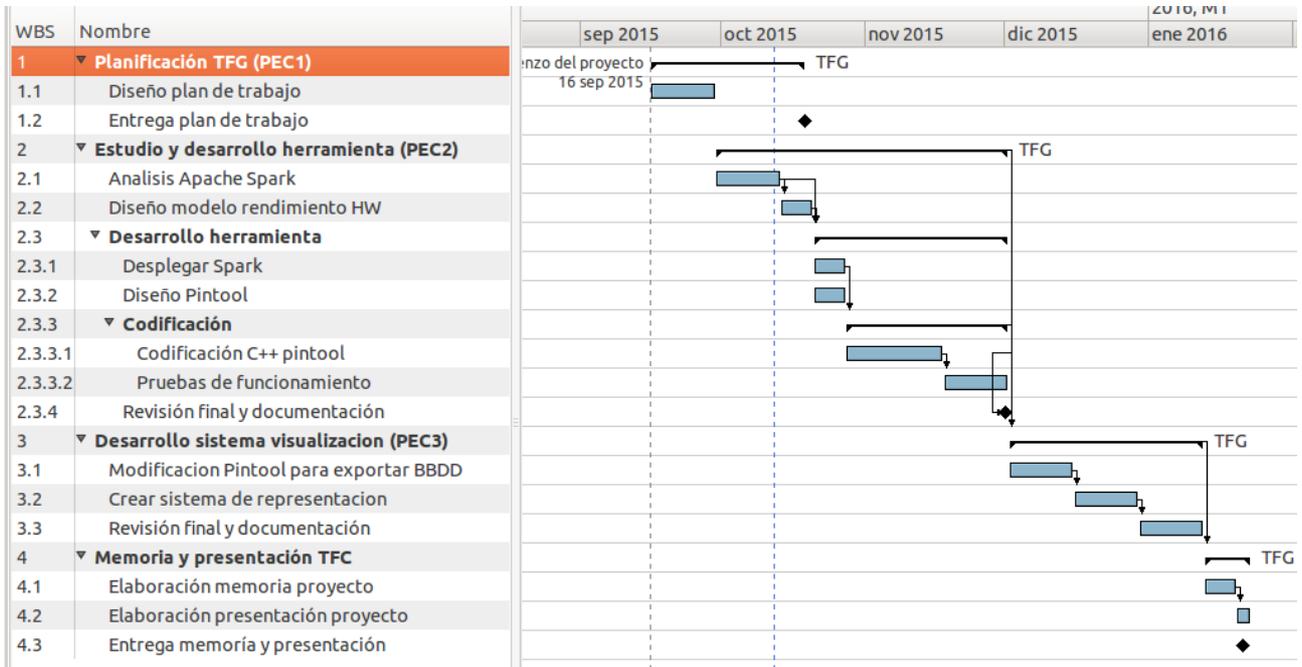
Según la planificación indicada en el aula, la fecha inicial coincide con la fecha de inicio del primer semestre del curso 2015/2016-1, mientras que la fecha para la entrega de la memoria y la presentación es el 21 de enero

### **3.2 Planificación**

La planificación propuesta es la siguiente

Podemos observar que los trabajos comienzan el 16 de septiembre, con la PEC1 y la elaboración del actual documento, y terminaría en 21 de enero con la entrega de la memoria y la presentación del proyecto.

**Diagrama 2. Planificación**



### 3.3 Esfuerzo estimado

Para la estimación de esfuerzos se establece una dedicación semanal 8 horas por cada 7 días naturales (una semana). Es el tiempo que teniendo en cuenta la complejidad del proyecto y los conocimientos con los que cuento a priori, a efecto del desarrollo de los trabajos.

Podemos ver en la siguiente tabla una estimación del esfuerzo para cada una de las tareas a desarrollar en el proyecto.

Tarea	Horas de dedicación aproximadas
Planificación TFC (PEC1)	12
Estudio Spark y desarrollo de Pintool (PEC2)	60
Herramienta de visualización (PEC3)	24
Memoria y presentación TFC	6

## 4 Desarrollo del proyecto

### 4.1 Apache Spark

Dado que estamos construyendo una herramienta para instrumentar trabajos en un entorno Apache Spark, he tenido que investigar y estudiar el funcionamiento de Apache Spark para modelar la solución. De la misma forma, he tenido que construir un cluster Apache Spark sobre el que realizar el conjunto de pruebas y validar la solución que he desarrollado para este proyecto.

#### 4.1.1 Estudio y funcionamiento y primeros pasos

Tal como podemos leer en la documentación, Apache Spark es un motor para el procesamiento masivo de información. Este motor es compatible con Hadoop<sup>vi</sup> y creo que competencia del mismo, ofreciendo algunas mejoras significativas que nos pueden hacer decantarnos sobre esta implementación.

- Aumento de rendimiento respecto a ejecución sobre entornos puros Hadoop
- Programación de aplicaciones paralelas con lenguajes de alto nivel como Scala<sup>vii</sup>, Python<sup>viii</sup> y R<sup>ix</sup>
- Capacidad de de combinar múltiples fuentes de datos de forma sencilla: SQL, machine learning, etc.

Apache Spark puede ser desplegado de diferentes formas:

- Standalone en forma de cluster
- Sobre Mesos/Yarn clusters
- Sobre Hadoop
- Funcionar de forma independiente en un único nodo

Para este proyecto hemos optado por desplegar una solución de cluster independiente, donde tenemos múltiples nodos para ejecutar las tareas que se les encargan.

Uno de los nodos que se denomina “Master” se encarga de recibir el trabajo, repartirlo entre los nodos esclavos, realizar el control de la ejecución y una vez terminada, mostrar los resultados.

A efecto de los trabajos desarrollados, hemos optado por la solución independiente porque nos permitía un mayor control de las modificaciones que hemos tenido que realizar a Spark para poder integrar el proceso de orquestación en el mismo.

Con seguridad las otras opciones existentes de despliegue de Spark pueden ser usadas pero no hemos validado este proyecto con las mismas, con lo que no sabemos con exactitud si las modificaciones que hemos realizado en Spark son suficientes o si es necesario realizar alguna

adaptación adicional.

Durante la fase de estudio hemos tenido que implementar un cluster pues de Apache Spark, con diferentes nodos que hemos configurado como maestro (Master) y esclavos (Slaves).

Dado que la configuración de esta arquitectura necesita de múltiples máquinas servidoras, nos hemos servido de un proveedor de infraestructura en la nube en formato IAAS para el despliegue la instalación. Esto nos ha permitido modelar un cluster complejo con varios nodos sin necesidad de adquirir hardware físico (que no sería factible dado que este es un proyecto de fin de estudios y por tanto no cuenta con presupuesto asignado para su realización).

Como sabemos, un proveedor IAAS es un proveedor de infraestructura que en nuestro caso, nos ha entregado sistemas operativos Linux Ubuntu con diferentes configuraciones de Hardware para los dos roles que vamos a desplegar.

La selección del proveedor IAAS dado que este es un proyecto experimental creo que no tiene una importancia relevante en el proyecto, y por eso no hemos realizado un estudio de capacidades o necesidades para la elección del proveedor. Dado que yo ya contaba con conocimientos y experiencia profesional en la solución de Amazon AWS<sup>x</sup>, he optado por usarla directamente.

Para la realización del proyecto he contratado recursos EC2 (Elastic Computing) que se resumen en

- 1 servidor dedicado para soportar el rol de Maestro y ser la máquina principal de compilación y desarrollo
- 6 servidores dedicados para soportar el rol de esclavo y donde realmente se ejecutan las tareas lanzadas en el cluster Spark

Aunque los sistemas y proveedores en la nube pueden ofrecer multitud de funcionalidades avanzadas, para este proyecto nos hemos centrado únicamente en la provisión de máquinas virtuales base, con sistemas operativos standard. Por ejemplo, Amazon ofrece una plantilla con Apache Spark configurado pero dado que no conozco como está implementada la solución, he preferido usar una máquina base limpia (S.O. Ubuntu) sobre la que hemos desplegado nuestra solución.

En primer lugar realice el despliegue de una máquina principal, sobre la que descargue y configure todo el software. Una vez realizada esta tarea, convertí la máquina a una plantilla AMI sobre la que desplegué el resto de nodos esclavos.

En caso de necesitar alguna modificación adicional de código o traspaso de información, usé SSH y SCP para transferir datos entre las máquinas.

Diagrama 3. Captura de pantalla instancias EC2

The screenshot shows the AWS Management Console interface for the EC2 Instances page. The top navigation bar includes the AWS logo, 'AWS', 'Services', and 'Edit' menus. The user's name 'Jose Manuel Garcia Sanchez' and the region 'Ireland' are displayed. The left sidebar contains navigation options for EC2 Dashboard, Events, Tags, Reports, Limits, INSTANCES (with 'Instances' selected), Spot Requests, Reserved Instances, Commands, Dedicated Hosts, IMAGES (with 'AMIs' and 'Bundle Tasks' listed), and ELASTIC BLOCK STORE (with 'Volumes' and 'Snapshots' listed). The main content area features a 'Launch Instance' button, 'Connect', and 'Actions' dropdown. A search bar allows filtering by tags and attributes. Below the search bar is a table of instances with columns for Name, Instance ID, Instance Type, Availability Zone, Instance State, Status Checks, Alarm Status, and Public DNS. The table lists seven instances, all in a 'stopped' state. The instance 'NODO1' is highlighted. Below the table, there is a scroll bar and a prompt to 'Select an instance above'.

<input type="checkbox"/>	Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS
<input type="checkbox"/>	PRINCIPAL	i-174b4bb6	c4.large	eu-west-1b	stopped		None	ec2-52-18-255-71.eu
<input type="checkbox"/>	NODO2	i-ada5d226	t2.medium	eu-west-1b	stopped		None	ec2-52-31-251-159.e
<input checked="" type="checkbox"/>	NODO1	i-d995d060	t2.medium	eu-west-1b	stopped		None	ec2-52-31-34-35.eu-1
<input type="checkbox"/>	NODO3	i-f0d3957b	t2.medium	eu-west-1b	stopped		None	ec2-52-17-95-79.eu-1
<input type="checkbox"/>	NODO4	i-fdd39576	t2.medium	eu-west-1b	stopped		None	ec2-52-48-87-136.eu
<input type="checkbox"/>	NODO5	i-fed39575	t2.medium	eu-west-1b	stopped		None	
<input type="checkbox"/>	NODO6	i-ffd39574	t2.medium	eu-west-1b	stopped		None	

A efecto de organizar la distribución de información en los nodos, establecí esta configuración de directorios:

- /servers como punto de montaje principal para todo el código y productos
- /servers/pin para el motor de PIN y la PIN tool desarrollada
- /servers/production para binarios de producción
- /servers/production/spark-production para despliegue de Apache Spark de producción
- /servers/development para desarrollo y test
- /servers/development/sparkanalyzer para todo el código desarrollado

El despliegue de cluster de Hadoop o de Spark puede necesitar la existencia de sistemas de ficheros compartidos para disponer de un repositorio de información común fácilmente accesible por todos los nodos que componen el cluster. En mi caso y para este proyecto de investigación he optado por no usarlo y simplificar la instalación de los sistemas. Por ejemplo, para la fase de pruebas y validación he copiado en local en cada nodo el conjunto de datos a estudiar. Esto en un sistema en producción no suele ser la solución, es necesario tenerlo en cuenta para reproducir este proyecto.

Como hemos visto en la figura 1, el conjunto de máquinas que hemos desplegado no es despreciable y en Amazon AWS el coste de las mismas depende de la configuración, los recursos asignados y si la máquina está encendida o apagada. Para controlar el coste (remitiéndome de nuevo a que este es un proyecto fin de estudios y que no cuenta con presupuesto, he optado por minimizar el numero de nodos encendidos y solamente arrancar la infraestructura cuando está realizando un determinado trabajo). Esta aproximación de arrancar solamente los nodos de trabajo es como se funciona realmente en un entorno Cloud, maximizando la potencia cuando es necesario y minimizando el coste cuando no está realizando tareas.

Así, durante el proyecto he tenido arrancada la máquina principal aproximadamente 8 horas diarias cinco días a la semana, y las máquinas esclavas (que en la figura 1 se denominan como nodos) únicamente durante las pruebas, que podrían ser varias horas a la semana (como máximo 8). El coste aproximado mensual de esta configuración ha resultado en unos 50€/mes, con un total aproximado de 200€ para la realización total del proyecto (teniendo en cuenta el tiempo total de ejecución desde el comienzo del curso a la entrega del PFG).

La configuración de memoria, CPU y disco por supuesto en un sistema en producción si debe ser estudiada adecuadamente. Para el proyecto únicamente he buscado un balance adecuado entre coste por hora y potencia. El nodo principal si que he tenido que dimensionarlo un poco mas grande (8GB de RAM y 8 CPU) porque en la fase de compilación y modificaciones de Spark tuve que realizar esta tarea decenas de veces y el tiempo empleado en las compilaciones (de aproximadamente 30 minutos cada vez) hizo que me planteara aumentar esta capacidad para avanzar mas rápidamente).

## 4.1.2 Despliegue e instalación. Compilación.

Como hemos mencionado, el despliegue de los productos lo hemos realizado sobre el directorio /servers.

En esta figura podemos ver una captura del directorio del nodo principal

*Diagrama 4. Captura de pantalla nodo principal*

```
System information as of Tue Jan  5 10:29:00 UTC 2016

System load: 0.0          Memory usage: 2%    Processes:      89
Usage of /:  41.1% of 7.74GB  Swap usage:  0%    Users logged in: 0

Graph this data and manage this system at:
  https://landscape.canonical.com/

Get cloud support with Ubuntu Advantage Cloud Guest:
  http://www.ubuntu.com/business/services/cloud

147 packages can be updated.
70 updates are security updates.

Last login: Sat Jan  2 10:13:06 2016 from cm-93-156-249-99.telecable.es
ubuntu@principal:~$ sudo su
root@principal:/home/ubuntu# cd /servers/
root@principal:/servers# ls
backup      development  pin          snipersim
data        lost+found  pin-2.14-71313-gcc.4.4.7-linux  test
Descargas  NPB         production
```

La compilación de Apache Spark se realiza mediante Maven<sup>xi</sup> y en nuestro caso, hemos aprovechado el script de compilación que el propio Spark incorpora y que permite automatizar la generación de distribuciones.

Es importante remarcar este punto, realmente nuestro proceso de instrumentación necesita de un Spark modificado en el que nosotros introducimos el parser de PIN.

La compilación en mi caso la he realizad con la ejecución de

```
./make-distribution.sh --name pin-Spark --tgz
```

Este fichero que genera es el que debemos distribuir a los nodos que conformen el cluster de spark.

### 4.1.3 Integración con PIN Tool

El proceso de integración de Spark con la PIN Tool ha sido bastante laborioso y con mucho proceso de ensayo/error.

El planteamiento del proyecto como mencionamos en la PEC1 era buscar una forma de introducir PIN en la ejecución de las tareas, de tal forma que pudiéramos instrumentar ese trabajo para extraer información del mismo.

A efecto de saber donde debíamos atacar, tuve que analizar el flujo de ejecución de una tarea desde que se lanza vía comandos hasta que finaliza. Hemos de tener en cuenta que Spark está realizado en Java y Scala, con lo que analizando el código fuente del producto podíamos de alguna forma intentar averiguar en que lugar es mas apropiado introducir el parser de PIN para nuestra instrumentación. Con esta idea en mente y poco a poco, siguiendo la ejecución e introduciendo código debug en el código fuente original, realizando compilaciones y pruebas, se puede trazar las clases y métodos de la ejecución del trabajo.

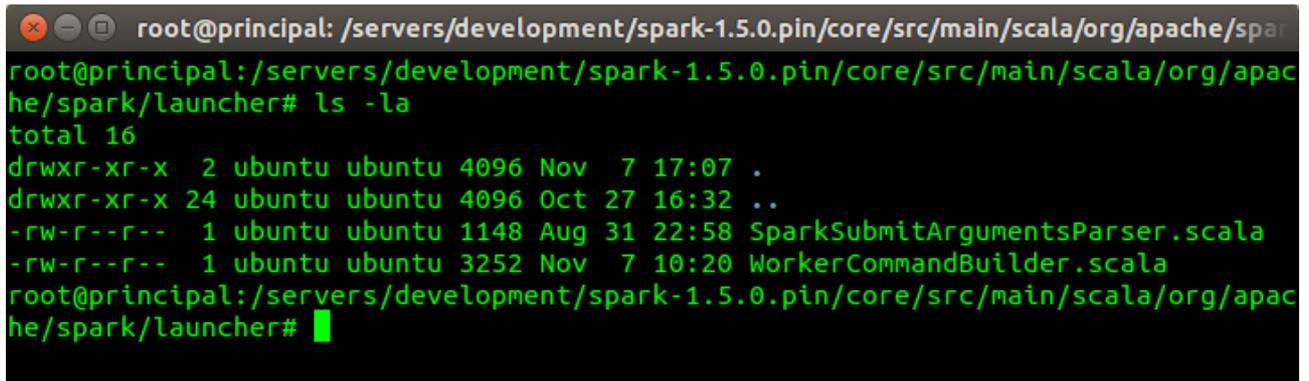
Me gustaría indicar que PIN puede funcionar en modo followexec, lo que permite lanzarlo en la ejecución general de Spark (imaginemos, en el proceso padre) y seguir todo child o thread que se lanzara a partir del padre. Este modo sin embargo lo que provocaría es que nuestra instrumentación contabilizara absolutamente todo el código ejecutado por el proceso padre, y no solamente el trabajo en particular. Por ejemplo, Spark nos ofrece un interfaz web para gestión de los trabajos y si instrumentamos desde el padre, la instrumentación también analizaría esta parte.

En este proyecto he buscado integrarnos totalmente en Spark de tal forma que fuéramos capaces de instrumentar únicamente el job en particular que se lanza en el cluster, y no todos los servicios de infraestructura generales. Esto nos permite tener un conocimiento muy profundo de que está pasando en la tarea o trabajo en cada uno de los nodos.

Como hemos comentado entonces, el sitio donde hemos modificado el código para incluir nuestro soporte es el fichero `WorkerCommandBuilder.scala` del código fuente de Spark. Este es el lugar que hemos elegido porque realmente es donde se genera el comando que se lanza vía Shell en los worker cuando reciben un trabajo. Digamos que se genera una línea que es la que se ejecuta, y si somos capaces de agregar información a esa línea, podremos incluir nuestro parser a PIN.

La ubicación del fichero en el código fuente de Spark es esta:

## Diagrama 5. Ubicación código Spark



```
root@principal: /servers/development/spark-1.5.0.pin/core/src/main/scala/org/apache/spark
root@principal: /servers/development/spark-1.5.0.pin/core/src/main/scala/org/apache/spark/launcher# ls -la
total 16
drwxr-xr-x  2 ubuntu ubuntu 4096 Nov  7 17:07 .
drwxr-xr-x 24 ubuntu ubuntu 4096 Oct 27 16:32 ..
-rw-r--r--  1 ubuntu ubuntu 1148 Aug 31 22:58 SparkSubmitArgumentsParser.scala
-rw-r--r--  1 ubuntu ubuntu 3252 Nov  7 10:20 WorkerCommandBuilder.scala
root@principal: /servers/development/spark-1.5.0.pin/core/src/main/scala/org/apache/spark/launcher# █
```

El fichero en si mismo no tiene una gran complejidad. La única problemática para mi fue que está construido en Scala y este lenguaje en mi caso era desconocido. Por tanto, dentro de la realización del proyecto también he tenido que aprender en la medida de lo posible programación Scala para realizar las adaptaciones.

En el caso de las modificaciones, he intentado parametrizar en la medida de lo posible todo a ficheros de configuración de tal forma que no sea necesario modificar el código Spark mas que una vez, y que se pueda cambiar la configuración de la instrumentación vía fichero externo en vez de tener que recompilar y distribuir Spark cada vez que se cambie un comportamiento.

Para esto, he usado un fichero `/etc/pinSpark/pinSpark.conf`. Este fichero se lee en el proceso de construcción del comando del Worker para generar dinámicamente en base a los valores de configuración la línea de comando exacta.

Las modificaciones del fichero se resumen en:

### ***Importamos un conjunto de librerías para facilitar la gestión del fichero de configuración.***

La primera nos permite acceder a objetos config (pares de valores del tipo Key = Value) y la segunda se una librería estandar de generación de aleatorios (la instrumentación genera un log local y para evitar duplicidades construimos un numero aleatorio)

```
/** Imports for pinSpark */
import com.typesafe.config.{ Config, ConfigFactory }
import scala.util.Random
```

### ***Modificar el método buildCommand para incluir nuestro código***

Lo que vamos en realizar realizando es agregando a un objeto Lista nuestro comando personalizado, que vamos construyendo con los valores leídos del fichero de configuración. Tenemos también una función personalizada para la generación del nombre aleatorio del fichero de log en particular para

ese trabajo. Cada trabajo en cada worker tiene un nombre diferente, dado que se lanza por worker (si un job se ejecuta en el cluster de 4 nodos, tendremos un fichero local de debug por nodo y un nombre de fichero de log independiente).

### Diagrama 6. Modificación Scala

```
cmd.add(s"-Xmx${memoryMb}M")
command.javaOpts.foreach(cmd.add)
addPermGenSizeOpt(cmd)
addOptionString(cmd, getenv("SPARK_JAVA_OPTS"))
val configPin = ConfigFactory.parseFile(new File("/etc/pinSpark/pinSpark.conf"))
val cmd2: List[String] = List(configPin.getString("pinCommand"))
val cmd21 = List.concat(cmd2, List(configPin.getString("pinToolSwitch")))
val cmd22 = List.concat(cmd21, List(configPin.getString("pinTool")))
val cmd23 = List.concat(cmd22, List(configPin.getString("pinOutSwitch")))
//Generate unique filename, prefix from config file, path fixed on /tmp
val logfile = randomFileName(configPin.getString("pinOutLogFile"), "log", configPin.getString("pinComputer
Name"), 10, 5)
val cmd24 = List.concat(cmd23, List(logfile))
val cmd25 = List.concat(cmd24, List(configPin.getString("pinAttachSwitch")))
println(cmd25)
val cmd3 = List.concat(cmd25, cmd)
println(cmd3)
cmd3
}

def randomFileName(prefix: String = "", suffix: String = "", computer: String = "", maxTries: Int = 10, na
meSize: Int = 10) = {
  val alphabet = ('a' to 'z') ++ ('A' to 'Z') ++ ('0' to '9') ++ ("_-")
  def generateName = (1 to nameSize).map(_ => alphabet(Random.nextInt(alphabet.size))).mkString
  val name1 = "/tmp/" + prefix + "_" + computer + "_" + generateName + "." + suffix
  name1
}
```

Un ejemplo del comando generado puede verse aquí (extraído del log de un worker Spark).

Como vemos lo primeros comandos son generados por nosotros e indicados como parser a PIN (antes del carácter -). Lo que aparece a posteriori corresponde al comando estandar de Spark.

```
INFO ExecutorRunner: Launch command: "/servers/pin/pin.sh" "-t"
"/servers/pin/source/tools/pinSpark/obj-intel64/pinSpark.so" "-o"
"/tmp/pinSpark_principal_SAI5d.log" "--" "/usr/lib/jvm/java-8-oracle/jre/bin/java" "-cp"
"/servers/production/spark-1.5.0-bin-pin-Spark/sbin/./conf:/servers/production/spark-1.5.0-bin-
pin-Spark/lib/spark-assembly-1.5.0-hadoop2.2.0.jar" "-Xms2048M" "-Xmx2048M" "-
Dspark.driver.port=42539" "org.apache.spark.executor.CoarseGrainedExecutorBackend" "--driver-
url" "akka.tcp://sparkDriver@10.0.0.108:42539/user/CoarseGrainedScheduler" "--executor-id" "0"
"--hostname" "10.0.0.108" "--cores" "2" "--app-id" "app-20160102101417-0000" "--worker-url"
"akka.tcp://sparkWorker@10.0.0.108:39526/user/Worker"
```

Vemos a continuación el contenido del fichero pinSpark.conf en la parte que toca a la integración con Spark (el fichero contiene mas datos de configuración que revisaremos en el apartado dedicado al diseño de la Tool)

*Diagrama 7. pinSpark.conf*

```
####Instrumentation configuration
##Base values
pinCommand = /servers/pin/pin.sh
pinToolSwitch = -t
pinTool = /servers/pin/source/tools/pinSpark/obj-intel64/pinSpark.so
pinOutSwitch = -o
pinOutLogFile = pinSpark
pinAttachSwitch = --
pinComputerName = principal
```

Los cuatro primeros valores realmente se van concatenando uno a uno y uno tras otro para generar la línea de comando. Si los modificamos y ponemos otra cosa, se generaría un comando de tipo  
\$pinCommand \$pinToolSwitch \$pinTool \$pinOutSwitch

Con este tipo de configuración podemos jugar y tener libertad de integración con otras herramientas si queremos.

PinOutLogFile y pinComputerName se usan con la función para generar el nombre aleatorio, de tal forma que sería el siguiente valor después de \$pinOutSwitch.

Así, el flujo quedaría como

```
$pinCommand $pinToolSwitch $pinTool $pinOutSwitch  
funcion(pinOutLogFile,pinComputerName) $pinAttachSwitch
```

Como vemos creo que nos permite una gran libertad si queremos cambiar el comportamiento de la integración con Spark. Los nombres realmente son descriptivos para una mejor comprensión, pero podrían ser param1, param2, etc.

## 4.2 *PIN Tool*

Dentro del proyecto indicamos como un objetivo disponer de una herramienta de análisis de bajo nivel del funcionamiento de Spark.

El propio Apache Spark cuenta con diferentes herramientas (propias y de terceros) que permiten extraer métricas de rendimiento del sistema.

¿Cual es entonces el interés de agregar una capa adicional de instrumentación?. Pues precisamente la capacidad que tenemos instrumentando de analizar el comportamiento a bajo nivel de una determinada ejecución. En este caso, y tal como hemos orientado el proyecto, creo que resulta sumamente interesante saber como se comporta un Worker a bajo nivel cuando le solicitan un trabajo.

Por supuesto, hemos de tener en cuenta que Apache Spark realmente está pensado para soportar un crecimiento scale-out pero aun así, en la solución de determinados problemas, puede ser interesante conocer el comportamiento a bajo nivel del Job para optimizar la ejecución del mismo.

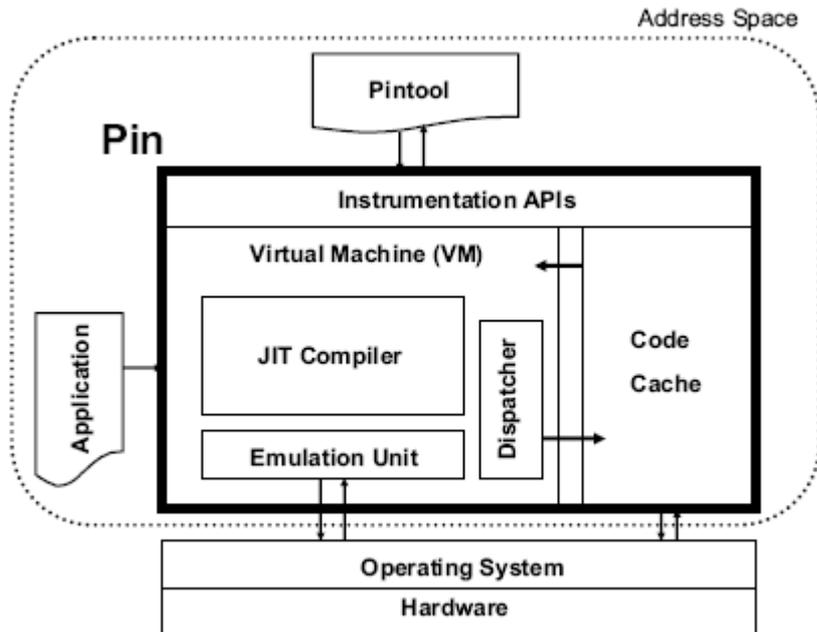
PIN es un software de instrumentación para arquitecturas x86-64, IA-32 e IA-64. Nos permite realizar análisis en tiempo de ejecución en archivos binarios. Esto significa que no es necesario recompilar las aplicaciones, y podemos obtener los análisis de aplicaciones en espacio de usuario sobre entornos Linux y Windows.

PIN está desarrollado<sup>xii</sup> y soportado por Intel, y está permitida su utilización sin coste para uso no comercial<sup>xiii</sup>.

### ***Funcionalidades de PIN***

- Soporta dos modos de instrumentación: Just-In-Time (o JIT)<sup>xiv</sup> y Probe. JIT ofrece la máxima funcionalidad, mientras que el modo Probe permite un conjunto limitado de funciones con una mayor velocidad de ejecución.
- Independencia de la plataforma: PIN está diseñado y programado siguiendo criterios de portabilidad.
- Muy eficiente en el proceso de attach y detach de los procesos en ejecución: PIN tiene la capacidad de capturar un proceso, extraer la información de funcionamiento y traza haciendo la instrumentación, para desconectarse a posteriori. Esta capacidad para conectarse y desconectarse es fundamental en la instrumentación de aplicaciones grandes<sup>2</sup>.

**Diagrama 8. Arquitectura PIN**



PIN consiste en una máquina virtual (VM), una cache y la API sobre la que funcionan las herramientas que trabajan con PIN. La máquina virtual como vemos consta del compilador JIT, un sistema de emulación y un dispatcher para comunicar con la cache.

Una vez que PIN toma control de una aplicación en ejecución, la VM coordina dicha ejecución para realizar el proceso de instrumentación. El JIT compila e instrumenta el código de la aplicación, que es pasado vía el dispatcher a la cache de código. El emulador permite que PIN ejecute instrucciones como llamadas al sistema, que no pueden ser tratadas de forma normal, y que deben ser pasadas directamente al sistema operativo.

### 4.2.1 Diseño

Dado que este es un proyecto que creo que tiene vocación de investigación (mas que generar un producto) el diseño de las funcionalidades de la PIN Tool creo que es menos importante que la generación de una tool que nos sirva de prueba de concepto de las capacidades que queremos explotar. Es evidente que la construcción de una tool para instrumentación de un determinado problema en Spark debería de conocer el problema a modelar e ir afinándose con el análisis de las

ejecuciones (en un proceso iterativo y continuo hasta disponer de la tool final).

Así, el diseño de mi PIN Tool es ciertamente genérico aunque incorpora una serie de características que considero básicas y que pueden ser aprovechadas como esqueleto para la construcción de otras tool que realicen otros análisis.

Los requisitos de diseño por tanto se basan mas que en un problema, en mis requisitos como desarrollador del proyecto.

- Uso de ficheros de configuración para almacenar información dinámica o cambiante. Debemos evitar el uso de variables estáticas y maximizar el uso de variables de configuración que se lean en ejecución para configurar la instrumentación. Mediante esta técnica evitamos tener que realizar compilaciones adicionales de la tool y simplificamos la integración con Spark.
- Granularidad en la ejecución de la instrumentación. El proceso de instrumentación impone un importantísimo overhead en la ejecución y debemos tenerlo en cuenta, permitiendo activar o desactivar (en la medida de lo posible) la instrumentación de determinadas funciones para favorecer la rapidez de ejecución.
- Control de la salida de la instrumentación para garantizar una adecuada gestión de las métricas. Dado que podemos realizar instrumentación en decenas de servidores simultáneos, tenemos que tener en cuenta como vamos a gestionar ese output para disponer, una vez finalizada la ejecución, de un resultado coherente y fiable.

Como vemos no especifico como requisito tal o cual métrica porque creo que desde el punto de vista del proyecto, es algo secundario. No obstante en esta tool voy a contemplar

- Contabilización de operaciones CPU: instrucciones, bloques, threads
- Contabilización de operaciones memoria: lecturas, escrituras, malloc y free
- Contabilización de operaciones en disco: Lecturas y escrituras (bytes)
- Contabilización de operaciones de red: Lecturas y escrituras (bytes)

Como nota importante, en la parte de operaciones en disco y dado que un binario puede leer multitud de datos no interesantes en la ejecución de la tarea Spark (por ejemplo, una librería), considero como requisito poder filtrar el directorio desde el que queremos que se contabilice el acceso a disco. Así, por ejemplo, podremos saber la cantidad de información que lee o escribe un job Spark del conjunto de datos de trabajo, para contabilizar y analizar la efectividad de dicho job.

## 4.2.2 Codificación

La pintool se programa usando el framework proporcionado por PIN que se basa en código C/C++.

En mi caso particular tengo tendencia a usar funciones C y estilo de programación C dado que es uno de mis lenguajes favoritos, aunque la pintool realmente está mezclando ambos tipos.

Pasamos pues a analizar cada una de las funciones principales de la pintool.

*int main(int argc, char \*argv[])*

Función principal. En ella preparamos la instrumentación leyendo del fichero de configuración los valores de como vamos a realizar la instrumentación

- Datos para conexión a BBDD para output
- Indicador de que y como vamos a instrumentar (se puede indicar en fichero de configuración)
- Inicialización de variables de estructura (array para almacenar e identificar ficheros interesantes para instrumentar las llamadas de sistema write y read)

El fichero de configuración está hardcoded a `/etc/pinSpark/pinSpark.conf` tal como hemos indicado anteriormente. En un capítulo anterior del documento hablamos de los parámetros que incluye el fichero referidos a la generación de la línea de comandos para la integración con Spark. Adicionalmente a estos, también tenemos parámetros de configuración que afectan a otras variables, y que realmente son genéricos (es decir, podríamos instrumentar cualquier binario con esta pintool, no es imprescindible que sea un trabajo Spark).

El fichero de configuración contiene por tanto tres partes adicionales

- Control de que cosas quiero instrumentar
- Control de desde que directorio quiero instrumentar
- Control de donde voy a almacenar la información de salida (aparte del fichero de log local)

Diagrama 9. *pinSpark.conf* (continuación)

```
##Choose values to instrument
##Always instrument ins, bbl and thread count
ifMalloc = 1;
ifFree = 1;
ifBarrier = 1;
ifMemoryRead = 1;
ifMemoryWrite = 1;
#Syscall
ifWrite = 1;
ifRead = 1;
ifRecv = 1;
ifSend = 1;
ifOpen = 1;
ifClose = 1;

##Syscalls specific configuration
#dirForSyscalls = /servers/production/spark-production
dirForSyscalls = /

####DB Store configuration
mysqlServer = principal
mysqlDatabase = sparkanalyzer
mysqlUser = sparkanalyzer
mysqlPassword = localpass
mysqlWorkerID = 0
mysqlWorkerIP = 10.0.0.108
```

Los primeros valores nos permiten decidir si se va a instrumentar una determinada métrica. Si no está a valor 1, no se instrumenta.

La variable `dirForSysCalls` es la que hemos mencionado que controla que ficheros son los interesantes para instrumentar las syscalls de read/write. Si no está en bajo esa raíz, no lo contabilizamos.

Las últimas variables hacen referencia a los datos de configuración para almacenar el output a la BD. Importante destacar que `mysqlWorkerID` debe ser diferente en cada nodo para identificar que genera cada worker. Es un valor que se lee y se inserta como identificador del servidor que instrumenta. Por ejemplo, si tenemos 6 nodos worker, podríamos asignar los valores de 1 a 6 en la configuración local de cada nodo (recordemos que el fichero `pinSpark.conf` es local a cada nodo y cada uno lo procesa de forma independiente. La configuración de todos los ficheros debe ser coherente y acorde a una computación distribuida).

Y por supuesto, la función `main()`, aparte de todo lo mencionado, lanza las funciones de instrumentación propiamente dichas.

### Diagrama 10. Funciones PIN

```
// Register Image to be called to instrument functions.
IMG_AddInstrumentFunction(Image, 0);
TRACE_AddInstrumentFunction(Trace, 0);
PIN_AddThreadStartFunction(ThreadStart, 0);
INS_AddInstrumentFunction(Instruction, 0);

PIN_AddSyscallEntryFunction(SyscallEntry, 0);
PIN_AddSyscallExitFunction(SyscallExit, 0);

// Register function to be called when the application exits
PIN_AddFiniFunction(Fini, 0);
```

### Instrumentación de métricas

La instrumentación de las diferentes métricas que recogemos se realiza en diferentes funciones. Vamos a reflejar en la memoria las partes mas interesantes y en las que hemos tenido que programar una solución ad-hoc para determinados problemas.

Llamadas al sistema<sup>xv</sup>

Una de las partes mas interesantes de la pintool era poder saber como se relaciona el job de spark con el sistema operativo en lo que respecta a intercambio de información. Dado el tipo de tarea para lo que se usa Spark, es evidente que el consumo de memoria, red y acceso a disco puede ser muy considerable por parte de los nodos esclavos.

La forma de que nosotros sepamos que está pasando realmente desde el punto de vista de la instrumentación la he basado en la interceptación de las llamadas al sistema operativo que realiza el job en particular en cada worker.

Una llamada al sistema es una solicitud de un ejecutable de un determinado servicio al sistema operativo. Existen como puede deducirse multitud de llamadas al sistema, cada una con sus particularidades y con un objetivo diferente.

Las llamadas al sistema se diferencia por un identificador y una serie de parámetros que se pasan a la llamada como datos necesarios para la ejecución de la misma.

Desde el punto de vista de la pintool, usamos esta función de PIN para interceptar una llamada al sistema cuando se produce

### Diagrama 11. Catch de llamadas al sistema

```
//Search for syscall
// For O/S's (Mac) that don't support PIN_AddSyscallEntryFunction(),
// instrument the system call instruction.

if (INS_IsSyscall(ins) && INS_HasFallThrough(ins))
{
    // Arguments and syscall number is only available before
    INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(SysBefore),
        IARG_INST_PTR, IARG_SYSCALL_NUMBER,
        IARG_SYSARG_VALUE, 0, IARG_SYSARG_VALUE, 1,
        IARG_SYSARG_VALUE, 2, IARG_SYSARG_VALUE, 3,
        IARG_SYSARG_VALUE, 4, IARG_SYSARG_VALUE, 5,
        IARG_END);

    // return value only available after
    INS_InsertCall(ins, IPOINT_AFTER, AFUNPTR(SysAfter),
        IARG_SYSRET_VALUE,
        IARG_END);
}
```

Como vemos en el código, cuando se produce la llamada al sistema, llamamos a SysBefore() pasando como referencia los datos de la llamada. Una vez que la llamada se ejecuto, nosotros llamaos a SysAfter para saber que ha ocurrido con la misma. Esto es importantísimo porque una cosa es que nosotros solicitemos un servicio al sistema operativo, por ejemplo, leer un dato del disco, y otra diferente que eso realmente se produzca y que si se realiza, sepamos cuantos bytes hemos leído realmente.

A efecto de identificar las syscall me he basado en documentación existente en internet y de las paginas del manual (man) de Linux. En esta url podemos ver un resumen de la definición para x86-64 (importante, las syscall pueden cambiar para 32bit. Dado que mi desarrollo y ejecución sería en x86-64, es lo que he utilizado)

<http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64>

### Diagrama 12. Ejemplo syscalls

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					

El flujo entonces cuando se produce una syscall seguiría con la llamada a Sysbefore(). El código que compone esta función lo que hace es identificar el tipo de Syscall y si es interesante para nosotros, modificar una variable semáforo que luego en el Sysafter() comprobamos para saber donde tenemos que acumular los resultados de la llamada.

Las comprobaciones de si la syscall es interesante o no son ligeramente diferentes, dependiendo del tipo de llamada.

### ***Diagrama 13. Detección de syscall interesante***

```
if ((long)num == 1){
    //Write syscall
    //Test if we instrument
    if (ifWrite==1){

        //Before instrument, check if file is in the array

        *out << "Write number: " << arg0 << endl;
        if (isFDinOpenFiles(arg0)==1){

            *out << "Interesting File" << endl;
            //Ok, we're on a file of interest
            isWrite = 1;
            isWriteFDValue = arg0;
        }
    }
}
```

En primer lugar detectamos el tipo de llamada. Si es read o write, tenemos que verificar que el fichero sea interesante para nosotros. Recordemos que tenemos un parámetro de configuración que nos dice la raíz desde la cual instrumentar estas syscall. Así, entonces, verificamos si la ruta del fichero es interesante. El que sea interesante o no lo decide la syscall open, que tiene que haberse ejecutado antes de leer o escribir en un fichero. En el write/read únicamente confirmamos si ese arg0 que tiene la ruta fue abierto antes y si es interesante. Finalmente, si todo es afirmativo, guardamos en una variable tipo isWrite, isRead, etc. que tipo de syscall estamos lanzando.

**Diagrama 14. Detección tipo syscall**

```
//open
if ((long)num == 2){
    //Open syscall
    //Test if we instrument
    if (ifOpen==1){

        //arg0 store memory address of first parameter, name of file to open
        char * address = (char *) arg0;

        //Comparefile from syscall with conf value

        if (contains(address,dirForSyscalls)==1)
        {
            //we must instrument
            isOpen=1;
            *out << "Open: File is interesting to instrument: " << address<< endl;

        }
    }
}
}
```

La llamada de open también tiene su particularidad. Lo que hacemos es confirmar si el fichero que abrimos está contenido en la raíz interesante para nosotros. Como vemos, hemos creado una función contains que confirmar si es interesante.

El resto de llamadas (close, read/write a la red) son mucho mas simples y en ellas únicamente verificamos el numero de la llamada.

**Diagrama 15. Syscalls de red**

```
if ((long)num == 45){
    //Recv syscall
    //Test if we instrument
    if (ifRecv==1){
        *out << "Recv syscall received: " << arg0 << endl;
        isRecv = 1;
    }
}

if ((long)num == 47){
    //Recvmsg syscall
    //Test if we instrument
    if (ifRecv==1){
        *out << "Recvmsg syscall received: " << arg0 << endl;
        isRecv = 1;
    }
}
```

Llegamos a este punto, hemos pasado por la Syscall y esta se ha ejecutado. Tenemos ahora que recuperar el resultado y ver que hacer con el. Esto se realiza en Sysafter().

Como hemos visto, hemos declarado variables semáforo para saber que tipo de llamada al sistema se realizo y actuar en consecuencia. En Sysafter() vamos simplemente analizando semáforo a semáforo hasta encontrar el que esta verde.

Ojo, es importante conocer que una llamada al sistema devuelve -1 si se produce un error. Como este retorno lo almacenamos en una variable UINT64, se guarda con un valor numérico determinado (sin signo). Este comportamiento nos costo una dedicación importante de debug ya que aunque una vez detectado es por supuesto evidente lo que estaba sucediendo, en tiempo de desarrollo y pruebas simplemente veíamos que alguna variable se desbocaba en tamaño y no sabíamos donde se producía.

Lo primero por tanto es verificar el retorno y si tenemos los semáforos. Si el retorno es -1, no hacemos nada con esa syscall porque es un error, simplemente reseteamos los semáforos.

### ***Diagrama 16. Detección de error (-1) en llamadas al sistema***

```
//Test if syscall() returns -1 error
//UINT64 vars don't store negative values

if (isWrite == 1 || isRead == 1 || isRecv == 1 || isSend == 1 || isOpen == 1 || isClose == 1){
if (ret == 18446744073709551615){

    *out << "Syscall error, return value: " << ret << endl;
    //Syscall is returning an error
    //No instrumentation
    isWrite = 0;
    isRead = 0;
    isRecv = 0;
    isSend = 0;
    isOpen = 0;
    isClose = 0;

}
}
```

Si el retorno es coherente, vamos uno a uno comprobando y acumulando. Realmente esta parte es bastante sencilla, las syscall retornan el numero de bytes operados y es lo que acumulamos al sumatorio de lo que lleva hasta el momento en esa syscall. Por ejemplo, si es una llamada a read(), tendremos que isRead=1 y entonces acumularemos ret al contador para las operaciones Read.

### Diagrama 16. Retorno de llamadas al sistema

```
if (isWrite == 1){
    out << "Add to write value: " << ret<< endl;
    writeSyscallCount+= ret;
    isWrite = 0;
}
if (isRead == 1){
    *out << "Add to read value: " << ret<< endl;
    readSyscallCount+= ret;
    isRead = 0;
}
if (isRecv == 1){
    *out << "Add to isrecv value: " << ret<< endl;
    recvSyscallCount+= ret;
    isRecv = 0;
}
if (isSend == 1){
    *out << "Add to issend value: " << ret<< endl;
    sendSyscallCount+= ret;
    isSend = 0;
}
```

En el caso de las operaciones Open y Close tenemos una particularidad. Estas operaciones usan un descriptor de fichero para identificar los que tienen abiertos (para leer y escribir). Este descriptor es dinámico y reutilizable. Así, el primer fichero abierto tendrá el FD 1, el segundo el FD 2 y así sucesivamente. Si se cierra un fichero, ese FD se va a reutilizar en la siguiente llamada a open.

Para controlar esta casuística, lo que hago es que cuanto en SysAfter tengo una llamada Open interesante, agrego a un array el FD correspondiente. Si luego llega un read o un write de un FD que está en el array de ficheros interesantes, debo contabilizarlo. Si no está en ese array, no me interesa (es lo que hemos visto anteriormente).

### Diagrama 17. Ficheros interesantes

```
if (isOpen == 1){
    //Return value is the file descriptor
    //We need take this file descriptor in account until a close syscall closes file
    //file descriptors value are reused
    addToOpenFiles(ret);
    *out << "Opening file FD: " << ret << endl;
    isOpen = 0;
}
```

Y por supuesto, si llega un close() lo que hacemos es quitar el FD del array de interesantes.

### Diagrama 18. Cierre de fichero

```
if (isClose == 1){
    //Closing File, we need to remove from array of interesting files
    //only remove if in this
    if (isFDinOpenFiles(ret)==1){
        removeToOpenFiles(ret);
        *out << "Closing file FD: " << ret << endl;
    }
    isClose = 0;
}
```

#### 4.2.3 Output de la instrumentación

La salida de la instrumentación se realiza mediante la función `partialLog` y en la función de finalización `Fini`. Dado que el proceso de instrumentación puede tomar varias horas dependiendo del problema y de la cantidad de nodos, he considerado que disponer de información periódica de el estado de las métricas según avanza el proceso puede resultar de ayuda e interesante, e incluso nos permite conocer datos en tiempo real de como avanza el proceso.

La salida de la instrumentación por tanto se va dando periódicamente hasta la finalización. El output periódico es acumulado, es decir, que simplemente vamos exponiendo las métricas que van aumentando según avanza la ejecución.

Las funciones que generan la salida son `partialLog()` y `Fini()`. A efecto de la salida ya hemos comentado que contamos con dos tipos de salida

- Local en un fichero en `/tmp` en cada worker
- Centralizada en una base de datos. En mi caso he optado por incorporar capacidad de escribir en base de datos MySQL a la `pintool`.

La salida local a fichero también la usamos para disponer de información detallada de debug en la `pintool`. La propia `pintool` maneja un objeto salida que uso a lo largo del programa para generar esa información de salida como vemos en este ejemplo

```
*out << "Error on query: " << err << endl;
```

La periodicidad de la información generada la controlo contando el número de instrucciones ejecutadas, y haciendo un partialLog() cada determinado número. Es una forma sencilla de sacar cada X minutos una salida sin complicarnos con timers o programaciones mas complejas.

**Diagrama 19. Salida parcial**

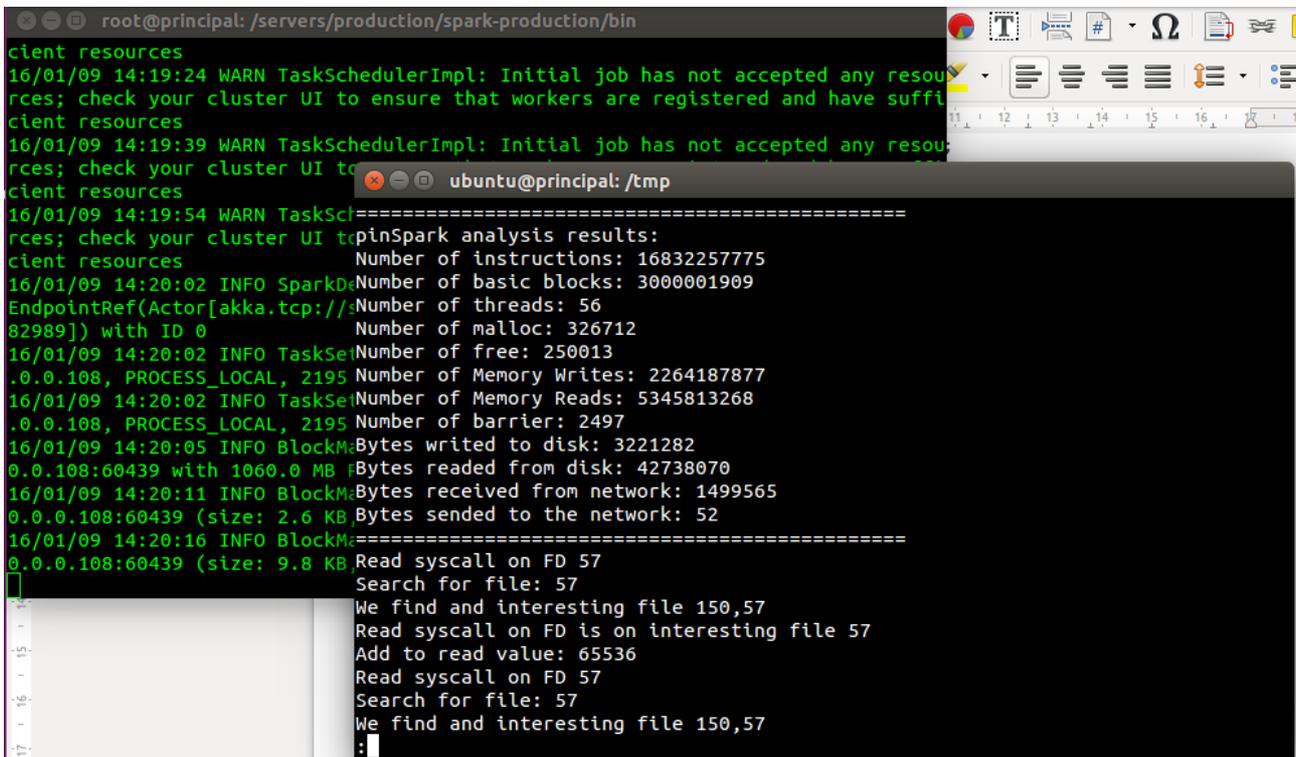
```
VOID CountBbl(UINT32 numInstInBbl)
{
    bblCount++;
    insCount += numInstInBbl;

    if (bblCount % 1000000000 ==0)
    {
        //Get partial statistics

        partialLog();
    }
}
```

Este es un ejemplo de una parte del output de una ejecución

**Diagrama 20. Captura de salida a fichero**



Desde el punto de vista de la salida de la información a la base de datos, he creado un esquema muy simple para almacenar la información. En partialLog y en Fini() entonces genero una cadena insert con los valores de la métrica, que luego inserto en la base de datos mediante la API de MySQL.

```
sprintf(query1,"insert into event(worker,workname,date,
insCount,bblCount,threadCount,mallocCount, freeCount, barrierCount, memoryReadCount,
memoryWriteCount,writeSyscallCount,readSyscallCount,recvSyscallCount,sendSSyscallCount)
values (%s,'%s',NOW(),%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%"
PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ")")
,worker,workname,insCount,bblCount,threadCount, mallocCount, freeCount, barrierCount,
memoryReadCount, memoryWriteCount, writeSyscallCount, readSyscallCount, recvSyscallCount,
sendSSyscallCount);
```

La gestión del acceso al log o a la base de datos la controlo con un mutex para evitar race conditions. Como se puede ver en el INSERT que genero, incorporo

- El ID del worker
- El nombre del worker
- Un timestamp de la instrucción
- Los valores de las métricas acumulados

El esquema de la tabla es el siguientes

**Diagrama 21. Esquema MySQL**

Column	Type	Default Value	Nullable	Character Set	Collation	Privileges	Extra
idevent	bigint(20)		NO			select,insert,update,references	auto_increment
worker	int(11)		YES			select,insert,update,references	
workname	varchar(45)		YES	latin1	latin1_swedish_ci	select,insert,update,references	
date	datetime		YES			select,insert,update,references	
insCount	bigint(20)		YES			select,insert,update,references	
bblCount	bigint(20)		YES			select,insert,update,references	
threadCount	bigint(20)		YES			select,insert,update,references	
mallocCount	bigint(20)		YES			select,insert,update,references	
freeCount	bigint(20)		YES			select,insert,update,references	
barrierCount	bigint(20)		YES			select,insert,update,references	
memoryReadCount	bigint(20)		YES			select,insert,update,references	
memoryWriteCount	bigint(20)		YES			select,insert,update,references	
writeSyscallCount	bigint(50)		YES			select,insert,update,references	
readSyscallCount	bigint(50)		YES			select,insert,update,references	
recvSyscallCount	bigint(50)		YES			select,insert,update,references	
sendSSyscallCount	bigint(50)		YES			select,insert,update,references	

Aparte de los campos citados anteriormente, tenemos un idevent que es el campo clave, autoincremental.

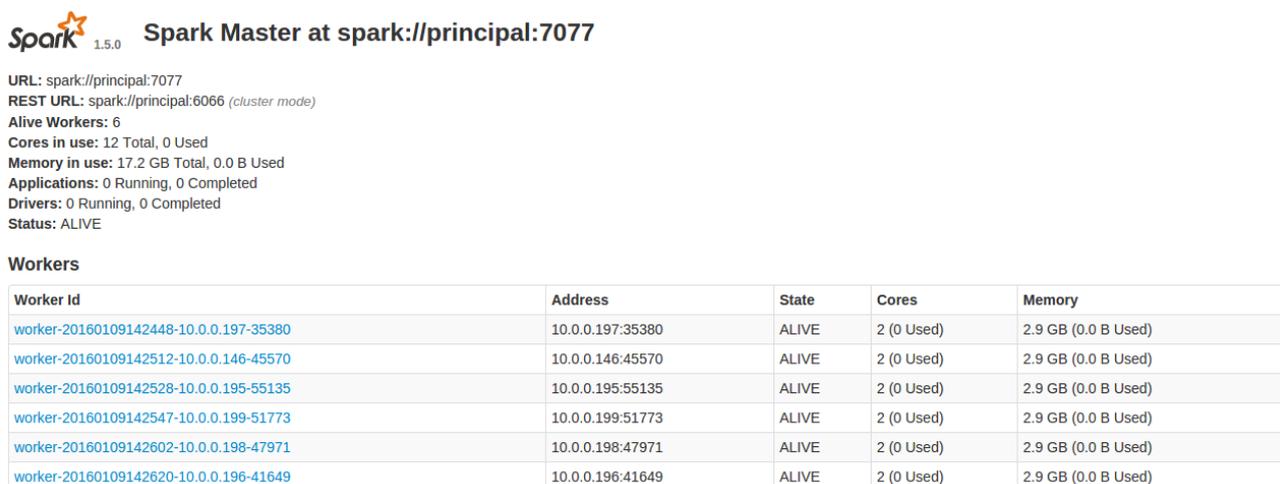
La BD, datos de conexión y demás se especifican en el fichero de configuración pinSpark.conf tal como mencionamos anteriormente.

## 5 Pruebas y validación

A efecto de realizar pruebas y validar el desarrollo del proyecto, he implementado un cluster de Apache Spark sobre máquinas virtuales en Amazon AWS EC2. El cluster lo compone un nodo principal (master) y seis nodos worker (slaves). Por tanto, la gestión del trabajo se realiza en el nodo principal, y los worker son los que realmente realizan los cálculos que se distribuyen en el cluster.

En la siguiente captura podemos observar la consola de gestión de Spark con todos los nodos conectados y sin ningún trabajo en ejecución

**Diagrama 22. Gestión cluster Spark**



The screenshot shows the Spark Master interface. At the top, it displays the Spark logo (version 1.5.0) and the text "Spark Master at spark://principal:7077". Below this, several status metrics are listed: URL, REST URL, Alive Workers (6), Cores in use (12 Total, 0 Used), Memory in use (17.2 GB Total, 0.0 B Used), Applications (0 Running, 0 Completed), Drivers (0 Running, 0 Completed), and Status (ALIVE). A section titled "Workers" contains a table with the following data:

Worker id	Address	State	Cores	Memory
<a href="#">worker-20160109142448-10.0.0.197-35380</a>	10.0.0.197:35380	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)
<a href="#">worker-20160109142512-10.0.0.146-45570</a>	10.0.0.146:45570	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)
<a href="#">worker-20160109142528-10.0.0.195-55135</a>	10.0.0.195:55135	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)
<a href="#">worker-20160109142547-10.0.0.199-51773</a>	10.0.0.199:51773	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)
<a href="#">worker-20160109142602-10.0.0.198-47971</a>	10.0.0.198:47971	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)
<a href="#">worker-20160109142620-10.0.0.196-41649</a>	10.0.0.196:41649	ALIVE	2 (0 Used)	2.9 GB (0.0 B Used)

La ejecución de los procesos se realiza con

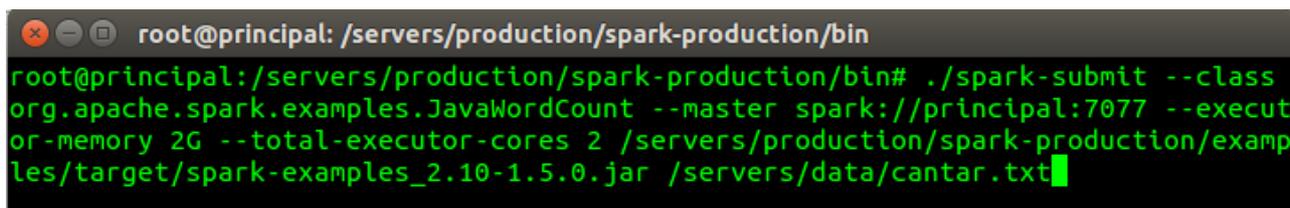
- `./start-master.sh` en el directorio `sbin` en el maestro
- `./start-slave.sh spark://principal:7077` en el directorio `sbin` en los worker

Por supuesto para el correcto funcionamiento del cluster es necesario que tengamos conectividad entre los nodos y una adecuada resolución de nombres.

La configuración y despliegue del cluster Spark realmente ha tomado una parte importante de la dedicación en el proyecto ya que no contaba con conocimientos previos en esta tecnología.

Como hemos comentado anteriormente en esta memoria, el proceso de instrumentación se produce porque hemos modificado el modo en el que Spark genera los trabajos para los worker. Así, para lanzar un trabajo debemos usar los comandos estándar de Spark. En mi caso utilizo el comando `spark-submit` que está presente en el comando `bin`.

### *Diagrama 23. Ejecución de trabajo Spark*

A terminal window with a dark background and light green text. The prompt is `root@principal: /servers/production/spark-production/bin`. The command entered is `./spark-submit --class org.apache.spark.examples.JavaWordCount --master spark://principal:7077 --executor-memory 2G --total-executor-cores 2 /servers/production/spark-production/examples/target/spark-examples_2.10-1.5.0.jar /servers/data/cantar.txt`.

```
root@principal: /servers/production/spark-production/bin
root@principal: /servers/production/spark-production/bin# ./spark-submit --class
org.apache.spark.examples.JavaWordCount --master spark://principal:7077 --execut
or-memory 2G --total-executor-cores 2 /servers/production/spark-production/examp
les/target/spark-examples_2.10-1.5.0.jar /servers/data/cantar.txt
```

En esta captura vemos como estamos lanzando un trabajo implementado en Java, indicando:

- La clase que vamos a lanzar (el problema)
- Cual es el nodo maestro
- La memoria que vamos a usar por worker para este problema
- El total de cores del cluster que vamos a dedicar a este problema
- El fichero que implementa la clase
- En particular para este trabajo, el fichero sobre el que vamos a contar las palabras

Si queremos lanzar un trabajo en el total de este cluster (que consta de 6 nodos de dos CPU), tendríamos que ejecutar

```
./spark-submit --class org.apache.spark.examples.JavaWordCount --master spark://principal:7077
--executor-memory 2G --total-executor-cores 12 /servers/production/spark-
production/examples/target/spark-examples_2.10-1.5.0.jar /servers/data/cantar.txt
```

Con esto podemos ver en la web de gestión del cluster que tenemos un trabajo en ejecución ocupando todos los nodos

## Diagrama 24. Nodos esclavos trabajando

### Workers

Worker Id	Address	State	Cores	Memory
worker-20160109142448-10.0.0.197-35380	10.0.0.197:35380	ALIVE	2 (2 Used)	2.9 GB (2.0 GB Used)
worker-20160109142512-10.0.0.146-45570	10.0.0.146:45570	ALIVE	2 (2 Used)	2.9 GB (2.0 GB Used)
worker-20160109142528-10.0.0.195-55135	10.0.0.195:55135	ALIVE	2 (2 Used)	2.9 GB (2.0 GB Used)
worker-20160109142547-10.0.0.199-51773	10.0.0.199:51773	ALIVE	2 (2 Used)	2.9 GB (2.0 GB Used)
worker-20160109142602-10.0.0.198-47971	10.0.0.198:47971	ALIVE	2 (2 Used)	2.9 GB (2.0 GB Used)
worker-20160109142620-10.0.0.196-41649	10.0.0.196:41649	ALIVE	2 (2 Used)	2.9 GB (2.0 GB Used)

### Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160109143840-0000	(kill) JavaWordCount	12	2.0 GB	2016/01/09 14:38:40	root	RUNNING	7 s

## 5.1 Pruebas de integración

Las pruebas de integración las he realizado en dos pasos

- Ejecución de múltiples trabajos y pruebas distintos en un único nodo a efecto de disponer de unas pruebas unitarias para validar la tool.
- Ejecución de un trabajo tipo en el cluster con diferentes números de nodos para identificar el impacto y el comportamiento.

### Unitarias

El primero de los pasos ha sido por supuesto el que ha consumido la mayor cantidad de tiempo ya que tiene mucho de ensayo/error. Las diferentes funcionalidades de la PinTool que hemos desarrollado y la integración de la misma en el flujo de Apache Spark, como se puede suponer, ha provocado múltiples pruebas para confirmar el funcionamiento de todas las partes.

Los principales problemas que me encontré en estas pruebas unitarias y que tuve que resolver los podríamos resumir como:

- Dificultad para confirmar que el código Scala que generamos modificando Spark era funcional sin compilar por completo y desplegar una instancia de Spark modificada. Esto último fue un consumidor de tiempo importante, no me quedo más remedio que compilar decenas de veces Spark y desplegarlo para realizar los tests
- Debug de las syscalls. Para realizar un debug de las llamadas al sistema antes de lanzar la instrumentación en un trabajo Spark, generé un conjunto de programas simples en C que realizaban escrituras/lecturas en disco y en la red, a efecto de validar el comportamiento de la tool. Recordemos que la tool no necesita a Spark para funcionar, podemos instrumentar cualquier binario con ella.

- Problema con los errores en la Syscall. Las llamadas al sistema, si terminan en error, devuelve un código -1 que no estaba tratando adecuadamente. Este retorno se almacenaba en una variable unsigned int de 64 bit y generaba un valor enorme que descuadraba las métricas, pero no causaba un error o problema en la ejecución. Ante esos valores sospechábamos un error de corrupción de datos pero no sabíamos exactamente a que se debía hasta la realización de una batería de pruebas y un debug (output a fichero local) de todos los pasos de la instrumentación. Una vez detectado por supuesto que teníamos un fallo funcional en la codificación (no debemos asignar un valor negativo al unsigned) las métricas volvieron a generar valores coherentes.

## **Genéricas**

Para realizar un test del sistema de instrumentación sobre el cluster, debíamos de buscar un trabajo que causara un overhead importante en los recursos del mismo. Nos interesaba especialmente que tuviera un componente importante de acceso a disco y memoria, ya que son recursos que sabemos se usan de forma masiva en Spark.

La elección de una implementación que ya venia en los ejemplos de Spark fue simplemente por economizar el tiempo de trabajo ya que este proyecto en general ha sido un consumidor muy importante de tiempo por la investigación que tuve que hacer y no quedaba mucho tiempo libre para la parte de las pruebas. La elección por tanto se centro en la implementación de un algoritmo de contar palabras en un fichero de texto.

Este algoritmo lee un fichero (que se reparte entre todos los nodos esclavos del cluster) y cuenta el numero de ocurrencias de cada una de las palabras contenidas en ese fichero. Como podemos imaginar, si el fichero es grande, la cantidad de trabajo que realiza es ingente. Para disponer de un fichero adecuado, decidí buscar un texto literario para que hubiera un vocabulario rico y multitud de palabras distintas. La elección fue un texto anónimo de mitología germánica, el cantar de los nibelungos. Este texto lo concatenamos múltiples veces para generar dos ficheros de control, uno de aproximadamente 2GB y otro de aproximadamente 9.3GB.

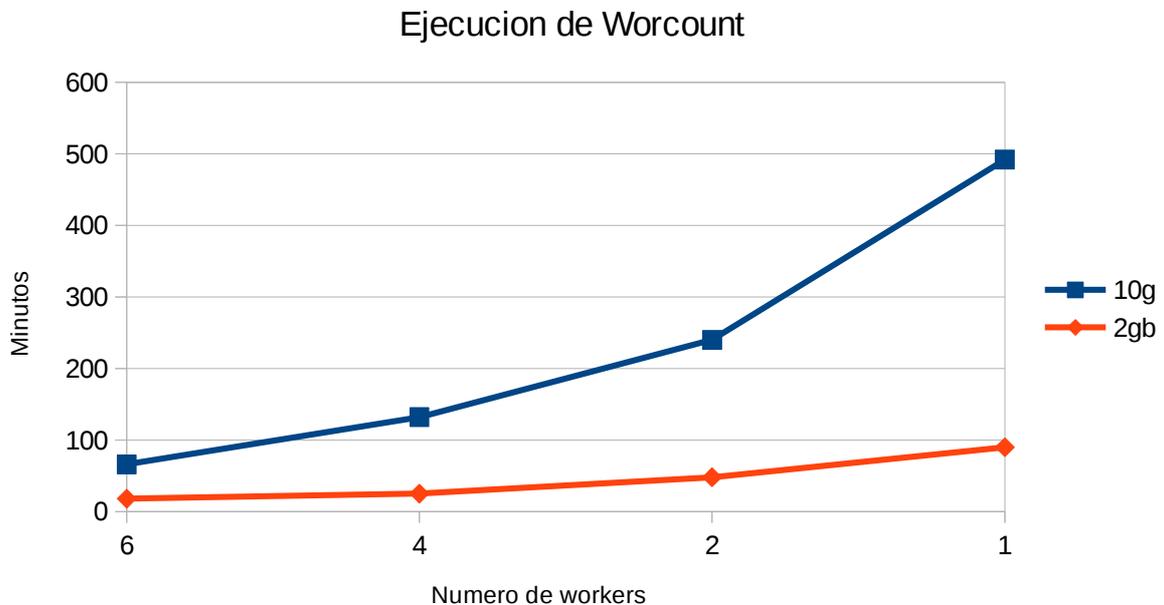
Cuando Spark recibe un trabajo determinado, ese trabajo se divide en unidades que son las que se reparten a los Worker mediante los procesos maestro/esclavo de Spark. Por supuesto, cuando aumenta el número de workers, disminuye el tiempo de ejecución total en el sistema (los tiempos de instrumentación son muy superiores a los de ejecución normal de los trabajos por el overhead).

**Diagrama 25. Tabla resultado ejecuciones Wordcount**

Tarea	Tamaño del fichero	Numero de workers (2CPU cada uno)	Tiempo de ejecución
Wordcount	9.3 GB	6	66 minutos
Wordcount	9.3 GB	4	132 minutos
Wordcount	9.3 GB	2	240 minutos
Wordcount	9.3 GB	1	492 minutos
Wordcount	2GB	6	18 minutos
Wordcount	2GB	4	25 minutos
Wordcount	2GB	2	48 minutos
Wordcount	2GB	1	90 minutos

Tanto en la tabla con en el siguiente gráfico podemos ver que los tiempos son lineales y proporcionales al tamaño del problema y al número de nodos trabajando en el mismo.

**Diagrama 26. Gráfico tiempo ejecución Wordcount**



A lo largo de todas las pruebas que realice se aprecia una correlación lineal entre el tiempo de ejecución y el número de nodos que participan. Esto creo que nos indica que el particionado del problema de ejemplo Wordcount con el que estamos trabajando realmente está bien construido y aprovecha adecuadamente las capacidades de Spark.

Así mismo, también en las métricas que obtenemos de la instrumentación podemos ver que cuando un solo nodo esclavo procesa el trabajo, el total de bytes leídos (usando como directorio raíz /servers/data) corresponde con el tamaño del fichero a analizar. En el caso de que distribuyamos el trabajo entre los nodos, cada uno lee una parte (mediante un offset del fichero) y el total acumulado de todos los nodos vuelve a corresponder con el tamaño del fichero a analizar.

He de decir que estas pruebas genéricas no han buscado probar la bonanza del algoritmo Wordcount o de Apache Spark, sino servir como validación de los trabajos realizados y sobre todo confirmar que el proyecto es funcional y que las decisiones de diseño que tomamos están cubiertas con el producto generado. Por tanto, no tengo unas conclusiones claras al respecto de las ejecuciones que realizamos, mas haya de las propias que puedo extraer como desarrollador del proyecto. Una vez dicho esto, me gustaría no obstante comentar los siguientes puntos que creo que deben tenerse en cuenta para instrumentar trabajos Spark

- La instrumentación genera un importantísimo overhead en la ejecución. Es necesario tenerlo en cuenta para la realización de pruebas ya que los tiempos de ejecución se disparan.
- Para instrumentar un trabajo Spark creo que hay que modelar un conjunto de datos mínimo que nos sirva como semilla para la toma de decisiones. Veo totalmente inviable instrumentar trabajos reales por la carga que conlleva (trabajos masivos con mucha información)
- El modelado de las métricas en la tool debe realizarse alineado con lo que queramos investigar respecto al funcionamiento a bajo nivel del trabajo Spark. Yo he incorporado una serie de métricas base pero es posible que un problema X requiera incluir alguna adicional.
- La integración de la instrumentación en Spark se hace a nivel de trabajo y nodo esclavo. Si queremos instrumentar los procesos padre en si, master y slave, podemos hacerlo por supuesto incluyendo en los scripts de inicio de los daemon el parser. Está fuera del objetivo de este proyecto.

## 6 Conclusiones y trabajos futuros

### 6.1 Justificación del proyecto

Este es el tercer proyecto fin de estudios que realizo desde que estoy en la UOC.

Comencé mis estudios de ITIS en el año 2004, que completé una vez finalizados con el Master en Ingeniería Informática de la UOC y ahora finalizando este ciclo, con el Grado en Ingeniería Informática.

La realización de un proyecto de final de carrera creo que tiene que permitir al estudiante plasmar no solamente lo aprendido durante los años de estudio, sino las inquietudes que todos llevamos dentro y que nos acompañan en los meses de trabajo con las asignaturas.

La UOC (en mi opinión) tiene un tipo de alumno que creo es sensiblemente diferente respecto al resto de Universidades tradicionales. Muchos de mis compañeros (por no decir todos) son profesionales contrastados y trabajan en diferentes áreas de la informática y en la UOC buscamos continuar nuestra formación para mejorar conocimientos y para por supuesto, tener mas oportunidades de promoción.

Dada esta doble vertiente de profesionales y de estudiantes, está claro que la realización de un proyecto fin de estudios no debería (en muchos de los casos) suponer un quebranto o un problema importante para personas que ya estamos desarrollando el trabajo de ingenieros en informática.

En mi caso particular, dedicándome al área de infraestructuras y comunicaciones, realizo proyectos con mucha mas complejidad y dedicación de la que puedo aplicar al estudio de varias asignaturas.

Una vez dicho esto, me gustaría no obstante hacer un punto y aparte y volver al inicio de estos párrafos donde comente que la realización del proyecto debería permitirnos plasmas nuestras inquietudes personales.

Todos los proyectos que realice en la UOC (incluyendo este) han estado apartados de mi linea de trabajo profesional y me han permitido expresar y aportar de alguna forma a la ingeniería informática, has haya del tramite de la realización del proyecto. Tengo la sensación de que nosotros los estudiantes tenemos el deber de devolver algo del conocimiento que hemos adquirido a lo largo de los años y por eso todos mis proyectos han tenido un componente de investigación y de generación de algo de valor para la ingeniería informática.

Tengo que agradecer a Cesc Guim su apoyo y el permitirme continuar con esta filosofía de trabajo y sobre todo, el favorecer que dado que siempre ha sido el consultor para mis proyectos fin de carrera, hemos podido seguir una linea en un área de conocimiento concreta lo que me hace estar extremadamente satisfecho con lo que hemos construido a lo largo de estos años.

## 6.2 Conclusiones

La carga de trabajo en este proyecto considero que ha sido media, con una excepción referida a las pruebas unitarias y generales que han tomado muchas horas de dedicación para validar el funcionamiento del proyecto.

He tenido que investigar en primer lugar el funcionamiento de Apache Spark, implementar un conjunto de servidores y un cluster, realizar pruebas de funcionamiento del mismo y estudiar a bajo nivel como funciona para integrarnos en la ejecución. Una vez realizado esto, he generado una distribución personalizada de Apache Spark con una compilación y la he desplegado a todo el cluster (7 nodos).

He tenido también que programar una PIN Tool para extraer información y realizar una batería de pruebas para validar todo el desarrollo.

El resultado del proyecto lo podemos dividir en estos hitos

- Modificación de Apache Spark para integración con PIN
- Desarrollo de una PIN Tool con métricas personalizadas
- Desarrollo de un modelo centralizado de output de la instrumentación

Dentro de la planificación y planteamiento del proyecto incluimos una tercera parte para disponer de una herramienta de visualización Web de los datos de salida de la instrumentación. Dada la dedicación necesaria para la realización de las pruebas, esta parte no la hemos realizado, quedando como un punto de mejora a futuro. Realmente una vez que tenemos la información de la instrumentación en tiempo real en una BBDD centralizada, la extracción de información de la misma se puede realizar con multitud de soluciones de código libre o comerciales de las que existen en el mercado.

Como líneas de trabajo futuras, creo que podríamos establecer las siguientes

- La PIN Tool en las métricas de las llamadas al sistema es monolítica. Sería muy interesante convertir ese código modificándolo para usar un fichero de configuración o plantilla y que pudiéramos establecer en runtime las syscall que queremos instrumentar.
- Configuración de una herramienta de visualización Web para explotar la información de la instrumentación.
- Implementar un script de despliegue e instalación de Apache Spark y la PIN Tool para despliegue rápido.

## 7 Anexos

### 7.1 Fichero de configuración *pinSpark.conf*

```
#####Instrumentation configuration
##Base values
pinCommand = /servers/pin/pin.sh
pinToolSwitch = -t
pinTool = /servers/pin/source/tools/pinSpark/obj-intel64/pinSpark.so
pinOutSwitch = -o
pinOutLogFile = pinSpark
pinAttachSwitch = --
pinComputerName = principal

##Choose values to instrument
##Always instrument ins, bbl and thread count
ifMalloc = 1;
ifFree = 1;
ifBarrier = 1;
ifMemoryRead = 1;
ifMemoryWrite = 1;
#Syscall
ifWrite = 1;
ifRead = 1;
ifRecv = 1;
ifSend = 1;
ifOpen = 1;
ifClose = 1;

##Syscalls specific configuration
#dirForSyscalls = /servers/production/spark-production
dirForSyscalls = /

#####DB Store configuration
mysqlServer = principal
mysqlDatabase = sparkanalyzer
mysqlUser = sparkanalyzer
mysqlPassword = localpass
mysqlWorkerID = 0
mysqlWorkerIP = 10.0.0.108
```

## 7.2 Codificación en C/C++ de la PIN Tool. PinSpark.cpp

```
#include "pin.H"
#include <iostream>
#include <fstream>
#include <ctime>
#include <string>
#include <cstdio>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
#include <pthread.h>

#ifdef TARGET_MAC
#include <sys/syscall.h>
#elif !defined(TARGET_WINDOWS)
#include <syscall.h>
#endif

//Includes for MySQL db store of output
#include <mysql.h>

using namespace std;

/*
=====
=== */
// Global variables
/*
=====
=== */

UINT64 insCount = 0; //number of dynamically executed instructions
UINT64 bblCount = 0; //number of dynamically executed basic blocks
UINT64 threadCount = 0; //total number of threads, including main thread
UINT64 mallocCount = 0; //Total number of malloc
UINT64 freeCount = 0; //Total number of free
UINT64 barrierCount = 0; //Total of barriers detected
UINT64 memoryReadCount = 0; //Total of memory reads
UINT64 memoryWriteCount = 0; //Total of memory writes

UINT64 writeSyscallCount = 0; //Total bytes writed
UINT64 readSyscallCount = 0; //Total bytes readed
UINT64 isWriteFDValue = 0; //Value of descriptor
```

```

UINT64 isReadFDValue = 0; //Value of descriptor
UINT64 recvSyscallCount = 0; //Total bytes received from network
UINT64 sendSyscallCount = 0; //Total bytes sended to network

UINT64 arrayFDOpenFiles[500]; //Array of FD descriptors of open files

//UINT64 test=-1;

//Declare variables for DB connection to store output of instrumentation
MYSQL *conn;
MYSQL_RES *res;
MYSQL_ROW row;

char *worker = (char *) malloc(100);
char *workname = (char *) malloc(100);

//Mutex for MySQL inserts
pthread_mutex_t lockmysql;

//Variables for instrument syscalls
int isWrite=0;
int isRead=0;
int isRecv=0;
int isSend=0;
int isOpen=0;
int isClose=0;
//Directory to instrument syscalls read/write. If file is contained on this
//directory we put bytes on vars, if outside ignore
char *dirForSyscalls= (char *) malloc(500);

//Vars to control instrumentation
//we can choose if we want or not to instrument based on fileconf
//if no conf var or value != from 1, not instrument
//ins, bbl and thread are instrumented always

int ifMalloc = 0;
int ifFree = 0;
int ifBarrier = 0;
int ifMemoryRead = 0;
int ifMemoryWrite = 0;

```

```

//Syscall
int ifWrite = 0;
int ifRead = 0;
int ifRecv = 0;
int ifSend = 0;
int ifOpen = 0; //To an adecuate management of read/write we need this, don't remove it
int ifClose = 0; //To an adecuate management of read/write we need this, don't remove it

/*
=====
===== */
/* Names of malloc and free */
/*
=====
===== */
#define MALLOC "malloc"
#define FREE "free"
#define BARRIER "pthread_cond_wait"
PIN_LOCK lock;

std::ostream * out = &cerr;

/*
=====
===== */
// Command line switches
/*
=====
===== */
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "", "File for Apache Spark instrumentation output");

KNOB<BOOL> KnobCount(KNOB_MODE_WRITEONCE, "pintool",
    "count", "1", "This tool instruments Apache Spark workloads");

/**
 * Returns 1 if char b is contained on char a, else returns 0
 * File comparison begins from the [0] index of the char value
 * for example
 *
 * a='/servers/spark/data/file1.data'
 * b='/servers/spark'
 *
 * returns 1

```

```

*
* a='/servers/spark/data/file1.data'
* b='/mydata/servers/spark'
*
* return 0
*
**/
int contains(char * a, char * b){

    int size=strlen(b);
    //printf("Size:%d\n",size);
    //printf("A.%s\n",a);
    //printf("B.%s\n",b);
    if (strncmp(a,b,size)==0){
        //a and b are equal until b size
        //printf("1.%s\n",a);
        return(1);

    } else {
        //printf("2.%s\n",b);
        return(0);

    }

}

//Put this FD descriptor on OpenFiles array
VOID addToOpenFiles(ADDRINT ret)

{

    *out << "Adding file: " << ret << endl;
    int i;
    for (i=0; i<499;i++){
        if (arrayFDOpenFiles[i]==987654321){
            //This value is dummy, empty space
            arrayFDOpenFiles[i]=ret;

            break;

        }

    }

}

```

```

        //printf("Insert on vector, Value for i %d, Vaalue for ret %ld\n",i,ret);

}

//Put this FD descriptor on OpenFiles array
VOID removeToOpenFiles(ADDRINT ret)

{

    *out << "Removing file: " << ret << endl;
    int i;
    for (i=0; i<499;i++){
        if (arrayFDOpenFiles[i]==ret){
            //We need to remove this value
            arrayFDOpenFiles[i]=987654321;
            break;
        }

    }

}

}

/**
 * Returns 1 if ret is in openfiles array
 * else returns 0
 *
 *
 */
int isFDinOpenFiles(ADDRINT ret)
{

    *out << "Search for file: " << ret << endl;
    int i;
    for (i=0; i<499;i++){
        //printf("Value on array position %d is %d\n",i,arrayFDOpenFiles[i]);
        if (arrayFDOpenFiles[i]==ret){

            //FD is on the array

            *out << "We find and interesting file " << i << ", " << ret<< endl;
            return (1);
        }
    }
}

```

```

        }

    }

    return(0);

}

```

```

VOID partialLog() {

    pthread_mutex_lock(&lockmysql);

    *out << "======" << endl;
    *out << "pinSpark analysis results: " << endl;
    *out << "Number of instructions: " << insCount << endl;
    *out << "Number of basic blocks: " << bblCount << endl;
    *out << "Number of threads: " << threadCount << endl;
    *out << "Number of malloc: " << mallocCount << endl;
    *out << "Number of free: " << freeCount << endl;
    *out << "Number of Memory Writes: " << memoryWriteCount << endl;
    *out << "Number of Memory Reads: " << memoryReadCount << endl;
    *out << "Number of barrier: " << barrierCount << endl;
    *out << "Bytes writed to disk: " << writeSyscallCount << endl;
    *out << "Bytes readed from disk: " << readSyscallCount << endl;
    *out << "Bytes received from network: " << recvSyscallCount << endl;
    *out << "Bytes sended to the network: " << sendSSyscallCount << endl;
    *out << "======" << endl;

    //Partial insert of values on MySQL database as instrumentation runs

    //worker, number of the worker, for example 0
    //workname, filename of output, we use it as work name

    char query1[800] = {0};

    sprintf(query1,"insert into event(worker,workname,date,

```



```

* Print out help message.
*/
INT32 Usage()
{
    cerr << "This tool instruments Apache Spark workloads for performance evaluation " << endl;

    cerr << KNOB_BASE::StringKnobSummary() << endl;

    return -1;
}

/*
===== */
// Analysis routines
/*
===== */

/*!
* Increase counter of the executed basic blocks and instructions.
* This function is called for every basic block when it is about to be executed.
* @param[in] numInstInBbl  number of instructions in the basic block
* @note use atomic operations for multi-threaded applications
*/
VOID CountBbl(UINT32 numInstInBbl)
{
    bblCount++;
    insCount += numInstInBbl;

    if (bblCount % 1000000000 == 0)
    {
        //Get partial statistics

        partialLog();
    }
}

/*
===== */
// Instrumentation callbacks
/*
===== */

```

```

/#!/
 * Insert call to the CountBbl() analysis routine before every basic block
 * of the trace.
 * This function is called every time a new trace is encountered.
 * @param[in] trace  trace to be instrumented
 * @param[in] v      value specified by the tool in the TRACE_AddInstrumentFunction
 *                   function call
 */
VOID Trace(TRACE trace, VOID *v)
{
    // Visit every basic block in the trace
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        // Insert a call to CountBbl() before every basic bloc, passing the number of instructions
        BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)CountBbl, IARG_UINT32, BBL_NumIns(bbl),
IARG_END);
    }
}

```

```

/#!/
 * Increase counter of threads in the application.
 * This function is called for every thread created by the application when it is
 * about to start running (including the root thread).
 * @param[in] threadIndex  ID assigned by PIN to the new thread
 * @param[in] ctxt         initial register state for the new thread
 * @param[in] flags        thread creation flags (OS specific)
 * @param[in] v            value specified by the tool in the
 *                         PIN_AddThreadStartFunction function call
 */
VOID ThreadStart(THREADID threadIndex, CONTEXT *ctxt, INT32 flags, VOID *v)
{
    threadCount++;
}

```

```

VOID Fini(INT32 code, VOID *v)
{
    pthread_mutex_lock(&lockmysql);

    *out << "===== " << endl;

```

```

*out << "pinSpark analysis results: " << endl;
*out << "Number of instructions: " << insCount << endl;
*out << "Number of basic blocks: " << bblCount << endl;
*out << "Number of threads: " << threadCount << endl;
*out << "Number of malloc: " << mallocCount << endl;
*out << "Number of free: " << freeCount << endl;
*out << "Number of Memory Writes: " << memoryWriteCount << endl;
*out << "Number of Memory Reads: " << memoryReadCount << endl;
*out << "Number of barrier: " << barrierCount << endl;
*out << "Bytes writed to disk: " << writeSyscallCount << endl;
*out << "Bytes readed from disk: " << readSyscallCount << endl;
*out << "Bytes received from network: " << recvSyscallCount << endl;
*out << "Bytes sended to the network: " << sendSSyscallCount << endl;
*out << "===== " << endl;

    //printPerformanceStats();

    //Partial insert of values on MySQL database as instrumentation runs

    //worker, number of the worker, for example 0
    //workname, filename of output, we use it as work name

    //Old values
    // sprintf(query1,"insert into event(worker,workname,date,
insCount,bblCount,threadCount,mallocCount, freeCount, barrierCount, memoryReadCount,
memoryWriteCount) values (%s,'%s',NOW(),%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64
",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ")",worker,workname,insCount,bblCount,threadCount, mallocCount,
freeCount, barrierCount, memoryReadCount, memoryWriteCount);

    char query1[800] = {0};

    sprintf(query1,"insert into event(worker,workname,date,
insCount,bblCount,threadCount,mallocCount, freeCount, barrierCount, memoryReadCount,
memoryWriteCount,writeSyscallCount,readSyscallCount,recvSyscallCount,sendSSyscallCount) values
(%s,'%s',NOW(),%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",
%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ",%" PRIu64 ")
",worker,workname,insCount,bblCount,threadCount, mallocCount, freeCount, barrierCount,
memoryReadCount, memoryWriteCount, writeSyscallCount, readSyscallCount, recvSyscallCount,
sendSSyscallCount);
    /*out << query1 << endl;

```

```

    int err;

    err=mysql_real_query(conn, query1,strlen(query1));

    if (err!=0){
        *out << "Error on query: " << err << endl;
        *out << query1 << endl;
    }

    mysql_close(conn);

    pthread_mutex_unlock(&lockmysql);
}

/**
 *
 * This funcion is executed on each malloc
 *
 */
VOID mallocCalculation(ADDRINT size, THREADID threadid)
{

    PIN_GetLock(&lock, threadid+1);
    mallocCount++;
    PIN_ReleaseLock(&lock);

}

VOID freeCalculation(ADDRINT size, THREADID threadid)
{

    PIN_GetLock(&lock, threadid+1);
    freeCount++;
    PIN_ReleaseLock(&lock);

}

VOID barrierCalculation(ADDRINT size, THREADID threadid)
{

```

```

        PIN_GetLock(&lock, threadid+1);
        barrierCount++;
        PIN_ReleaseLock(&lock);
    }

VOID memoryReadCalculation(VOID * ip, UINT32 size)
{

    memoryReadCount++;

}

VOID memoryWriteCalculation(VOID * ip, UINT32 size)
{

    memoryWriteCount++;

}

// Print syscall number and arguments
VOID SysBefore(ADDRINT ip, ADDRINT num, ADDRINT arg0, ADDRINT arg1, ADDRINT arg2, ADDRINT
arg3, ADDRINT arg4, ADDRINT arg5)
{
#if defined(TARGET_LINUX) && defined(TARGET_IA32)
    // On ia32 Linux, there are only 5 registers for passing system call arguments,
    // but mmap needs 6. For mmap on ia32, the first argument to the system call
    // is a pointer to an array of the 6 arguments
    if (num == SYS_mmap)
    {
        ADDRINT * mmapArgs = reinterpret_cast<ADDRINT *>(arg0);
        arg0 = mmapArgs[0];
        arg1 = mmapArgs[1];
        arg2 = mmapArgs[2];
        arg3 = mmapArgs[3];
        arg4 = mmapArgs[4];
        arg5 = mmapArgs[5];
    }
#endif

```

```

if ((long)num == 1){
    //Write syscall
    //Test if we instrument
    if (ifWrite==1){

        //Before instrument, check if file is in the array

        *out << "Write number: " << arg0 << endl;
        if (isFDinOpenFiles(arg0)==1){

            *out << "Interesting File" << endl;
            //Ok, we're on a file of interest
            isWrite = 1;
            isWriteFDValue = arg0;
        }
    }

}

if ((long)num == 0){
    //Read syscall
    //Test if we instrument
    if (ifRead==1){
        *out << "Read syscall on FD " << arg0 << endl;
        //Before instrument, check if file is in the array
        if (isFDinOpenFiles(arg0)==1){
            //Ok, we're on a file of interest
            isRead = 1;
            isReadFDValue = arg0;
            *out << "Read syscall on FD is on interesting file " << arg0 << endl;
        }
    }

}

if ((long)num == 45){
    //Recv syscall
    //Test if we instrument
    if (ifRecv==1){
        *out << "Recv syscall received: " << arg0 << endl;
        isRecv = 1;
    }
}

```

```

}

if ((long)num == 47){
    //Recvmsg syscall
    //Test if we instrument
    if (ifRecv==1){
        *out << "Recvmsg syscall received: " << arg0 << endl;
        isRecv = 1;
    }
}

if ((long)num == 44){
    //Send syscall
    //Test if we instrument
    if (ifSend==1){
        *out << "Send syscall received: " << arg0 << endl;
        isSend = 1;
    }
}

if ((long)num == 46){
    //Sendmsg syscall
    //Test if we instrument
    if (ifSend==1){
        *out << "Sendmsg syscall received: " << arg0 << endl;
        isSend = 1;
    }
}

//open
if ((long)num == 2){
    //Open syscall
    //Test if we instrument
    if (ifOpen==1){

        //arg0 store memory address of first parameter, name of file to open
        char * address = (char *) arg0;

        //Comparefile from syscall with conf value

        /*out << "Open: File to compare 1: " << address<< endl;

```

```

        /*out << "Open: File to compare 2: " << dirForSyscalls << endl;
        if (contains(address,dirForSyscalls)==1)
        {
            //we must instrument
            isOpen=1;
            *out << "Open: File is interesting to instrument: " << address<< endl;

        }
    }

}

if ((long)num == 3){
    //Close syscall
    //Test if we instrument
    if (ifClose ==1){

        *out << "Close syscall requested " << endl;
        isClose=1;

    }
}

}

```

```

VOID SysAfter(ADDRINT ret)
{

    //Test if syscall() returns -1 error
    //UINT64 vars don't store negative values

    if (isWrite == 1 || isRead == 1 || isRecv == 1 || isSend == 1 || isOpen == 1|| isClose == 1){
        if (ret == 18446744073709551615){

            *out << "Syscall error, return value: " << ret << endl;

```

```

        //Syscall is returning an error
        //No instrumentation
        isWrite = 0;
        isRead = 0;
        isRecv = 0;
        isSend = 0;
        isOpen = 0;
        isClose = 0;

    }
}

if (isWrite == 1){

    *out << "Add to write value: " << ret<< endl;
    writeSyscallCount+= ret;
    isWrite = 0;

}

if (isRead == 1){
    *out << "Add to read value: " << ret<< endl;
    readSyscallCount+= ret;
    isRead = 0;
}

if (isRecv == 1){
    *out << "Add to isrecv value: " << ret<< endl;
    recvSyscallCount+= ret;
    isRecv = 0;
}

if (isSend == 1){
    *out << "Add to issend value: " << ret<< endl;
    sendSSyscallCount+= ret;
    isSend = 0;
}

if (isOpen == 1){
    //Return value is the file descriptor
    //We need take this file descriptor in account until a close syscall closes file
    //file descriptors value are reused
    addToOpenFiles(ret);
    *out << "Opening file FD: " << ret << endl;
    isOpen = 0;
}

```

```

if (isClose == 1){
    //Closing File, we need to remove from array of interesting files
    //only remove if in this
    if (isFDinOpenFiles(ret)==1){
        removeToOpenFiles(ret);
        *out << "Closing file FD: " << ret << endl;

    }
    isClose = 0;
}

```

```

}

```

```

VOID SyscallEntry(THREADID threadIndex, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v)
{

```

```

    SysBefore(PIN_GetContextReg(ctxt, REG_INST_PTR),
        PIN_GetSyscallNumber(ctxt, std),
        PIN_GetSyscallArgument(ctxt, std, 0),
        PIN_GetSyscallArgument(ctxt, std, 1),
        PIN_GetSyscallArgument(ctxt, std, 2),
        PIN_GetSyscallArgument(ctxt, std, 3),
        PIN_GetSyscallArgument(ctxt, std, 4),
        PIN_GetSyscallArgument(ctxt, std, 5));

```

```

}

```

```

VOID SyscallExit(THREADID threadIndex, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v)
{

```

```

    SysAfter(PIN_GetSyscallReturn(ctxt, std));
}

```

```

LOCALFUN VOID Image(IMG img, VOID *v)
{

```

```

    // Instrument the malloc() and free() functions. Print the input argument
    // of each malloc() or free(), and the return value of malloc().

```

```

    //

```

```

    // Find the malloc() function.

```

```

    RTN mallocRtn = RTN_FindByName(img, MALLOC);

```

```

    if (RTN_Valid(mallocRtn))

```

```

    {

```

```

        //Test if we instrument
        if (ifMalloc == 1){
RTN_Open(mallocRtn);
// Instrument malloc() to count operations
RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocCalculation,
                IARG_ADDRINT, MALLOC,
                IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_THREAD_ID,
                IARG_END);
RTN_Close(mallocRtn);}
}

// Find the free() function.
RTN freeRtn = RTN_FindByName(img, FREE);
if (RTN_Valid(freeRtn))
{
        //Test if we instrument
        if (ifFree == 1){
RTN_Open(freeRtn);
// Instrument free() to count operations
RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)freeCalculation,
                IARG_ADDRINT, FREE,
                IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_THREAD_ID,
                IARG_END);
RTN_Close(freeRtn);}
}

//Search for barriers

RTN barrierRtn = RTN_FindByName(img, BARRIER);
if (RTN_Valid(barrierRtn)){
        //Test if we instrument
        if (ifBarrier==1){
RTN_Open(barrierRtn);
RTN_InsertCall(barrierRtn, IPOINT_BEFORE, (AFUNPTR)barrierCalculation,
IARG_ADDRINT, BARRIER,
                IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_THREAD_ID,
                IARG_END);
RTN_Close(barrierRtn);}

        }
}

VOID Instruction(INS ins, VOID *v){
        if (INS_IsMemoryRead(ins))

```

```

{
    //Test if we instrument
    if (ifMemoryRead==1){
INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR) memoryReadCalculation,
IARG_INST_PTR,IARG_MEMORYWRITE_SIZE, IARG_END);}
    }

    if (INS_IsMemoryWrite(ins))
{
    //Test if we instrument
    if (ifMemoryWrite==1){
INS_InsertPredicatedCall(ins, IPOINT_BEFORE,(AFUNPTR)
memoryWriteCalculation,IARG_INST_PTR, IARG_MEMORYWRITE_SIZE, IARG_END);}

    }

    //Search for syscall
    // For O/S's (Mac) that don't support PIN_AddSyscallEntryFunction(),
    // instrument the system call instruction.

if (INS_IsSyscall(ins) && INS_HasFallThrough(ins))
{
    // Arguments and syscall number is only available before
INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(SysBefore),
                IARG_INST_PTR, IARG_SYSCALL_NUMBER,
                IARG_SYSARG_VALUE, 0, IARG_SYSARG_VALUE, 1,
                IARG_SYSARG_VALUE, 2, IARG_SYSARG_VALUE, 3,
                IARG_SYSARG_VALUE, 4, IARG_SYSARG_VALUE, 5,
                IARG_END);

    // return value only available after
INS_InsertCall(ins, IPOINT_AFTER, AFUNPTR(SysAfter),
                IARG_SYSRET_VALUE,
                IARG_END);
}

}

int split(char *string, char *delim, char *tokens[])
{
    int count = 0;
    char *token;
    char *stringp;

```

```

stringp = string;
while (stringp != NULL) {
    token = strtok(&stringp, delim);
    tokens[count] = token;
    count++;
}
return count;
}

```

```

char *trimwhitespace(char *str)
{
    char *end;

    // Trim leading space
    while(isspace(*str)) str++;

    if(*str == 0) // All spaces?
        return str;

    // Trim trailing space
    end = str + strlen(str) - 1;
    while(end > str && isspace(*end)) end--;

    // Write new null terminator
    *(end+1) = 0;

    return str;
}

```

```

int main(int argc, char *argv[])
{

    char *server= (char *) malloc(100);
    char *user= (char *) malloc(100);
    char *password= (char *) malloc(100);
    char *database= (char *) malloc(100);

    //Initialize array for FD descriptors on Syscall
    int i;
    for (i=0;i<499;i++)
    {
        arrayFDOpenFiles[i]=987654321; //We're sure that no FD are going to be used
    }
}

```

```
//Read values from file conf
//We set malloc on values to strcpy
```

```
static const char filename[] = "/etc/pinSpark/pinSpark.conf";
FILE *file = fopen ( filename, "r" );
char * tokens[2];

if ( file != NULL )
{
int count;
char linea[128];
while ( fgets ( linea, 128, file ) != NULL ) /* read a line */
{
count=split(linea,"=",tokens);
//trim trash from vars
tokens[0] = trimwhitespace(tokens[0]);
tokens[1] = trimwhitespace(tokens[1]);

/*out << "###" << endl;

if (strcmp(tokens[0],"mysqlServer")==0)
{
strcpy(server,tokens[1]);
}

if (strcmp(tokens[0],"mysqlDatabase")==0)
{
strcpy(database,tokens[1]);
}

if (strcmp(tokens[0],"mysqlUser")==0)
{
strcpy(user,tokens[1]);
}

if (strcmp(tokens[0],"mysqlPassword")==0)
{
strcpy(password,tokens[1]);
```

```
}  
    if (strcmp(tokens[0],"mysqlWorkerID")==0)  
{  
    strcpy(worker,tokens[1]);  
}  
  
if (strcmp(tokens[0],"dirForSyscalls")==0)  
{  
    strcpy(dirForSyscalls,tokens[1]);  
}  
  
if (strcmp(tokens[0],"ifMalloc")==0 )  
{  
  
    ifMalloc = atoi(tokens[1]);  
  
}  
  
if (strcmp(tokens[0],"ifFree")==0 )  
{  
  
    ifFree=atoi(tokens[1]);  
  
}  
  
if (strcmp(tokens[0],"ifBarrier")==0 )  
{  
  
    ifBarrier=atoi(tokens[1]);  
  
}  
  
if (strcmp(tokens[0],"ifMemoryRead")==0)  
{  
  
    ifMemoryRead=atoi(tokens[1]);  
  
}  
  
if (strcmp(tokens[0],"ifMemoryWrite")==0)  
{
```

```
        ifMemoryWrite=atoi(tokens[1]);
    }

    if (strcmp(tokens[0],"ifWrite")==0)
    {

        ifWrite=atoi(tokens[1]);

    }

    if (strcmp(tokens[0],"ifRead")==0)
    {

        ifRead=atoi(tokens[1]);

    }

    if (strcmp(tokens[0],"ifRecv")==0)
    {

        ifRecv=atoi(tokens[1]);

    }

    if (strcmp(tokens[0],"ifSend")==0)
    {

        ifSend=atoi(tokens[1]);

    }

    if (strcmp(tokens[0],"ifOpen")==0)
    {

        ifOpen=atoi(tokens[1]);

    }

    if (strcmp(tokens[0],"ifClose")==0)
    {

        ifClose=atoi(tokens[1]);

    }

}
```

```

        }
    }
    fclose ( file );
} else

{
    //Error on conf file access
    *out <<
"===== " << endl;
    *out << " Error reading /etc/pinSpark/pinSpark.conf  " << endl;
    *out << "===== " <<
endl;
    exit(1);
}

conn = mysql_init(NULL);

//Connect to mysql
if (!mysql_real_connect(conn,server,user,password,database,0,NULL,0))
{
    //Error on DB connection
    *out << "===== " <<
endl;
    *out << " Error on Mysql DB connection, check conf  " << endl;
    *out << "===== " <<
endl;
    exit(1);
}
mysql_autocommit(conn, 1);

if( PIN_Init(argc,argv) )
{
    return Usage();
}
// Initialize the pin lock
PIN_InitLock(&lock);

PIN_InitSymbols();
string fileName = KnobOutputFile.Value();

```

```

//Output filename
if (!fileName.empty()) {

    out = new std::ofstream(fileName.c_str());
    //Setting workname to filename. We use this value for DB store of works
    strcpy(workname,fileName.c_str());
    *out << "Salida: " << workname << endl;
}

// Register function to be called to instrument traces
//TRACE_AddInstrumentFunction(Trace, 0);

// Register function to be called for every thread before it starts running
//PIN_AddThreadStartFunction(ThreadStart, 0);

// Register Image to be called to instrument functions.
IMG_AddInstrumentFunction(Image, 0);
TRACE_AddInstrumentFunction(Trace, 0);
PIN_AddThreadStartFunction(ThreadStart, 0);
INS_AddInstrumentFunction(Instruction, 0);

PIN_AddSyscallEntryFunction(SyscallEntry, 0);
PIN_AddSyscallExitFunction(SyscallExit, 0);

// Register function to be called when the application exits
PIN_AddFiniFunction(Fini, 0);

cerr << "=====" << endl;
cerr << "This application is instrumented by pinSpark" << endl;
//err << "Valor negativo: " << test << endl;
if (!KnobOutputFile.Value().empty())
{
    cerr << "See file " << KnobOutputFile.Value() << " for analysis results" << endl;
}
cerr << "=====" << endl;

// Start the program, never returns
PIN_StartProgram();

return 0;
}

```

### 7.3 Codificación en Scala del fichero para Apache Spark. *WorkerCommandBuilder.scala*

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements. See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.spark.launcher

import java.io.File
import java.util.{HashMap => JHashMap, List => JList, Map => JMap}

import scala.collection.JavaConversions._

import org.apache.spark.deploy.Command

/** Imports for pinSpark */
import com.typesafe.config.{ Config, ConfigFactory }
import scala.util.Random

/**
 * This class is used by CommandUtils. It uses some package-private APIs in SparkLauncher, and since
 * Java doesn't have a feature similar to `private[spark]`, and we don't want that class to be
 * public, needs to live in the same package as the rest of the library.
 */
private[spark] class WorkerCommandBuilder(sparkHome: String, memoryMb: Int, command:
Command)
  extends AbstractCommandBuilder {

  childEnv.putAll(command.environment)
  childEnv.put(CommandBuilderUtils.ENV_SPARK_HOME, sparkHome)
}
```

```

override def buildCommand(env: JMap[String, String]): JList[String] = {
  val cmd = buildJavaCommand(command.classPathEntries.mkString(File.pathSeparator))
  cmd.add(s"-Xms${memoryMb}M")
  cmd.add(s"-Xmx${memoryMb}M")
  command.javaOpts.foreach(cmd.add)
  addPermGenSizeOpt(cmd)
  addOptionString(cmd, getenv("SPARK_JAVA_OPTS"))
  val configPin = ConfigFactory.parseFile(new File("etc/pinSpark/pinSpark.conf"))
  val cmd2: List[String] = List(configPin.getString("pinCommand"))
  val cmd21 = List.concat(cmd2,List(configPin.getString("pinToolSwitch")))
  val cmd22 = List.concat(cmd21,List(configPin.getString("pinTool")))
  val cmd23 = List.concat(cmd22,List(configPin.getString("pinOutSwitch")))
  //Generate unique filename, prefix from config file, path fixed on /tmp
  val logfile =
randomFileName(configPin.getString("pinOutLogFile"),"log",configPin.getString("pinComputerName"),10,
5)
  val cmd24 = List.concat(cmd23,List(logfile))
  val cmd25 = List.concat(cmd24,List(configPin.getString("pinAttachSwitch")))
  println(cmd25)
  val cmd3 = List.concat(cmd25,cmd)
  println(cmd3)
  cmd3
}

def randomFileName(prefix: String = "", suffix: String = "", computer: String = "", maxTries: Int =
10, nameSize: Int = 10) = {
  val alphabet = ('a' to 'z') ++ ('A' to 'Z') ++ ('0' to '9') ++ (" _")
  def generateName = (1 to nameSize).map(_ =>
alphabet(Random.nextInt(alphabet.size))).mkString
  val name1 = "/tmp/" + prefix + "_" + computer + "_" + generateName + "." + suffix
  name1
}

def buildCommand(): JList[String] = buildCommand(new JHashMap[String, String]())
}

```

## 8 Glosario de términos

### ***Instrumentación***

Proceso mediante el cual podemos analizar la ejecución de un determinado programa o binario.

### ***Proceso/Binario***

Programa a ejecutar

### **C++**

Lenguaje de programación extensión de C que incorpora conceptos de orientación a objetos.

### ***Worker***

Proceso que se ejecuta en un nodo Esclavo de Spark

### ***Trabajo/Job***

Ejecución que se lanza en un cluster de Apache/Spark. Usualmente esta ejecución se reparte entre los nodos del cluster

### ***Maestro/Nodo Maestro***

Proceso de Apache Spark que se encarga de coordinar la ejecución de trabajos.

### ***Esclavo/Nodo Esclavo***

Proceso de Apache Spark que se encarga de recibir trabajos, ejecutarlos y devolver el resultado al nodo maestro

## 9 Bibliografía

- *Apuntes UOC*
- *Apache Spark*, <http://spark.apache.org>
- *Relación de llamadas al sistema*, <http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64>
- *Paginas del manual (manpages)*

## 10 Referencias

- i UOC: <http://www.uoc.edu>
- ii PIN: <http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>
- iii Apache Spark: <http://spark.apache.org/>
- iv PIN: <http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>
- v [https://en.wikipedia.org/wiki/Infrastructure\\_as\\_a\\_service](https://en.wikipedia.org/wiki/Infrastructure_as_a_service)
- vi <https://hadoop.apache.org/>
- vii <http://www.scala-lang.org/>
- viii <https://www.python.org/>
- ix R Core Team (2015). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. URL <http://www.R-project.org/>
- x <http://aws.amazon.com>
- xi <https://maven.apache.org>
- xii *Pin: a dynamic binary instrumentation tool*: <http://www.pintool.org>
- xiii <http://www.cs.virginia.edu/kim/publicity/pin/LICENSE>
- xiv *Just-in-Time compilation*: [http://en.wikipedia.org/wiki/Just-in-time\\_compilation](http://en.wikipedia.org/wiki/Just-in-time_compilation)
- xv [https://es.wikipedia.org/wiki/Llamada\\_al\\_sistema](https://es.wikipedia.org/wiki/Llamada_al_sistema)