Comparing scalability of message queue system: ZeroMQ vs RabbitMQ

Nicolás Estrada
Departamento de Informática
Universidad Técnica Federico Santa María
Santiago, Chile
Email: nicolas.estrada@alumnos.usm.cl

Hernán Astudillo
Departamento de Informática
Universidad Técnica Federico Santa María
Santiago, Chile
Email: hernan@inf.utfsm.cl

***Abstract*—Modern web apps handle huge and increasing numbers of users and operations. A rise of event-driven architecture approach and message queue systems provide a new alternative to face this scenario evidencing the lack of quantitative measurements comparing performance and scalability between specific message queue products. This article proposes a prototype architecture applied in ZeroMQ and RabbitMQ, used for measure the impact of (1) the number of messages over performance, and (2) the numbers of consuming nodes over scalability. The results show that for both criteria, the degradation threshold of ZeroMQ is higher than RabbitMQ, thus more scalable and faster.**

Keywords: Event-Driven architecture, Business Integration, Distributed Systems, Message Queue, Broker.

I. Introducción

En un mundo globalizado como el de hoy, las plataformas de comunicación y los sistemas de información se ven enfrentados a un gran desafío, proveer datos en todo momento, en todo lugar y de forma segura, consistente y confiable. Es más, muchas aplicaciones sociales o de control de procesos críticos requieren que todo esto sea en tiempo real e instantáneo, para así satisfacer la experiencia de usuario o reaccionar ante alarmas o niveles críticos en la industria. Existen varias propuestas arquitectónicas para enfrentar estos desafíos, una de las más populares es SOA (arquitectura orientada a servicios) que desacopla la capa de datos con la de clientes mediante un conjunto definido de servicios (llamadas remotas) que manejan las reglas del negocio [6].

Últimamente se ha puesto énfasis en arquitecturas orientadas a eventos (*EDA* por su sigla en inglés) incorporando llamadas y procesos asíncronos que reaccionan ante un flujo de datos entrantes [6]. Este documento pretende profundizar en este tema y evaluar tecnologías de Colas de Mensajes como eje principal.

A continuación, en la Sección III, se detalla la problemática a enfrentar y trabajos relacionados a ella, para después en la Sección IV definir la propuesta y pruebas a realizar. Finalmente, los resultados se detallan en la Sección V y las conclusiones en la Sección VI.

II. Contexto

En los sistemas orientados a eventos hay un productor quien es capaz de detectar y notificar de eventos relacionados con el negocio, así como también existe la contraparte, el consumidor cuyo propósito es recibir estos sucesos. Además se describen cinco principios de *EDA* [4]:

1) **Individualidad**: cada evento se transmite individualmente, los productores no permiten acumulación de eventos o envío por lote.
2) ***Push***: las notificaciones enviadas son recibidas por los consumidores, a diferencia de otros sistemas *request-driven* que utilizan un esquema de *polling* en el cual cada consumidor solicita los eventos periódicamente.
3) **Inmediatez**: el consumidor procesa el evento una vez que lo recibe, ya sea simple procesamiento o guardarlo para ser utilizado más tarde (Procesamiento de Eventos Complejos).
4) ***One-way***: la comunicación en EDA es *dispara y olvida*, es decir, el productor envía la notificación y no espera respuesta.
5) **Libre de comandos**: los eventos no contienen instrucciones ni comandos a ejecutar, en este caso es el consumidor es el que posee la lógica de cómo procesará el evento.

Por otro lado, sistemas EDA son comúnmente implementados con mecanismo de comunicación *publish-and-subscribe* permitiendo a los productores enviar el mensaje una sola vez y que uno o más consumidores lo reciban. Esta característica es considerada opcional [4] y además, los principios descritos anteriormente son muy similares a al patrón de diseño que propone por Hohpe [9]: *Broadcast communication* (comunicación uno a muchos), *Timeliness* (eventos son emitidos a medida que ocurren y no almacenados para su posterior proceso), *Asynchrony* (productores envían sin esperar respuesta), *Fine Grained* (se reciben eventos simples e individuales), *Ontology* (subscripción e interés por eventos) y *Complex Event Processing* (procesamiento de eventos complejos, una de las posible alternativas es realizarlo a través de detección de patrones).

Hohpe [9] ha caracterizado patrones basados en mensaje dentro de los cuales el Enrutamiento y la Transformación de mensajes pueden ser reutilizados en EDA, incluso los patrones de procesamiento descritos por Hohpe, tales como *Pipes and Filter*, *Message Filter*, *Aggregator* y *Content Enrichment* son utilizados en EDA como pilares fundamentales en Procesamiento de Eventos Complejos [3] (CEP por su sigla en inglés).

978-1-4673-9143-6/15/$31.00 ©2015 IEEE

Bruns and Dunkel [3] proponen patrones de arquitectura y patrones de diseño enfocados en el procesamiento de eventos (EDA y CEP). Los patrones de arquitectura que describe para EDA son:

- **Capas**: considera 3 niveles, monitoreo, procesamiento y manejo de eventos.
- **Agentes**: cada procesador de eventos es un agente individual (EPA por su sigla en inglés).
- **Segmentación**: en este patrón los eventos se procesan en serie pasando por una cadena definida de consumidores y/o productores.

Por otro lado, los patrones de diseño que proponen en el mismo artículo [3] son:

- **Consistencia de Eventos**: el cual propone un paso de limpieza inicial de los eventos entrantes.
- **Reducción de Eventos**: se minimiza la cantidad de eventos mediante 3 mecanismos propuestos: Filtrado (dejando afuera eventos no necesarios), Enrutamiento Basado en Contenido (consumidores sólo reciben eventos en los cuales están interesados), y por último Ventanas de Tiempo Deslizables donde se considera un rango de tiempo de eventos a analizar
- **Transformación de Eventos**: tiene 2 aristas: Traducción, que implica unificación de formatos para que los eventos sean compatibles en el sistema y Enriquecimiento de Contenido, que sugiere la agregación de contenido para uso futuro accediendo a bases de datos.
- **Síntesis de Eventos** que considera Correlación por dominio, por tiempo o ubicación, que permite Cambios de Granularidad de eventos (agregación) y Cambio de significado (basado en patrones de eventos).

Uno de los propósitos de esta investigación es probar las capacidades intrínsecas de sistemas de cola de mensaje como base para la construcción de un sistema orientado a eventos, considerando que muchos de los principios de mensajería son aplicables o son la base de sistemas orientados a eventos. A continuación en la Sección III se describe el problema y la propuesta del presente estudio.

III. Problema y trabajos relacionados

Dunkel et al[5], proponen una arquitectura orientada a eventos para un sistema de control de tráfico que apoya la toma de decisiones, argumentando que arquitecturas actuales no están diseñadas para recibir un flujo continuo de datos y que incluso, *SOA* no es apropiado para sistemas orientados a eventos, debido al alto flujo de estos y sus dependencias complejas.

En la Tabla I se muestra una comparación entre ambos patrones de arquitectura (EDA versus SOA). Dentro de los aspectos a considerar, la comunicación asíncrona y el patrón *Publish-and-Suscribe* son los principales argumentos en los que se basa este documento para abordar el problema de procesamiento de datos masivos, es decir, tratar cada mensaje como un evento.

Tabla I: Comparativa entre EDA y SOA

	EDA	**SOA**
Enfoque	Eventos, enrutamiento de mensajes	Descomposición, capas separadas y poco acopladas
Patrón	Publish-and-Subscribe	Request-and-Reply
Comunicación	Asíncrono	Síncrono
En la práctica	Provee flexibilidad a la organización, adaptable	La organización se adapta a los componentes diseñados
Trascendencia	Nuevo, poca literatura e investigación, un explosivo interés a partir del 2011	Maduro, mucha literatura e investigación
Definiciones	Sólo se comparte la semántica de cómo se envían los mensajes	Se define para cada servicio un estándar, estados y requerimientos
Aplicable a	Interacción horizontal, ideal para integrar múltiples sistemas o mantener procesos autónomos	Interacción vertical, cohesión fuerte en los procesos
Implementación	Utilizando colas de mensajes	Sobre protocolo HTTP

Un sistema orientado a eventos, debe ser capaz de reaccionar ante una secuencia de eventos e incluso generarlos. Las capacidades intrínsecas de filtrado, enrutamiento y procesamiento de una arquitectura, facilitan o resultan ser idóneas para situaciones en que se reciben altos volúmenes de eventos, como por ejemplo: Dunkel et al [5] proponen una arquitectura de referencia para un sistema de control de tráfico; Wan et al [13] proponen una arquitectura orientada a eventos para comunicación máquina a máquina *M2M* y llevan a cabo un caso de estudio de control de vehículos; Li [10] realiza un estudio de los sistemas orientados a eventos y *middleware*, reflexionando la necesidad de que se desarrollen nuevas arquitecturas y *middleware* basado en eventos.

El *Middleware* en general puede ser definido como *software* diseñado para construir sistemas distribuidos a gran escala [7]. También permite conectar a múltiples procesos corriendo en varias máquinas a través de la red [2]. La mensajería consiste en una comunicación *peer-to-peer* entre aplicaciones [7]. Los sistemas de colas de mensajes cuentan con un agente externo al cual los clientes están conectados, por el cual pueden enviar mensajes a cualquier cliente conectado o también recibirlos [2]. Sistemas distribuidos pueden comunicarse de dos formas:

De forma síncrona donde las aplicaciones envían mensajes a otros y esperan su respuesta. Por otro lado, en la comunicación asíncrona las aplicaciones envían mensajes y continúan procasándolos sin esperar la respuesta [8].

Estos sistemas de mensajería usualmente soportan 2 tipos de patrones de mensajes: punto-a-punto (*p2p*) y Emisor-Subscriptor (*pub/sub*). Los dos estilos de mensajería son arquitecturas centralizadas y distribuidas. En arquitecturas centralizadas, todos los procesos se comunican con un servidor común. Por otro lado, en arquitecturas distribuidas, los procesos se comunican con componentes locales de mensajería, los que a su vez se comunican a través de la red para entregar los mensajes en lugar de los emisores y receptores [12].

A. Propuesta y tecnologías utilizadas

Se han propuesto varias alternativas que consideran *EDA* como alternativa para flujo de eventos, sin embargo, en el ámbito de integración a escala organizacional, las herramientas existentes son por lo general, propietarias o difíciles de implementar a medida. Es por esto, que se pretende profundizar en

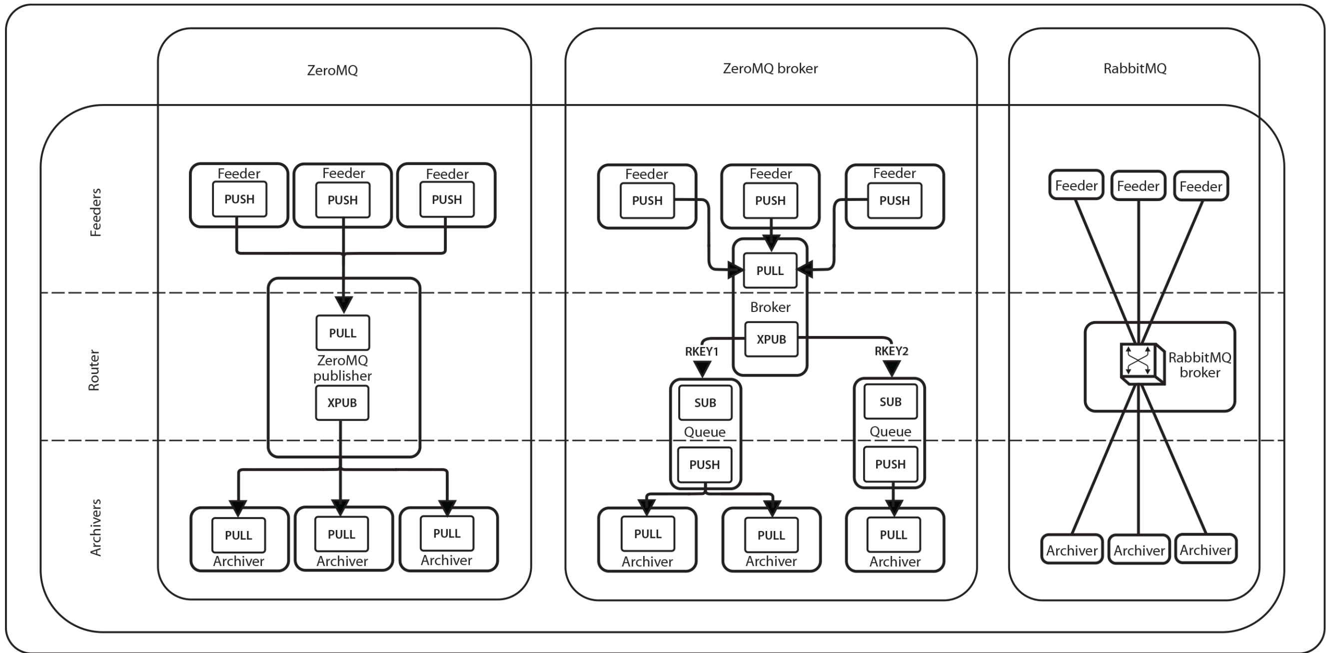

Fig. 1: Casos de estudio implementados

EDA como alternativa de integración y sentar las bases para la construcción de una plataforma que facilite la integración de sistemas. Para ello se propone estudiar tecnologías existentes.

De esta forma, en la búsqueda de desarrollar una plataforma de integración utilizando *EDA*, se compara un *broker* como RabbitMQ con una librería de *sockets* y colas de mensajes, ZeroMQ. Este trabajo tiene el objetivo de elegir una tecnología apropiada para el envío de mensajes (con filtrado y subscripción) para una plataforma de integración.

1) RabbitMQ: RabbitMQ [1] es un *broker* de mensajes - un intermediario para mensajería. Entrega una plataforma común donde las aplicaciones pueden enviar y recibir mensajes.

RabbitMQ ofrece persistencia de mensaje, confirmación de entrega de mensajes (desde el receptor), confirmación de envío (desde el emisor) y alta disponibilidad. También ofrece enrutamiento flexible a través de un intercambiador que envía los mensajes a las colas que están conectadas según tipo de mensaje, tópico o filtro.

2) ZeroMQ: ZeroMQ [2] es una librería de *sockets* para mensajería que implementa varios patrones de conexión. Es multiplataforma y soporta la mayoría de los lenguajes.

Por otro lado, soporta patrones de mensajería como: *pub-sub*, *push-pull* y *router-dealer* a una alta velocidad en operaciones I/O asíncronas.

Al ser una librería liviana, es fácil de implementar usando el lenguaje preferido.

[1] http://www.rabbitmq.com
[2] http://www.zeromq.org

IV. Propuesta y experimentos

A. Pruebas de Rendimiento

La prueba consiste en productores que envían mensajes a un servidor (*broker*) que es capaz de dirigir estos a consumidores que tenga registrado utilizando un *routing key* (llave que identifica el tipo de mensaje). Finalmente, los consumidores almacenan en disco una copia del mensaje completo y otra copia comprimida.

B. Modelos a evaluar

Se proponen *brokers* para cada una de las tecnologías a estudiar. En la Figura 1 se distinguen las propuestas.

ZeroMQ broker: Se implementa un *broker* como se aprecia en la figura. Los productores de mensajes envían a través de un *socket PUSH* recibidos por el *broker* utilizando el *socket PULL*. Para el filtrado de mensajes a través de *routing key* se implementa una cola por cada tipo de subscripción. Cada cola se registra para una llave específica utilizando *sockets XPUB* y *SUB*, es decir, todos los mensajes que vengan con ese identificador, van a ser recibidos por la cola. De esta manera, los consumidores se conectan a la cola indistintamente según el *routing key* que desean recibir usando *sockets PUSH* y *PULL* respectivamente, todo esto bajo un patrón de ordenamiento secuencial (*Round-Robin*) en caso de que haya más de un consumidor conectado.

RabbitMQ: Se configura RabbitMQ para el envío de mensajes no persistentes, así como la cola y el *exchange* son temporales. Esto se realiza para efectos de disminuir la sobrecarga de RabbitMQ sobre ZeroMQ en aspectos de rendimiento (buscando igualdad de condiciones), para lo cual

también se utiliza *auto acknowledge*, para enviar mensajes sin esperar confirmación.

C. Parámetros y entornos de prueba

En la Tabla II se observan las máquinas empleadas para las pruebas de rendimiento.

Tabla II: Especificación de Máquinas utilizadas

Característica	*Feeders*	*Broker*	*Archivers*
Sist. Operativo	Centos 6.4 64bit		
CPU	Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz		
Núcleos	8	1	8
RAM	4 GB	4 GB	4 GB
Almacenamiento	200 GB	12 GB	200 GB

En la Figura 2 se observa la topología de los servidores, hay un servidor que funciona como *broker*, otro que funciona como *feeder* (productor) que puede ejecutar varios procesos y de la misma forma el servidor que contiene los *archivers* (consumidor).

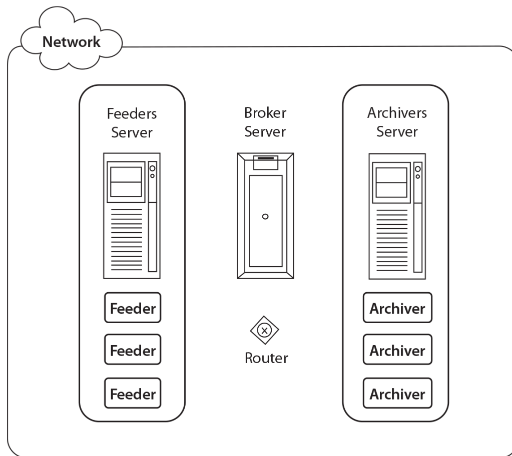

Fig. 2: Diagrama de servidores

A continuación se definen los parámetros y pruebas a realizar.

D. Pruebas de I/O

En primer lugar se define una prueba de rendimiento con la intención de obtener el límite máximo de operaciones sobre disco de cada proceso consumidor.

1) Determinando el límite de operaciones sobre disco por consumidor: En este caso se utiliza la implementación simple de ZeroMQ con 1 productor y 1 consumidor. Se limita el envío de mensajes por intervalos para obtener un punto de comparación preliminar entre tecnologías: 100, 500, 1.000, 2.500, 5.000, 7.500, 10.000, 20.000, 50.000 (mensajes/segundo) y sin límite.

E. Pruebas de rendimiento de los sistemas de mensajería

Una vez determinado el límite de I/O de un consumidor, se realizan pruebas de desempeño de ambas tecnologías para estudiar como se comportan procesando una gran cantidad de mensajes. Para esto se definen dos experimentos.

1) Determinando el mejor desempeño según cantidad de consumidores: Variar la cantidad bajo el límite óptimo encontrado y sin límites, para así determinar el número óptimo de consumidores. Se fijan pruebas para: 1, 2, 3, 5 y 10 consumidores.

2) Determinando el rendimiento del broker con múltiples consumidores: Para determinar el desempeño del *broker*, se utiliza la cantidad de consumidores que produce el mejor desempeño, variando los límites: 100, 500, 1.000, 2.500, 5.000, 7.500, 10.000, 20.000, 50.000 y 100.000 (mensajes/segundo).

F. Limitaciones y consideraciones

La propuesta y los experimentos descritos considera uno de los dos criterios de escalabilidad descritos por Abbott y Fisher [1], esto es el *throughput* (mensajes procesados por unidad de tiempo) dejando de lado la latencia.

Por otro lado, el presente estudio además de entregar resultados, entrega una implementación de los casos de estudio, lo que permite implementarlos bajo otros ambientes o utilizando otras tecnologías lo que sirve como referencia para sistemas de mensajería y arquitecturas orientadas a eventos.

V. Resultados

Se implementaron los casos de prueba [3], a continuación se pueden observar los resultados obtenidos.

1) Determinando el límite de operaciones sobre disco por consumidor: Para ambas tecnologías, se corrieron las pruebas descritas en la Sección IV-D1. Los resultados obtenidos se pueden observar en la Figura 3. Se utilizó un productor, un *broker* y un consumidor para determinar el límite de envío de mensajes por segundo según la implementación y *hardware* utilizado. El gráfico en cuestión es un gráfico logarítmico, donde se observa para ambas tecnologías el mismo tope en cuanto a rendimiento, debido a que se trata del máximo de operaciones en disco que puede realizar cada proceso consumidor.

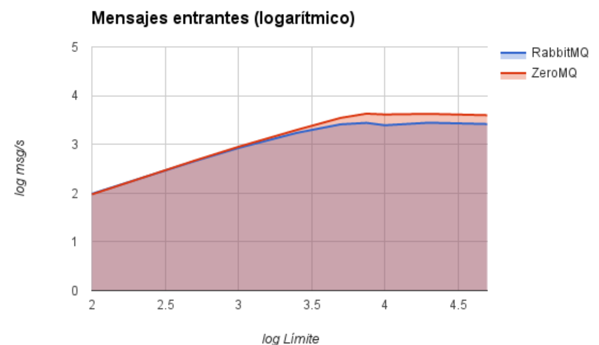

Fig. 3: Límite de operaciones en disco por comsumidor

[3] https://bitbucket.org/nestrada/zmq_rmq_eda/(branchrmq-zmq/develop_zmq-broker)

Para este primer modelo implementado, es decir, ZeroMQ (ver Figura 1) se determinó el límite de operaciones en disco por consumidor. Si se observa la Figura 3, el límite cuando las operaciones por disco se estancan es prácticamente el mismo para ambas tecnologías (límite 10.000 mensajes enviados por segundo), sin embargo, ZeroMQ procesa mayor cantidad de mensajes por segundo, aproximadamente un tercio de orden de magnitud. Esto significa que ante un flujo entrante de mensajes, ZeroMQ es más rápido procesándolos.

2) Determinando el mejor desempeño según cantidad de consumidores: Una vez determinado el límite de operaciones por proceso consumidor, se utiliza el óptimo para realizar pruebas con varios consumidores. En este caso se utiliza un productor, un *broker* y como límite 20.000 mensajes por segundo, variando la cantidad de consumidores como se define en la Sección IV-E1. En la Figura 4 se observan los resultados obtenidos. En el caso de ZeroMQ, el rendimiento (msg/s) alcanzado es superior al obtenido por RabbitMQ, este último después de 3 consumidores no mejora el desempeño a diferencia de ZeroMQ que sigue aumentando.

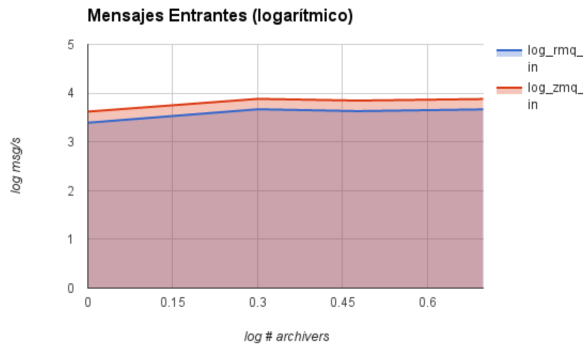

Fig. 4: Gráfico de rendimiento con flujo fijo según cantidad de consumidores

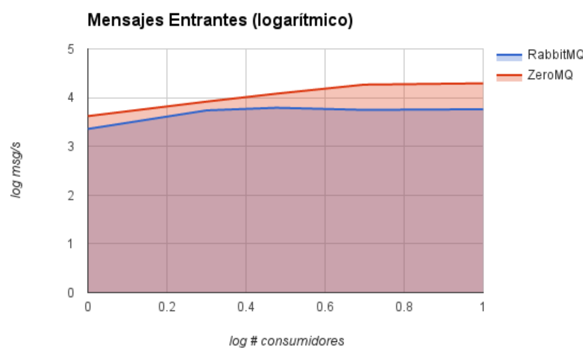

Fig. 5: Gráfico de rendimiento según cantidad de consumidores

Así mismo, con el rendimiento óptimo obtenido anteriormente, se realiza una prueba de escalabilidad similar, variando la cantidad de consumidores pero sin limitar la cantidad de mensajes enviados. En este caso se compara el modelo ZeroMQ broker versus RabbitMQ. En la Figura 5, ZeroMQ tienen un mayor desempeño y se estanca a partir de los 5 consumidores, en cambio RabbitMQ se estanca a partir de los 3 consumidores.

Adicionalmente, en la Figura 4 se observa que ambas alternativas se comportan de forma similar. Sin embargo, en este caso hay que considerar que el flujo entrante de mensajes no sobrepasa la capacidad de los consumidores ni la del broker, al contrario de lo que ocurre en la Figura 5 donde el rendimiento se degrada en puntos distintos para ambas implementaciones.

3) Determinando el rendimiento del broker con múltiples consumidores: Con las pruebas definidas en la Sección IV-E2 y utilizando los resultados anteriores, se pone a prueba el *broker*. Se utiliza un productor, un *broker* y tres consumidores, variando los límites de mensajes a enviar por segundo.
Los resultados obtenidos se observan en la Figura 6 donde se puede observar que el rendimiento utilizando ZeroMQ es medio orden de magnitud superior al logrado por RabbitMQ.

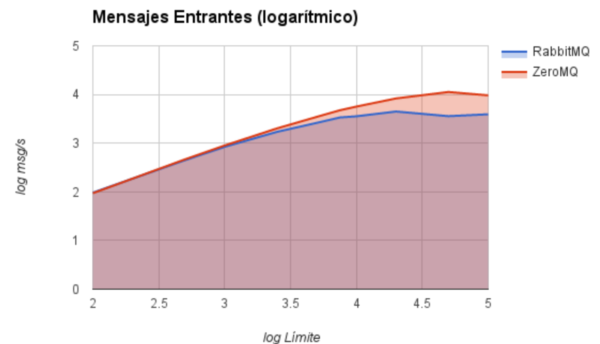

Fig. 6: Gráfico de rendimiento del *broker* con 3 consumidores

Finalmente, con 3 consumidores, se observa el rendimiento frente a un flujo de eventos creciente. Es así como en la Figura 6 se observa como el umbral de degradamiento para ZeroMQ es medio orden de magnitud mayor al de RabbitMQ, lo que se traduce en que ante la misma configuración y carga, el modelo ZeroMQ broker llega a procesar 11.000 mensajes por segundos y RabbitMQ 4.500. Esto se puede observar en el gráfico no logarítmico correspondiente a la Figura 7.

VI. Conclusiones

Ambas tecnologías tienen un rendimiento alto, son capaces de procesar miles de mensajes por segundo y cuentan con la capacidad de filtrado de mensajes. Se pueden configurar distintas topologías, de forma más personalizable en el caso de ZeroMQ.

RabbitMQ tiene filtrado y enrutamiento de mensajes incorporado, por otro lado, para ZeroMQ se tuvo que implementar una cola intermedia para filtrar y dirigir mensajes.

Las pruebas para medir los límites de operaciones sobre disco resultaron iguales para ambas tecnologías (con el mismo límite), lo cual es lógico, ya que ambos corrieron sobre el mismo ambiente de desarrollo.

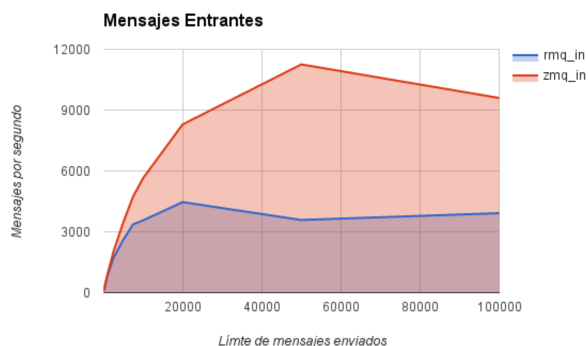

Fig. 7: Gráfico de rendimiento del *broker* con 3 consumidores no logarítmico

El umbral de degradamiento de ZeroMQ es medio orden de magnitud más alto que el de RabbitMQ en ambos casos (rendimiento y escalabilidad), lo que implica que es más escalable y rápido.

Tomando aspectos de implementación, ZeroMQ permite construir soluciones más a medida, mientras que RabbitMQ resulta ser una caja negra por sobre la cual operar.

Como trabajo futuro se propone realizar pruebas de rendimiento para el caso descrito por Sun et al [11] y por otro lado, implementar un caso de estudio con una red de sensores en tiempo real con un flujo de eventos numerosos para así ver que tan idóneo es un sistema de cola de mensajes como tecnología orientada a eventos y qué tan capaz es para Procesamiento de Eventos Complejos en búsqueda de patrones, alerta temprana e integración.

VII. Agradecimientos

Este trabajo fue parcialmente financiado por Fondecyt (Proyecto Basal CCTVal FB0821) y UTFSM (proyecto DGIP 24.15.74).

Bibliografía

[1] Martin L. Abbott and Michael T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. 1st ed. Addison-Wesley Professional, Dec. 2009. ISBN: 0137030428. URL: http://www.worldcat.org/isbn/0137030428.

[2] Philip A. Bernstein. "Middleware: A Model for Distributed System Services". In: *Commun. ACM* 39.2 (Feb. 1996), pp. 86–98. ISSN: 0001-0782. DOI: 10.1145/230798.230809. URL: http://doi.acm.org/10.1145/230798.230809.

[3] Ralf Bruns and Jürgen Dunkel. "Towards pattern-based architectures for event processing systems". In: *Software: Practice and Experience* (2013), n/a–n/a. ISSN: 1097-024X. DOI: 10.1002/spe.2204.

[4] K. Mani Chandy and W. Roy Schulte. *Event Processing - Designing IT Systems for Agile Companies*. McGraw-Hill, 2010, pp. I–XVII, 1–235. ISBN: 978-0-07-163350-5.

[5] Jürgen Dunkel et al. "Event-driven architecture for decision support in traffic management systems." In: *Expert Syst. Appl.* 38.6 (2011), pp. 6530–6539. URL: http://dblp.uni-trier.de/db/journals/eswa/eswa38.html#DunkelFOO11.

[6] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321127420.

[7] Sushant Goel, Hema Sharda, and David Taniar. "Message-Oriented-Middleware in a Distributed Environment". English. In: *Innovative Internet Community Systems*. Ed. by Thomas Böhme, Gerhard Heyer, and Herwig Unger. Vol. 2877. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 93–103. ISBN: 978-3-540-20436-7. DOI: 10.1007/978-3-540-39884-4_8. URL: http://dx.doi.org/10.1007/978-3-540-39884-4_8.

[8] Mark Haner. *The Java Message Service 1.0.2b*. 2001.

[9] Gregor Hohpe. *Programming Without a Call Stack – Event-driven Architectures*. 2006. URL: http://www.eaipatterns.com/docs/EDA.pdf.

[10] Chung-Sheng Li. "Real-time event driven architecture for activity monitoring and early warning". In: *Emerging Information Technology Conference, 2005*. 2005, 4 pp.–. DOI: 10.1109/EITC.2005.1544382.

[11] Dou Sun et al. "SEDA4BPEL: A staged event-driven architecture for high-concurrency BPEL engine". In: *Computers and Communications (ISCC), 2010 IEEE Symposium on*. 2010, pp. 744–749. DOI: 10.1109/ISCC.2010.5546728.

[12] Stefan Tai, ThomasA. Mikalsen, and Isabelle Rouvellou. "Using Message-Oriented Middleware for Reliable Web Services Messaging". English. In: *Web Services, E-Business, and the Semantic Web*. Ed. by ChristophJ. Bussler et al. Vol. 3095. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 89–104. ISBN: 978-3-540-22396-2. DOI: 10.1007/978-3-540-25982-4_9. URL: http://dx.doi.org/10.1007/978-3-540-25982-4_9.

[13] Jiafu Wan et al. "M2M Communications for Smart City: An Event-Based Architecture". In: *Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on*. 2012, pp. 895–900. DOI: 10.1109/CIT.2012.188.