Modelado y Simulación de Colas de Mensajes Distribuidas (*Message Oriented Middleware*)

Entregable #2

Carlos Martín Flores González

Carné: 2015183528

Profesor: César Garita

Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Maestría en Computación

MC-7205 Tema Selecto de Investigación

3 de Mayo del 2018

Índice general

Capítulo 1

Informe de avance

1.1. Avance de actividades

Las siguientes son las actividades propuestas en el cronograma de trabajo para este entregable:

1. **Revisión de Literatura, 100 % - Terminada** De acuerdo a la recomendación del profesor, se dedicó un apartado a recolectar y trabajos relacionados con el tema propuesto.

   a) Selección de literatura relevante: Terminada

   b) Preparación del documento: Terminado. Ver Sección 2.1

2. **Adaptación de aplicación de referencia, 100 % - Terminada**. Luego de evaluar literatura relevante sobre el tema propuesto y valorar y probar aplicaciones que fueron utilizadas en varios artículos, se decidió optar por la aplicación *CloudStore*[1]. Esta es una aplicación que se realizó con el fin de demostrar las herramientas del *CloudScale Method*[2]. Para más detalles, ver Sección 2.2

   a) Selección de aplicación de referencia: Terminada

   b) Adaptar aplicación para que utilice MOM: Terminada

   c) Ejecutar pruebas de carga/rendimiento: se realizaron pruebas iniciales sobre la aplicación de referencia con el fin de explorar su comportamiento. Si bien estas mediciones representan una referencia inicial, se considera que en el trabajo futuro de modelado se podrían generar otras mediciones o bien refinamientos de las ya existentes. Es por esto que se considera que las mediciones en su fase exploratoria se consideran cubiertas pero al mismo tiempo se continuará trabajando en las mismas en un futuro próximo.

[1] https://github.com/CloudScale-Project/CloudStore
[2] http://www.cloudscale-project.eu/

1.2. Cumplimiento de objetivos

1. **Objetivo 1:** Revisión del estado del arte de trabajos relacionados con enfoques de predicción y medición del rendimiento en sistemas de software basados en componentes y en *middleware* orientado a mensajes.

   → Este objetivo se ha completado. Para más detalles, ver Seccion 2.1.

2. **Objetivo 2:** Adaptar un sistema de software de referencia para el cual ya exista un modelo de rendimiento y simulaciones para que se integre y comunique con *middleware* orientado a mensajes.

   → Este objetivo se ha completado. Para más detalles, ver Sección 2.2.

3. **Objetivo 3:** Comparar la solución implementada bajo diferentes cargas de trabajo

   → Se hicieron mediciones iniciales en el rendimiento de la aplicación en su estado original y en la versión adaptada. Ver Sección 2.3

1.3. Entregables

Entregable 1. Revisión de literatura: 100 %.

Entregable 2. Aplicación adaptada para que soporte comunicación basada en mensajes: 100 %

Entregable 3. Medición de rendimiento en aplicación adaptada: 100 %

Capítulo 2

Entregables

2.1. Revisión de la Literatura

2.1.1. Introducción

Los métodos de predicción de rendimiento basados en modelos permiten a los arquitectos de software evaluar el rendimiento de los sistemas de software durante las primeras etapas de desarrollo. Estos modelos de predicción se centran en los aspectos relevantes de la arquitectura y de la lógica del negocio, dejando de lado detalles de la infraestructura subyacente. Sin embargo, estos detalles son esenciales para generar predicciones de rendimiento que sean precisas.

Para los ingenieros, es una práctica común simular el modelo de un artefacto antes de construirlo. Modelos de diseños de autos, circuitos electrónicos, puentes, entre otros, son simulados para entender el impacto de decisiones de diseño en varias atributos de calidad de interés como seguridad, consumo de energía o estabilidad. La habilidad de predecir las propiedades de un artefacto en base a su diseño sin necesidad de construirlo, es una de las características centrales de una disciplina de ingeniería. A partir de esta visión de lo que se considera una disciplina de ingeniería establecida se podría decir entonces que la ingeniería de software es apenas una disciplina de ingeniería [1]. Esto porque frecuentemente los ingenieros de software carecen del entendimiento del impacto de decisiones de diseño en atributos de calidad como rendimiento o confiabilidad. Como resultado, se intenta probar la calidad del software mediante costosos ciclos de prueba y error.

El no entender el impacto en las decisiones de diseño puede ser costoso y riesgoso. Probar software significa que ya se ha hecho un esfuerzo en su implementación. Por ejemplo, si las pruebas revelan problemas de rendimiento, es muy probable que la arquitectura necesita ser modificada, lo que puede conllevar a costos adicionales. Estos costos surgen debido a que en sistemas de software empresarial un bajo rendimiento es principalmente el efecto de una arquitectura inadecuada que efecto de código.

La ingeniería de rendimiento de software(SPE por sus siglas en inglés) es una disciplina que se centra en incorporar aspecto de rendimiento dentro del proceso de desarrollo de software, con el objetivo de entregar software de confiable de acuerdo con propiedades de rendimiento particulares. Los modelos de rendimiento predictivos son una de las herramientas empleadas en SPE. Construidos en las fases tempranas del proceso de desarrollo de software, los modelos ayudan a predecir el rendimiento eventual del software y de esta forma guiar el desarrollo, para eso los modelos de predicción de rendimiento deben capturar todos los componentes relevantes del sistema.

Para aplicaciones de software moderans, esto puede implicar modelar complejas capas tales como máquinas virtuales o *middleware* de mensajería. Componer todos estos modelos puede resultar una tarea costosa e ineficiente. En su lugar, un modelo abstracto de la aplicación se puede construir primero y luego ir agregando los modelos de los componentes del sistema.

En este trabajo se propone la construcción de un modelo de rendimiento para un sistema que utilice *middleware* orientado a mensajes con el fin de evaluar la influencia en el rendimiento de dicho sistema. Se propone evaluar esta influencia por medio de un ejemplo: tomar una aplicación de referencia con el fin de obtener sus métricas actuales de rendimiento, adaptarla para que utilice *middleware* orientado a mensajes y luego medir su rendimiento y generar un modelo a partir de esto.

2.1.2. Estrategia

Esta sección presenta la estrategia emprendida para cubrir el cuerpo de conocimiento relacionado con ingeniería de rendimiento de software La estrategia general se basó en un proceso iterativo de identificación y lectura de artículos, luego identificar y leer artículos relevantes a partir de referencias y citas bibliográficas.

2.1.3. Preguntas de Investigación

Las preguntas de investigación seleccionadas para conducir la revisión de la literatura fueron:

PI1 ¿Cuáles enfoques de predicción y medición del rendimiento en sistemas de software basados en componente se han propuesto?

PI2 ¿Cuáles enfoques de predicción y medición de rendimiento de software se han utilizado para *middleware* orientado a mensajes?

PI3 ¿Qué retos y oportunidades existen con estos enfoques en la actualidad?

PI4 ¿Qué herramientas hay disponibles para modelaje de rendimiento de software?

2.1.4. Estrategia de Búsqueda

Con el fin de identificar el primer conjunto de artículos relevantes, se hizo una revisión preliminar con Google Scholar[1] porque con este motor de búsqueda se puede abarcar una amplio número de artículos y actas académicas de diferentes fuentes. El criterio de búsqueda se basó en búsquedas de palabras derivadas del tema de investigación y las preguntas de investigación. Se incluyeron palabras como "component-based software", "software performance engineering", "software performance modeling", "middleware oriented messages", "modeling middleware oriented messages", "broker modeling", "palladio component model", "modeling software architecture"

La búsqueda de la literatura se realizó en Marzo del 2018 usando las siguientes bases de datos electrónicas:

- ACM *Digital Library*
- IEEE *Explore Digital Library*
- Safari Books Online[2]
- Google Scholar

2.1.5. Criterio de Selección

Se aplicó el siguiente criterio de inclusión de artículos para esta revisión:

- Estudios que dan a conocer enfoques de análisis de rendimiento de software basado en componentes
- Estudios sobre modelaje de rendimiento de software
- Estudios sobre modelaje de rendimiento en *middleware* orientado a mensajes
- Estudios, aplicaciones o casos de uso de *Palladio Component Model*
- Estudios que reporten sobre éxito, fracaso y retos de la ingeniería de rendimiento de sofware basado en componentes

Los criterios de exclusión de artículos fueron los siguiente:

- Estudios relacionados ingeniería de rendimiento de software pero que no cubren o cubren muy poco aspectos sobre modelaje y simulación de arquitecturas.
- Artículos publicados hace más de 15 años atrás (rango aceptable 2003-2018).

[1] http://scholar.google.com
[2] https://www.safaribooksonline.com

2.1.6. Resultado de la Revisión

La literatura consultada se podría llegar a dividir en dos grupos, estudios sobre:

- Ingeniería de rendimiento de software: enfoques, modelado de ingeniería de rendimiento, herramientas y retos de investigación
- *Middleware* orientado a mensajes: características y modelado del rendimiento de estos sistemas.

A continuación se presenta el desarrollo de los temas seleccionados.

2.1.7. Ingeniería de rendimiento de software (*Software Performance Engineering*)

Una definición comúnmente utilizada para definir ingeniería de rendimiento de software es la que que brinda Woodside [18]: "Ingeniería de rendimiento de software (*Software Performance Engineering* - SPE) representa toda la colección de actividades de ingeniería de software y análisis relacionados utilizadas a través del ciclo de desarrollo de software que están dirigidos a cumplir con los requisitos de rendimiento".

De acuerdo con este mismo autor, los enfoques para ingeniería de rendimiento puede ser divididos en dos categorías: basadas en mediciones y basadas en modelos. La primera es la más común y utiliza pruebas, diagnóstico y ajustes una vez que existe un sistema en ejecución que se puede medir, es por esto que solamente puede ser utilizada conforme se va acercando el final del ciclo de desarrollo de software. Al contrario del enfoque basado en mediciones, el enfoque basado en modelos se centra en las etapas iniciales del desarrollo y inición con el enfoque de ingeniería de rendimiento propuesto por Smith [19]. Como el nombre lo indica, en este enfoque los modelos son clave para hacer predicciones cuantitativas de qué tan bien un arquitectura puede cumplir sus expectativas de rendimiento.

Se han propuesto otras clasificaciones de enfoques para SPE pero, con respecto a la evaluación de sistemas basados en componentes, en [14] se deja clasificación a un lado debido a que se argumenta que la mayoría de enfoques de modelaje toman alguna medición como entrada y a la mayoría de los métodos de medición los acompaña algún modelo.

Ingeniería de rendimiento basada en mediciones

Los enfoques basados en mediciones prevalencen en la industria [19] y son típicamente utilizados para verificación(¿el sistema cumple con su requisito de rendimiento?) o para localizar y arreglar *hot-spots* (cuáles son las partes que tienen peor rendimiento en el sistema). La medición de rendimiento se remota al inicio de la era de la

computación, lo que ha generado una amplia gama de herramientas como generadores de carga y monitores para crear cargas de trabajo ficticias y llevar a cabo la medición de un sistema respectivamente.

Las pruebas de rendimiento aplican técnicas basadas en medición y usualmente esto es hecho luego de las pruebas funcionales o de carga. Las pruebas de carga verifican el funcionamiento de un sistema bajo cargas de trabajo pesadas, mientras que las pruebas de rendimiento son usadas para obtener datos cuantitativos de características de rendimiento, como tiempos de respuesta, *throughput* y utilización de hardware para una configuración de un sistema bajo una carga de trabajo definida.

2.1.8. Ingeniería de rendimiento a través de modelado

La importancia del modelado del rendimiento está motivada por el riesgo de problemas graves de rendimiento [20] y la creciente complejidad de sistemas modernos, lo que hace difícil abordar los problemas de rendimiento al nivel de código. Cambios considerables en el diseño o en las arquitecturas pueden ser requeridos para mitigar los problemas de rendimiento. Por esta razón, la comunidad de investigación de modelado de rendimiento intenta luchar contra el enfoque de "arreglar las cosas luego" durante el proceso de desarrollo. Con la aplicación de modelo del rendimiento de software se busca encontrar problemas de rendimiento y alternativas de diseño de manera temprana en el ciclo de desarrollo, evitando así el costo y la complejidad de un rediseño o cambios en los requerimientos.

Las herramientas de modelado de rendimiento ayudar a predecir la conducta del sistema antes que este sea construido o bien, evaluar el resultado de un cambio antes de su implementación. El modelado del rendimiento puede ser usado como una herramienta de alerta temprana durante todo el ciclo de desarrollo con mayor precisión y modelos cada vez más detallados a lo largo del proceso. Al inicio del desarrollo un modelo no puede ser validado contra un sistema real, por esto el modelo representa el conocimiento incierto del diseñador. Como consecuencia de esto el modelo hace suposiciones que no necesariamente se van a dar en el sistema real, pero que van a ser útiles para obtener una abstracción del comportamiento del sistema. En estas fases iniciales, la validación se obtiene mediante el uso del modelo, y existe el riego de conclusiones erróneas debido a su precisión limitada. Luego, el modelo puede ser validado contra mediciones en el sistema real (o parte de este) o prototypos y esto hace que la precisión del modelo se incremente.

En [21] se sugiere que los métodos actuales tienen que superar un número de retos antes que puedan ser aplicados en sistemas existentes que enfrentan cambios en su arquitectura o requerimientos. Primero, debe quedar claro cómo se obtienen los valores para los parámetros del modelo y cómo se pueden validar los supuestos. Estimaciones

basadas en la experiencia para estos parámetros no son suficientes y mediciones en el sistema existente son necesarias para hacer predicciones precisas. Segundo, la caracterización de la carga del sistema en un entorno de producción es problemática debido a los recursos compartidos (bases de datos, hardware). Tercero, deben desarrollarse métodos para capturar parámetros del modelo dependientes de la carga. Por ejemplo un incremento en el tamaño de la base de datos probablemente incrementará las necesidades de procesador, memoria y disco en el servidor.

Técnicas comúnes de modelado incluyen redes de colas, extensiones de estas como redes de colas en capas y varios tipos de redes de Petri y procesos de álgebra estocástica.

2.1.9. Modelado de Rendimiento

En SPE, la creación y evaluación de modelos de rendimiento es un concepto clave para evaluar cuantitativamente el rendimiento del diseño de un sistema y predecir el rendimiento de otras alternativas de diseño. Un modelo de rendimiento captura el comportamiento relevante al rendimiento de un sistema para identificar el efecto de cambios en la configuración o en la carga de trabajo en el rendimiento. Permite predecir los efectos de tales cambios sin necesidad de implementarlo y ponerlo en producción, que podrían ser no solamente tareas costosas sino también un desperdicio en el caso que un el hardware con el que se cuenta pruebe ser insuficiente para soportar la intensidad de la carga de trabajo. [12]

La forma del modelo de rendimiento puede comprender desde funciones matemáticas a formalismos de modelado estructural y modelos de simulación. Estos modelos varían en sus características clave, por ejemplo, las suposiciones de modelado de los formalismos, el esfuerzo de modelado requerido y el nivel de abstracción.

En cuanto a técnicas de simulación, a pesar que estas permiten un estudio más detallado de los sistemas que modelos analíticos, la construcción de un modelo de simulación requiere de conocimiento detallado tanto de desarrollo de software como de estadística [19]. Los modelos de simulación también requieren usualmente de mayor tiempo de desarrollo que los modelos analíticos. En [18] se menciona que “la construcción de un modelo de simulación es caro, algunas veces comparable con el desarrollo de un sistema, y, los modelos de simulación detallados puede tardar casi tanto en ejecutarse como el sistema.

Enfoques de ingeniería de rendimiento para software basado en componentes propuestos

La encuesta llevada a cabo en [14] proporciona una clasificación de enfoques de medición y predicción de rendimiento para sistemas de software basados en componentes. Otra clasificación es la que se expone en [22] donde se presenta una revisión de

métodos de predicción de rendimiento basado en modelos para sistemas en general, pero no analiza los requerimientos propios para sistemas basados en componentes.

De acuerdo con [14] durante los últimos diez años, los investigadores han propuesto muchos enfoques para evaluar el rendimiento (tiempos de respuesta, *throughput*, utlización de recursos) de sistemas de software basados en componentes. Estos enfoques tratan con predicción de rendimiento y medición del rendimiento. Los primeros analizan el rendimiento esperado de un diseño de software basado en componentes para evitar problemas de rendimiento en la implementación del sistema, lo que podría llevar a costos substanciales para rediseñar la arquitectura. Los otros analizan el rendimiento observable de sistemas de software basados en componentes implementados para entender sus propiedades, determinar su capacidad máxima, identificar componentes críticos y para remover cuellos de botella.

Métodos de evaluación de rendimiento Los enfoques se agruparon en dos grandes grupos: enfoques principales que proporcionan procesos de evaluación de rendimiento completo y enfoques suplementarios que se centran en aspectos específicos como medición de componentes individuales o modelaje de las propiedades de rendimiento de los conectores de un componente.

Enfoques principales

- **Enfoques de predicción basados en UML**: los enfoques en este grupo se enfocan en la predicción de rendimiento en tiempo de diseño para sistemas de software basado en componentes modelados con el Lenguaje de Modelado Unificado (UML por sus siglas en inglés). UML 2.0 tiene la noción de componente de software como una clase extendida. UML permite modelar el comportamiento de un un componente con diagramas de sequencia, actividad y colaboración. El alojamiento de los componentes puede ser descrito como con diagramas de despliegue(*deployment*).
  - CB-SPE - *Component-Based Software Performance Engineering*
- **Enfoques de predicción basados en Meta-Modelos propietarios**: Los enfoques en este grupo apuntan a las predicciones de rendimiento de tiempo de diseño. En lugar de usar UML como lenguaje de modelado para desarrolladores y arquitectos, estos enfoques tienen meta-modelos propietarios.
  - CBML - *Component-Based Modeling Language*
  - PECT - *The Prediction Enabled Component Technology*
  - COMQUAD - *Components with Quantitative properties and Adaptivity*
  - KLAPPER

- ROBOCOP
- Palladio

- **Enfoques de predicción centrados en *middleware***: hacen énfasis en la influencia del *middleware* en el rendimiento de un sistema basado en componentes. Por lo tanto miden y modelan el rendimiento de plataformas *middleware* como JavaEE o .Net. Se basan en la suposición que la lógica de negocio de los componentes como tal tienen poco impacto en el rendimiento general del sistema y por eso no requieren un modelado detallado.

  - NICTA

- **Enfoques basados en especificaciones formales**: estos enfoques siguen teorías fundamentales de especificación de rendimiento y no toman en cuenta marcos de trabajo (*frameworks*) de medición y predicción.

  - RESOLVE
  - HAMLET

- **Enfoques de monitoreo para sistemas implementados**: asumen que un sistema basado en componentes ha sido implementado y puede ser probado. El objetivo es encontrar problemas de rendimiento en un sistema en ejecución, identificar cuellos de botella y adaptar el sistema para que pueda lograr los requerimientos de rendimiento.

  - COMPAS
  - TESTEJB
  - AQUA
  - PAD

Enfoques Suplementarios

- **Enfoques de monitoreo para componentes implementados**: El objetivo de los enfoques de medición para implementaciones de componentes de software individuales es derivar especificaciones de rendimiento parametrizadas a través de mediciones múltiples. El objetivo es obtener el perfil de uso, dependencias y la plataforma de implementación a partir de la especificación de rendimiento, de modo que pueda usarse en diferentes contextos.

  - RefCAM
  - COMAERA
  - ByCounter

- **Enfoques de predicción con énfasis en conectores de componentes**: Estos enfoques asumen un lenguaje de descripción de componentes existente y se centran en modelar y medir el impacto en el rendimiento de las conexiones entre componentes. Estas conexiones pueden implementarse con diferentes técnicas *middleware*.

  - Verdickt
  - Grassi
  - Becker
  - Happe

Recientemente varios enfoques de predicción basados en meta-modelos propietarios han sido propuestos para optimización de diseño de arquitecturas, modelado de calidad de servicio y escalabilidad. PerOpteryx [23] es un enfoque de optimización de diseño de arquitecturas que manipula modelos especificados en *Palladio Component Model* (PCM) [1] y utiliza el algoritmo evolucionario multi-objetivo NSGA-II. Para análisis de rendimiento utiliza redes de colas en capas. *Descartes Modeling Language* [24] es un lenguaje de modelado de arquitecturas para modelar calidad de servicio y aspectos relacionados con la gestión de recursos de los sistemas, las infraestructuras y los servicios de tecnología de información dinámicos modernos. *CloudScale*[3] [25] es un enfoque de diseño y evolución de aplicaciones y servicios escalables en la nube. En CloudScale se identifica y gradualmente se resuelven problemas de escalabilidad en aplicaciones existentes y también permite el modelado de alternativas de diseño y el análisis del efecto de la escalabilidad en el costo. Cabe mencionar que estos últimos enfoques han sido influenciado en gran medida por el trabajo llevado a cabo en PCM.

2.1.10. Modelado de Arquitecturas de Software con el *Palladio Component Model (PCM)*

El *Palladio Component Model* es un enfoque de modelaje para arquitecturas de software basados en componentes que permite predicción de rendimiento basada en modelos. PCM contribuye el proceso de desarrollo de ingeniería basado en componentes y proporciona conceptos de modelaje para describir componentes de software, arquitectura de software, despliegue (*deployment*) de componentes y perfiles de uso de sistemas de software basados en componentes en diferentes submodelos (Figura 2.1).

- **Especificaciones de componentes** son descripciones abstractas y paramétricas de los componentes de software. En las especificaciones de software se proporciona una descripción del comportamiento interno del componente así como las

[3] http://www.cloudscale-project.eu/

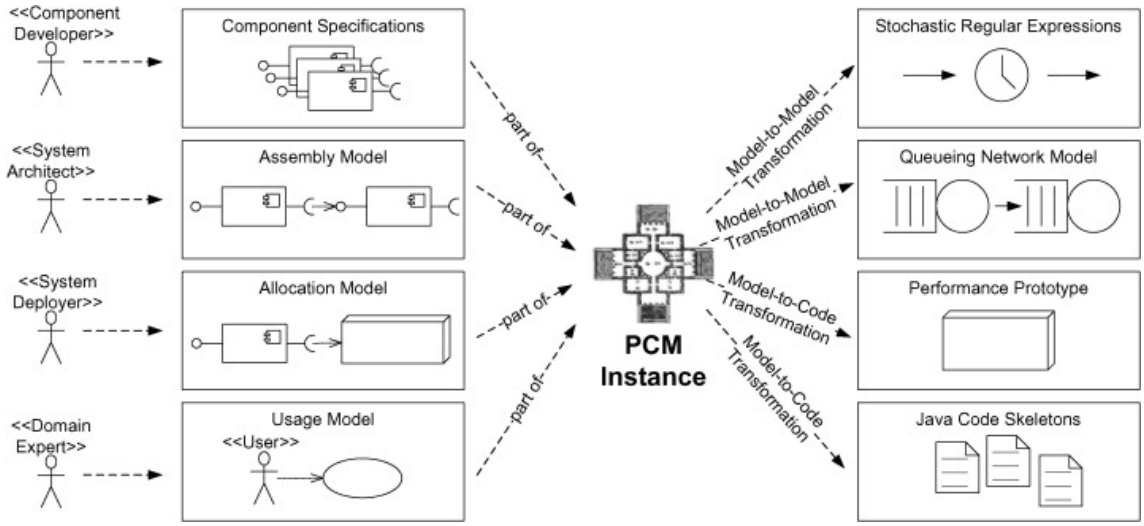

Figura 2.1: Instancia de un modelo PCM. Tomado de [13]

demandas de sus recursos en RDSEFFs (*Resource Demanding Service EFFect specifications*) utilizando una sintaxis similar a los diagramas de actividad de UML.

- **Un modelo de ensamblaje** (*assembly model*) especifica qué tipo de componentes se utilizan en una instancia de aplicación modelada y si las instancias del componente se replican. Además, define cómo las instancias del componente se conectan representando la arquitectura de software.

- El entorno de ejecución y los recursos, así como el despliegue (*deployment*) de instancias de componentes para dichos contenedores de recursos se definen en un **modelo de asignación** (*allocation model*).

- El **modelo del uso** especifica la interacción de los usuarios con el sistema utilizando una sintaxis similar al diagrama de actividades de UML proporcionando una descripción abstracta de la secuencia y la frencuencia en que los usuarios activan las operaciones disponibles en un sistema.

Un modelo PCM abstrae un sistema de software a nivel de arquitectura y se anota con consumos de recursos que fueron medidos previamente u otros que son estimados. El modelo puede entonces ser usado en transformaciones de modelo-a-modelo o modelo-a-texto a un modelo de análisis en particular (redes de colas o simulación de código) que puede ser analíticamente resuelto o simulado para obtener resultados sobre el rendimiento y predicciones del sistema modelado. Los resultados del rendimiento y las predicciones pueden ser utilizadas como retroalimentación para evaluar y mejorar el diseño inicial, permitiendo así una evaluación de calidad de los sistemas de software en base a un modelo [12].

2.1.11. Retos de investigación asociados a Ingeniería de rendimiento

Informes recientes como [6] y [7] señalan que los principales retos en investigación de ingeniería de rendimiento en la actualidad están asociados con generación de modelos a partir de código, integración de la evaluación de rendimiento en el ciclo de desarrollo y modelado de rendimiento de sistemas en la nube.

Con respecto a integración en ciclos de desarrollo de software y principalmente en aquellos basados en DevOps, una tendencia hacia una estrecha integración entre equipos desarrollo y operaciones, en [7] se destaca que la necesidad de tal integración está orientada por el requerimiento de adaptar aplicaciones empresariales a los cambios del ambiente del negocio continuamente. El rendimiento describe las propiedades del sistema con respecto a su ejecución y uso de recursos. Métricas comunes son tiempo de respuesta, *throughput* y la utilización de los recursos. Los objetivos de rendimiento para aplicaciones empresariales son típicamente definidos al establecer cotas superiores/inferiores para estas métricas y transacciones de negocio específicas.

Actividades de administración del rendimiento

Evaluación del rendimiento basado en modelos:

- La representación de la memoria principal y del recolector de basura aún no están explícitamente integrados ni considerados en los modelos de rendimiento
- La selección de técnicas apropiadas de solución requiere de mucha experiencia

Extracción de modelos de rendimiento y de cargas de trabajo:

- La precisión de los modelos puede llegar a expirar si no son actualizados cuando hay cambios. Mecanismos de detección son requeridos para aprender cuando los modelos son antiguos y hay que actualizarlos.
- La extracción de capacidades de rendimiento se basa en una combinación de software y los recursos de hardware en los que se implementa. Este enfoque combinado es compatible con la precisión de la predicción, pero está menos calificado con respecto a la portabilidad de los conocimientos a otras plataformas. Una dirección de investigación futura podría ser extraer modelos separados (por ejemplo, *middleware* y modelos de aplicación separados).

Modelos de rendimiento durante etapas diseño

- Los retos de utilizar modelos de rendimiento en etapas tempranas de desarrollo es que usualmente es difícil validar la precisión de los modelos hasta que un

sistema en ejecución exista. Las predicciones de rendimiento basadas en suposiciones, entrevistas y pruebas previas pueden ser también imprecisas y por tanto las decisiones que se hagan a partir de estas predicciones.

En relación con sistemas y arquitecturas en la nube como las basadas en microservicios otras tecnologías importantes para *deployment* como, virtualización basada en contenedores y soluciones de orquestación de contenedores han emergido. Estas tecnologías permiten explotar plataformas en la nube, proporcionando altos niveles de escalabilidad, disponibilidad y portabilidad para microservicios. A pesar del hecho de que es una necesidad inherente contar con escalabilidad y elasticidad, la ingeniería de rendimiento para microservicios hasta ahora ha tenido poca atención por parte de las comunidades tanto de microservicios como de comunidades de investigación de ingeniería de rendimiento [6]. Un gran cuerpo de conocimiento y buenas prácticas para ingeniería de rendimiento para desarrollo tradicional de software y arquitecturas está disponible. Sin embargo, sus aplicaciones en DevOps imponen tanto retos como oportunidades.

2.1.12. Retos de investigación en microservicios

Pruebas de rendimiento

- Reemplazar y compensar pruebas extensivas de integración y de sistema por un control detallado de los entornos de producción
- Alinear las pruebas de rendimiento y pruebas de regresión de rendimiento con prácticas de entrega continua, por ejemplo, acelerar las estas de pruebas correspondientes.
- Selección dinámica y semi automática de pruebas de rendimiento

Monitoreo

- Instrumentación para monitoreo distribuido arquitecturas microservicios políglotas
- Métricas adicionales para monitorear microservicios
- Técnicas de detección precisa de anomalías en arquitecturas de microservicios

Modelado del rendimiento: de momento no existen enfoques para modelado del rendimiento de microservicios

- Adoptar modelos de rendimiento para casos de uso como planeamiento de capacidad, confiabilidad y resiliencia

- Buscar abstracciones de modelado apropiadas
- Extracción automática de modelos de rendimiento
- Aprender del comportamiento de la infraestructura e integrarlo en los modelos de rendimiento

2.1.13. Herramientas para modelaje de rendimiento de software

En [19] se llevó a cabo un extensa encuesta sobre el estado del arte de herramientas de modelado de rendimiento de software. Como resultado, se lograron identificar varias herramientas para construir modelos de rendimiento pero, a pesar de esto, muchas de ellas fueron consideradas como no-maduras de acuerdo a los criterios de selección aplicados.

Primero, la herramienta debería de demostrar que se puede aplicar para el modelado de arquitecturas de software, este criterio excluyó a muchas herramientas de bajo nivel. Segundo, debe de estar liberada y disponible y esto excluyó a muchas herramientas descritas en artículos de investigación. Por último, debe ser madura y medianamente estable y este fue requerimiento que no cumplieron mucha de las herramientas publicadas en artículos de investigación.

Al final, se seleccionarion seis herramientas que se consideraron suficientemente maduras y promisorias las cuales se podrían ajustar para abordar problemas de modelado de arquitecturas.

1. Java Modeling Tools (JMT): es un conjunto de programas en Java que apoyan el modelado y análisis de modelos de redes de colas. Dentro de las ventajas que tiene es que puede obtener soluciones computacionalmente baratas de modelos de redes de colas. También cuenta con una extensa interfaz gráfica de usuario que ayuda a los usuarios a través del proceso de modelado y análisis, esto baja su curva de aprendizaje. Una de sus limitaciones es que se ha señalado que es desafiante modelar sistemas de software industrial usando un formalismos simples de modelos de redes de colas. La herramienta se encuentra disponible en `http://jmt.sourceforge.net`.

2. Layered Queueing Network Solver (LQNS): es una herramienta de redes de colas(LQN por sus siglas en inglés) en capas y tiene el conjunto de características más completo de varias herramientas de LQN en capas comunes. Las LQNs ofrecen una forma fácil de expresar el uso de los recursos y encontrar cuellos de botella en servidores. Los algoritmos de análisis para LQN pueden escalar a sistemas grandes y puede ser usados para modelar grandes cantidades de solicitudes concurrentes. Dentro de sus limitaciones se menciona el hecho que LQN son menos adecuadas para modelar res-

tricciones de tamaño de memoria que los modelos de redes de colas. De igual forma, en su manual de usuario se listan una serie de defectos conocidos y pocos detalles sobre estos son proporcionados. La herramienta se encuentra en http://www.sce.carleton.ca/rads/lqns/.

3. Palladio-Bench: implementa el *Palladio Component Model*. El software se encuentra disponible como un *plug-in* de Eclipse IDE y puede ser obtenido en el sitio Web http://www.palladio-simulator.com/. Esta herramienta puede ahorrar tiempo cuando se modelan componentes que son reutilizados en otros modelos. Es posible especificar las demandas de recursos dependientes de la entrada, probabilidades condicionales y contadores de iteraciones. Por ejemplo, el tiempo de ejecución de un componente puede hacerse dependiente del tamaño del archivo que va a procesar. Las limitaciones que se han reportado de esta herramienta incluyen poco soporte para concurrencia y del modelado del ambiente de ejecución.

4. Möbius: es una herramienta que implementa múltiples lenguajes de modelado y inclusive permite combinarlos para explotar su expresividad o para combinar múltiples aspectos de un sistema dentro de un modelo. Dentro de los lenguajes de modelaje ofrecidos, el SAN, *Buckets and Balls* y PEPA puede ser utilizados para modelado de rendimiento. Varios de los formalismos soportados por Möbius se basan en cadenas de Markov y por lo tanto sufren del problema de explosión espacial del estado. Puede haber una reducción en el número de estados de uno a dos órdenes de magnitud utilizando técnicas de agrupamiento, pero lo que esto significa en la práctica aún no está claro. La página Web de la herramienta es https://www.mobius.illinois.edu/

5. SPE-ED: implementa el enfoque SPE documentado por Smith y Williams [26]. Es una de las herramientas más maduras que se adecua a la predicción temprana del rendimiento. Utiliza modelos simples y que son fáciles de construir y resolver. SPE-ED solamente soporta redes de tipo $M/M/n$, lo que significa que los tiempos de llegada y servicio solamente pueden ser modelar teniendo distribuciones exponenciales. También se menciona que SPE-ED reporta solamente tiempos medios, lo cual es una limitación importante para alguien que quiera basar sus decisiones en valores extremos. SPE-ED se puede encontrar en http://www.spe-ed.com.

6. QPME: De acuerdo con el sitio Web *Petri Nets World*[4] QPME es una de las pocas herramientas de redes de Petri que sportan análisis de rendimiento avanzado y es la única herramienta actual que soporta redes de colas de Petri jerárquicas. El formalismo de modelado de redes de colas de Petri ha demostrado ser muy adecuado para representar

[4] http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html

sistemas de comercio electrónico distribuidos pero, por la falta de buenas herramientas es que el desarrollo de QPME fue motivado. QPME se ocupa de reducir el problema de explosión del espacio de estado. Se señala que con QPME no se puede modelar restricciones de memoria y que de momento no hay técnicas de análisis disponible, solamente de simulación. Otra limitación es que solamente se puede modelar llamadas sincrónicas y esto no lo hace adecuado para modelar sistemas de paso de mensajes. QPME se puede encontrar en https://se.informatik.uni-wuerzburg.de/tools/qpme/.

2.1.14. *Middleware* Orientado a Mensajes

Junto con el crecimiento de Internet, los sistemas distribuidos han crecido a una escala masiva. Hoy en día, estos sistemas puede involucrar a miles de entidades distruidas globalmente. Esto ha motivado el estudio de modelos de comunicación y sistemas flexibles, que puedan reflejar la naturaleza dinámica y desacoplada de las aplicaciones.

La integración de tecnologías heterogéneas es una de las áreas principales en donde la comunicación basada en mensajes juega un papel clave. Más y más compañías se enfrentan al problema de integrar sistemas y aplicaciones heterogéneas dentro y fuera de la organización ya sea por fusiones, adquisiciones, requisitos comerciales o simplemente un cambio en la direción tecnológica. No es raro encontrar una gran cantidad de tecnologías y plataformas dentro de una solo compañía, desde productos de código libre y comerciales hasta sistemas y equipos heredados (*legacy*).

La comunicación basada en mensajes también ofrece la habilidad de procesar solicitudes de manera asincrónica, proporcionando a los arquitectos y desarrolladores soluciones para reducir o eliminar cuellos de botella en un sistema e incrementar la productividad del usuario final y del sistema en general. Dado que la comunicación basada en mensajes provee un alto grado de desacoplamiento entre componentes, los sistemas que utilizan esta tecnología también logran contar con altos grados de agilidad y flexibilidad en su arquitectura.

A los sistemas de mensajería de aplicación-a-aplicación que se utilizan sistemas de negocios se les denomina genéricamente sistemas de mensajería empresarial o *middleware* orientado a mensajes [9]. MOM permite a dos o más aplicaciones intercambiar información en forma de mensajes. Un mensaje, en este caso, es un paquete autocontenido de datos de negocio y encabezados de enrutamiento de red. Los datos de negocio contenidos en un mensaje puede ser cualquier cosa, van a depender de cada negocio, y usualmente contiene información acerca de algún tipo de transacción. En sistemas de mensajería empresariales, los mensajes informan a una aplicación de la ocurrencia de algún evento en otro sistema. La tasa de mensajes depende de la capacidad de la implementación de MOM o *broker*. El retraso depende de la latencia en el *broker* y en los caminos de entrada y salida [4].

Al usar MOM, los mensajes son trasmitidos desde una aplicación a otra a través de la red. Productos de *middleware* empresarial aseguran que los mensajes se distribuyan correctamente entre las aplicaciones. Además estos productos usualmente proporcionan tolerancia a fallos, balanceo de carga, escalabilidad y soporte transaccional para sistemas que necesitan que necesitan intercambiar de manera confiable grandes cantidades de mensajes.

Los fabricantes de MOM usan diferentes formatos de mensajes y protocolos de red para intercambiar mensajes pero la semántica básica es la misma. Una interfaz de programación de aplicación (API, por sus siglas en inglés) se utiliza para crear un mensaje, cargar los datos, asignar información de enrutamiento y enviar el mensaje. La misma API se utiliza para recibir los mensajes producidos por otras aplicaciones.

En todos los sistemas modernos de mensajería empresarial, las aplicaiones intercambian mensajes a través de canales virtuales llamados destinos(*destinations*). Cuando un mensaje se envía, se dirije a un destino (por ejemplo una cola o un tópico) no a una aplicación específica. Cualquier aplicación qu subscriba o registre un interés en ese destino puede recibir el mensaje. De esta forma, las aplicaciones que reciben mensajes y aquellas que envían mensajes están desacopladas. Los emisores y receptores no están enlazados uno con otro de ninguna forma y pueden enviar y recibir mensajes como mejor les parezca.

Un concepto clave de MOM es que los mensajes son entregados de forma asincrónica desde un sistema a otros sobre la red. El entregar un mensaje de manera asicrónica significa que el emisor no requiere esperar a que el mensaje sea recibido o manejado por el receptor, él es libre de enviar el mensaje y continuar su procesamiento. Los mensajes asicrónicos son tratados como unidades autónomas: cada mensaje es autocontenido y lleva consigo todos los datos necesarios para ser procesado.

En comunicación de mensajes asincrónicos, las aplicaciones usan una API para contruir un mensaje, luego pasarlos al MOM para su entrega a uno o varios recipientes (Figura 2.2). Un mensaje es un paquete de datos que es enviado desde una aplicación a otra sobre la red. El mensaje debe de ser autodescriptivo en el sentido que debe de contener todo el contexto necesario para que permit a los recipientes llevar a cabo su trabajo de forma independiente.

La arquitecturas de MOM de hoy en día varían en su implementación. Van desde las arquitecturas centralizadas que dependen de un servidor de mensajes para realizar enrutamiento a arquitecturas descentralizadas que distribuyen el procesamiento del servidor hacia los clientes. Una variedad de protocolos como `TCP/IP`, `HTTP`, `SSL` y `IP` son empleados como capa de transporte de red.

Los sistemas de mensajería están compuestos por clientes de mensajería (*messaging clients*) y algún tipo de servidor MOM. Los clientes envían mensajes al servidor de mensajería el cual distribuye estos mensajes a otros clientes. El cliente es una aplica-

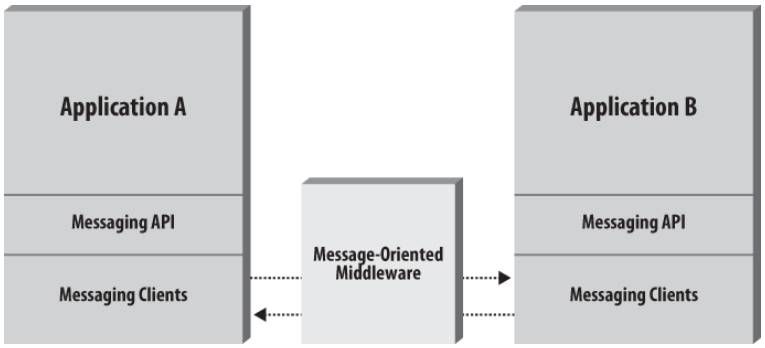

Figura 2.2: *Middleware* Orientado a Mensajes. Tomado de [9]

ción de negocio o componente que es usa una API de mensajería.

2.1.15. Modelos de mensajería

Punto-a-Punto

Este modelo de mensajería permite a los clientes enviar y recibir mensajes de forma síncrona como asíncrona a través de canales virtuales conocidos como colas. En este modelo a los productores de mensajes se les llama emisores (*senders*) y a los consumidores se les conoce como receptores (*receivers*). El modelo punto-a-punto ha sido tradicionalmente un modelo basado en *polling*, en donde los mensajes son solicitados desde la cola en lugar de ser puestos en el cliente automáticamente. Los mensajes que se envían a una cola son recibidos por uno y solo un *receiver*, aunque pueden haber otros *receivers* escuchando en la cola por el mismo mensaje. Este es un modelo que promueve acoplamiento esto porque generalmente el *sender* conocer cómo el mensaje va a ser utilizado y quién lo va a recibir.

Publish-and-Subscribe

En este modelo, los mensajes son publicados en un canal virtual llamado tópico. Los productores de mensajes son conocidos como *publishers* y a los consumidores se les llama *subscribers*. Los mensajes pueden ser recibidios por múltiples *subscribers*, a diferencia del modelo punto-a-punto. Cada *subscriber* recibe una copia de cada mensaje. Este es un modelo basado en *push* en donde los mensajes son automáticamente transmitidos a los consumidores sin que estos tengan que solicitarlos o revisar la cola por nuevos mensajes.

Este modelo tiende a ser menos acoplado que el modelo punto-a-punto debido a que el publicador del mensaje generalmente no está conciente de cuántos subscritores hay y lo qué van a hacer estos con el mensaje.

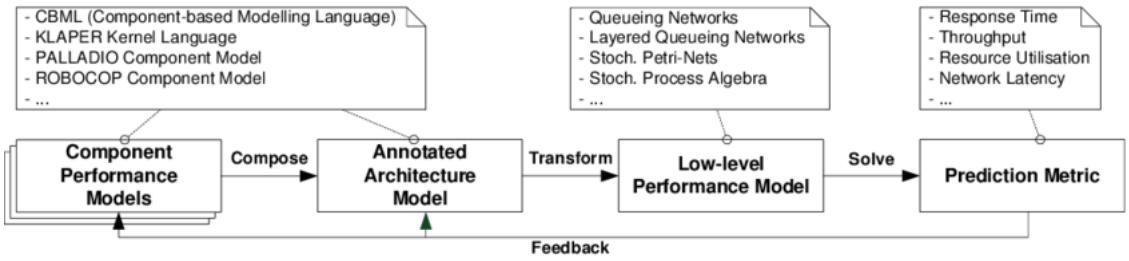

Figura 2.3: Predicción de rendimiento dirigido por modelos. Tomado de [13]

2.1.16. Enfoques de ingeniería de rendimiento para *middleware* orientado a mensajes

En [13] se extiende el *Palladio Component Model* con *performance completions* para *middleware* orientado a mensajes. *Performance completions* [27] proporcionan el concepto general de incluir detalles de bajo nivel de ambientes de ejecución en modelos de rendimiento. Con la extensión del modelo, los arquitectos de software puede especificar y configurar comunicación basada en mensajes utilizando un lenguaje basado en patrones de mensajería.

La influencia en el rendimiento de *middleware* orientado a mensajes fue estudiada en [5]. Se consideró el *middleware* como un factor determinante del rendimiento en sistemas distribuidos y se hizo un mayor enfoque en su modelado y evaluación. Se propuso un enfoque basado en medición en combinación con modelos matemáticos para predecir el rendimiento de aplicaciones J2EE[5]. Las mediciones proporcionan los datos necesarios para calcular los valores de entrada del modelo de red de cola. El cálculo refleja el comportamiento de la aplicación en particular. La red de cola se resuelve para derivar métricas de rendimiento tales como tiempos de respuesta y *throughput* de la aplicación.

La investigación llevada a cabo en [4] presenta un modelo abstracto de *middleware* orientado a mensajes basado en Apache Qpid junto con el uso de modelos exógenos de autorregressión (ARX por sus siglas en inglés) que describen el comportamiento del *middleware* durante condiciones de cuellos de botella. Los modelos ARX son modelos autorregresivos en donde la salida depende de la salida anterior así como de estimulos externos. Estos componentes son integrados para producir una técnica generalizada de calibración para rendimiento del *middleware* y detección de cuellos de botella en el mismo.

En [8] se construyen modelos de rendimiento para *middleware* que implementa el estándar *Java Messaging Service* (JMS). Se utiliza análisis de código y mediciones experimentales de implementaciones de JMS populares para mostrar situaciones en las que el rendimiento observado no es predecido de forma preciesa por otros modelos. Se

[5] *Java Enterprise Edition*

proporciona un análisis técnico detallado de los efectos observados como base esencial para futuros trabajos de modelado. Por último, se diseña un modelo de rendimiento que captura estos efectos y se valida el modelo utilizando mediciones experimentales.

Un un modelo analítico $M/M/1$ con políticas*first in - first out* y prioridad de colas fue diseñado y desarrolloado en [3] para evaluar el rendimiento de *middleware* orientado a mensajes y llamados a procedimientos remotos (RPC por sus siglas en inglés). Los modelos comparan el rendimiento de *middleware* orientado a mensajes y RPC con prioridad de colas y analizan el *throughput* de estos paradigmas de comunicación. Varios parámetros de entrada son usados para determinar la configuración óptima para lograr el máximo rendimiento. Los resultados prueban que al utilizar *middleware* orientado a mensajes con prioridad de cola, el *throughput* del sistema puede ser mejorado.

2.1.17. Posicionamiento de la investigación con respecto a la literatura consultada

A partir de lo recolectado se pudo conocer que mucho del esfuerzo que se ha hecho en este campo de estudio ha tenido baja aceptación en la industria pero que, por otro lado, compañías y sistemas que han utilizado estos métodos de modelaje y evaluación han obtenido buenos resultados. Queda claro también que la aplicación de estos modelos permiten entender la influencia que tienen tanto de los diferentes componentes de software como el impacto que pudiera generar en ellos durante la ejecución de un sistema.

También se pudo reconocer que existe necesidad en aplicar enfoques de modelado del rendimiento en el desarrollo de sistemas modernos. No existen enfoques de modelado de rendimiento para microservicios que hoy en día es un estilo de arquitectura de software sumamente popular. Es por esto que se considera que la realización de estudios exploratorios para determinar la influencia en el rendimiento que tienen diferentes componentes, librerías y productos de software sobre un sistema van a brindar nuevo conocimiento para dar a conocer factores que favoren o desfavorecen el uso de los mismos y, además de esto, podría representar un marco de referencia inicial por medio del cual se pueda evaluar la adopción de estas tecnologías *a priori*.

Por último, durante la búsqueda de artículos se encontraron muy pocos de ellos provenientes de universidades de latinoamérica. Esto representa, a juicio del autor, una oportunidad para estudiar, probar y generar conocimiento sobre estos temas.

2.2. Aplicación de referencia

Durante la elaboración de la revisión de la literatura, se procedió a estudiar algunas de las aplicaciones que se utilizaron en la realización de los respectivos estudios con el fin de tomar alguno de estos como referencia para la investigación propuesta. Para seleccionar una aplicación que se adecue al tema propuesto, se establecieron los siguientes criterios:

1. Aplicación de código libre
2. Que cuente con modelos de rendimiento hechos en *Palladio Workbench*
3. Desarrollada en un lenguaje de programación y con herramientas de actualidad
4. Que continue siendo mantenida y mejorada por sus autores

Del listado anterior, quizás el criterio más restrictivo es el #2 pero al mismo tiempo se considera el de mayor importancia debido a que en esta investigación se están dando los primeros pasos con el modelado de arquitecturas utilizando Palladio y no se deseaba empezar desde cero sino más bien tomar un modelo de referencia e ir haciendo pequeñas modificaciones y probar los resultados.

Durante la exploración de aplicaciones candidatas se pudo constatar que muchas de ellas fueron creadas para probar los objetivos de un estudio en particular y luego cayeron en desuso. A pesar que varias de ella se pueden acceder fácilmente por medio de Internet, carecen de mantenimiento y documentación y esto causa que sean difíciles de evaluar. Una de las motivaciones principales detrás de la elección de método Palladio para modelado de arquitecturas de software, es que este hay sido un proyecto que cuenta con un nivel de madurez importante y está en constante mantenimiento y mejora por parte de su equipo de desarrollo. Aún así se pudo encontrar aplicaciones que utilizaron modelos de Palladio pero se encontraban en “abandono”.

CloudScale, un proyecto generado a partir de la experiencia de Palladio, es un método que permite identificar y gradualmente resolver problemas de escalabilidad en aplicaciones existentes. *CloudScale* también permite modelar alternativas de diseño y el análisis de su efecto en la escalibilidad y costo [25]. El sitio Web de *CloudScale*, `http://www.cloudscale-project.eu/` contiene publicaciones e información general acerca de este método, así también como del *CloudScale Environment*, un entorno de desarrollo integrado creado para modelaje y análisis.

Con el fin de validar *CloudScale*, varias aplicaciones fueron creadas. Una ventaja de las aplicaciones que utlizan *CloudScale* es que deben de tener un modelo basado en Palladio para su posterior procesamiento. Otra de las cosas que hicieron que las aplicaciones creadas para esta metología resultaran atractivas para esta investigación, es que

el proyecto *CloudScale* es reciente por lo que estas aplicaciones no se consideran como "viejas" y se les da mantenimiento.

Una de estas aplicaciones que se crearon para probar esta metolodología es *CloudStore*, una aplicación Web de código libre que emula una tienda (*e-commerce*) de libros en línea. Su objetivo principal es ser utilizada para el análisis de las características de los sistemas en la nube como escalabilidad, capacidad, elasticidad y eficiencia. Fue desarrollada como la aplicación de muestra para validar las herramientas de *CloudScale*.

La aplicación fue desarrollada en Java utilizando la librerías de *Spring* y se ejecuta en un servidor Tomcat. Utiliza una base de datos MySQL. Para mayor información sobre *CloudStore*, se puede visitar su repositorio en GitHub: `https://github.com/CloudScale-Project/CloudStore`.

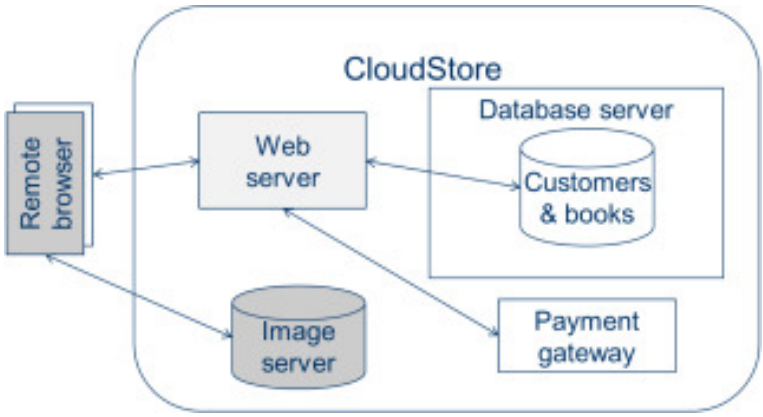

Figura 2.4: Arquitectura Conceptual de CloudStore. Tomado de [13]

En la figura 2.4 se muestra la arquitectura conceptual de *CloudStore*. Tiene un diseño simple: una aplicación Java que se ejecuta en un servidor Web y que realiza consultas a una base de datos MySQL. Para llevar a cabo los pagos, la aplicación se comunica con un *gateway* para pagos, un servicio Web publicado en un servidor externo al del *CloudStore* que simula el tiempo de respuesta de podría tomar realizar un pago.

Una característica de *CloudStore* particularmente atractiva para este estudio es precisamente el *gateway* de pagos. La interacción entre *CloudStore* y este *gateway* de pagos representa un clásico escenario de integración de sistemas. En su versión original, *CloudStore* realiza llamadas a este *gateway* de pagos de manera *ad hoc* y es aquí en donde la introducción de comunicación basada en mensajes puede ser una mejor opción al actual debido que:

- Permite que *CloudStore* se desacople del *gateway* de pagos: la aplicación no necesita saber información específica de este *gateway* ni la forma en cómo debe interactuar con él, esto se lo delega a un tercero.
- La comunicación basada en mensajes puede garantizar el envío de peticiones al gateway de pagos.

2.2.1. Adaptación de *CloudStore*

Con el fin de adaptar *CloudStore* para que soporte comunicación basada en mensajes cuando se ejecute un pago, se propone lo siguiente:

1. Selección de una solución para comunicación basada en mensajes

2. Selección de librería(s) de código para realizar comunicación entre *CloudStore* y la solución seleccionada en el punto anterior

3. Implementar un consumidor de mensajes de petición de pago: una aplicación que estará monitoreando(*pooling*) y procesando mensajes provenientes de la solución seleccionada en el punto #1 y que los entregará al *gateway* de pagos. Adicionalmente esta aplicación comunicará al *CloudStore* el estado del procesamiento del pago

4. Reemplazar el código existente que se dedica a realizar el pago contra el *gateway* de pagos por nuevo código que se comunicará con la solución de mensajería

Para el punto #1, se seleccionó ActiveMQ[6], el servidor más popular de mensajería de código libre.

Al estar utlizando ActiveMQ, el cual es un servidor que implementa el estándar *Java Messaging Service*, se va a utilizar la librería `javax.jms` con versión 1.1.

La aplicación consumidora/procesadora de mensajes será una aplicación Java que se va a ejecutar como un servicio.

La arquitectura adaptada de *CloudStore* para que utilice comunicación basada en mensajes se muestra en la figura 2.5. Ahora, en lugar de realizar una llamada *ad hoc* al *gateway* de pagos, la aplicación se comunica con una cola de mensajería para pagos tanto para el envío como para la recepción del pago. A este esquema de comunicación se le conoce como *request-reply*, esto porque la aplicación que inicia la petición se queda a la espera del resultado.

Para realizar un pago, *CloudStore* enviará un mensaje a una cola de pagos, en la figura 2.5 se identifica como "Payment Queue". La aplicación consumidora de mensajes, estará monitoreando esta cola constantemente con el fin de tomar los mensajes y procesarlos contra el *gateway* de pagos. Una vez que el pago se haya efectuado, la aplicación consumidora envía el resultado del pago a otra cola identificada en la figura 2.5 como "Payment Status Queue". *CloudStore* estará monitoreando los mensajes provenientes de esta cola con el fin de conocer el estado del pago.

[6] http://activemq.apache.org/

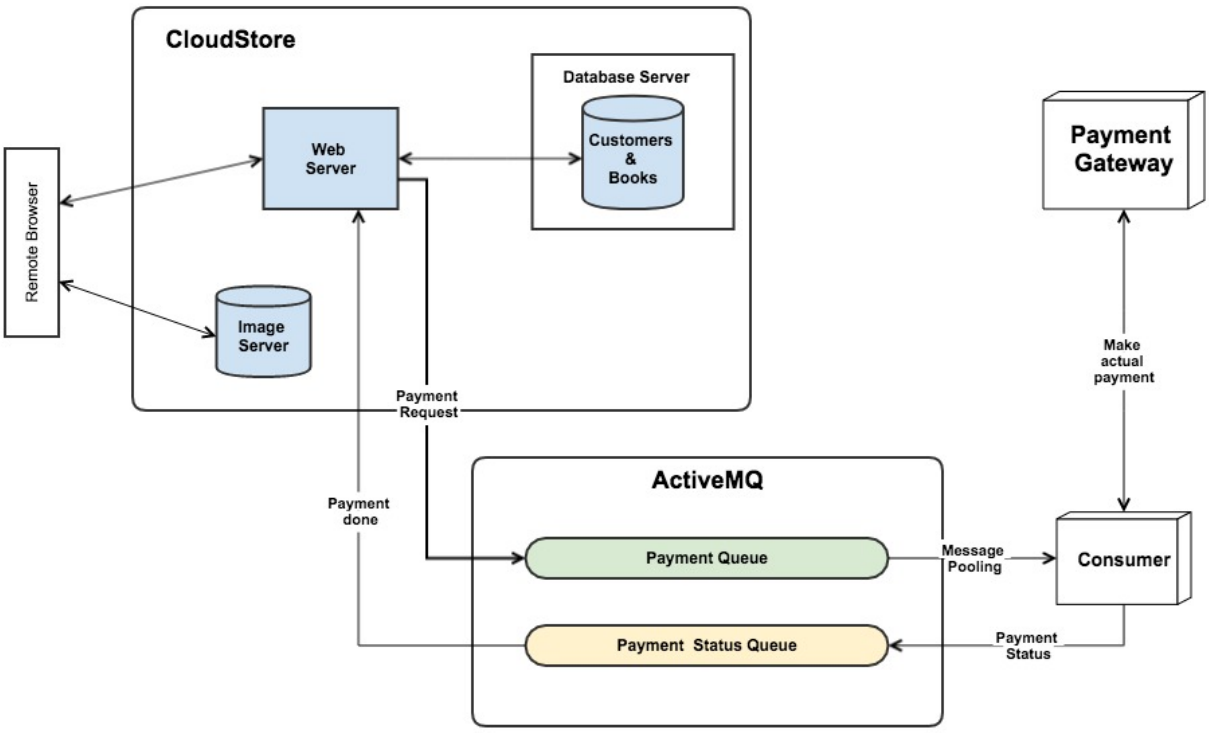

Figura 2.5: Arquitectura Conceptual de CloudStore utilizando comunicación basada en mensajería

Aplicación original *vs.* Aplicación modificada

Para la adaptación de *CloudStore*, se tomó del repositorio de código de este proyecto que se encuentra en `https://github.com/CloudScale-Project/CloudStore` y se copió a un nuevo repositorio: `https://github.com/tema-selecto-investigacion/cloudstore-boot`. Esto se hizo con el fin de tener mayor control y visibilidad sobre los cambios hechos. En la figura 2.6 se muestra el repositorio creado para *CloudStore*.

Seguidamente, se siguieron las instrucciones para su puesta en ejecución: instalación dependencias de código necesarias y creación de *scripts* para crear la base de datos y agregar registros a la misma. Una vez que la aplicación se construyó y se configuró para que trabajara junto con una base de datos MySQL local, la misma se pudo ejecutar en el puerto `:8080`.

Luego de ejecutar la aplicación localmente y probar que funcione adecuadamente, se procedió a crear un ambiente de producción en Amazon AWS. Para esto, se creó un ambiente con un servidor Apache Tomcat 8 y una base de datos MySQL utilizando los servicios de *Elastic BeanStalk* y *Relational Database Services*(RDS) respectivamente. A este ambiente se llamó `cloudstore-plain`. Se configuró la aplicación para que cuando

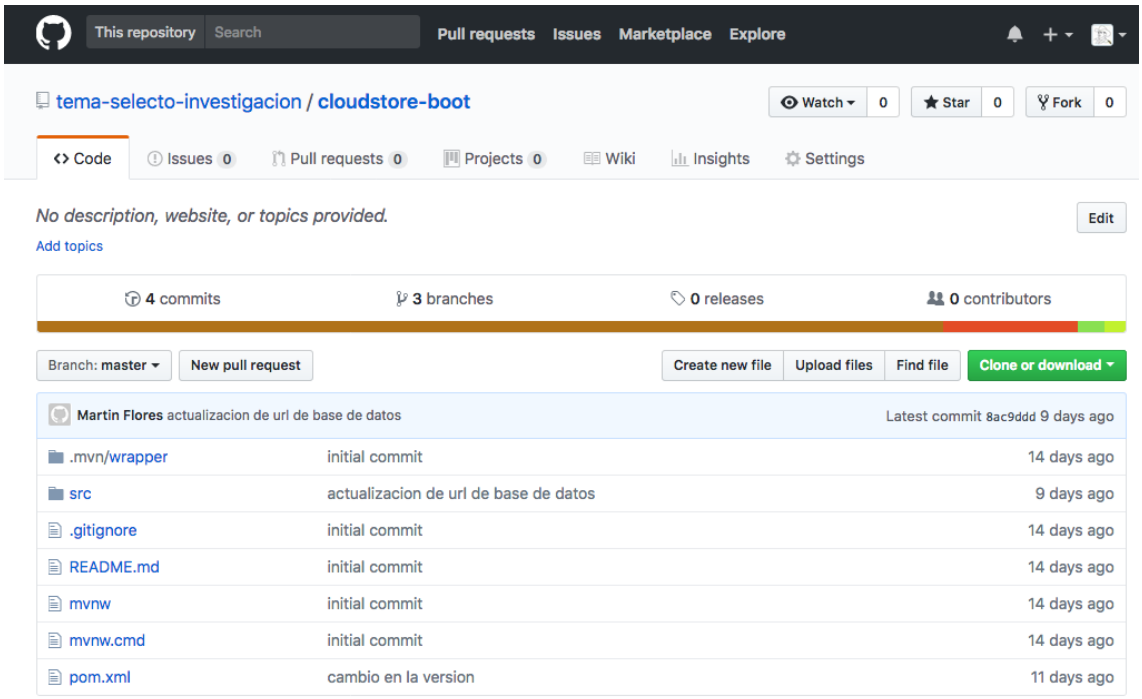

Figura 2.6: Repositorio de código para CloudStore en GitHub

fuera instalada en este ambiente se conectara a la base de datos en AWS.

Cuando ambas, la aplicación local y la instalada en AWS pudieron ser ejecutarse satisfactoriamente, se empezó con la adaptación de *CloudStore*. Lo primero fue instalar e iniciar una instancia de ActiveMQ localmente. En su configuración por defecto, ActiveMQ se ejecuta en el puerto `:61616` y su aplicación Web de administración en el puerto `:8161`. Se crearon dos colas de mensajes: `payment-queue`, en donde se enviarán los mensajes(peticiones) de pago y `payment-status-queue`, en donde se enviarán los mensajes del estado del pago. Luego se hizo un nuevo *branch* (rama) en el repositorio de código para los nuevos cambios que se iban a realizar. Este *branch* lleva por nombre `jms`. Se instaló la dependencia `javax.jms 1.1` para que la aplicación pudiera comunicarse por medio del estándar *Java Messaging Service*. Se cambió el flujo de trabajo de *CloudStore* para que en lugar de llamar al *gateway* de pago de manera *ad hoc*, se creará un mensaje con la información de la compra para luego enviar este a la cola de mensajes `payment-queue`. Por último, se agregará código para monitorear y consumir el resultado del pago, el cual debería de ser entregado por la cola `payment-status-queue`. A este estilo de comunicación se le conoce como *request-reply*: la parte que envia la comunicación debe de aportar algún tipo de encabezado y/o meta-dato al mensaje para su posterior identificación. Luego de enviar el mensaje, la aplicación se pone en modo de monitoreo para filtrar y procesar solamente aquellos mensajes que fueron generados por esta.

Se creó un nuevo proyecto en Java llamado `jms-receiver`, cuyo repositorio de códi-

go se encuentra en https://github.com/tema-selecto-investigacion/jms-receiver (Figura 2.7). Este nuevo proyecto va a llevar a cabo las siguientes actividades: monitoreo y consumo constante de mensajes de la cola `payment-queue`, realizar una llamada al *gateway* de pagos, obtener el resultado del pago y enviarlo por la cola `payment-status-queue`.

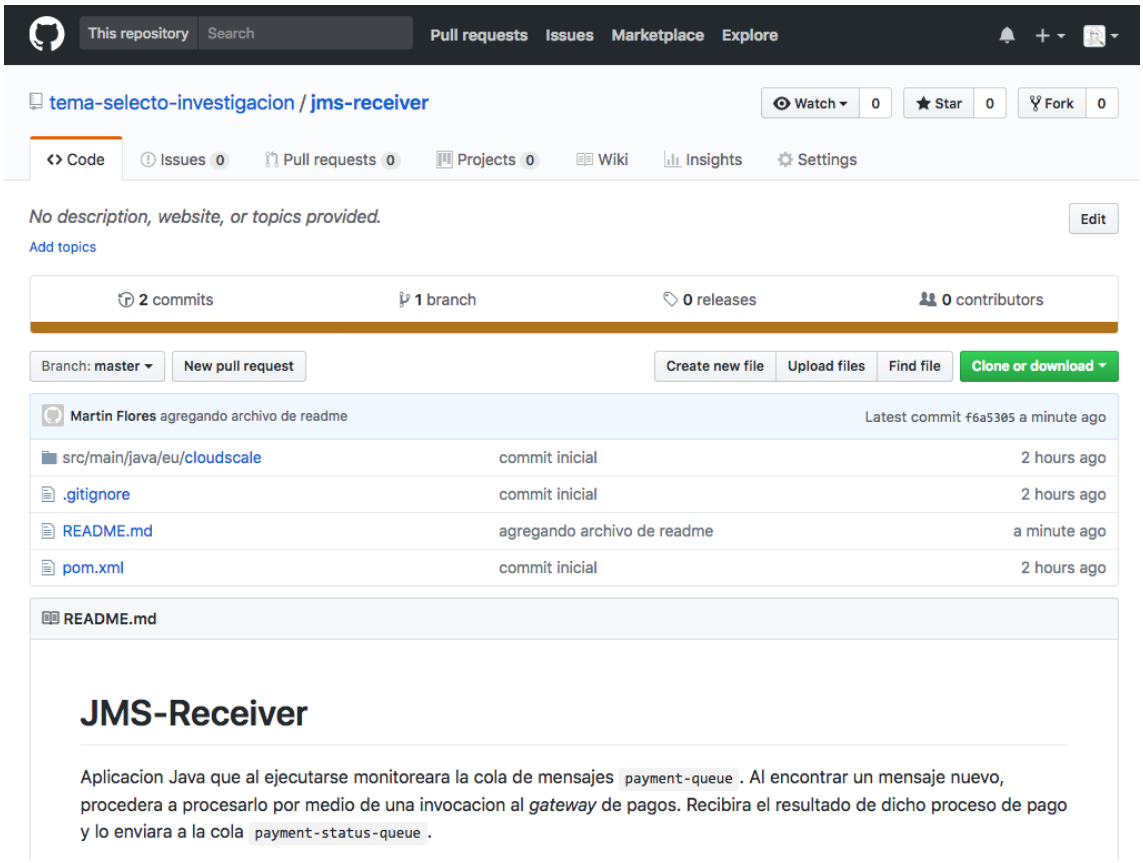

Figura 2.7: Repositorio de código para el proyecto `JMS-Receiver` en GitHub

Primero se probaron los cambios localmente y se pudo comprobar como ahora los mensajes con las peticiones de pagos viajaban a través de las colas de mensajería en lugar de ser generadas a lo interno de *CloudStore*.

Para poner estos nuevos cambios en un ambiente de producción se realizaron las siguientes actividades:

- Se creó un nuevo ambiente en *Elastic BeanStalk* llamado `cloudstore-jms`
    - En este nuevo ambiente se va a instalar la aplicación de *CloudStore* modificada que vive en el *branch* `jms`.
- Se provisionó una nueva máquina virtual en el servicio *Amazon Elastic Compute Cloud* (EC2). Una máquina virtual con Ubuntu 16.04, Java y ActiveMQ
    - En esta máquina virtual se va a ejecutar la aplicación `jms-receiver` y el servicio de ActiveMQ con las colas de `payment-service` y `paymente-status-queue`

- Se configuró la versión de *CloudStore* modificada para que se comunicara con las colas de mensajería que se encuentran en la nueva máquina virtual provisionada

Se puede comparar los cambios hechos a la versión original de *CloudStore* en el siguiente *Pull Request*: https://github.com/tema-selecto-investigacion/cloudstore-boot/pull/1.

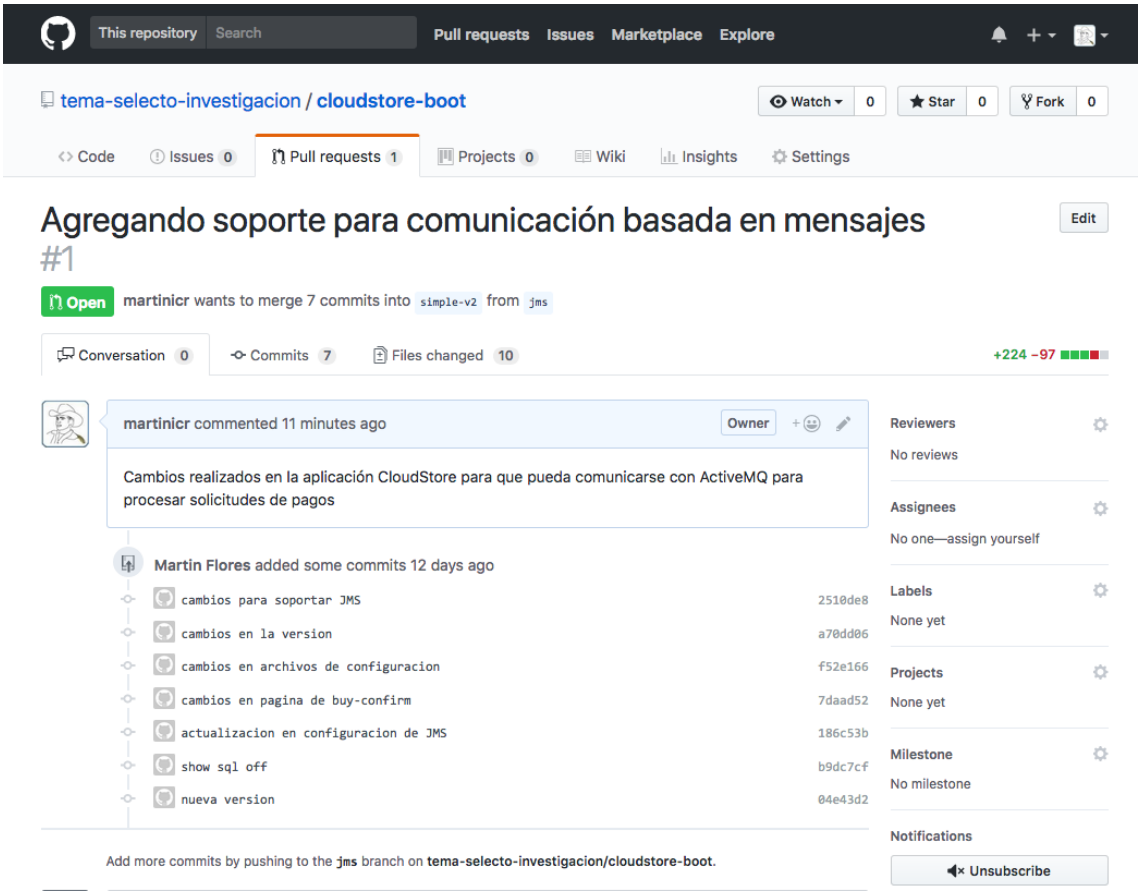

Figura 2.8: Diferencias entre el código original de *CloudStore* y el implementado

En la figura 2.9 se pueden ver que los dos ambientes de *CloudStore* se fueron puestos en producción de forma satisfactoria.

A continuación se presentan las dos versiones de *CloudStore* ejecutándose en un ambiente de producción. En la figura 2.10 se muestra la versión original y el la figura 2.11 la versión modificada. Ambas aplicaciones lucen similar, pero apuntan a dominios diferentes y también difieren en la forma en la que se realiza el proceso de pago[7].

[7]Ambas aplicaciones podrían permanecer no disponibles durante ciertos periódos con el fin de no incurrir en costos innecesarios. En caso de querer probar la aplicación y este no se encuentre disponible, se puede poner en contacto con el autor para su puesta en ejecución.

Figura 2.9: Los dos ambientes de *CloudStore* en AWS

2.3. Mediciones de rendimiento iniciales

Para realizar mediciones de rendimiento, los autores de *CloudStore* crearon una herramienta de pruebas de carga basada en Apache JMeter[8] a la cual llamaron *Distributed JMeter*. Esta herramienta se usa para crear cargas de trabajo las cual simulan múltiples usuarios realizando peticiones a un servidor Web el cual esté ejecutando *CloudStore*. *Distributed JMeter* se ejecuta en un entorno Linux en EC2 en AWS. Envía solicitudes `HTTP` a servicio de balanceo de carga *Elastic Load Balancer*. JMeter registra los datos de todas las solicitudes y respuestas `HTTP`, lo que permite la medición de las caracteríistas del sistema tal y cómo el número de solicitudes que son exitosas, caducan o fallan. *Distributed JMeter* se encuentra disponible en `https://github.com/CloudScale-Project/Distributed-Jmeter`.

En la figura 2.12 se muestra la instalación de *CloudStore* junto con el generador de pruebas de carga *Distributed JMeter*, los componentes funcionales de *CloudStore* (servidor Web, base de Datos, servidor de imágenes y *gateway* de pagos) instalados en la platforma AWS.

- El *Elastic load balancer* es parte de AWS y es utilizado para distribuir tráfico entrante a la aplicación a través de múltiples instancias EC2. En la instalación de ambas versiones de *CloudStore*, el servicio de balanceo se configura a través de *Elastic BeanStalk*.

- Un servidor Web instalado en varias instancias EC2 que se ejecuta bajo un sistema operativo Linux y que consiste de dos componentes:

  1. Un proxy Web basado en Nginx que reenvía peticiones `HTTP` a la aplicación de *CloudStore*
  2. La aplicación de *CloudStore*

[8] `https://jmeter.apache.org`

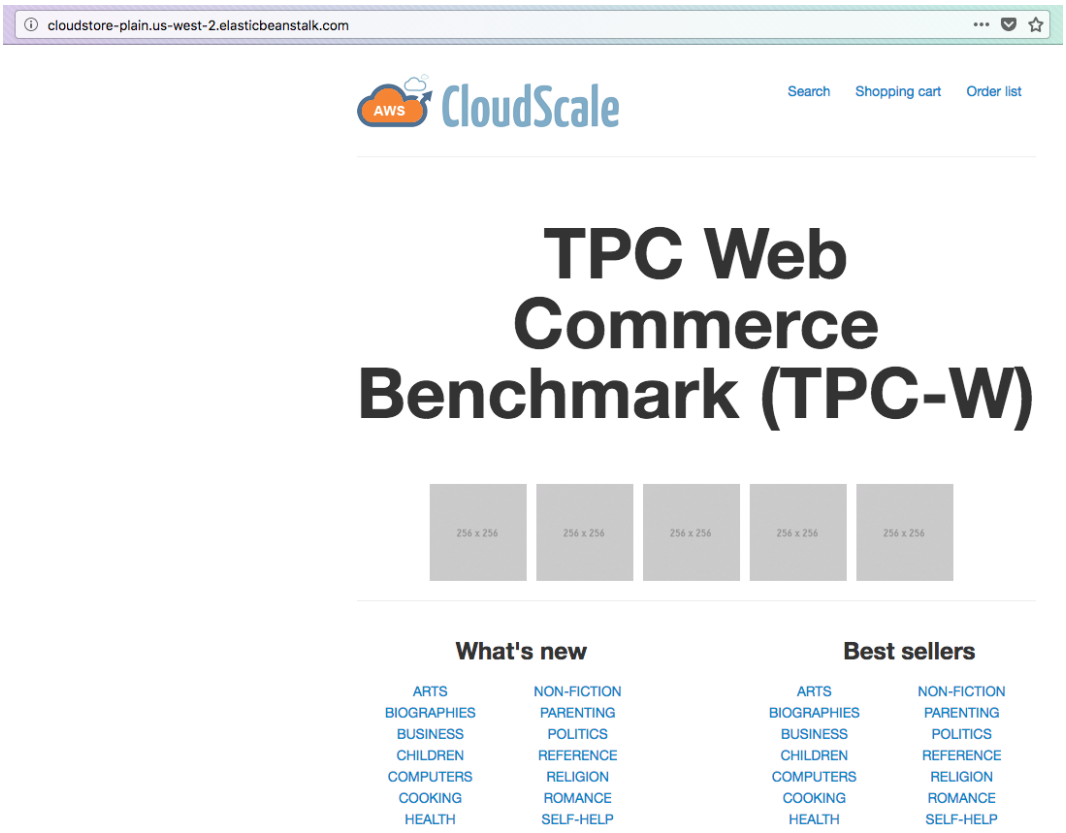

Figura 2.10: Versión original de *CloudStore* (*plain*) en AWS

- Un servidor de base datos que se proporciona a través de RDS y que ejecuta una base de datos MySQL.
- Para ambas versiones de *CloudStore* se prescindió del servidor de imágenes que se propone en la instalación original de la figura 2.12. En su lugar, se modificó *CloudStore* para que cada vez que se necesitara entregar una imagen, se entregara una por defecto
- Un *gateway* de pagos, un generador simple de tiempos de respuesta que se introducido en lugar de un servicio de pagos externo. Con el fin de imitar un servicio real independiente a *CloudStore*, el generador de tiempos de respuesta introduce un retraso definido a partir de una distribución estadística. El generador de tiempos de respuesta se instaló en una plataforma Heroku, la cual es completamente independiente a AWS. El repositorio de código para el generador de tiempos de respuesta se encuentra disponible en `https://github.com/CloudScale-Project/Response-Generator`

2.3.1. Generación de cargas de trabajo con JMeter

En *Distributed JMeter* se ponen a disposición varias pruebas de carga para evaluar principalmente las características de escalabilidad de la aplicación (recordar que la

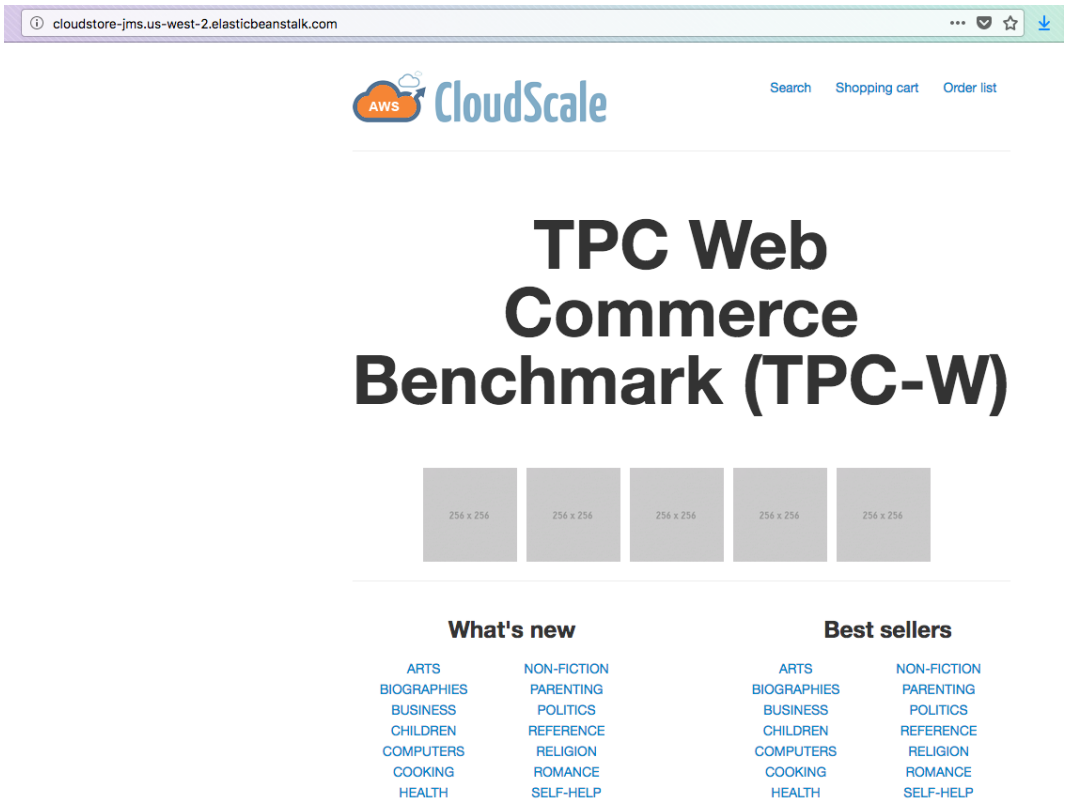

Figura 2.11: Versión de *CloudStore* adaptada para MOM en AWS

aplicación *CloudStore* fue creada validar los modelos y metodología de *CloudScale*). Estas pruebas estan basadas en escenarios en donde miles de usuarios están accediendo de forma aleatoria a las diferentes páginas Web que componen *CloudStore* por un tiempos de 10 - 15 minutos. A partir de esto, los autores esperan experimentar cambios el comportamiento de la aplicación debido al crecimiento/disminución en las instancias en donde se ejecuta.

Con el fin de explorar las características de *CloudStore* y del impacto de agregar comunicación basada en mensajes para el procesamiento de pagos, se toma como referencia el escenario de prueba llamado `cloudscale-max` que se pone a disposición en *Distributed JMeter*, en donde se simulan 2000 usuarios realizando peticiones a las páginas Web de *CloudScale* durante 15 minutos de acuerdo a una distribución de probabilidades proporcionada dentro de la prueba. Se toma esta prueba como base y se modifica para utilizar 100 usuarios en lugar de 2000 y la duración de la se cambió a 2 - 3 minutos (La duración se cambió a 2 minutos, pero por lo general los hilos -*threads* - de JMeter tienden a no finalizar luego del tiempo especificado). La intención detrás de esta modificación es la de explorar el comportamiento de *CloudStore* a una escala la cual permita dar mejor seguimiento a las acciones que se dan a lo interno de la aplicación y de esta forma lograr tener un mejor entendimiento de la misma. También se busca, al contrario de los autores de *CloudStore* no introducir cargas de trabajo que pudieran hacer que el ambiente de AWS cambiara porque esto podría impactar con los resultados

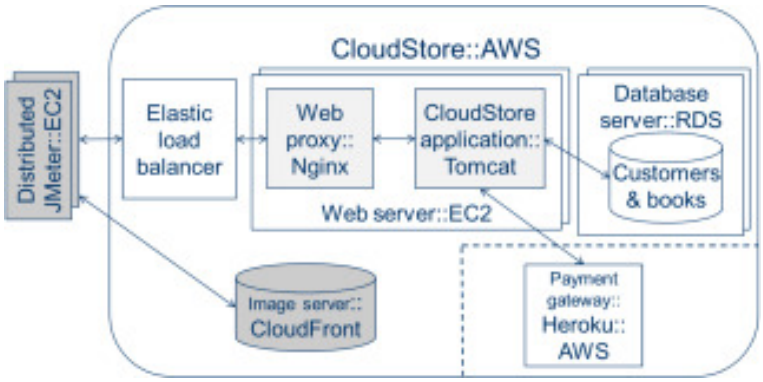

Figura 2.12: Instalación de *CloudStore* en AWS

de los tiempos de respuesta.

2.3.2. Pruebas sobre versión original de *CloudStore*

En la figura 2.13 muestra JMeter configurado con la prueba `cloudscale-max`.

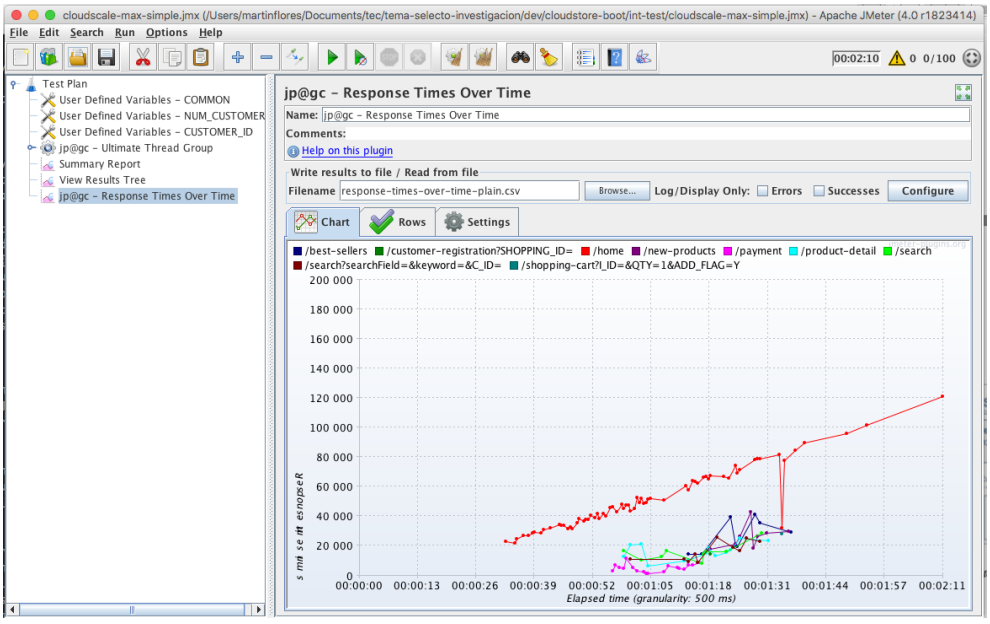

Figura 2.13: JMeter con la prueba `cloudscale-max`

Luego de ejecutar la prueba se pudo obtener los tiempos de respuesta sobre las páginas que componen la versión original de *CloudStore*. En esta investigación se está particularmente interesado en la página(recurso Web) llamado `/payment` ya que es en este en donde se ejecut la lógica del pago, en dónde se realiza la conexión con el *gateway* de pagos. En la figura 2.14 se muestran los resultados obtenidos. La línea de color fucsia es la que representa los tiempos de respuesta que experimentaron las solicitudes a `/payment`. Los resultados para `/payment` se pueden ver con mayor detalle en la figura 2.15.

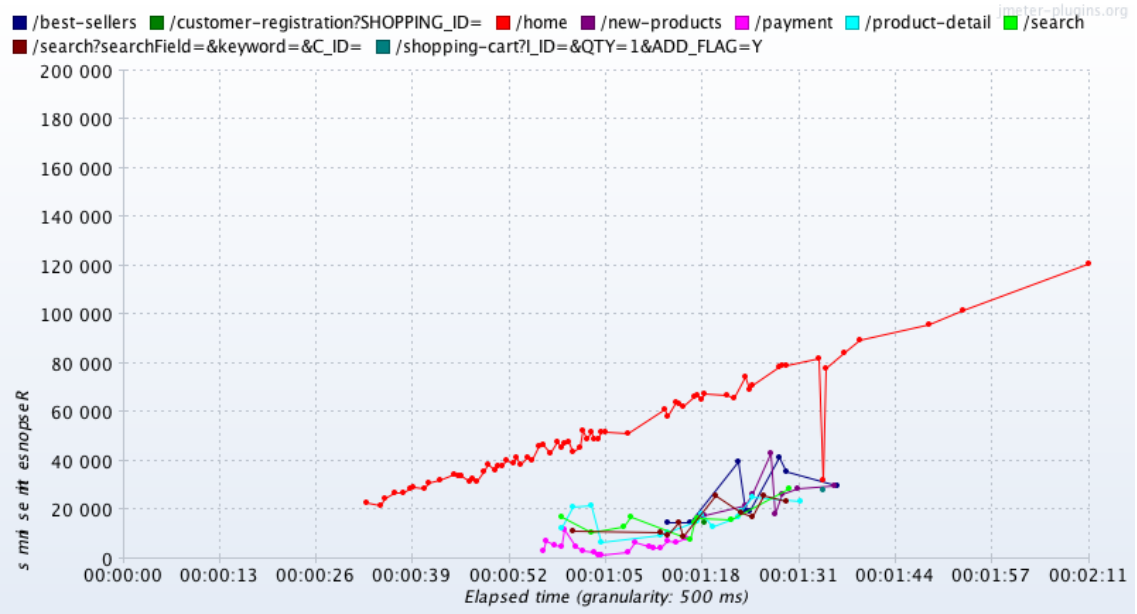

Figura 2.14: Tiempos de respuesta de las páginas de la versión original *CloudStore* (cloudstore-plain)

En promedio, las páginas Web de *CloudStore* tomaron 30 segundos en ser servidas. La página de inicio, `/home`, representada en la línea roja de la figura 2.14 muestra un crecimiento sostenido desde el inicio hasta el fin de la prueba. La página de inicio realiza varios llamados a la base de datos con el fin de mostrar productos e información de la tienda, por lo que para esta página, la base de datos podría resultar el cuello de botella.

Con respecto a `/payment` (figura 2.15), se obtuvieron tiempos de respuesta de 5 segundos en promedio. En la programación detrás de `/payment` solamente se lleva a cabo la invocación y conexión con el *gateway* de pagos. En los resultados para `/payment` también se obtuvieron varios respuestas con código `503 - Service Unavailable`, lo que indica que en un ambiente de producción real, varias solicitudes de pago no pudieron ser atendidas debido a que el servidor Web o bien el *gateway* de pagos no fueron capaces de procesar la solicitud.

Esta prueba se ejecutó desde la computadora personal del autor de esta investigación. Cabe mencionar de que cuando se ejecutó la misma prueba desde una máquina virtual en EC2 los tiempos de respuesta que se obtuvieron fueron alrededor de 40 % más rápido, las tendencias en cuanto al comportamiento de la aplicación se mantuvo similar.

2.3.3. Pruebas sobre versión modificada de *CloudStore*

Se tomó la misma prueba, `cloudscale-max`, como referencia para esta prueba. Solamente se cambió el *host* al cual apuntaba anteriormente para que realizara las pruebas sobre la versión modificada de *CloudStore*.

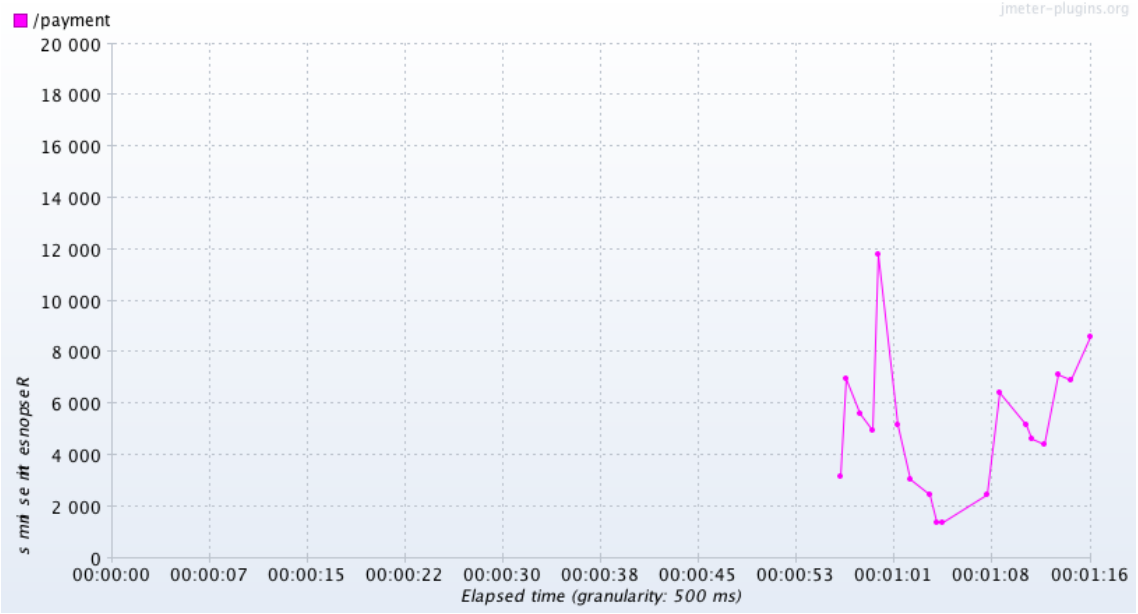

Figura 2.15: Tiempos de respuesta de `/payment` en la versión original de *CloudStore*

Para la versión modificada de *CloudStore* (figura 2.16), el tiempo promedio de respuesta de una página es de 31 segundos. Se puede notar que el comportamiento de la aplicación se mantiene similar aunque sí se experimentan cambios sobre `/payment`.

En la figura 2.17 se muestra el detalle de los tiempos de respuesta obtenidos en `/payment`. El tiempo de respuesta promedio de esta versión es de 35 segundos y mantuvo un crecimiento sostenido desde el inicio hasta el fin de la prueba. Este comportamiento es congruente con el comportamiento de la cola de mensajes: las solicitudes que tomaron menor tiempo fueron las que entraron de primero en la cola, las que duraron más fueron las que llegaron de último y por lo tanto fueron procesadas al final. Un dato interesante a tomar en cuenta es que en la versión modificada de *CloudStore* el 100 % de las solitudes a `/payment` fueron exitosas, a diferencia de la versión original en donde se obtuvieron varias respuestas con código `503 - Service Unavailable`, principalmente hacia el final de la prueba. Esto sugiere que existen ocasiones en que el *gateway* de pagos (el servicio generador de tiempos de respuesta) no puede manejar tantas solicitudes al mismo tiempo pero que cuando estas solicitudes se manejan por medio de comunicación basada en mensajes, cada solicitud de pago es servida adecuadamente.

En la prueba sobre la versión modificada de *CloudStore*, solamente se puso a disposición un consumidor de mensajes, es decir, solamente una instancia de `jms-receiver` fue ejecutada. Cuando se ejecutaron varias instancias de `jms-receiver` al mismo tiempo, se experimentaron tiempos de respuesta más bajos. Cuando se probaron hasta 5 instancias al mismo tiempo, el tiempo bajó casi 50 %, y todas las solicitudes fueron exitosas. En la configuración original de *CloudStore* este tipo de problemas se hubiera re-

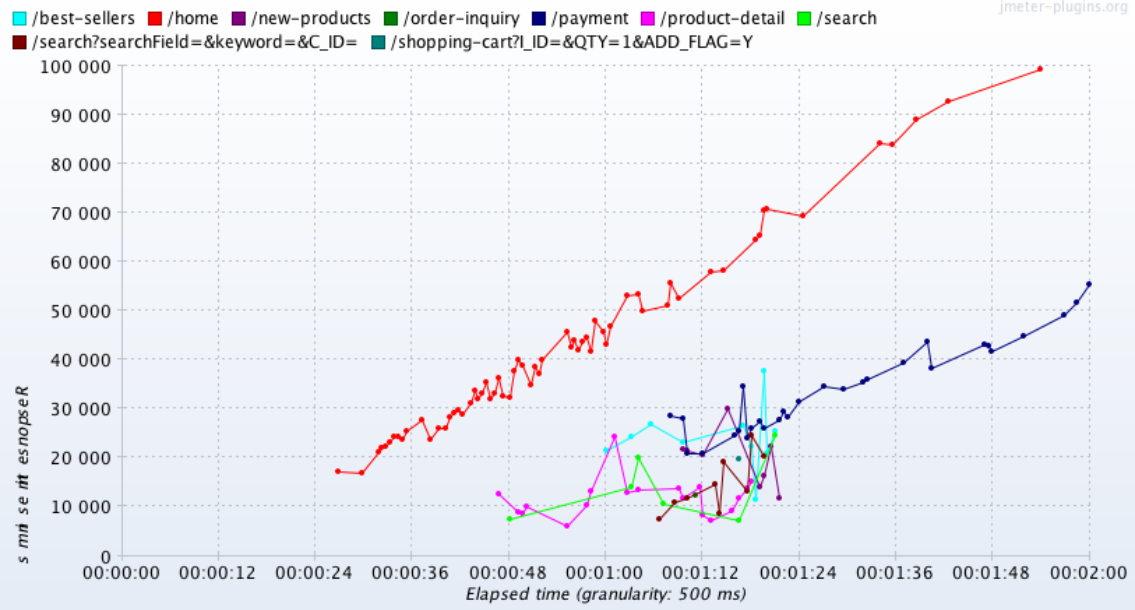

Figura 2.16: Tiempos de respuesta de las páginas de la versión modificada *CloudStore* (cloudstore-jms)

sulto por medio de la puesta en producción de más instancias.

El proyecto *CloudScale* (del cual proviene *CloudStore*) tiene su énfasis en modelado y rendimiento de escalabilidad, por lo que muchas de las pruebas que se incluyen en *Distributed JMeter* están destinadas a estresar la infraestructuras para que estan puedan crecer y nuevas instancias de máquinas virtuales o bases de datos sean creadas. Luego de estas pruebas iniciales se considera que en *CloudScale* también puede existir la oportunidad de evaluar el efecto que tiene en la escalabilidad el uso de comunicación basada en mensajes.

2.3.4. Mediciones sobre máquina virtual que ejecuta ActiveMQ

En la figura 2.18 se muestran las métricas del uso del CPU y disco duro de la máquina virtual que estaba ejecutando ActiveMQ y el consumidor de mensajes (`jms-receiver`).

Durante la ejecución de la prueba de la sección anterior, la máquina virtual no experimentó operaciones de lectura/escritura en disco. Las colas de mensajes `payment-queue` y `payment-status-queue` no están configuradas para persistir datos y además *CloudStore* delega almacenamiento y lectura de datos a la base de datos MySQL. En cuanto al uso del CPU, la máquina virtual experimentó un incremento de 0.5 % del uso del CPU durante la ejecución de la prueba.

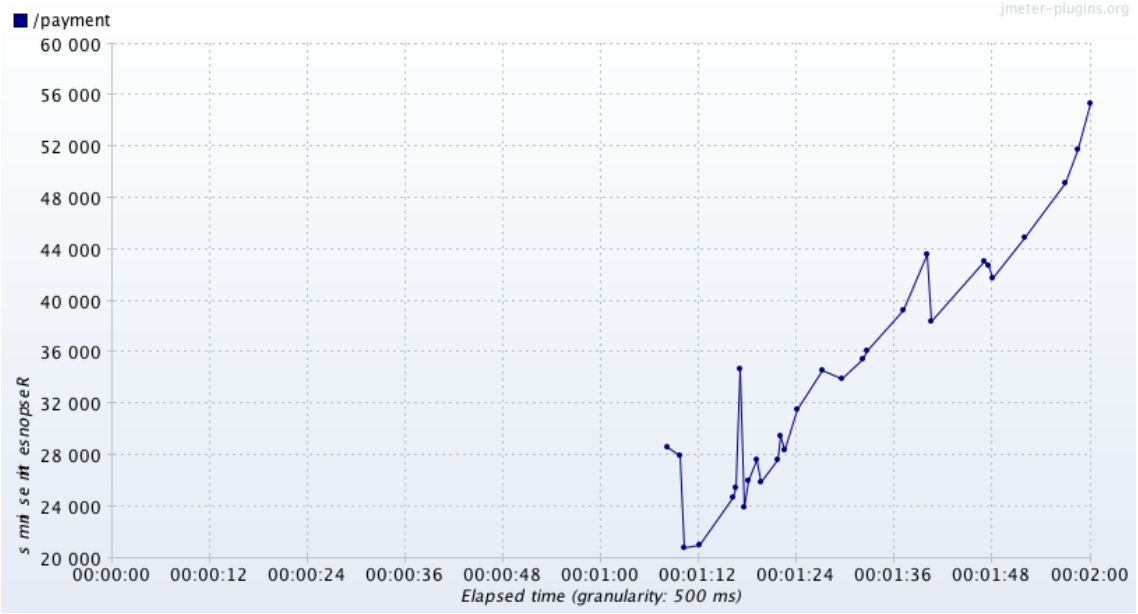

Figura 2.17: Tiempos de respuesta de `/payment` en la versión modificada de *CloudStore*

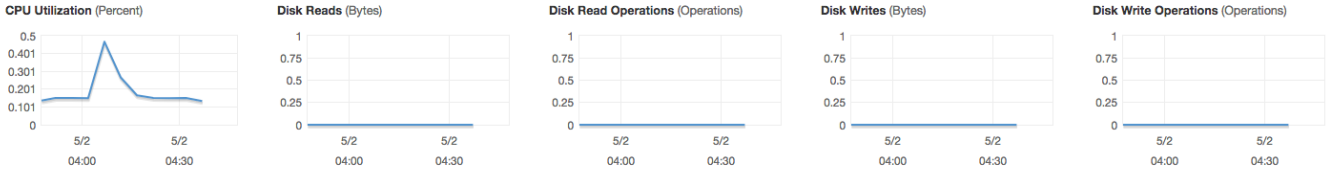

Figura 2.18: Métricas del rendimiento de la máquina virtual que ejecuta ActiveMQ en EC2

2.4. Trabajo Futuro

Las mediciones que se presentan en este informe se hicieron con el fin de empezar a probar y conocer cómo se comporta la aplicación de referencia. En las próximas semanas se pretende seguir experimentando con estas y otras mediciones con el fin de identificar características relevantes que puedan servir de entrada para el modelo de la aplicación y su posterior simulación.

Tal y como se mencionó en la sección 2.2, una de las principales características por la cual se escogió *CloudStore* como aplicación de referencia, es porque para esta aplicación ya se cuenta con un modelo basado en Palladio. Este modelo fue realizado por los autores de *CloudStore* y se encuentra en el repositorio de código del mismo en GitHub. En este modelo se incluyen experimentos, los cuales son pruebas de carga que el *Palladio Workbench* utiliza para simular el uso que le podrían dar los usuarios.

Para las próximas semanas, se pretende empezar a trabajar con el *Palladio Workbench* y el modelo de *CloudStore*. Ejecutar experimentos/pruebas y evaluar resultados. Luego se agregarán nuevos componentes a este modelo: estos componentes serán los que representen el *middleware* orientado a mensajes y la aplicación consumidora de

mensajes. En esta parte, artículos que se han recopilado sobre el tema y las mediciones hechas servirán como entrada para modelar estos componentes.

Por último, se ejecutarán simulaciones sobre el modelo modificado y se compararán las métricas obtenidas con las que se han obtenido cuando se utilizó JMeter para probar *CloudStore*. En este punto intentará saber si el modelo propuesto es uno que puede describir bien el comportamiento de esta aplicación.

Capítulo 3

Propuesta de contenidos de informe final

Los siguientes son los contenidos que se proponen para el informe final:

1. Resumen (*Abstract*)
2. Introducción: generalidades de los temas que se van a cubrir así como de la estructura del artículo
3. Ingeniería de rendimiento de software:
   a) Ingeniería de rendimiento a través de modelado
   b) Modelado de Arquitecturas de Software con el *Palladio Component Model* (PCM)
4. *Middleware* Orientado a Mensajes: resumen de sección 2.1.14
5. Modelado de rendimiento en una aplicación: *CloudStore*
   a) Generalidades de *CloudStore*
   b) *CloudStore* con *middleware* orientado a mensajes
   c) Modelado de *middleware* orientado a mensajes en *CloudStore*
6. Resultados: dar a conocer los resultados de las simulaciones sobre el modelo de *CloudStore* y si los mismos logran predecir de alguna forma el comportamiento de la aplicación
7. Trabajos Relacionados
8. Conclusión
9. Bibliografía

Bibliografía

[1] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press. 2016.

[2] Bruce Snyder, Dejan Bosnanac, Rob Davies. *ActiveMQ in Action*. Manning, 2011.

[3] S.S. Alwakeel and H.M. Almansour. *Modeling and Performance Evaluation of Message-oriented Middleware with Priority Queuing*. Information Technology Journal, 10: 61-70. 2011. DOI http://dx.doi.org/10.3923/itj.2011.61.70

[4] Chew, Zen Bob. *Modelling Message-oriented-middleware Brokers Using Autoregressive Models for Bottleneck Prediction*. PhD Thesis. Queen Mary, University of London. 2013. https://qmro.qmul.ac.uk/jspui/handle/123456789/8832

[5] Yan Liu and Ian Gorton. *Performance prediction of J2EE applications using messaging protocols*. In Proceedings of the 8th international conference on Component-Based Software Engineering (CBSE'05), George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, and Clemens Szyperski (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-16. 2005. DOI http://ezproxy.itcr.ac.cr:2075/10.1007/11424529_1

[6] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. *Performance engineering for microservices: Research challenges and directions*. In Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, 2017, pages 223-226. DOI: https://doi.org/10.1145/3053600.3053653

[7] Andreas Brunnert, André van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Roman Herbst, Pooyan Jamshidi, Reiner Jung, Jóakim von Kistowski, Anne Koziolek, Johannes Kroß, Simon Spinner, Christian Vögele, Jürgen Walter, and Alexander Wert. *Performance-oriented devops: A research agenda*. Technical Report SPEC-RG-2015-01, SPEC Research Group - DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), 2015. http://arxiv.org/abs/1508.04752

[8] Tomáş Martinec, Lukáş Marek, Antonín Steinhauser, Petr Tůma, Qais Noorshams, Andreas Rentschler, and Ralf Reussner. *Constructing performance model of JMS middleware platform*. In Proceedings of the 5th ACM/SPEC international conference on Performance engineering (ICPE '14). ACM, New York, NY, USA, 123-134. 2014. DOI: https://ezproxy.itcr.ac.cr:2878/10.1145/2568088.2568096

[9] Mark Richards, Richard Monson-Haefel, David Chappell. *Java Message Service*. O'Reilly Media. Segunda Edición. 2009.

[10] Nikolaus Huber, Steffen Becker, Christoph Rathfelder, Jochen Schweflinghaus, and Ralf H. Reussner. 2010. *Performance modeling in industry: a case study on storage virtualization*. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10), Vol. 2. ACM, New York, NY, USA, 1-10. DOI: http://ezproxy.itcr.ac.cr:2075/10.1145/1810295.1810297

[11] Stephen Becker, Heiko Koziolek and Ralf Reussner. *The Palladio component model for model-driven prediction*. Journal of Systems and Software 82:3-22. Elsevier Science Inc. 2009. DOI: http://dx.doi.org/10.1016/j.jss.2008.03.066

[12] Qais Noorshams. *Modeling and Prediction of I/O Performance in Virtualized Environments*. PhD thesis, Karlsruhe Institute of Technology (KIT), 2015. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000046750

[13] Jens Happe, Holger Friedrich, Steffen Becker, and Ralf H. Reussner. *A pattern-based performance completion for Message-oriented Middleware*. In Proceedings of the 7th international workshop on Software and performance (WOSP '08). ACM, New York, NY, USA, 165-176. 2008. DOI: http://ezproxy.itcr.ac.cr:2075/10.1145/1383559.1383581

[14] Heiko Koziolek. 2010. *Performance evaluation of component-based software systems: A survey. Perform.* Eval. 67, 8 (August 2010), 634-658. DOI: http://ezproxy.itcr.ac.cr:2075/10.1016/j.peva.2009.07.007

[15] Thijmen de Gooijer, Anton Jansen, Heiko Koziolek, and Anne Koziolek. 2012. *An industrial case study of performance and cost design space exploration*. In Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12). ACM, New York, NY, USA, 205-216. DOI: https://ezproxy.itcr.ac.cr:2878/10.1145/2188286.2188319

[16] Christoph Rathfelder, Steffen Becker, Klaus Krogmann and Ralf Reussner. *Workload-aware System Monitoring Using Performance Predictions Applied to a Large-scale E-Mail System*. 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, Helsinki, 2012, pp. 31-40. DOI: 10.1109/WICSA-ECSA.212.11.

[17] Misha Strittmatter and Amine Kechaou. *The media store 3 case study system*. Technical Report 2016,1, Faculty of Informatics, Karlsruhe Institute of Technology, 2016. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/3792054

[18] Murray Woodside, Greg Franks, and Dorina C. Petriu. *The Future of Software Performance Engineering*. Future of Software Engineering (FOSE '07), pages 171-187, May 2007. DOI 10.1109/FOSE.2007.32

[19] Thijmen de Gooijer. *Performance Modeling of ASP.Net Web Service Applications: an Industrial Case Study*. Master's thesis, Malardalen University, Vasteras, Sweden, 2011.

[20] Steffen Becker. *Palladio-Bench Screencasts*. http://www.palladio-simulator.com/tools/screencasts/ Accesado el 20 de Marzo del 2018.

[21] Yan Jin, Antony Tang, Jun Han, and Yan Liu. *Performance Evaluation and Prediction for Legacy Information Systems*. 29th International Conference on Software Engineering (ICSE'07), pages 540-549, May 2007.

[22] Simonetta Balsamo, Antinisca DiMarco, Paola Inverardi, and Marta Simeoni. *Model-based performance prediction in software development: A survey*. IEEE Trans. Softw. Eng., 30(5):295–310, May 2004.

[23] Anne Koziolek, Heiko Koziolek, and Ralf Reussner. *PerOpteryx: automated application of tactics in multi-objective software architecture optimization*. In Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS (QoSA-ISARCS '11). ACM, New York, NY, USA, 33-42. 2011. DOI=http://ezproxy.itcr.ac.cr:2075/10.1145/2000259.2000267

[24] Samuel Kounev and Fabian Brosig and Nikolaus Huber. *The Descartes Modeling Language*. Technical Report. Department of Computer Science, University of Wuerzburg. 2014.

[25] Gunnar Brataas, Erlend Stav, Sebastian Lehrig, Steffen Becker, Goran Kopčak, and Darko Huljenic. 2013. *CloudScale: scalability management for cloud systems*. In Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13), Seetharami Seelam (Ed.). ACM, New York, NY, USA, 335-338. 2013. DOI: https://ezproxy.itcr.ac.cr:2878/10.1145/2479871.2479920

[26] Connie U Smith and Lloyd G Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[27] Murray Woodside, Dorin Petriu, and Khalid Siddiqui. *Performance-related Completions for Software Specifications*. In Proceedings of the 24th International Conference on Software Engineering, pages 22–32, New York, NY, USA, 2002.