Facultad
de
Ciencias

Especificación y ejecución de benchmarks para arquitecturas centradas en el dato (Specification and execution of benchmarks for data-centered architecture)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERIA INFORMATICA

Autor: Martín de la Rubia, Luis

Director: Zorrilla Pantaleón, Marta Elena

Co-Director: Drake Monayo, Jose Maria

Junio - 2018

Resumen

El volumen, variedad y velocidad con la que se están generando datos ha traído consigo la aparición de numerosas tecnologías para su ingesta, tratamiento y uso en diferentes ámbitos. También ha contribuido al cambio en el modo de concebir los sistemas, pasando de entornos centralizados a distribuidos y escalables donde el dato es el centro de la arquitectura. Este proyecto se centra en la especificación y ejecución de un modelo que permita la definición de pruebas de benchmark adaptables a diferentes casos de usos y a distintas etapas de desarrollo del mismo. Su justificación se encuentra en la falta de estándares para realizar una comparativa entre tecnologías que se adapte a casos de uso propios y aplicables en entornos reales cuando otros procesos también existen. Este proyecto presenta el modelo medicionado y valida su adecuación con dos casos de uso propios de la Industria 4.0. Se desplegará su ejecución y se realizarán la medición de las métricas especificadas en los modelos.

Palabras Clave— Big data, sistemas distribuidos, benchmarks, métricas, Industria 4.0

Abstract

The volume, variety and speed with which data is being generated has led to the appearance of numerous technologies for its ingestion, management and use in different areas. It has also contributed to change the way in which systems are conceived, moving from centralized to distributed and scalable environments where the data is the center of architecture. This project focuses on the specification and execution of a model that allows the definition of benchmark tests adaptable to different use cases and to different stages of development. Its justification lies in the lack of standards to make a comparison between technologies that can be adapted to own use cases and applicable in real environments where other processes also exist. This project presents the mentioned benchmark and validates its suitability applying it to two use cases of the Industry 4.0 arena. Their execution will be deployed and the metrics specified in them will be measured.

Keywords— Big data, distributed system, benchmarks, performance metrics, Industry 4.0

Agradecimientos

Este trabajo ha sido financiado por el Gobierno de España a través de los proyectos:

- TIN2014-56158-C4-2-P: "Sistemas ciber-físicos de criticidad mixta sobre plataformas multinúcleo"
- TIN2017-86520-C3-3_R: "Sistemas informáticos predecibles y confiables para la Industria 4.0"

Así mismo mi agradecimiento al Ministerio de Educación, Cultura y Deporte, por la beca de colaboración que me ha sido concedida durante el curso 2017/2018.

Índice general

1 Introducción **3**

2 P3forI4: Plataforma de tercera generación propuesta **6**
2.1 Prometheus 7

3 Modelo de datos para la descripción de pruebas de Benchmark **9**

4 Caso de uso I: Empresa de distribución eléctrica **16**
4.1 Modelo 17
4.2 Validación del modelo 20
4.3 Métricas 22

5 Caso de uso II: P3SimMast **26**
5.1 Modelo 28
5.2 Validación del modelo 30
5.3 Métricas 32

6 Conclusiones y líneas futuras **35**

Bibliografía **35**

Apéndices **37**

A Modelo XML del caso de uso I: Empresa de distribución eléctrica **37**

B Modelo XML del caso de uso II: P3SimMast **44**

Capítulo 1

Introducción

La informática avanza cada vez más rápido y trae consigo nuevas posibilidades y retos. Actualmente estamos envueltos en la era del big data (ver Figura 1), donde volúmenes ingentes de datos y de distinta índole se generan, procesan y analizan para poder, a su vez, desarrollar procesos de comunicación, monitorización, integración o de inteligencia de negocio que permitan añadir valor a los negocios. Como consecuencia, se está produciendo un cambio de paradigma en la arquitectura de las aplicaciones, pasando de arquitecturas cliente/servidor a sistemas distribuidos y escalables que se se adaptan dinámicamente para hacer frente al volumen de computación que requieren en cada instante. Estos nuevos sistemas reciben el nombre de plataformas de tercera generación.

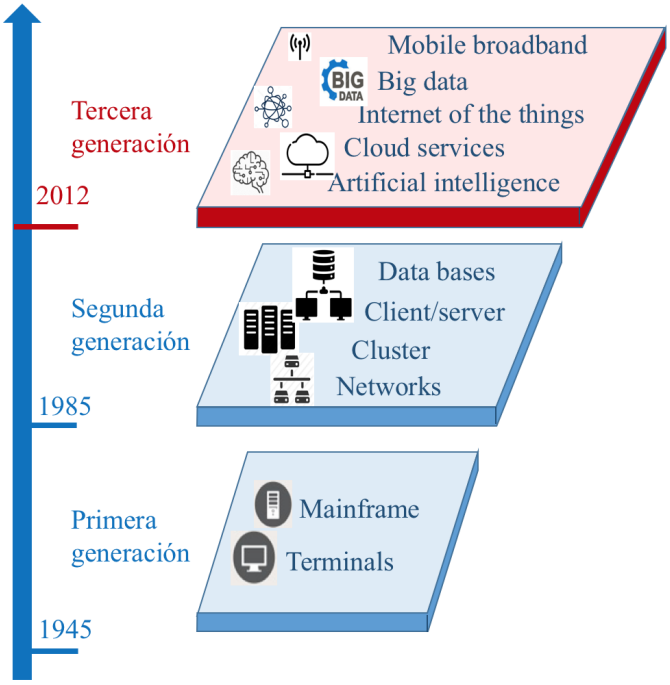

Figura 1: Evolución de las plataformas.

Estas plataformas se caracterizan por disponer de alta capacidad de computación tanto en entornos fijos como móviles (Ubiquitous & Mobile Computing), gestionar información masiva en tiempo real (Big Data) sin tener que almacenarla para procesarla en diferido, escalar dinámicamente la capacidad de procesamiento en base a su virtualización y deslocalización (Cloud & Fog Computing), facilitar la interacción de las aplicaciones con entornos inteligentes (IE & IoT) y redes sociales (Social Network) y hacer uso de nuevos paradigmas de computación basados en la inteligencia artificial (AI) y el aprendizaje automático (Machine Learning).

Existen diferentes opciones para implementar las plataformas de tercera generación. Nuestro trabajo se centra en plataformas compuestas por conjuntos de nodos de procesamiento físico o virtual, redes de comunicación y software de intermediación gestionados bajo dos paradigmas:

- **DaaS (Data as a Service)**: El dato constituye el nivel más alto de la arquitectura software. Los tipos de datos (topics) definen la información que representan, al igual que los requisitos no funcionales de la plataforma (persistencia, durabilidad, disponibilidad, seguridad, integridad, etc.). Las aplicaciones se conciben como subscriptoras a los tipos de datos disponibles en el sistema, y como publicadores de los nuevos datos que generan y que ponen a disposición de quien quiera utilizarlos ahora o en el futuro.

- **PaaS (Platform as a Service)**: Las plataformas están constituidas por conjuntos de recursos computacionales físicos disponibles en el propio entorno (fog) y/o recursos computacionales virtuales externos (cloud) que pueden ser reclutados dinámicamente en función de la carga de trabajo que se genere.

El grupo ISTR de la UC, dentro del proyecto del plan Nacional "Sistemas Informáticos Predecibles y confiables para la Industria 4.0" (TIN2017-86520-C3-3_R), está desarrollando la plataforma de tercera generación P3forI4 para dar soporte a la Industria 4.0.

La cuarta revolución industrial o Industria 4.0 (ver Figura 2), aborda la transformación de los procesos productivos en base al uso de la abundante información disponible sobre la totalidad del ciclo de vida de las cadenas de producción, logísticas y de consumo de cualquier sector industrial. Mientras que la tercera revolución industrial generalizó la digitalización de los ámbitos productivos logísticos y de consumo proporcionando ingentes volúmenes de información, la cuarta revolución industrial surge como consecuencia del IoT y la posibilidad de comunicación e interacción de todos los elementos de la cadena (máquinas, procesos y personas) por lo que su fin es desarrollar métodos de explotación de esa información para la mejora cualitativa de los procesos industriales y para la generación de nuevos valores añadidos basados en su procesamiento.

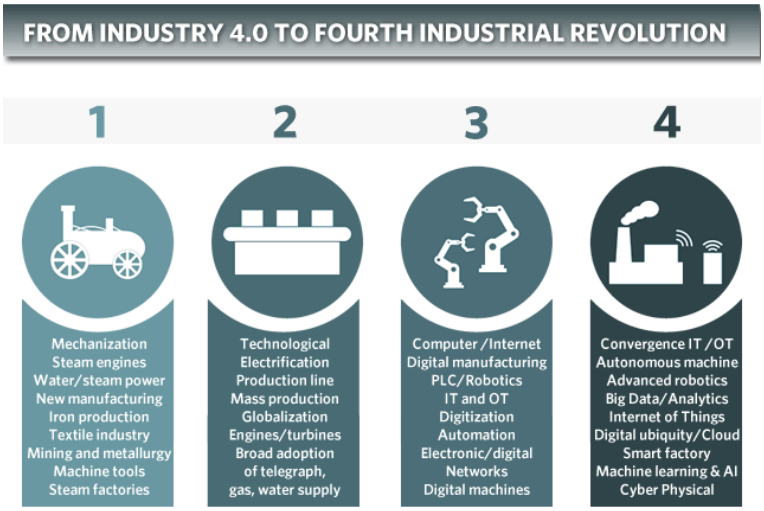

Figura 2: Transformación de la industria.[1]

Con el fin de elegir las tecnologías para construir esta plataforma fue necesario estudiar y comparar las existentes, implementar varios casos de uso y aplicar diferentes pruebas de evaluación. Uno de los problemas a los que nos enfrentamos es que dada la variabilidad de casos de uso y tecnologías, la definición e implementación de pruebas de rendimiento resultaban costosas por tenerse que definir e implementar manualmente; ya que esto requiere disponer del código fuente de las aplicaciones para adaptarlo a las interfaces de las plataformas, reemplazar los elementos que no están disponible con emuladores, e introducir y administrar herramientas de instrumentación y monitorización para evaluar la prueba. Actualmente no hay un benchmark que se puede utilizar directamente, salvo algunos casos de uso desarrollados por empresas como Yahoo [2] o Facebook [3]. Por ello uno de los objetivos del proyecto es diseñar e implementar una herramienta MDE (Model Driven Engineering) que permita configurar, desplegar, ejecutar y comparar métricas de rendimiento de aplicaciones desarrolladas bajo una plataforma de tercera generación. Para ello es necesario disponer de un modelo de datos que describa las pruebas con independencia de las tecnologías de la plataforma y de la naturaleza de las propias pruebas.

El modelo propuesto debe incluir información sobre todos los aspectos de la prueba. La plataforma donde se ejecuta, las fuentes que generan los datos, las tareas de procesamiento y las métricas que deben ser evaluadas. Este se ha diseñado para que permita la comparación de las prestaciones de diferentes plataformas en la ejecución de una misma aplicación; el efecto sobre la ejecución de la aplicación del escalado y configuración de la plataforma; la repercusión de la ejecución de la aplicación sobre otras aplicaciones que también se están ejecutando en la plataforma; y la evaluación de la eficiencia de la descomposición en componentes de la aplicación para su ejecución sobre la plataforma.

Mi participación en este proyecto ha sido mediante una beca de colaboración con el Departamento de Ingeniería Informática y Electrónica, que se encuentra en el proceso de implementar dicha herramienta. Mi contribución personal ha sido la participación en el diseño del modelo de datos para describir las pruebas de benchmark, la validación del modelo de datos para casos de usos concretos y el análisis, y puesta a punto de las tecnologías utilizadas, en especial, la monitorización mediante la configuración de la herramienta Prometheus y el análisis de las métricas.

Capítulo 2

P3forI4: Plataforma de tercera generación propuesta

Aunque existen muchas herramientas software para implementar plataformas de tercera generación, nuestro grupo de investigación ha planteado una solución usando tecnologías del proyecto de software libre Apache [4] principalmente (ver Figura 3). En concreto, las tecnologías seleccionadas son:

- **Zookeeper**: Es un servicio de coordinación para sistemas distribuidos, incluye servicios como registro de nombres o gestión de la configuración, proporciona gran disponibilidad gracias a la redundancia de los servicios, así si algún nodo no está disponible se puede preguntar a otro [5]. Nosotros lo usamos como middleware de soporte de información distribuida.
- **AVRO**: Es un sistema de serialización de información. Utiliza JSON para la definir tipos de datos y protocolos y serializa los datos en un formato binario compacto [6].
- **Kafka**: Es un sistema de mensajería distribuido basado en Publicadores y Subscriptores. Los publicadores introducen los flujos de datos (topics) y los subscriptores leen estos en crudo o una vez procesados. Cada topic lo mantiene en una o varias particiones que son inmutables y accesibles secuencialmente hasta que se pida su eliminación.[7].
- **Spark**: Es un motor de análisis para el procesamiento de grandes cantidades de información de forma distribuida [8]. Este se utiliza como planificador de la ejecución de las tareas de procesamiento.
- **Cassandra**: Es una base de datos altamente escalable, eventualmente consistente y distribuido de estructuras clave-valor. Su arquitectura se compone de varios nodos conectados mediante un protocolo P2P (peer to peer) [9].
- **MemSQL**: Es otra alternativa de base de datos distribuida y en memoria, pero que garantiza la consistencia de los datos en todo momento. Su arquitectura está formada con varios nodos en el que uno asume el rol de maestro y el resto son esclavos, las peticiones llegan al maestro que divide el trabajo entre los nodos esclavos [10].
- **Prometheus**: Consiste en una herramienta encargada de la monitorización de métricas, recoge las medidas de aplicación y las guarda para su posterior análisis [11].

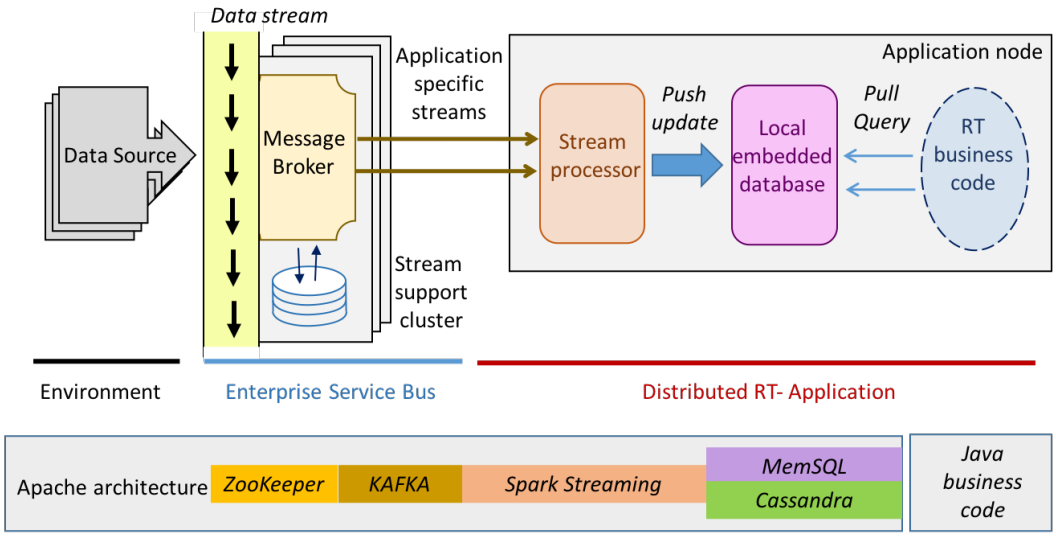

Figura 3: Esquema de la plataforma P3forI4.

2.1 Prometheus

Prometheus es un herramienta open-source para monitorizar sistemas y gestionar alertas. Fue originalmente creado por SoundCloud pero fue adoptado por numerosas empresas y ahora dispone de una gran comunidad [11]. Funciona recogiendo datos de aplicaciones durante su ejecución para su posterior análisis.

En la Figura 4, se muestra la arquitectura que usa Prometheus. Como entrada de datos puede utilizar aplicaciones instrumentadas, "Exporters", que son programas desarrollados por Prometheus o gente de la comunidad que habilita la recolección de métricas, por ejemplo, bases de datos, sistemas operativos, sistema de almacenamiento, protocolos http o cualquier tipo de aplicación. Luego en el servidor se almacenan todos los datos a los que posteriormente se puede acceder desde la interfaz web que ofrece Prometheus, bien desde la API de consulta o mediante Grafana, una aplicación web para la representación de analíticas en forma de gráficas. Prometheus también ofrece un sistema de alertas.

Para la instrumentación de aplicaciones creadas por el usuario, Prometheus proporciona unas librerías cliente para recoger los datos que se deseen y de manera sencilla ponerlos a disposición del servidor para que los recoja. Estas librerías permiten almacenar los datos de diversas maneras. Es posible crear contadores, "Gauge", indicador visual similar a los disponibles en los cuadros de mandos de los vehículos, y obtener la serie temporal de alguna variable en un periodo de tiempo para visualizarlo en histogramas

Otra opción para la recolección de métricas es la de usar los programas que proporciona Prometheus o la comunidad, denominados Exporters, que exponen métricas de diversas aplicaciones para que el servidor de Prometheus pueda obtenerlas. Esta será la estrategia a seguir en caso de no disponer del código para instrumentar las aplicaciones legadas que se vayan a medir. Los exporters que nosotros usaremos son el Node exporter y el JMX exporter. Con el primero podremos obtener

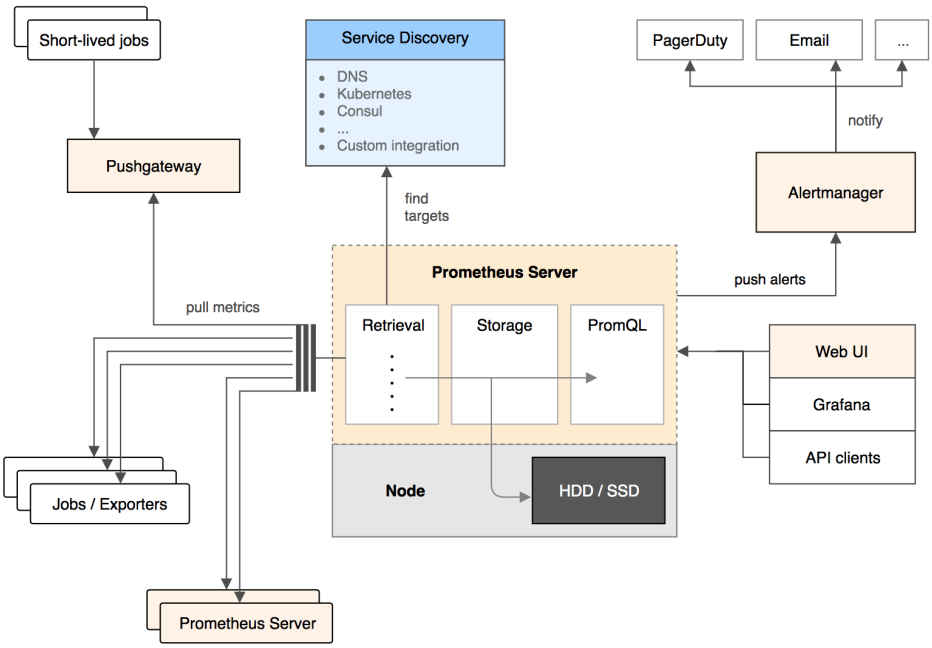

Figura 4: Arquitectura de Prometheus.[11]

todas las métricas que proporciona el sistema operativo donde se ejecute como estadísticas de la cpu, memoria, wifi, etc. El segundo nos ayudará a capturar todas las métricas de las máquinas virtuales Java, ya sean de las aplicaciones del usuario escritas en java como del software usado en la infraestructura en la que se ejecutará la prueba de benchmark como son Kafka, Zookeeper o Spark.

Capítulo 3

Modelo de datos para la descripción de pruebas de Benchmark

El modelo de datos para la descripción de una prueba de benchmark incluye toda la información que se requiere para satisfacer los siguientes casos de uso [12]:

- Comparación de tecnologías: comparar las prestaciones y rendimiento de diferentes plataformas, cada una de ellas implementadas con una tecnología diferente.
- Evaluación de escalado y configuración de la plataforma: evaluar el dimensionado de los recursos que constituyen la plataforma y la configuración de los recursos de la misma.
- Diseño modular de las aplicaciones: evaluar las alternativas del diseño de componentes de las aplicaciones en lo que concierne al uso eficiente de los recursos frente al escalado de la plataforma.
- Valoración del efecto sobre el sistema del despliegue de una aplicación: valorar el efecto que sobre un entorno en operación produce el despliegue y ejecución de nuevas aplicaciones.

La Figura 5 muestra los tipos de datos de más alto nivel del modelo que describe una prueba de benchmark. La clase raíz, *P3BM_Proof*, identifica la prueba con información básica sobre ella y constituye un contenedor de elementos del modelo (*P3BM_Element*). La clase abstracta *P3BM_NamedElement* se introduce para establecer que todos los elementos del modelo tengan nombre e identificador. Un modelo puede referenciar a otros modelos (*referencedProof*), y a través de ello, se facilita la organización del modelo con secciones reutilizables e intercambiables entre pruebas.

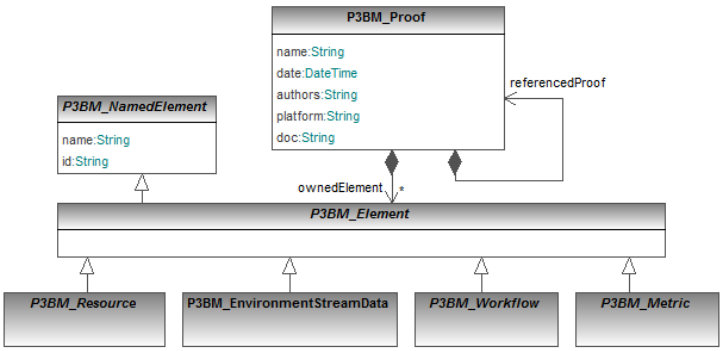

Figura 5: Clases raíces del modelo de una prueba de benchmark.

Las clases *P3BM_Resource*, *P3BM_EnvironmentStreamData*, *P3BM_Workflow* y *P3BM_Metric* son los cuatro tipos de elementos que intervienen en la prueba de benchmark: las características de los recursos que constituyen la plataforma, lo flujos de datos que introduce el entorno y que constituyen la carga de trabajo, las tares de procesamiento que se ejecutan y las medidas que deben ser evaluadas en la ejecución de la prueba.

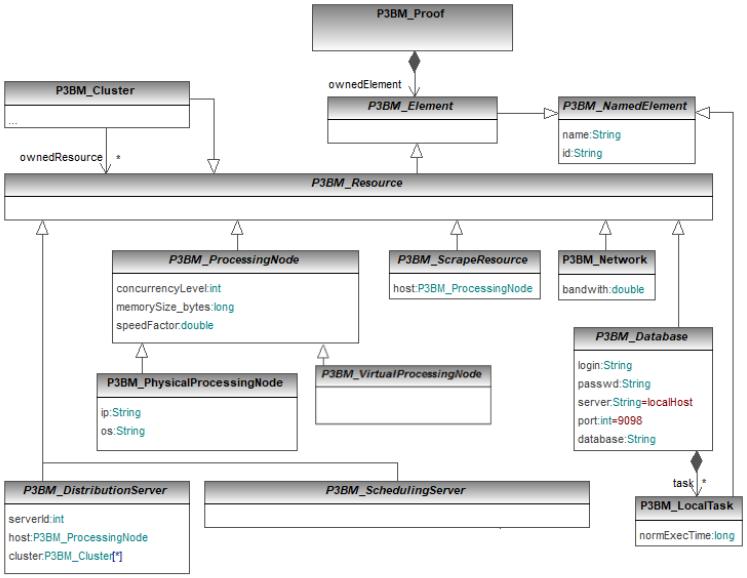

Figura 6: Tipos de datos genéricos que describen los recursos de la plataforma.

La Figura 6 muestra tipos de datos genéricos que describen la información de los recursos de la plataforma. *P3BM_ProcessingNode* caracteriza los nodos de procesamiento de la plataforma. En este nivel de abstracción solo se incluye información general como el número de nodos de los que dispone (concurencyLevel), la cantidad de memoria (memorySyze_bytes) y su capacidad de procesamiento relativa (speedFactor). Se contempla sus implementaciones como nodos físicos disponibles

en el entorno (P3BM_PhysicalProcessingNode) o como recursos virtuales contratados en la nube (P3BM_VirtualProcessingNode). P3BM_Network caracteriza el efecto sobre la prueba de benchmark de la red de comunicaciones junto al servicio de comunicaciones que la gestiona indicando el ancho de banda del que se dispone (bandwith).

P3BM_SchedulingServer identifica y caracteriza el planificador distribuido de tareas de la plataforma que gestiona la asignación de las tareas de procesamiento a los recursos de la plataforma. P3BM_DistributionServer describe la presencia de un recurso de gestión del sistema distribuido que facilita el acceso seguro a la información compartida desde todos los nudos, así como la replicación en segundo plano de la información almacenada en los nodos para garantizar su integridad frente a caída de los mismos. P3BM_Database caracteriza los recursos de almacenamiento de datos utilizados en ciertas tareas que persisten la información. Por último, P3BM_Cluster, permite agrupar conjuntos de recursos presentes en la plataforma a fin de que puedan ser referenciados como una agrupación.

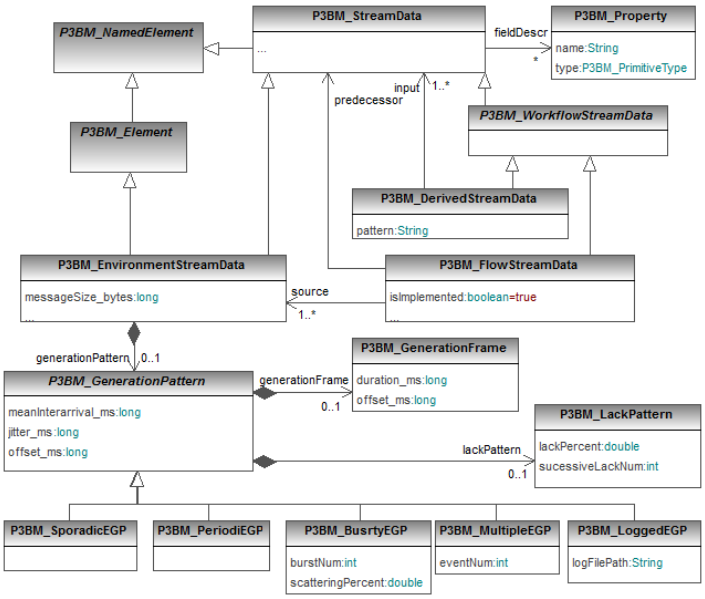

Figura 7: Tipos de datos que describen un flujo de datos.

La Figura 7 muestra los tipos de datos que caracterizan a los flujos de datos de una prueba de benchmark. Cualquier tipo de flujo de datos tiene como raíz el tipo de datos abstracto P3BM_StreamData y tiene como atributos identificadores heredados de P3BM_NamedElement, una longitud de mensaje (messageSize_bytes) y un conjunto de campos específicos (fieldDescription) definidos con un identificador (clave) y un tipo de dato. Se consideran tres tipos de flujos de datos:

- P3BM_EnvironmentStreamData: Describe los flujos de datos que son generados por el entorno y que definen la carga de trabajo a la que está sometido el sistema en la prueba. Los flujos de datos de este tipo son declarados dentro de la lista ownedElement de la prueba. Si la prueba de benchmark se ejecuta sobre un entorno real y el flujo de eventos existe, la descripción sólo incluye su identificación, por el contrario, si la prueba se lleva a cabo en fase la fase de prototipo se le asocia una estructura de datos (P3BM_GenerationPattern), que proporciona

la información necesaria para que su generación sea emulada por el gestor del benchmark. En la Figura 7 se muestra las alternativas de generación que se han definido. P3BM_LackPattern define el patrón de generación de datos faltantes, si fuera preciso modelarlo.

- P3BM_FlowStreamData: Este tipo de flujos de datos, representa el medio con el que se describe el flujo de control y de datos entre los componentes que ejecutan las tareas de las aplicaciones que se encuentran desplegados en cualquier nudo de la plataforma. Los flujos de datos de este tipo se declaran agregados (workflowEvent) dentro de los elementos P3BM_Workflow que describen las aplicaciones. Los atributos source y predecessor, se introducen con el fin de seguir las trazas del flujo de control de las aplicaciones y evaluar las latencias de las tareas y los tiempos de ejecución de las aplicaciones.
- P3BM_DerivedStreamData: Representa flujos de datos virtuales que resultan de aplicar una función de filtro a uno o varios flujos de datos que se generan en la prueba de benchmark. Los dos atributos específicos del tipo de dato que describe estos eventos son la lista de flujos de datos que se referencian en la función de filtro (input), y la propia función de filtro (pattern) que se formula como una sentencia EPL [13] que es de tipo SQL pero con una cláusula RETAIN que restringe la consulta a ventanas temporales o por número de datos tanto de forma estática como deslizante.

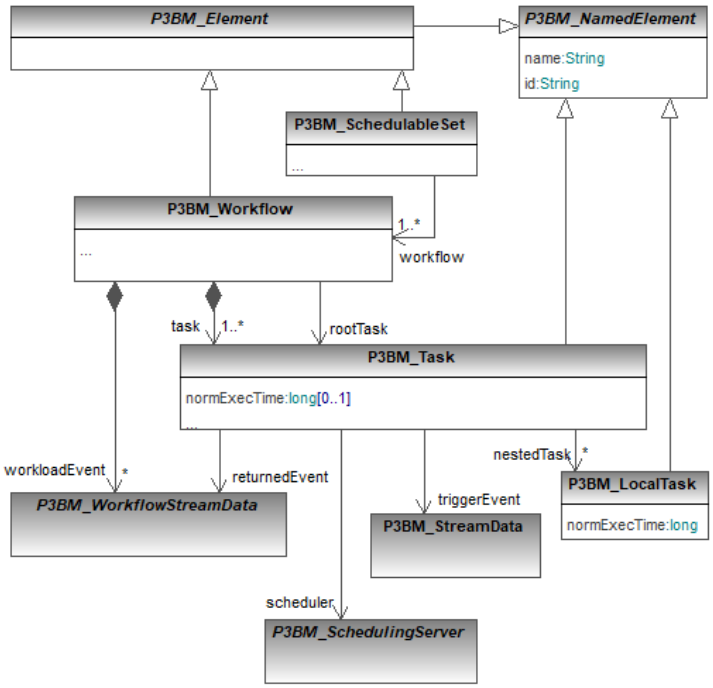

Figura 8: Tipos de datos que describen un flujo de procesamiento.

La Figura 8 muestra el tipo de datos P3BM_Workflow que describe un flujo de procesamiento en una prueba de benchmark. Se compone de tareas de procesamiento (task) y de los enventos de control de flujo (workflowStreamData). El atributo rootTask indica qué tarea inicia dicho flujo de procesamiento.

Una tarea representa una actividad que es parte de un flujo de procesamiento y que ejecuta un componente software desplegable. La tarea (P3BM_Task) incluye referencia triggerEvent que indica el flujo de datos que inicia su ejecución, también la referencia returnedEvent indica el flujo de dato que se genera al finalizar la tarea para así tener la trazabilidad completa de la entrada y salida de cada tarea. El atributo scheduler indica el recurso de la plataforma que reparte la carga de procesamiento. El atributo normExecTime representa el tiempo medio de ejecución de la tarea normalizado a un nodo de procesamiento de referencia (speedFactor=1.0). Este atributo es solo usado en la fase de prototipo de la plataforma para realizar la simulación del componente. P3BM_LocalTask hace referencia a las operaciones locales a la tarea, que se hacen haciendo uso de otros recursos de invocación, por ejemplo, una llamada a la base de datos dentro de esa tarea.

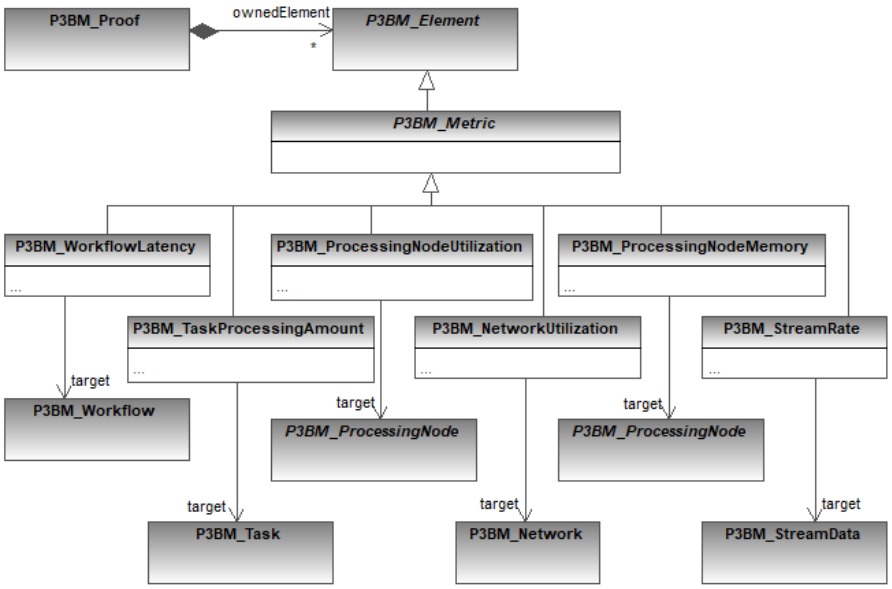

Figura 9: Tipos de datos que describen métricas que pueden ser evaluadas en el benchmark.

La Figura 9 se muestran los tipos de datos que describen las medidas que se pueden evaluar durante la ejecución de la prueba de bechmark. Como lo habitual es tratar con código legado, no se basan en la instrumentación del código, sino en requerir las medidas de los servicios de monitorización que ofrecen los recursos o el middleware. Por ello, la información que incluye el modelo sólo especifica la naturaleza de la medida e identifica el elemento al que se aplica (target). Las métricas que pueden aplicarse a una prueba dependen de los objetivos del benchmark y pueden ser muy variadas. Las métricas ahora incluidas en el modelo de datos son:

- P3BM_WorkflowLatency: Latencia del flujo de procesamiento.
- P3BM_TaskProcessingAmount: Cantidad de procesamiento de la tarea.
- P3BM_ProcessingNodeUtilization: Utilización de la cpu del nodo indicado.
- P3BM_NetworkUtilization: Utilización de la red sobre la que se ejecuta la prueba de benchmark.

- P3BM_ProcessingNodeMemory: Cantidad de utilización de uso de la memoria del nodo indicado.
- P3BM_StreamRate: Frecuencia de mensajes y e información que se generan en la prueba de un determinado flujo de datos.

Otras métricas sobre consumo de energía, fiabilidad, disponibilidad, etc. pueden ser incluidas si los servicios de monitorización de la plataforma las evalúan.

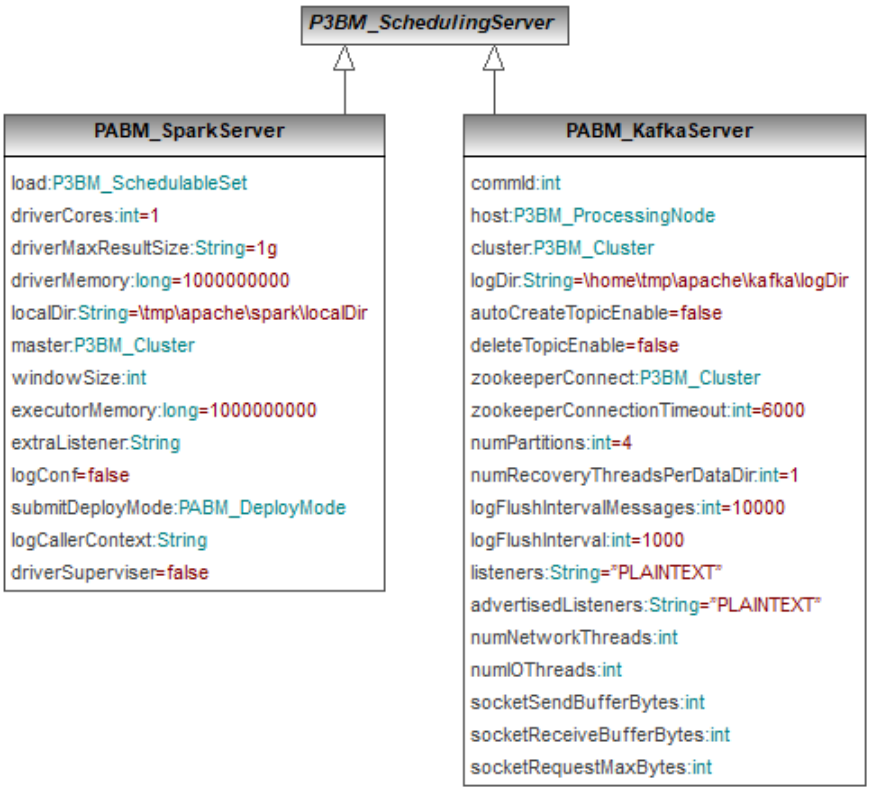

Figura 10: Personalización de los tipos de datos para la plataforma P3forI4.

Las Figuras 10 y 11 muestran la especificación para describir pruebas de benchmark en la plataforma P3forI4 basada en tecnología del proyecto de software libre Apache. Los tipos de datos abstactos *P3BM_SchedulingServer*, *P3BM_DistributionServer*, *P3BM_ScrapeResource* y *P3BM_Database* describen la estrategia de despliegue de cada elemento con la tecnología concreta que se use. Estas incluyen los datos de configuración necesarios para desplegar dichos recursos.

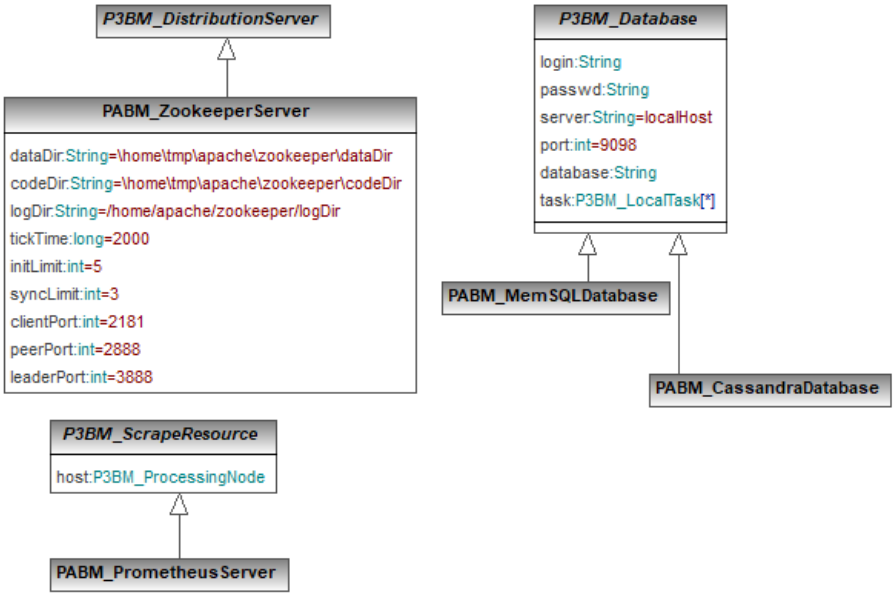

Figura 11: Personalización de los tipos de datos para la plataforma P3forI4.

Capítulo 4

Caso de uso I: Empresa de distribución eléctrica

Con objeto de mostrar la validez del modelo, en este capítulo se muestra cómo se instancia para un caso de uso real propuesto para una empresa del sector eléctrico.

Con la implantación de los contadores eléctricos ya pueden realizar la lectura de ellos de forma remota, evitando así que el personal de la compañía se tenga que desplazar hasta cada domicilio. Cada madrugada, los contadores envía una curva horaria (CH) de consumo (un valor por cada hora) junto con un parámetro que permite saber si el valor es correcto o no. A partir de las 4:00 am se empiezan a procesar las curvas de los 700000 clientes con objeto de almacenarlas en una base de datos para mensualmente facturar a sus clientes por consumo real. Este proceso tarda actualmente en un solo servidor aproximadamente unas 12h. También la lectura real del contador (RM) llega una vez al mes con el consumo real del cliente.

El procesamiento de las curvas no es trivial. Como se observa en la Figura 12, se basa en leer diariamente cada curva, comprobar si es válida, esto es no tiene ninguna componente del vector con el flag a false, y en ese caso se almacena en la base de datos; en caso contrario, se debe rellenar cada lectura incorrecta con el valor obtenido de los consumos históricos del cliente antes de guardar. Cuando llega una lectura real, una vez al mes, se debe recalcular los consumos estimados por los reales y guardar esta última estimación como el consumo real del cliente.

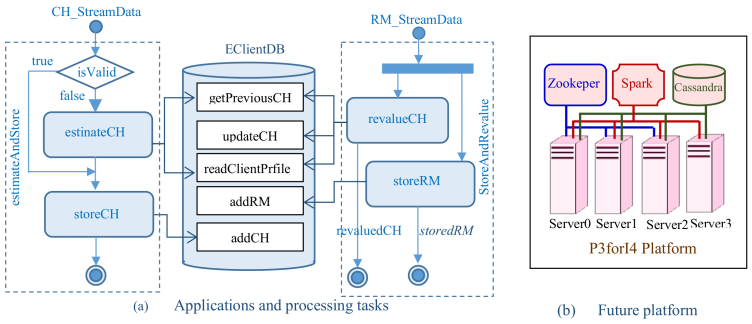

Figura 12: Aplicaciones y futura plataforma del ejemplo.

Debido a la previsión del aumento de clientes, esta plataforma debe escalar para poder realizar el procesamiento en un tiempo razonable. Por ello se ha reproducido esta funcionalidad en una arquitectura escalable y distribuida utilizando el ecosistema Apache. En particular, como se muestra en la Figura 12 se utiliza Zookeeper para garantizar la integridad de los datos, Spark para la ejecución de las tareas de forma distribuida y como base de datos Cassandra.

Para la evaluación del sistema en esta nueva plataforma se usarán datos históricos de la compañía que se componen de un mes de curvas horarias, un año de consumos y un año de perfiles. Cabe señalar que este TFG no se encarga de la implementación de este procesamiento sino de verificar que la especificación del benchmark es suficiente para poder ejecutar y obtener las métricas especificadas.

4.1 Modelo

A continuación se presenta gráficamente la definición del caso de uso. En la Figura 13, se compone de las estructuras de datos que describen la plataforma, la fuente de esos datos, las tareas de procesamiento y las métricas que deben de ser evaluadas.

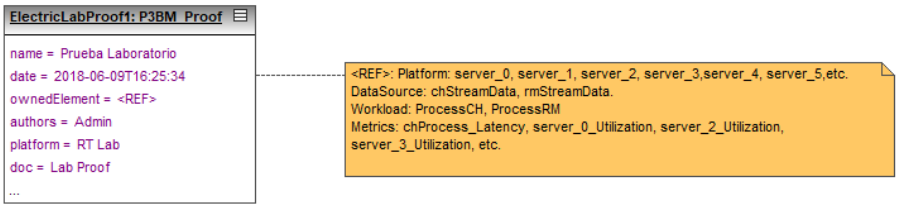

Figura 13: Definicion de la prueba.

Figura 14: Descripción de los recursos del caso de uso I.

En la Figura 14 se describe la plataforma. De cada nudo de procesamiento (server_?) se proporciona sus identificadores (name, id, ip, os), el número de procesadores (concurrencyLevel), la memoria de que disponen (memorySize_bytes) y su capacidad de procesamiento (speedFactor), que en este caso, es igual a la del mainframe en el que ha sido evaluada la carga de procesamiento de las

Figura 15: Descripción de los recursos de la prueba del caso de uso I.

tareas. Los zk_Server_? describe los gestores de distribución. Mantiene 9 réplicas de la información en los nudos de procesamiento del zkCluster (ver Figura 15), por lo que garantiza la integridad de los datos frente a la caída de un nudo. Sincroniza las réplicas cada 4 segundos (tickTime*syncLimit) y comprueba la operatividad de los nudos cada 10 segundos (tickTime*initLimit). De cada recurso encargado de la mensajería (kafka_server_?) entre las aplicaciones del sistema se proporciona a parte de la información básica, datos de la configuración de Kafka como deleteTopicEnable, para indicar si se habilita el borrado de tópicos si el cliente lo desea.

En la Figura 15, sparkServer describe el planificador de la ejecución de las tareas sobre la plataforma distribuida. Describe los nudos de la plataforma (master) en los que despliegan las tareas de las aplicaciones especificadas (apps), el número de procesadores que utiliza en cada nudo (driverCores), y la longitud en mensaje de las tandas que utiliza (windowSize). Los resultados son persistidos en una base de datos de tipo Cassadra (cassndra_server_?), cuyos datos de acceso están incluidos (port, database). También se define los clústers, conjunto de nodos, sobre que se desplegará la aplicacion, así como el recurso para la recolecta de las métricas, prServer, indicando en que nodo se alojará (host).

En la Figura 16 se muestra como ejemplo uno de los dos flujos de datos de entrada que se procesan. El flujo chStreamData se transmite en mensajes de 4096 bytes (messageSize_bytes), se distribuye en 80 particiones (numPartitions) para garantizar la máxima concurrencia, es replicados por triplicado (numReplication), se mantiene persistido durante 5 horas (logRettention_ms) y tiene declarado el atributo isValid (fieldDescr). Se genera (generationPattern) con los datos del fichero chHistoricalData.csv, que se introducen durante 30 minutos a partir de las 3 horas de la mañana. Los datos se inyectan (generationFrame) con un marco de repetición diario (duration_ms).

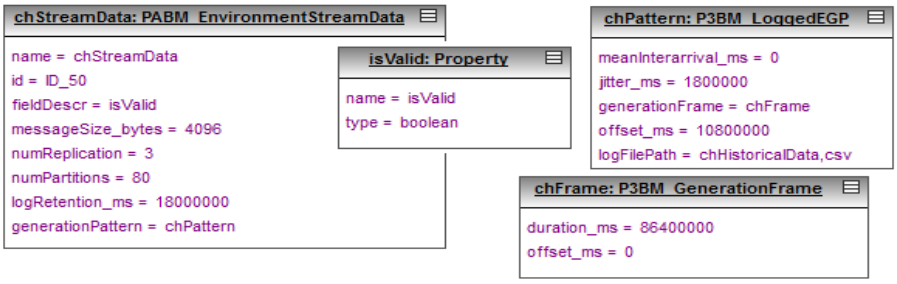

Figura 16: Descripción del flujo de entrada chStreamData del caso de uso I.

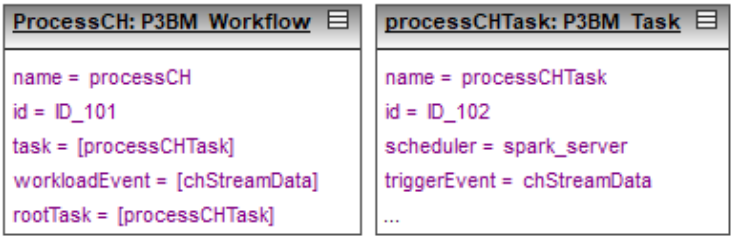

Figura 17: Descripción de la aplicación processCH del caso de uso I.

En la Figura 17, se muestra como ejemplo la descripción de la aplicación processCH, está compuesta por una tarea processCHTask, que se iniciará con el flujo de datos chStreamData (triggerEvent) y será manejado la ejecución por el planificador spark_server.

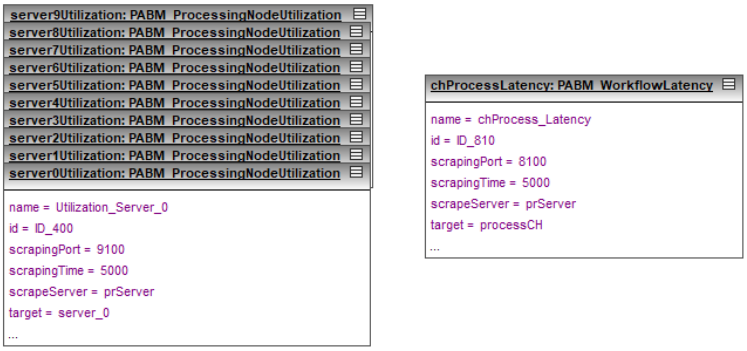

Figura 18: Descripción de las métricas a aplicar en el caso de uso I.

En la Figura 18 se describen las métricas que se aplican en la prueba. Estas son el porcentaje de utilizacion de la CPU de los diez nodos de procesamiento, y el tiempo de ejecución de la aplicación que procesa las curvas diarias.

La representación de este modelo de datos en formato XML se puede encontrar en el Apéndice A.

4.2 Validación del modelo

En esta sección comprobaremos que en el modelo de datos descrito previamente se encuentra toda la información para poder desplegar la plataforma, iniciar el procesamiento y realizar las monitorización que se haya especificado.

El primer paso es desplegar todos los recursos necesarios, esto es, ZooKeeper, Kafka, Spark y Cassandra. Para desplegar Zookeeper tenemos que primero configurarlo, para ello se debe escribir el fichero de configuración zookeeper.properties con la información que aparece en el modelo. En este caso de uso se ejecuta con el servidor replicado en todos los nodos como se indica con el atributo cluster.

```
tickTime=2000
initLimit=5
clientPort=2181
peerPort=2888
leaderPort=3888
initLimit=5
syncLimit=2
server.0=server_0:2888:3888
server.1=server_1:2888:3888
server.2=server_2:2888:3888
server.3=server_3:2888:3888
server.4=server_4:2888:3888
server.5=server_5:2888:3888
server.6=server_6:2888:3888
server.7=server_7:2888:3888
server.8=server_8:2888:3888
```

Una vez que la configuración está escrita desplegamos Zookeeper en todos los nodos que se indican ejecutando el siguiente comando:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Ahora que tenemos desplegado el Zookeper necesitamos poner en marcha el Kafka, al igual que la anterior herramienta tiene un fichero de configuración, server.properties, en el que debemos incluir la informacion indicada en el modelo. Para el id que necesita podemos usar el mismo que identifica al servidor. Hace falta también indicar los servidores de Zookeeper disponibles, que sabemos pues están en la descripción de los recursos.

```
broker.id=0
port=9092
auto.create.topics.enable=true
delete.topic.enable=true
num.partitions=1
log.flush.interval.messages=10000
log.flush.interval.ms=1000
zookeeper.connect= server_0:2181, server_1:2181,
    server_2:2181,server_3:2181, server_4:2181, server_5:2181,
    server_6:2181, server_7:2181, server_8:2181
```

Con la configuración definida al igual que con Zookeeper arrancamos los servidores ejecutando el siguiente comando en todos los servidores de la plataforma como indica la descripción de la prueba.

```
bin/kafka-server-start.sh config/server.properties
```

Para terminar de configurar Kafka es necesario definir los tópicos, es decir, los flujos de datos descritos en el modelo de la prueba. En este caso solo tenemos chStreamData como flujo de datos en el sistema. Se crea con los valores definidos como atributos en el modelo, ejecutando el siguiente comando en cualquiera de los nodos que usan Kafka.

```
bin/kafka-topics.sh --create --zookeeper server_0:2181,
    server_1:2181, server_2:2181,server_3:2181, server_4:2181,
    server_5:2181, server_6:2181, server_7:2181, server_8:2181 --
    replication-factor 3 --partitions 80 --topic chStreamData
```

Los valores de –zookeeper son todos los servidores donde se desea que se replique el tópico creado, los valores –replication-factor y –partitions se obtienen de los atributos numReplication y numPartitions respectivamente de la descripción del modelo. Con esto ya tenemos los servidores de Kafka y Zookeeper desplegados y configurados, A continuación pasaremos a iniciar la base de datos Cassandra, para ello también tenemos que configurarlo a través del fichero de configuracion cassandra.yaml. Los datos que deberemos modificar son los siguientes:

```
cluster_name= globalCluster
seeds= server_0, server_1, server_2, server_3, server_4, server_5,
    server_6, server_7, server_8, server_9
listen_address= server_0
```

En cada servidor el listen_address será la dirección de cada nodo, en este ejemplo se muestra para el server_0. A continuación para desplegar Spark, a partir del modelo sabemos que el nodo que actuará como master es el server_0, y los workers serán sobre el clúster que trabaje en este caso globalCluster, por lo que se desplegará en todos los nodos. Para iniciar el master se ejecutará este comando en server_0.

```
./sbin/start-master.sh -c 4
```

Con -c indicamos el número de cores que usará Spark, como se define en el modelo, serán 4. Para iniciar los workers se ejecutara el siguiente comando en el resto de nodos de la plataforma indicando la dirección del servidor master.

```
./sbin/start-slave.sh -h spark:\\server_0:7077
```

Por último para lanzar la aplicación que se va a ejecutar sobre Spark, necesitaríamos saber el paquete donde se encuentra la clase principal de la aplicación así como su nombre, son datos que faltan en el modelo de datos.

```
-/spark-sumbit -class
    es.unican.electricity.consumers.curves.CurveProcessor --deploy-mode
    cluster --master spark://\textcolor{red}{server\_0:6066}
    \textcolor{red}{electricity-consumer-curves-assembly-1.0.
    jar}
```

Con este comando Spark, ejecutará la aplicación electricity-consumer-curves-assembly-1.0.jar en modo clúster sobre toda la plataforma. Para finalizar, hay que iniciar la producción del flujo de datos, como conocemos el patrón de generación, que es a partir de un fichero de datos llamado chHistoricalData.csv hacemos uso de un programa llamado streamingestor que a partir de un fichero te permite introducir los valores en kafka del tópico que le indiques. Se ejecuta con el siguiente comando:

```
java -jar streamingestor.jar -f chHistoricalData.csv -t 15000 -T
    chStreamData -d 6000 -p server_0:9092, server_1:9092,
   server_2:9092, server_3:9092, server_4:9092, server_5:9092,
   server_6:9092, server_7:9092, server_8:9092, server_9:9092
```

Con esto hemos comprobado que los valores necesarios para poner en marcha la prueba de benchmark se encuentran en el modelo de datos que hemos definido.

4.3 Métricas

En esta sección analizaremos cómo a partir de la descripción de la prueba de benchmark podemos realizar la configuración del servidor Prometheus de monitorización y cómo se pueden realizar las medidas indicadas.

Una de las métricas que tenemos que medir es ProcessingNodeUtilization que proporciona el porcentaje de utilización de la CPU durante la ejecución de la prueba de benchmark. Para ello primero hay que configurar el Node exporter, ya que es el que nos proporciona la información del sistema operativo. Para que el programa nos ofrezca solo la información de la CPU es necesario indicarlo al invocarlo con el siguiente comando que se ejecutará en todos los nodos donde queremos medir esta métrica. Este comando además desactiva las demás funciones que por defecto envía este exporter.

```
./node_exporter --no-collector.arp --no-collector.bcache
   --no-collector.bonding --no-collector.conntrack
   --no-collector.diskstats --no-collector.edac --no-collector.entropy
   --no-collector.exec --no-collector.filefd --no-collector.filesystem
   --no-collector.hwmon --no-collector.infiniband --no-collector.ipvs
   --no-collector.loadavg --no-collector.mdadm --no-collector.meminfo
   --no-collector.netdev --no-collector.netstat --no-collector.nfs
   --no-collector.nfsd --no-collector.sockstat --no-collector.stat
   --no-collector.textfile --no-collector.time --no-collector.timex
   --no-collector.uname --no-collector.vmstat --no-collector.wifi
   --no-collector.xfs --no-collector.zfs
```

Para que Prometheus lea la información de este programa hay que indicarlo en su fichero de configuración que se llama prometheus.yml. Como conocemos que el Node exporter publica las métricas en el puerto 9100 por lo que , el fichero de configuración quedaría de la siguiente forma:

```
global:
        scrape_interval:     15s
        evaluation_interval: 15s

        scrape_configs:
        - job_name: Utilization_Server_0
                scrape_interval: 5s
                static_configs:
                - targets: ['server_0:9100']

        - job_name: Utilization_Server_1
                scrape_interval: 5s
                static_configs:
```

```
            - targets: ['server_1:9100']

    - job_name: Utilization_Server_2
            scrape_interval: 5s
            static_configs:
            - targets: ['server_2:9100']

    //Lo mismo con el resto de nodos de la plataforma
```

Para iniciar el servidor de Prometheus simplemente se ejecuta el siguiente comando:

```
./prometheus --config.file=prometheus.yml
```

Una vez que el servidor de Prometheus está en marcha lo que recibe el servidor por cada core de la CPU del nodo es lo siguiente:

```
# HELP node_cpu Seconds the cpus spent in each mode.
# TYPE node_cpu counter
node_cpu{cpu="cpu0",mode="guest"} 0
node_cpu{cpu="cpu0",mode="guest_nice"} 0
node_cpu{cpu="cpu0",mode="idle"} 230.7
node_cpu{cpu="cpu0",mode="iowait"} 3.1
node_cpu{cpu="cpu0",mode="irq"} 0
node_cpu{cpu="cpu0",mode="nice"} 3.79
node_cpu{cpu="cpu0",mode="softirq"} 0.62
node_cpu{cpu="cpu0",mode="steal"} 0
node_cpu{cpu="cpu0",mode="system"} 11.94
node_cpu{cpu="cpu0",mode="user"} 41.18
```

Con estos datos, preguntando al servidor de Prometheus entre el momento en el que comenzó el benchmark y en la que acabó podemos saber el porcentaje de utilización ya que tenemos el tiempo gastado en modo sin trabajo (idle) y el tiempo gastado en el resto de modos.

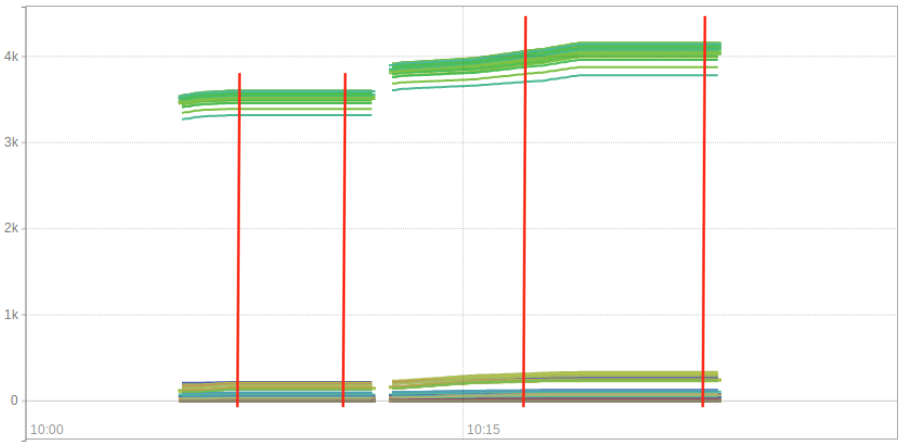

Figura 19: Resultados de la monitorización del uso de CPU con Prometheus.

La interfaz web de Prometheus te permite ver los resultados en una gráfica como se muestra en la Figura 19. En ella se visualizan dos ejecuciones de este programa (delimitadas por las líneas rojas

verticales). La primera de 1 minuto y la segunda de 5. En el eje X se encuentra el tiempo y el eje Y el valor de la variable. Las líneas superiores representan el tiempo en modo idle de cada CPU de cada nodo. Se observa que durante la ejecución del programa los valores en idle no aumentan, mientras que las curvas del resto de los modos que se observan en la parte inferior aumentan cuando se está realizando el trabajo. Los valores en el modo idle son tan elevados debido a que el exporter para Prometheus recoge estos valores de las variables del sistema desde que se inicializa y por ello el cálculo del consumo se la CPU se debe realizar como la diferencia entre el valor final e inicial de la ejecución del programa del modo user.

La otra métrica que se requiere medir es el WorkflowLatency de la aplicación que procesa las curvas diarias (processCH), esto se refiere al tiempo que tarda en realizar el procesamiento de una curva. Como el código es legado no podemos introducir métricas en él, pero conocemos que el procesamiento siempre finaliza guardando la curva en la base de datos y la estructura de la tabla donde se guarda la base de datos es la siguiente:

```
CREATE TABLE IF NOT EXISTS curvas(
cups varchar,
dia date,
valor_h01 double,
valor_h02 double,
valor_h03 double,
...
flag_h01 int,
flag_h02 int,
flag_h03 int,
...
prl_h01 int,
prl_h02 int,
prl_h03 int,
...
created timestamp,
inserted timestamp,
before_select_consumption timestamp,
after_select_consumption timestamp,
after_select_profile timestamp,
PRIMARY KEY(cups, dia));
```

Se compone de un identificador del cliente, la fecha de la curva, los 25 valores obtenidos del contador con sus 25 flags y 25 valores de prelación que indican cuántas veces se ha recalculado esa hora para mejorar la precisión. Junto a eso se encuentran los valores de created, que indica cuando entró en el sistema dicha curva, y el valor inserted, que muestra cuando se insertó en la base de datos la curva. También se incluyen valores de tiempo de las distintas fases del procesado de las curvas. Para conocer la latencia de esta aplicación nos interesan los valores de cuándo entró al sistema y cuándo se inserto. Con el fin de obtener esos datos hay que preguntar a Cassandra por ellos, nosotros obtendremos el máximo, el mínimo y la media de tiempos. Para ello, se debe programar una función que halle la diferencia entre inserted y created y después calcule los valores máximo, mínimo y la media de este experimento.

```
create function resta(t1 timestamp, t2 timestamp)
returns null on null input
 returns bigint
```

```
 language java
 as $$
        java.util.Calendar c1 = new java.util.
           GregorianCalendar();
        java.util.Calendar c2 = new java.util.
           GregorianCalendar();
        c1.setTime(t1);
        c2.setTime(t2);
        return c1.getTimeInMillis()-c2.getTimeInMillis();
$$;
```

A continuación, realizamos la consulta para obtener los tres valores deseados de la siguiente forma:

```
select max(resta(inserted,created))as MAX_Latency, min(resta
   (inserted,created)) as MIN_Latency, avg(resta(inserted,
   created))as AVG_Latency from curvas;
```

Los valores que nos retorna son las latencias en ms del workflow processCH una vez procesado 700.000 curvas con una inyección de 15000 tuplas por segundo:

```
 max_latency | min_latency | avg_latency
-------------+-------------+-------------
        62774|        1015 |       42945

(1 rows)
```

Capítulo 5

Caso de uso II: P3SimMast

El segundo caso de uso se trata del simulador P3SimMast es una aplicación Java que evalúa mediante un proceso simulación el comportamiento temporal de un sistema informático descrito mediante un modelo Mast2 [14]. El proceso de simulación se describe gráficamente en la Figura 19. La entrada a la aplicación es un modelo Mast2 que describe el comportamiento del sistema que se analiza. El modelo se formula como un string XML conforme al schema W3C Mast2_mdl.xsd. La salida global del proceso de simulación es un informe sobre el comportamiento del sistema analizado, constituido por un modelo de resultados que se formula como un string XML conforme al schema W3C Mast2_res.xsd.

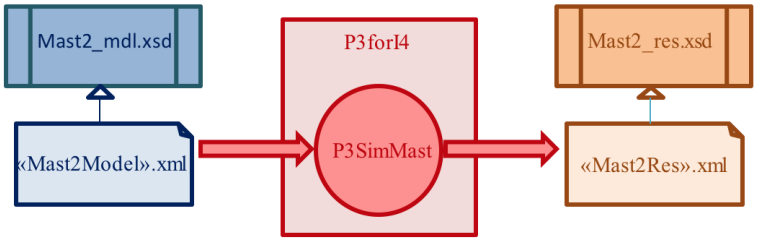

Figura 20: Entrada/salida del proceso de simulación.

El proceso de simulación requiere tres fases:

1. Enumeración de las configuraciones de inicialización que requiere el proceso de simulación del modelo como se muestra en la Figura 21.
2. Simulación del sistema bajo cada una de los estados de arranque.
3. Integración de los resultados obtenidos en cada simulación parcial en unos resultados que describan el comportamiento global del sistema.

Las fases 1 y 3 son globales, pero las simulaciones de la fase 2 son paralelizables y pueden ser distribuidas para su ejecución concurrente en la plataforma distribuida P3forI4 como se muestra en la Figura 22.

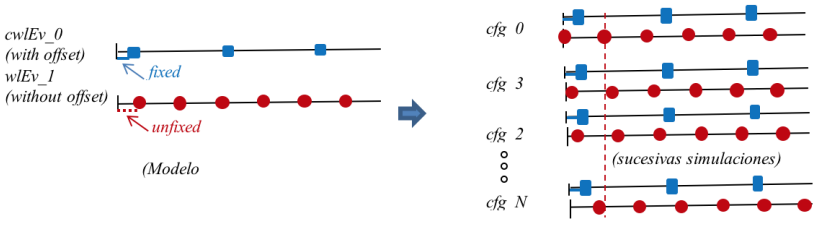

Figura 21: Origen de las configuraciones de inicialización.

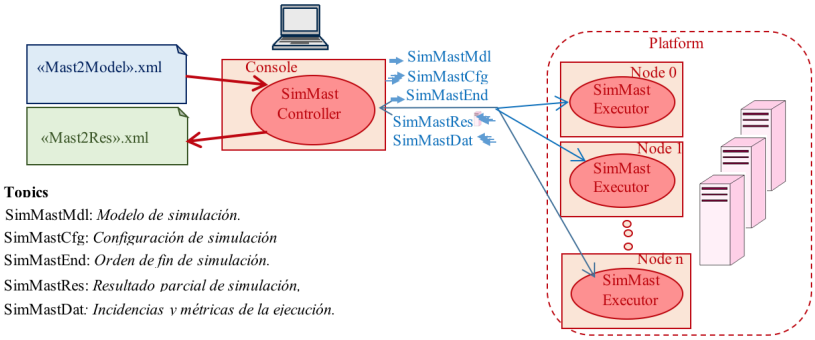

Figura 22: Paralelización del proceso de simulación

Para ello se hará uso de nuestra plataforma P3forI4 como se muestra en la Figura 23. Usaremos Zookeeper para la integridad de la información y Kafka para el intercambio de mensajes entre el programa que interpreta el modelo de entrada y los ejecutores programados en java, que son los encargados de realizar la simulación con la configuración que reciban.

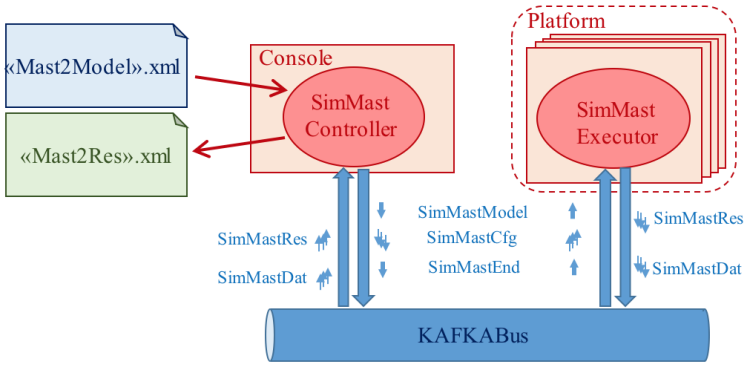

Figura 23: Recursos para la ejecución.

Se quiere realizar una prueba de benchmark para medir el impacto que tiene sobre el sistema donde se ejecuta la simulación, midiendo el porcentaje de utilización de CPU de cada uno de los nodos y la carga de la red que supone el intercambio de mensajes entre los componentes.

5.1 Modelo

A continuación se describe el modelo de datos de la prueba de benchmark para este caso de uso.

Figura 24: Definicion de la prueba del caso de uso II.

En la Figura 24, se muestra la descripción principal de la prueba indicando las estructuras de datos que describen la plataforma, la fuente de esos datos, las tareas de procesamiento y las métricas que deben de ser evaluadas.

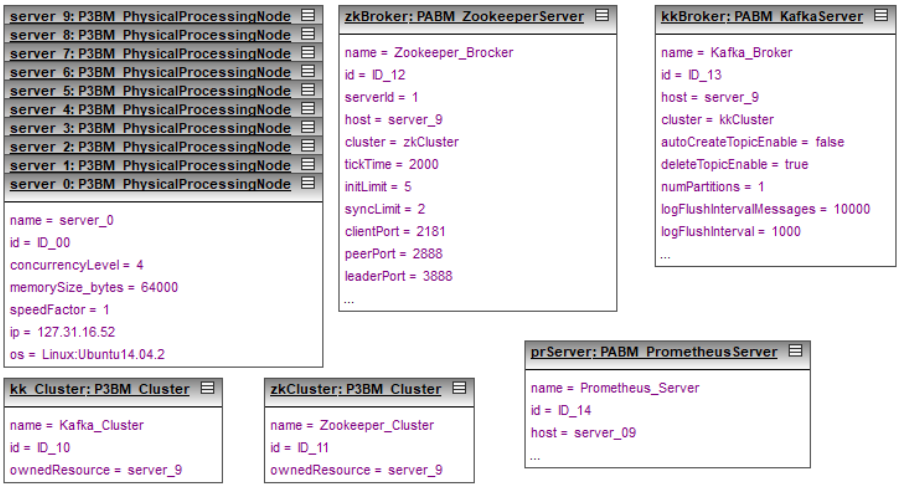

Figura 25: Descripción de los recursos de la prueba del caso de uso II.

En la Figura 25 se describe la plataforma. Al igual que en el caso de uso anterior de cada nodo de procesamiento (server_?) se proporciona sus identificadores (name, id, ip, os), el número de procesadores (concurrencyLevel), la memoria de que disponen (memorySize_bytes) y su capacidad de procesamiento (speedFactor), que en este caso, es igual a la del mainframe en el que ha sido evaluada la carga de procesamiento de las tareas. El zkBroker describe el gestor de distribución Zookeper, en este caso de uso solo se cuenta con uno ya que no se considera necesario la redundancia de la información. De la misma forma solo se dispone de un recurso para la mensajería entre las

aplicaciones (kkBroker) de la que se ofrece la información necesaria para configurarlo. Se definen los clústers tanto de Zookeeper como de Kafka para que puedan ser referenciados, aunque en este caso el clúster estaría solo compuesto por el server_9. También se define el recurso para la monitorización de las métricas (prServer), que también se alojara en el server_9.

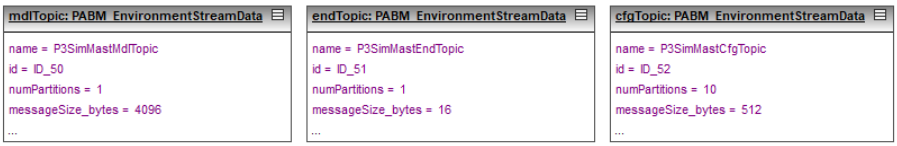

Figura 26: Descripción de los flujos de entrada del caso de uso II.

En la Figura 26 se muestran los flujos de datos presentes en la ejecución de la aplicación (mdlTopic, endTopic, cfgTopic), se indica el número de particiones en las que se distribuirá (numPartitions) y el tamaño de de los mensajes que enviará.

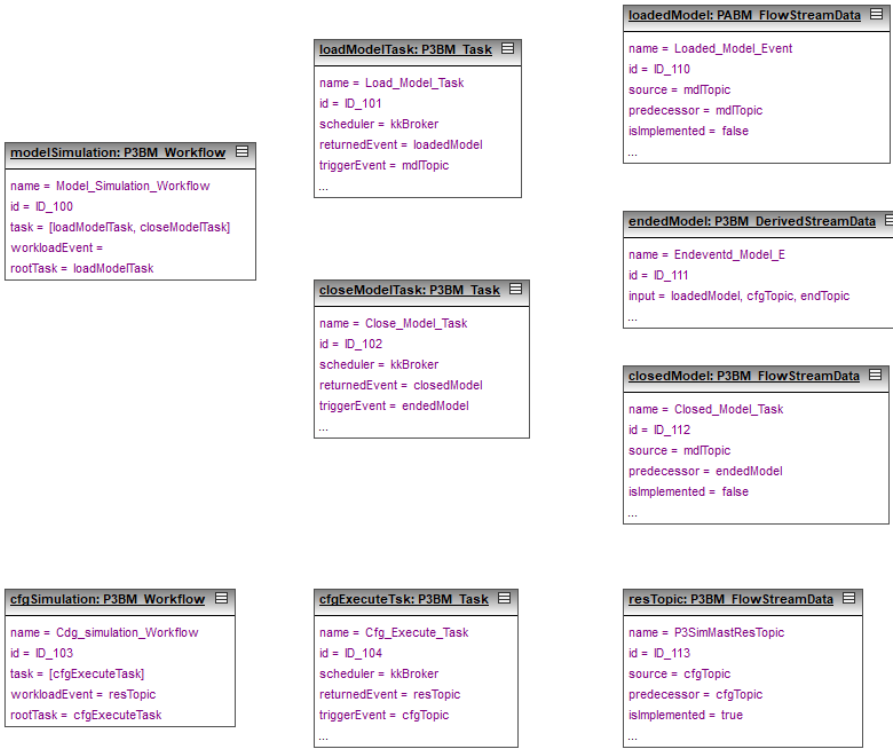

Figura 27: Descripción de las aplicación del caso de uso II.

En la Figura 27, se muestra los dos flujos de trabajo que existen, el que genera las configuraciones y las envía a cada worker (modelSimulation) que se compone de dos tareas, la que carga el modelo (loadModelTask) el que retorna, loadedModel, indicando que se ha cargado, y la tarea closeModelTask que indica cuando finaliza la ejecución. Por otro lado, el flujo de trabajo que realiza la simulación con la configuración que le llegue, se ejecuta mediante la tarea cgfExecute que devuelve

el flujo de datos, resTopic, con los resultados de la simulación.

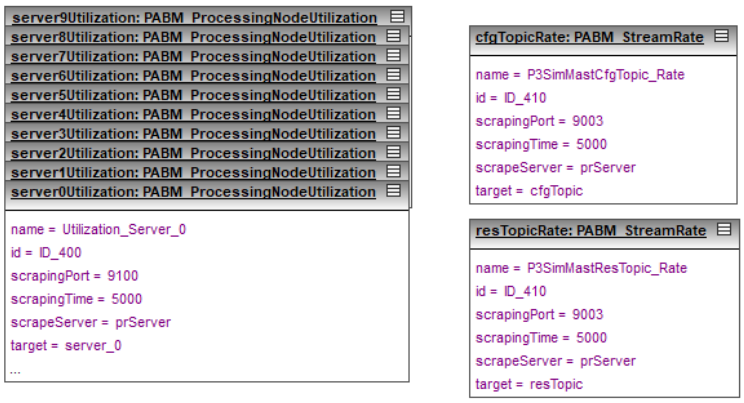

Figura 28: Descripción de las métricas a aplicar en el caso de uso II.

En la Figura 28 se describen las métricas que se aplican en la prueba. Estas son el porcentaje de utilización de la CPU de los diez nodos de procesamiento, y la media de bytes consumidos por el tópico que se le indica, en este caso se medirá para los tópicos cfgTopic y resTopic.

La representación de este modelo de datos en formato XML se puede encontrar en el Apéndice B.

5.2 Validación del modelo

Como en el caso de uso anterior en esta sección comprobaremos que en el modelo de datos descrito previamente se encuentra toda la información para poder desplegar la plataforma, iniciar el procesamiento y realizar las monitorización que se haya especificado.

El primer paso es desplegar todos los recursos necesarios, esto es, ZooKeeper y Kafka. Esto se realizará de manera muy similar al anterior caso de uso. Para iniciar el Zookeeper tenemos que escribir su fichero de configuración, zookeeper.properties, con la información que aparece en el modelo. En este caso de uso solo disponemos de un nodo de Zookeeper.

```
tickTime=2000
initLimit=5
clientPort=2181
peerPort=2888
leaderPort=3888
initLimit=5
syncLimit=2
```

Se inicia el servidor ejecutando el siguiente comando solo en el server_9 que es donde el modelo de datos lo indica.

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

A continuación, como en el caso de uso anterior iniciaremos Kafka, para ello realizaremos la modificación al fichero de configuración, server.properties, indicándole la información que aparece en el modelo y en este caso el único servidor de Zookeeper que existe en la plataforma.

```
broker.id=0
port=9092
auto.create.topics.enable=false
delete.topic.enable=true
num.partitions=1
log.flush.interval.messages=10000
log.flush.interval.ms=1000
zookeeper.connect= server_9:2181
```

Con el fichero de configuración de Kafka terminado lo iniciaremos solo en el server_9 con el siguiente comando:

```
bin/kafka-server-start.sh config/server.properties
```

Una vez desplegado el servidor de Kafka crearemos los flujos de datos (topics) presentes en el sistema, con los valores indicados en el modelo y como servidor de Zookeeper, el único que existe en el sistema el server_9.

```
bin/kafka-topics.sh --create --zookeeper server_9:2181 --
    replication-factor 1 --partitions 10 --topic
    P3SimMastCfgTopic

bin/kafka-topics.sh --create --zookeeper server_9:2181 --
    replication-factor 1 --partitions 1 --topic
    P3SimMastEndTopic

bin/kafka-topics.sh --create --zookeeper server_9:2181 --
    replication-factor 1 --partitions 1 --topic
    P3SimMastResTopic

bin/kafka-topics.sh --create --zookeeper server_9:2181 --
    replication-factor 1 --partitions 1 --topic
    P3SimMastMdlTopic
```

Por último, queda desplegar la propia aplicación, por un lado los ejecutores, que se encargarán del procesamiento de la simulación, y por otro el controlador, que leerá el modelo Mast2 y generará las configuraciones. Como en el caso de uso anterior nos hace falta los nombres de los programas para saber como ejecutarlos, pero en el modelo falta la especificación para ello. Aún así como los conocemos, se lanzaría en primer lugar los ejecutores con el siguiente comando;

```
java -jar P3SimMastExecutor.jar server_9:9092 server_9:2181 0
```

Este comando sería ejecutado en todos los nodos del sistema ya que son donde se realizará la simulación. Para finalizar hace falta lanzar el controlador que leerá el modelo XML y enviarà las configuraciones a través de Kafka de la siguiente forma:

```
java -jar P3SimMastController.jar Mast2Model.xml server_9:9092
    server_9:2181
```

Al controlador se le pasa como parámetros el archivo XML con la descripción del sistema que se quiere simular y los servidores de Kafka y ZooKeeper para que pueda conectarse a enviar los datos. Con esto hemos comprobado que los valores necesarios para poner en marcha la prueba de benchmark se encuentran en el modelo de datos que hemos definido a excepción del nombre de los aplicativos.

5.3 Métricas

Al igual que en el caso de uso anterior comprobaremos en esta sección la validez de la información sobre las métricas para realizar la configuración del servidor Prometheus de monitorización y realizar las medidas indicadas.

La primera métrica que se describe es ProcessingNodeUtilization, la configuración tanto de Prometheus como del Node exporter, que revela las métricas sobre el sistema operativo, se realizaría de la misma forma que en el caso de uso anterior (Sección 4.3), porque se quiere saber la información de los 10 nodos, la única diferencia es que en vez de desplegar el servidor de Prometheus sobre el server_0 en este caso es sobre el server_9, por lo que el fichero de configuración se tiene que situar en el nodo correcto, y el siguiente comando se ejecutará en el server_9:

```
./prometheus --config.file=prometheus.yml
```

La otra métrica que tenemos que medir es el StreamRate que representa la media de bytes consumidos por el tópico que se le indique. Para ello haremos uso del JMX exporter que es el que nos permite obtener las métricas de aplicaciones Java. Este exporter es un java agent, el cual es una aplicación que permite modificar el bytecode de cada clase, de esta manera consigue las métricas que compartirá con Prometheus. Lo primero que hemos de hacer para poner en marcha este exporter es configurarlo, para ello requiere un fichero en YAML en el que se indica cómo se presentará la información que se envíe y cúales de todas las métricas disponibles estarán disponible para Prometheus. Para medir la métrica que deseamos se obtiene desde el programa consumidor de Kafka, por lo que el java agent será lanzado junto a los ejecutores. Estableceremos que queremos que las métricas salgan en minúsculas con lowercaseOutputName y pondremos en la withelist los datos que enviará el exporter sobre el tópico que se querrá controlar.

```
---
lowercaseOutputName: true
whitelistObjectNames: ['kafka.consumer:type=consumer-fetch-
    manager-metrics,topic=P3SimMastCfgTopic,*', 'kafka.consumer:
    type=consumer-fetch-manager-metrics,topic=
    P3SimMastResTopic,*']
```

Como se describe en el modelo los flujos de datos que nos interesan saber su media de bytes enviados son P3SimMastCfgTopic y P3SimMastResTopic. A continuación, para desplegar el exporter es necesario ejecutarlo junto al programa del ejecutor, por lo que el comando para iniciar el ejecutor quedaría:

```
java -javaagent:./jmx_prometheus_javaagent-0.3.1.jar=9003:
    config.yaml -jar P3SimMastExecutor.jar Mast2Model.xml
    server_9:9092 server_9:2181
```

Siendo config.yaml el fichero de configuración anteriormente creado, 9003 el puerto especificado en el modelo y la aplicación P3SimMastExecutor.jar que usamos para la ejecución de la plataforma sin las métricas. Como en el caso del Node exporter para leer los recursos de la CPU, hay que indicarle a Prometheus dónde está escribiendo los datos el exporter por lo que a su fichero de configuración se le añade la definición de las aplicaciones nuevas.

```
global:
                scrape_interval:     15s
                evaluation_interval: 15s
```

```
scrape_configs:
- job_name: Utilization_Server_0
        scrape_interval: 5s
        static_configs:
        - targets: ['server_0:9100']

        //Los 9 nodos restantes para la
            medida de la CPU

- job_name: P3SimMastTopic_RateNode0
        scrape_interval: 5s
        static_configs:
        - targets: ['server_0:9003']

- job_name: P3SimMastTopic_RateNode1
        scrape_interval: 5s
        static_configs:
        - targets: ['server_1:9003']

//Los 9 nodos restantes para la medida del
    StremRate
```

Una vez que el servidor de Prometheus se pone en marcha, de la misma forma que en el caso anterior, los resultados que se obtienen por cada nodo y tópico indicado en las configuraciones. Se presenta un ejemplo de la ejecución para el tópico P3SimMastCfgTopic:

```
# HELP The average number of records consumed per second for
     topic P3SimMastCfgTopic
kafka_consumer_consumer_fetch_manager_metrics_records_consumed_rate
    {client_id="consumer-2",topic="P3SimMastCfgTopic",}
    15044.0
# HELP The maximum number of bytes fetched per request for
    topic P3SimMastCfgTopic
kafka_consumer_consumer_fetch_manager_metrics_fetch_size_max
    {client_id="consumer-2",topic="P3SimMastCfgTopic",}
    7274145.0
# HELP The average number of bytes fetched per request for
    topic P3SimMastCfgTopic
kafka_consumer_consumer_fetch_manager_metrics_fetch_size_avg
    {client_id="consumer-2",topic="P3SimMastCfgTopic",}
    403029.0
# HELP The average number of bytes consumed per second for
    topic P3SimMastCfgTopic
kafka_consumer_consumer_fetch_manager_metrics_bytes_consumed_rate
    {client_id="consumer-2",topic="P3SimMastCfgTopic",}
    2424214.0
# HELP The average number of records in each request for
    topic P3SimMastCfgTopic
kafka_consumer_consumer_fetch_manager_metrics_records_per_request_avg
```

```
{client_id="consumer-2",topic="P3SimMastCfgTopic",}
14383.0
```

Con esta información obtenemos el dato bytes_consumed_rate que representa la media de bytes consumidos del tópico indicado, por la aplicación, que es el dato que nosotros deseamos.

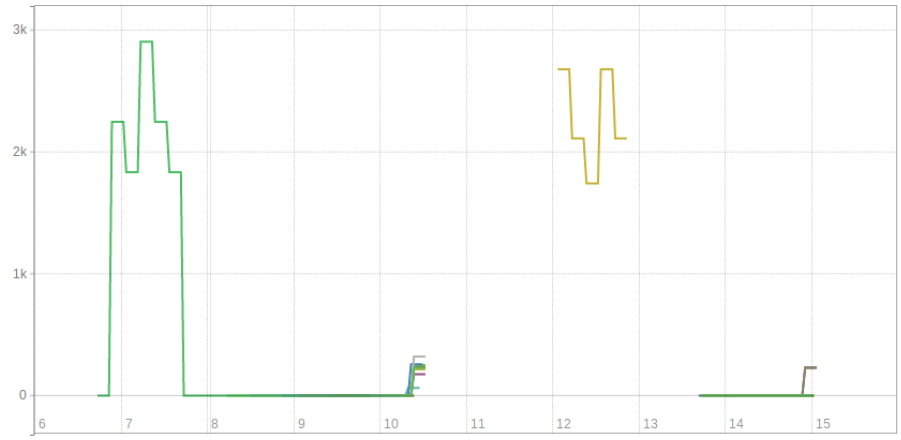

Figura 29: Resultados de la monitorización del uso de cpu con Prometheus.

La interfaz web de Prometheus te permita observar los datos en una gráfica como la de la Figura 29 en la que el eje X representa el tiempo, en este caso son minutos, y en el eje Y el valor de la variable en ese instante. En la gráfica aparecen varias ejecuciones de la aplicación entre las que han ocurrido paradas para lanzar el sistema otra vez.

Capítulo 6

Conclusiones y líneas futuras

La finalidad del proyecto se puede resumir en tres tareas principales: 1) definir especificaciones de benchmark para casos de uso de la Industria 4.0 usando el modelo desarrollado en el seno del grupo de investigación de ISTR; 2) mostrar su completitud para que una herramienta MDE que se construya en el futuro pueda realizar la ejecución de las pruebas en la infraestructura disponible y 3) instrumentar la arquitectura para recoger las métricas de rendimiento especificadas.

Como hemos podido comprobar, a través de los dos casos de uso, es posible desplegar la plataforma y obtener las métricas que en ellas se solicitaban por lo que se considera que los objetivos del proyecto se han cumplido satisfactoriamente. El metamodelo para la descripción de las pruebas de benchmark a fin de facilitar su configuración, despliegue y ejecución en plataformas de tercera generación va a ser presentado en las Jornadas de la Sociedad de Ingeniería del Software y Tecnologías de Desarrollo de software en el siguiente trabajo:

"Descripción de pruebas de benchmark para plataformas de tercera generación"Martin de la Rubia L.,Algorri M., Zorrilla M., Drake J.M.. SISTEDES'2018, Sevilla, Septiembre, 2018.

El siguiente paso sería intentar encontrar más aplicaciones propias de la Industria 4.0 que se beneficiarían al ser ejecutadas en una plataforma centrada en el dato, para comprobar si el modelo se adapta o debe ser extendido de algún modo.

Así mismo, es necesario desarrollar la herramienta que haga posible la realización de todas las tareas de configuración y de despliegue, de forma automática al menos para nuestra plataforma P3forI4, a partir de una descripción en XML o a través de unos formularios para introducir los datos del modelo. Posteriormente, lo ideal sería poder automatizar este proceso para cualquier tecnología disponible, añadiendo su especificación al modelo y su forma de configuración a la herramienta. Por otro lado, para la realización de métricas más variadas, detalladas y precisas será necesario generar nuevos programas o soluciones, principalmente para software legado ya que no se puede actuar sobre el código fuente.

Durante la realización del proyecto, he aprendido mucho sobre estas nuevas tecnologías y cómo afrontar la evolución que está sufriendo el mundo de la informática. También me ha permitido conocer la importancia que está adquiriendo los datos en las organizaciones, llevándolas a generar nuevas arquitecturas sobre las que desarrollar sus aplicaciones. Por último me ha servido para descubrir cómo es el trabajo de un grupo de investigación en la universidad, gracias a la beca de colaboración y así he apreciado lo costoso que es desarrollar una gran idea desde los inicios, hasta ver que es algo real y válido.

Bibliografía

[1] *Industry 4.0: the fourth industrial revolution – guide to Industrie 4.0.* URL: https://www.i-scoop.eu/industry-4-0/.

[2] S. Chintapalli y col. "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming". En: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Mayo de 2016, págs. 1789-1792. DOI: 10.1109/IPDPSW.2016.138.

[3] Guoqiang Jerry Chen y col. "Realtime Data Processing at Facebook". En: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, págs. 1087-1098. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2904441. URL: http://doi.acm.org/10.1145/2882903.2904441.

[4] *The Apache Software Foundation.* 2017. URL: http://www.apache.org/.

[5] Junqueira F. y Reed B. *ZooKeeper: Distributed process Coordination.* Ed. por O'Reilly Media. 2014.

[6] *The Apache Avro project: a data serialization system.* URL: http://avro.apache.org/.

[7] *Apache Kafka project: A distributed streaming platform.* 2017. URL: http://kafka.apache.org/.

[8] *Apache Spark: A fast and general engine for large-scale data processing.* 2017. URL: http://spark.apache.org/.

[9] *Apache Cassandra.* URL: http://cassandra.apache.org/.

[10] *MEMSQL: Tutorials overview.* 2018. URL: https://docs.memsql.com/tutorials/v5.8/tutorials-overview.

[11] *Prometheus Overview.* 2018. URL: https://prometheus.io/docs/introduction/overview/.

[12] Martin de la Rubia L. Algorri M. Zorrilla M. Drake J.M., ed. *Descripción de pruebas de benchmark para plataformas de tercera generación.* Jornadas de la Sociedad de Ingeniería del Software y Tecnologías de Desarrollo de software. 2018.

[13] *ORACLE: "Overview of the Event Processing Language (EPL)".* 2008. URL: https://docs.oracle.com/cd/E13213_01/wlevs/docs30/epl_guide/overview.html.

[14] Michael Gonzalez Harbour y Julio Medina, eds. *MAST: a model and tools to predict response times in event-driven real-time systems.* Presentation at Tutor'2016. 1st Tutorial on Tools for Real-Time Systems, en CPS (Cyber-Physical Systems) Week 2016, Viena (Austria), April 2016. 2016.

Apéndice A

Modelo XML del caso de uso I: Empresa de distribución eléctrica

```
<?xml version="1.0" encoding="UTF-8"?>
<P3BM_PROOF
name="Prueba␣Laboraorio" date="2018-06-09T16:25:34"
 authors="Admin"
 doc="Lab␣Proof"
 platform="RT␣Lab"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="P3BM.xsd">
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
           server0" name="server_0" concurrecyLevel="4" speedFactor=
           "1.0" ip="127.31.16.52" hostID="server_0" os="Linux:
           Ubuntu␣14.04.2" memorySize_bytes="64000"/>
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
           server1"  name="server_1" concurrecyLevel="4" speedFactor
           ="1.0" ip="127.31.16.46" hostID="server_1" os="Linux:
           Ubuntu␣14.04.2" memorySize_bytes="64000"/>
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
           server2"  name="server_2" concurrecyLevel="4" speedFactor
           ="1.0" ip="127.31.16.51" hostID="server_2" os="Linux:
           Ubuntu␣14.04.2" memorySize_bytes="64000"/>
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
           server3"  name="server_3" concurrecyLevel="4" speedFactor
           ="1.0" ip="127.31.16.47" hostID="server_3" os="Linux:
           Ubuntu␣14.04.2" memorySize_bytes="64000"/>
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
           server4"  name="server_4" concurrecyLevel="4" speedFactor
           ="1.0" ip="127.31.16.55" hostID="server_4" os="Linux:
           Ubuntu␣14.04.2" memorySize_bytes="64000"/>
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
           server5"  name="server_5" concurrecyLevel="4" speedFactor
```

```
    ="1.0" ip="127.31.16.58" hostID="server_5" os="Linux:
    Ubuntu␣14.04.2" memorySize_bytes="64000"/>
<ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
    server6"  name="server_6" concurrecyLevel="4" speedFactor
    ="1.0" ip="127.31.16.23" hostID="server_6" os="Linux:
    Ubunt␣14.04.2" memorySize_bytes="64000"/>
<ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
    server7"  name="server_7" concurrecyLevel="4" speedFactor
    ="1.0" ip="127.31.16.54" hostID="server_7" os="Linux:
    Ubuntu␣14.04.2" memorySize_bytes="64000"/>
<ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
    server8" name="server_8" concurrecyLevel="4" speedFactor=
    "1.0" ip="127.31.16.49" hostID="server_8" os="Linux:
    Ubuntu␣14.04.2" memorySize_bytes="64000"/>
<ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
    server9" name="server_9" concurrecyLevel="4" speedFactor=
    "1.0" ip="127.31.16.53" hostID="server_9" os="Linux:
    Ubuntu␣14.04.2" memorySize_bytes="64000"/>
<ownedElement xsi:type="P3BM_Cluster" id="zkCluster" name="
    Zookeeper_Cluster"ownedResource="server_0,␣server_1,␣
    server_2,␣server_3,␣server_4,␣server_5,␣server_6,␣
    server_7,␣server_8"/>
<ownedElement xsi:type="P3BM_Cluster" id="globalCluster"
    name="Zookeeper_Cluster"ownedResource="server_0,␣server_1
    ,␣server_2,␣server_3,␣server_4,␣server_5,␣server_6,␣
    server_7,␣server_8,␣server_9"/>
<ownedElement xsi:type="PABM_SparkServer" name="sparkServer"
     id="ID_41" host="server_0" driverCores="4" master="
    globalCluster" windowSize="100"/>


<ownedElement xsi:type="PABM_ZookeeperServer" id="zkServer0"
     name="ZkServer0"serverId="0" clientPort="2181" cluster="
    zkCluster" host="server_0" leaderPort="3888" peerPort="
    2888" codeDir="" dataDir="" ticTime="2000" initLimit="5"
    syncLimit="2"/>
<ownedElement xsi:type="PABM_ZookeeperServer" id="zkServer1"
     name="ZkServer1" serverId="1" clientPort="2181" cluster=
    "zkCluster" host="server_1" leaderPort="3888" peerPort="
    2888" codeDir="" dataDir="" ticTime="2000" initLimit="5"
    syncLimit="2"/>
<ownedElement xsi:type="PABM_ZookeeperServer" id="zkServer2"
     name="ZkServer2" serverId="2" clientPort="2181" cluster=
    "zkCluster" host="server_2" leaderPort="3888" peerPort="
    2888" codeDir="" dataDir="" ticTime="2000" initLimit="5"
    syncLimit="2"/>
<ownedElement xsi:type="PABM_ZookeeperServer" id="zkServer3"
     name="ZkServer3" serverId="3" clientPort="2181" cluster=
```

```
    "zkCluster" host="server_3" leaderPort="3888" peerPort="
    2888" codeDir="" dataDir="" ticTime="2000" initLimit="5"
    syncLimit="2"/>
<ownedElement xsi:type="PABM_ZookeeperServer" id="zkServer4"
     name="ZkServer4" serverId="4" clientPort="2181" cluster=
    "zkCluster" host="server_4" leaderPort="3888" peerPort="
    2888" codeDir="" dataDir="" ticTime="2000" initLimit="5"
    syncLimit="2"/>
<ownedElement xsi:type="PABM_ZookeeperServer" id="zkServer5"
     name="ZkServer5" serverId="5" clientPort="2181" cluster=
    "zkCluster" host="server_5" leaderPort="3888" peerPort="
    2888" codeDir="" dataDir="" ticTime="2000" initLimit="5"
    syncLimit="2"/>
<ownedElement xsi:type="PABM_ZookeeperServer" id="zkServer6"
     name="ZkServer6" serverId="6" clientPort="2181" cluster=
    "zkCluster" host="server_6" leaderPort="3888" peerPort="
    2888" codeDir="" dataDir="" ticTime="2000" initLimit="5"
    syncLimit="2"/>
<ownedElement xsi:type="PABM_ZookeeperServer" id="zkServer7"
     name="ZkServer7" serverId="7" clientPort="2181" cluster=
    "zkCluster" host="server_7" leaderPort="3888" peerPort="
    2888" codeDir="" dataDir="" ticTime="2000" initLimit="5"
    syncLimit="2"/>
<ownedElement xsi:type="PABM_ZookeeperServer" id="zkServer8"
     name="ZkServer8" serverId="8" clientPort="2181" cluster=
    "zkCluster" host="server_8" leaderPort="3888" peerPort="
    2888" codeDir="" dataDir="" ticTime="2000" initLimit="5"
    syncLimit="2"/>
<ownedElement xsi:type="PABM_ZookeeperServer" id="zkServer9"
     name="ZkServer9" serverId="9" clientPort="2181" cluster=
    "zkCluster" host="server_9" leaderPort="3888" peerPort="
    2888" codeDir="" dataDir="" ticTime="2000" initLimit="5"
    syncLimit="2"/>


<ownedElement xsi:type="PABM_KafkaServer" id="kafkaServer0"
     name="kafkaServer0" cluster="globalCluster" host="
    server_0" autoCreateTopicEnable="true" deleteTopicEnable=
    "true" logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numPartition="1"/>
<ownedElement xsi:type="PABM_KafkaServer" id="kafkaServer1"
     name="kafkaServer1" cluster="globalCluster" host="
    server_1" autoCreateTopicEnable="true" deleteTopicEnable=
    "true" logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numPartition="1"/>
<ownedElement xsi:type="PABM_KafkaServer" id="kafkaServer2"
     name="kafkaServer2" cluster="globalCluster" host="
    server_2" autoCreateTopicEnable="true" deleteTopicEnable=
```

```
    "true" logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numPartition="1"/>
<ownedElement xsi:type="PABM_KafkaServer" id="kafkaServer3"
     name="kafkaServer3" cluster="globalCluster" host="
    server_3"
autoCreateTopicEnable="true" deleteTopicEnable="true"
    logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numPartition="1"/>
<ownedElement xsi:type="PABM_KafkaServer" id="kafkaServer4"
     name="kafkaServer4" cluster="globalCluster" host="
    server_4" autoCreateTopicEnable="true" deleteTopicEnable=
    "true" logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numPartition="1"/>
<ownedElement xsi:type="PABM_KafkaServer" id="kafkaServer5"
     name="kafkaServer5" cluster="globalCluster" host="
    server_5" autoCreateTopicEnable="true" deleteTopicEnable=
    "true" logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numPartition="1"/>
<ownedElement xsi:type="PABM_KafkaServer" id="kafkaServer6"
     name="kafkaServer6" cluster="globalCluster" host="
    server_6" autoCreateTopicEnable="true" deleteTopicEnable=
    "true" logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numPartition="1"/>
<ownedElement xsi:type="PABM_KafkaServer" id="kafkaServer7"
     name="kafkaServer7" cluster="globalCluster" host="
    server_7" autoCreateTopicEnable="true" deleteTopicEnable=
    "true" logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numPartition="1"/>
<ownedElement xsi:type="PABM_KafkaServer" id="kafkaServer8"
     name="kafkaServer8" cluster="globalCluster" host="
    server_8" autoCreateTopicEnable="true" deleteTopicEnable=
    "true" logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numPartition="1"/>
<ownedElement xsi:type="PABM_KafkaServer" id="kafkaServer9"
     name="kafkaServer9" cluster="globalCluster" host="
    server_9" autoCreateTopicEnable="true" deleteTopicEnable=
    "true" logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numPartition="1"/>

<ownedElement xsi:type="PAMB_CassandraDatabase" name="
    cassandraServer0" id="cassandraServer0" host="server_0"
    port="9042" database="electricity"/>
<ownedElement xsi:type="PAMB_CassandraDatabase" name="
    cassandraServer1" id="cassandraServer1" host="server_1"
    port="9042" database="electricity"/>
<ownedElement xsi:type="PAMB_CassandraDatabase" name="
    cassandraServer2" id="cassandraServer2" host="server_2"
    port="9042" database="electricity"/>
```

```
<ownedElement xsi:type="PAMB_CassandraDatabase" name="
   cassandraServer3" id="cassandraServer3" host="server_3"
   port="9042" database="electricity"/>
<ownedElement xsi:type="PAMB_CassandraDatabase" name="
   cassandraServer4" id="cassandraServer4" host="server_4"
   port="9042" database="electricity"/>
<ownedElement xsi:type="PAMB_CassandraDatabase" name="
   cassandraServer5" id="cassandraServer5" host="server_5"
   port="9042" database="electricity"/>
<ownedElement xsi:type="PAMB_CassandraDatabase" name="
   cassandraServer6" id="cassandraServer6" host="server_6"
   port="9042" database="electricity"/>
<ownedElement xsi:type="PAMB_CassandraDatabase" name="
   cassandraServer7" id="cassandraServer7" host="server_7"
   port="9042" database="electricity"/>
<ownedElement xsi:type="PAMB_CassandraDatabase" name="
   cassandraServer8" id="cassandraServer8" host="server_8"
   port="9042" database="electricity"/>
<ownedElement xsi:type="PAMB_CassandraDatabase" name="
   cassandraServer9" id="cassandraServer9" host="server_9"
   port="9042" database="electricity"/>

<ownedElement xsi:type="PABM_ScrapeResource" id="prServer"
   name="Prometheus_Server" host="server_0"/>

<!-- ***** DATA STREAM WORKLOAD
   ************************************-->
<ownedElement xsi:type="PABM_EnvironmentStreamData" name="
   chStreamData" id="ID_50"
        messageSize_bytes="4096"
        numReplication="3"
        numPartition="80"
        logRetention_ms="18000000">
        <fieldDescr name="isValida"  primitiveType="boolean"
           />
        <generationPattern xsi:type="P3BM_LoggedEGP"
                meanInterarrival_ms="0"
                jitter_ms="1800000"
                offset_ms="10800000"
                logFilePath="chHistoricalData,csv">
                <P3BM_GenerationFrame duration_ms="86400000"
                   />
        </generationPattern>
</ownedElement>

<ownedElement xsi:type="PABM_EnvironmentStreamData" name="
   rmStreamData" id="ID_51" numPartition="80"
   numReplication="3">
```

```
        <generationPattern xsi:type="P3BM_LoggedEGP"
            meanInterarrival_ms="0" jitter_ms="600000"
            offset_ms="3600000" logFilePath="rmHistoricalData
            ,csv">
                <P3BM_GenerationFrame duration_ms="86400000"
                    />
        </generationPattern>
</ownedElement>

<!-- ***** APPLICATIONS ************************************
    -->
<ownedElement xsi:type="P3BM_Workflow" name="processCH" id="
    ID_101" rootTask="ID_102">
<task name="processCHTask" id="ID_102" scheduler="ID_100"/>
</ownedElement>

<!-- ***** METRICS
    ****************************************-->
<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_0Utilization" name="Utilization_server_0" target="
    server_0" port="9000" scrapeServer="prServer"
    scrapingTime="5000"/>
<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_1Utilization" name="Utilization_server_1" target="
    server_1" port="9000" scrapeServer="prServer"
    scrapingTime="5000"/>
<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_2Utilization" name="Utilization_server_2" target="
    server_2" port="9000" scrapeServer="prServer"
    scrapingTime="5000"/>
<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_3Utilization" name="Utilization_server_3" target="
    server_3" port="9000" scrapeServer="prServer"
    scrapingTime="5000"/>
<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_4Utilization" name="Utilization_server_4" target="
    server_4" port="9000" scrapeServer="prServer"
    scrapingTime="5000"/>
<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_5Utilization" name="Utilization_server_5" target="
    server_5" port="9000" scrapeServer="prServer"
    scrapingTime="5000"/>
<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_6Utilization" name="Utilization_server_6" target="
    server_6" port="9000" scrapeServer="prServer"
    scrapingTime="5000"/>
<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_7Utilization" name="Utilization_server_7" target="
```

```
            server_7" port="9000" scrapeServer="prServer"
            scrapingTime="5000"/>
        <ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
            server_8Utilization" name="Utilization_server_8" target="
            server_8" port="9000" scrapeServer="prServer"
            scrapingTime="5000"/>
        <ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
            server_9Utilization" name="Utilization_server_9" target="
            server_9" port="9000" scrapeServer="prServer"
            scrapingTime="5000"/>
        <ownedElement xsi:type="PAMB_WorkflowLatency" id="ID_810"
            name="chProcessLatency" target="processCH" port="9003"
            scrapeServer="prServer" scrapingTime="8100"/>
</P3BM_PROOF>
```

Apéndice B

Modelo XML del caso de uso II: P3SimMast

```
<?xml version="1.0" encoding="UTF-8"?>
<P3BM_PROOF name="P3SimMast_10servers" date="2018-05-16T06:30:00"
    authors="Jose␣M.␣Drake" doc="Ejecucion␣de␣SimMast␣para␣comprobar␣
    medida␣de␣metricas␣con␣Prometheus" platform="Lab_TR_10P" xmlns:
    xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
    noNamespaceSchemaLocation="P3BM.xsd">
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
            server5" name="server_0" concurrecyLevel="4" speedFactor=
            "1.0" ip="127.31.16.52" hostID="server_0" os="Linux:
            Ubuntu␣14.04.2" memorySize_bytes="64000"/>
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
            server6"  name="server_1" concurrecyLevel="4" speedFactor
            ="1.0" ip="127.31.16.46" hostID="server_1" os="Linux:
            Ubuntu␣14.04.2" memorySize_bytes="64000"/>
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
            server7"  name="server_2" concurrecyLevel="4" speedFactor
            ="1.0" ip="127.31.16.51" hostID="server_2" os="Linux:
            Ubuntu␣14.04.2" memorySize_bytes="64000"/>
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
            server8"  name="server_3" concurrecyLevel="4" speedFactor
            ="1.0" ip="127.31.16.47" hostID="server_3" os="Linux:
            Ubuntu␣14.04.2" memorySize_bytes="64000"/>
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
            server9"  name="server_4" concurrecyLevel="4" speedFactor
            ="1.0" ip="127.31.16.55" hostID="server_4" os="Linux:
            Ubuntu␣14.04.2" memorySize_bytes="64000"/>
        <ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
            server10"  name="server_5" concurrecyLevel="4"
            speedFactor="1.0" ip="127.31.16.58" hostID="server_5" os=
            "Linux:Ubuntu␣14.04.2" memorySize_bytes="64000"/>
```

```
<ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
    server11"  name="server_6" concurrecyLevel="4"
    speedFactor="1.0" ip="127.31.16.23" hostID="server_6" os=
    "Linux:Ubuntu␣14.04.2" memorySize_bytes="64000"/>
<ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
    server12"  name="server_7" concurrecyLevel="4"
    speedFactor="1.0" ip="127.31.16.54" hostID="server_7" os=
    "Linux:Ubuntu␣14.04.2" memorySize_bytes="64000"/>
<ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
    server13" name="server_8" concurrecyLevel="4" speedFactor
    ="1.0" ip="127.31.16.49" hostID="server_8" os="Linux:
    Ubuntu␣14.04.2" memorySize_bytes="64000"/>
<ownedElement xsi:type="P3BM_PhysicalProcessingNode" id="
    server14" name="server_9" concurrecyLevel="4" speedFactor
    ="1.0" ip="127.31.16.53" hostID="server_9" os="Linux:
    Ubuntu␣14.04.2" memorySize_bytes="64000"/>

<ownedElement xsi:type="P3BM_Cluster" id="kkCluster" name="
    Kafka_Cluster" ownedResource="server_9"/>
<ownedElement xsi:type="P3BM_Cluster" id="zkCluster" name="
    Zookeeper_Cluster" ownedResource="server_9"/>

<ownedElement xsi:type="PABM_ZookeeperServer" id="zkBroker"
    name="Zookeeper_Brocker"      serverId="1" clientPort="
    2181" cluster="zkCluster" host="server_9" leaderPort="
    3888" peerPort="2888" codeDir="" dataDir="" ticTime="2000
    " initLimit="5" syncLimit="2"/>
<ownedElement xsi:type="PABM_KafkaServer" id="kkBroker"
    name="Kafka_Broker" commId="5" cluster="kkCluster" host="
    server_9" advertisedListeners="PLAINTEXT://server_14:9093
    " autoCreateTopicEnable="false" deleteTopicEnable="true"
    logDirs="" logFlushInterval_ms="1000"
    logFlushIntervalMessage="10000" numIOThreads="2"
    numNetworkThreads="2" numPartition="1"
    numRecoveryThreadsPerDataDir="1" socketReceiveBufferBytes
    ="102400" socketRequestMaxBytes="104857600"
    socketSendBufferBytes="102400"
    zookeeperConnectionTimeOut_ms="6000"/>
<ownedElement xsi:type="PABM_ScrapeResource" id="prServer"
    name="Prometheus_Server" host="server_9"/>

<ownedElement xsi:type="PABM_EnvironmentStreamData" id="
    mdlTopic" name="P3SimMastMdlTopic" numPartition="1"
    messageSize_bytes="4096"/>
<ownedElement xsi:type="PABM_EnvironmentStreamData" id="
    endTopic" name="P3SimMastEndTopic" numPartition="1"
    messageSize_bytes="16"/>
<ownedElement xsi:type="PABM_EnvironmentStreamData" id="
```

```
    cfgTopic" name="P3SimMastCfgTopic"  numPartition="10"
    messageSize_bytes="512"/>

<ownedElement xsi:type="P3BM_Worflow" id="modelSimulation"
    name="Model_Simulation_Workflow"  rootTask="loadModelTask
    ">
          <workflowStreamData xsi:type="P3BM_FlowStreamData"
             id="loadedModel" name="Loaded_Model_Event"
             isImplemented="false"  source="mdlTopic"
             predecessor="mdlTopic" />
          <workflowStreamData xsi:type="P3BM_DerivedStreamData
             " id="endedModel" name="Ended_Model_Event" input=
             "loadedModel␣cfgTopic␣endTopic" pattern=""/>
          <workflowStreamData xsi:type="P3BM_FlowStreamData"
             id="closedModel" name="Closed_Model_Event"
             predecessor="endedModel" source="mdlTopic"
             isImplemented="false"/>
          <task id="loadModelTask" name="Load_Model_Task"
             returnedEvent="loadedModel" triggerEvent="
             mdlTopic" scheduler="kkBroker"/>
          <task id="closeModelTask" name="Close_Model_Task"
             triggerEvent="endedModel" returnedEvent="
             closedModel" scheduler="kkBroker"/>
</ownedElement>
<ownedElement xsi:type="P3BM_Worflow" id="cfgSimulation"
    name="Cfg_simulation_Workflow"  rootTask="cfgExecuteTask"
    >
          <workflowStreamData xsi:type="P3BM_FlowStreamData"
             id="resTopic" name="P3SimMastResTopic"
             isImplemented="true"  source="cfgTopic"
             predecessor="cfgTopic"/>
          <task id="cfgExecuteTask" name="Cfg_Execute_Task"
             triggerEvent="cfgTopic" returnedEvent="resTopic"
             scheduler="kkBroker"/>
</ownedElement>


<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_0Utilization" name="Utilization_server_0"
target="server_0" port="9000" scrapeServer="prServer"
    scrapingTime="5000"/>
<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_1Utilization" name="Utilization_server_1"
target="server_1" port="9000" scrapeServer="prServer"
    scrapingTime="5000"/>
<ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
    server_2Utilization" name="Utilization_server_2"
target="server_2" port="9000" scrapeServer="prServer"
```

```
        scrapingTime="5000"/>
      <ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
        server_3Utilization" name="Utilization_server_3"
      target="server_3" port="9000" scrapeServer="prServer"
        scrapingTime="5000"/>
      <ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
        server_4Utilization" name="Utilization_server_4"
      target="server_4" port="9000" scrapeServer="prServer"
        scrapingTime="5000"/>
      <ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
        server_5Utilization" name="Utilization_server_5"
      target="server_5" port="9000" scrapeServer="prServer"
        scrapingTime="5000"/>
      <ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
        server_6Utilization" name="Utilization_server_6"
      target="server_6" port="9000" scrapeServer="prServer"
        scrapingTime="5000"/>
      <ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
        server_7Utilization" name="Utilization_server_7"
      target="server_7" port="9000" scrapeServer="prServer"
        scrapingTime="5000"/>
      <ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
        server_8Utilization" name="Utilization_server_8"
      target="server_8" port="9000" scrapeServer="prServer"
        scrapingTime="5000"/>
      <ownedElement xsi:type="PABM_ProcessingNodeUtilization" id="
        server_9Utilization" name="Utilization_server_9"
      target="server_9" port="9000" scrapeServer="prServer"
        scrapingTime="5000"/>
      <ownedElement xsi:type="PABM_StreamRate" id="cfgTopicRate"
        name="P3SimMastCfgTopic_Rate"
      target="cfgTopic" port="9003" scrapeServer="prServer"
        scrapingTime="5000"/>
      <ownedElement xsi:type="PABM_StreamRate" id="resTopicRate"
        name="P3SimMastResTopic_Rate"
      target="resTopic" port="9003" scrapeServer="prServer"
        scrapingTime="5000"/>

</P3BM_PROOF>
```