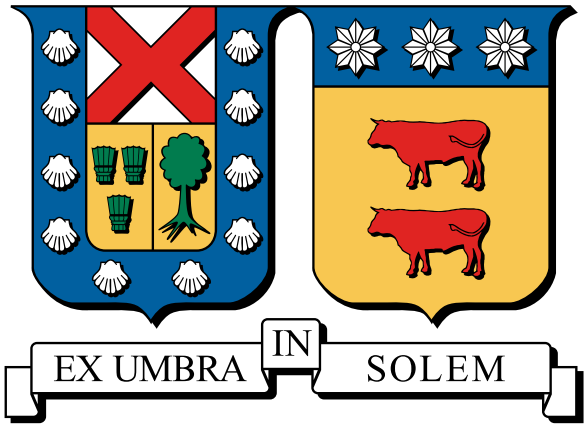
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INFORMÁTICA

SANTIAGO – CHILE



"ARQUITECTURA ORIENTADA A EVENTOS PARA REDES DE SENSORES APLICADA A CONTROL DE TRÁFICO"

NICOLÁS ALEJANDRO ESTRADA IRRIBARRA

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS DE LA INGENIERÍA EN INFORMÁTICA

PROFESOR GUÍA: **DR. HERNÁN ASTUDILLO**
PROFESOR CORREFERENTE: **DR. RAÚL MONGE**

Enero – 2017

TITULO DE LA TESIS:

ARQUITECTURA ORIENTADA A EVENTOS PARA REDES DE SENSORES APLICADA A CONTROL DE TRÁFICO

AUTOR:

NICOLÁS ALEJANDRO ESTRADA IRRIBARRA

TRABAJO DE TESIS, presentado en cumplimiento parcial de los requisitos para el Grado de Magíster en Ciencias de la ingeniería Informática de la Universidad Técnica Federico Santa María.

Dr. Hernán Astudillo
_____
Director de Tesis

Dr. Raúl Monge
_____
Co-referente Interno

Dr. Fabio Kon
_____
Co-referente Externo

Dr. Marcelo Mendoza
_____
Presidente Comisión Examen

Resumen

Las arquitecturas dirigidas por eventos (EDA) comenzaron a desarrollarse a principios de este siglo y tomaron fuerza este último tiempo. En la búsqueda de una nueva propuesta en el ámbito de procesamiento de eventos complejos (CEP), este documento presenta una prueba de escalabilidad entre una arquitectura orientada a eventos y una tradicional utilizando datos sintéticos.

Finalmente, se diseñó e implementó un prototipo orientado a eventos utilizando datos reales bajo una simulación *trace-driven* con el fin de procesar eventos en tiempo real y reconocer patrones complejos generando alarmas ante anomalías detectadas.

Del presente estudio, se obtuvo que EDA se comporta de mejor forma que una arquitectura tradicional para sistemas basados en sensores. Además, el prototipo implementado para CEP fue capaz de procesar eventos y detectar anomalías en tiempo real sin acceder a una base de datos históricos.

Abstract

Modern web applications and platforms needs to provide data in real-time basis considering multiple sources, clients, locations and factors like service disruption and malicious attacks. Several systems process huge amounts of events occurring in the real life, events that needs to be processed to detect anomalies, patterns and key business logic to react under unfavorable circumstances. Lately, there is an increasing interest for event-driven architectures (EDA) that includes asynchronous calls and processes that reacts under a data stream.

Looking for a new approach for complex event processing (CEP), this document presents results of performance and scalability tests comparing EDA to traditional architectures. Finally, an event-driven prototype was implemented using a real dataset under a trace-driven simulation to accomplish real-time event processng and detecting complex patterns generating alarm notifications for anomalies.

Thus, the results shows that EDA performs better than traditional architectures for sensor-based systems. Also, the implemented CEP's prototype was able to process and detect anomalies in real-time without historical database access.

Publicaciones relacionadas

- *Comparing scalability of message queue system: ZeroMQ vs RabbitMQ* [10], N. Estrada y H. Astudillo, XLI Conferencia Latinoamericana de Informática, Simposio Latinoamericano de Sistemas de Información de Gran Escala (SLSIGE). Arequipa, Perú, Octubre del 2015. Aceptado, publicado y presentado.

Contenido

Lista de Figuras

Lista de Tablas

Lista de Algoritmos

Capítulo 1

Introducción

1.1 Contexto

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 entregar una buena 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 maneja las reglas del negocio [11]. Sin embargo, 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.

En este trabajo, se diseña una arquitectura de referencia para sistemas orientados a eventos basados en sensores y un prototipo funcional implementado sobre un caso de prueba con datos reales de un sistema de control de tráfico en Santiago.

En el Capítulo 2 se detalla el estado del arte referente a soluciones actuales y arquitecturas orientadas a eventos, para luego en el Capítulo 5 mostrar experimentos previos a la implementación con el objetivo de comparar tecnologías de mensajería para su posterior

utilización dando paso al Capítulo 5 donde se presenta la propuesta a implementar junto con la arquitectura de referencia. Con la arquitectura propuesta se implementa un prototipo aplicado a un sistema basado en una red de sensores para el control de tráfico que se observa en el Capítulo 6. Finalmente, los resultados y conclusiones son presentados en el Capítulo 7 y 8 respectivamente.

1.2 Objetivos

A continuación se describen los objetivos del presente trabajo.

1.2.1 Objetivo General

1. Diseñar y validar una arquitectura distribuida orientada a eventos aplicable a Ciudades Inteligentes [7], es decir, que considere un flujo de datos desde múltiples fuentes y sea capaz de procesar eventos complejos

1.2.2 Objetivos Específicos

1. Implementar un prototipo funcional de una arquitectura orientada a eventos
2. Comparar escalabilidad de una arquitectura orientada a eventos versus una tradicional
3. Obtener un conjunto de datos empírico para el control de tráfico (*Trace-Driven Simulation*)
4. Evaluar capacidad de arquitectura propuesta para identificar patrones de eventos complejos

1.3 Hipótesis

Con la arquitectura de referencia detallada en la Sección 4.1 se plantean dos hipótesis:

1. Una arquitectura orientada a eventos tiene un mejor desempeño en cuanto a criterios de escalabilidad para un sistema enmarcado en Redes de Sensores. En este caso, el rendimiento de la arquitectura propuesta (eventos procesados por unidad de tiempo) será significativamente más alto (al menos un 20%) al aumentar la cantidad de eventos y sensores en el sistema respecto de un enfoque centralizado y jerárquico.

2. Utilizando una arquitectura orientada a eventos es posible procesar eventos complejos en tiempo real sin acceder a un repositorio de datos.

Capítulo 2

Arquitectura Orientada a Eventos

Una Arquitectura Orientada a Eventos (EDA por sus siglas en inglés, *Event-driven architecture*) es un patrón de arquitectura de *software* diseñado para producir, consumir, detectar y reaccionar ante un flujo de eventos. Generalmente se trata de emisores y receptores de eventos en el cual cada consumidor debe reaccionar cuando recibe un evento, ya sea ejecutando procedimientos específicos o procesando y generando un nuevo evento [17].

En un enfoque más de procesos, un productor es capaz de detectar y de notificar eventos relacionados con las reglas de negocio, así como también existe su homólogo, el consumidor, cuyo propósito es recibir estos sucesos. Se describen cinco principios en EDA [6]:

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 el consumidor es el que posee la lógica de cómo se procesará el evento.

Por otro lado, los sistemas orientados a eventos son comúnmente implementados con el 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 por Chandy y Schulte [6] y además, los principios descritos anteriormente son muy similares a los del patrón de mensajería *Message Routing* que propone Hohpe [16] los cuales que se enumeran a continuación:

1. ***Broadcast communication***: comunicación de uno a muchos.
2. ***Timeliness***: los eventos son emitidos a medida que ocurren y no almacenados para su posterior proceso).
3. ***Asynchrony***: los productores envían sin esperar que éstos sean procesados por quien los recibe.
4. ***Fine Grained***: los eventos que se reciben son simples e individuales.
5. ***Ontology***: existe una nomenclatura por la cual los receptores se subscriben y expresan su interés por una parte o totalidad de los eventos.
6. ***Complex Event Processing***: Procesamiento de Eventos Complejos (CEP por sus siglas en inglés) es la fase posterior a la recepción de eventos individuales que recae en la posibilidad de procesarlos y generar a su vez eventos con un nivel de complejidad más, los cuales corresponden a eventos con información agregada. Dentro de los mecanismos de CEP se encuentra la detección de patrones y las ventanas deslizantes.

Además, Hohpe [16] 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 [4].

Bruns y Dunkel [4] proponen patrones de arquitectura y patrones de diseño enfocados en el procesamiento de eventos (EDA y CEP). Los patrones de arquitectura que describen 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.

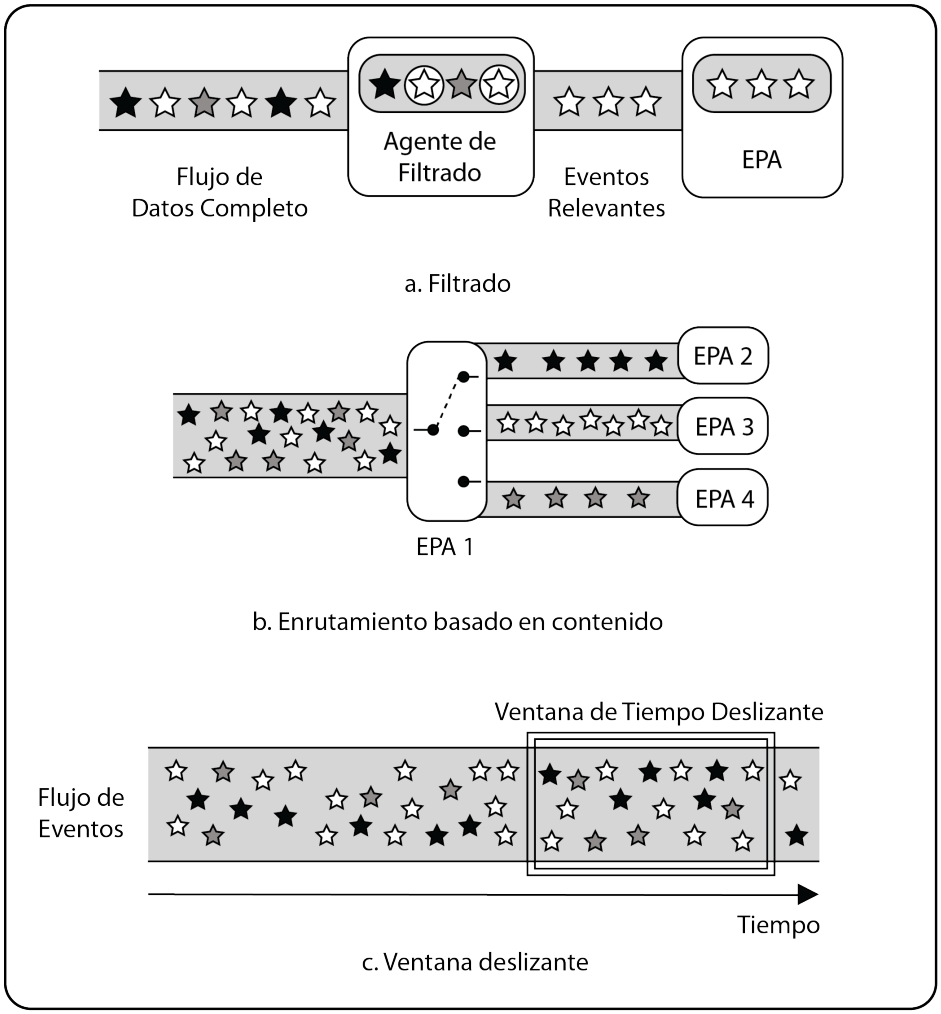


Figura 2.1: Patrón de diseño para reducción de eventos propuesto por Bruns y Dunkel [4]

Bruns y Dunkel [4] también proponen patrones de diseño orientados a eventos:

- **Consistencia de Eventos**: el cual propone un paso de limpieza inicial de los eventos entrantes.
- **Reducción de Eventos**: Para lidiar con la gran cantidad de eventos que reciben hay varios patrones de diseño, dentro de los cuales los autores [4] proponen 3 mecanismos que se pueden observar en la Figura 2.1:

  1. *Filtrado*: en la etapa de filtrado, todos los eventos que no son necesarios se remueven del flujo de datos entrante. Sólo los eventos relevantes pasan a la siguiente fase. Generalmente los criterios de filtrado son simples.
  2. *Enrutamiento Basado en Contenido*: según el contenido del evento se va despachando y dividiendo el flujo entrante de datos en diferentes flujos más pequeños dirigidos a aquellos agentes interesados en un cierto tipo de eventos (basado en información que contiene). Con este mecanismo, la cantidad de eventos no se reduce, pero cada agente procesador sólo recibe lo que necesita y no es necesario que procese todo el flujo de datos que llega.
  3. *Ventanas de Tiempo Deslizantes*: se utiliza una foto instantánea de un espacio temporal en el cual se procesa los eventos más recientes en búsqueda de patrones o datos relevantes. Se trata de una partición del flujo entrante basado en distancia temporal, donde básicamente se considera un rango de tiempo de eventos a analizar.

- **Transformación de Eventos** que se pueden observar en la Figura 2.2:

  1. *Traducción*: implica unificación de formatos para que los eventos sean compatibles en el sistema, lo cual puede ser usado para transformar los datos en un formato específico utilizado por un agente procesador o para ser serializados/deserializados y enviados a través de la red.
  2. *Enriquecimiento de Contenido*: sugiere la agregación de contenido para uso futuro reduciendo el acceso a bases de datos, agregando datos en una etapa temprana para reducir futuras consultas. Esto requiere mayor espacio de almacenamiento y agregar campos al evento.

- **Síntesis de Eventos**: considera Correlación por dominio, por tiempo o ubicación, lo que permite Cambios de Granularidad de eventos (agregación) y Cambios de significado (basado en patrones de eventos). Este patrón de diseño se puede observar en la Figura 2.3.

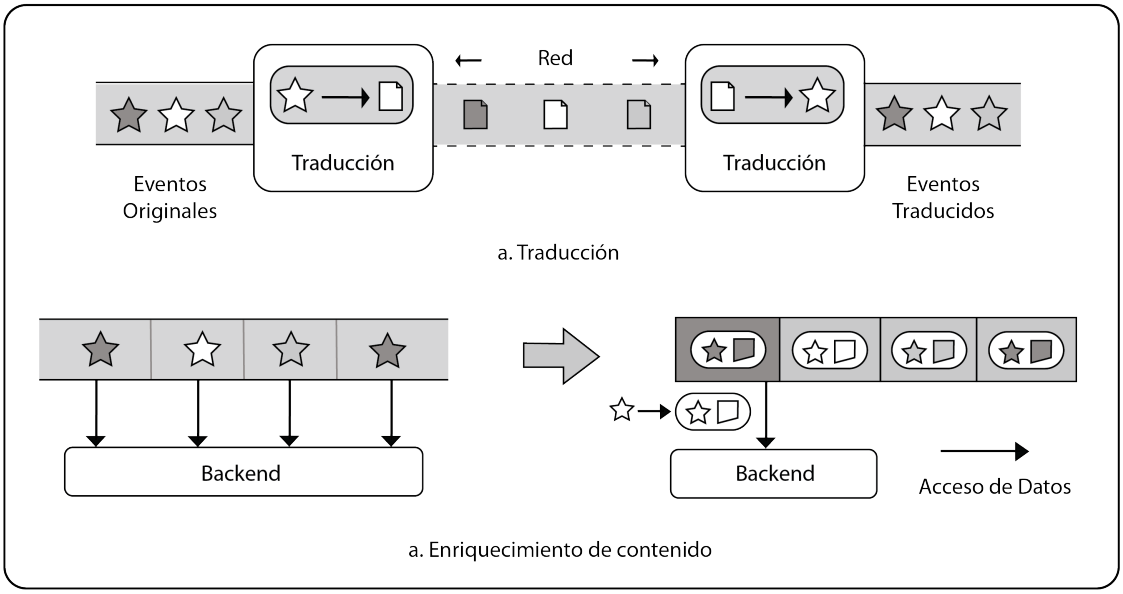


Figura 2.2: Patrón de diseño para transformación de eventos propuesto por Bruns y Dunkel [4]

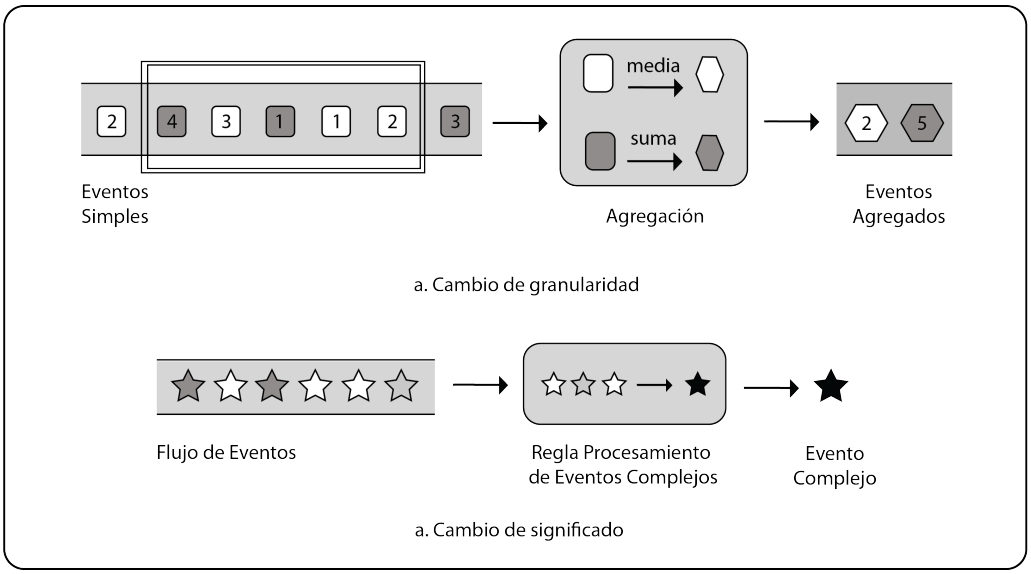


Figura 2.3: Patrón de diseño para correlación de eventos propuesto por Bruns y Dunkel [4]

En la Tabla 2.1 se muestra una comparación entre EDA y uno de los patrones más populares (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 2.1: Comparación entre EDA y SOA (elaboración propia)

	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

2.1 Procesamiento de Eventos Complejos

Ericsson y Berndtsson [9] definen el Procesamiento de Eventos Complejos como una tecnología para reconocer patrones en la nube o en un flujo de eventos para así, darle al sistema en el cual funciona, la capacidad de detectar y reaccionar frente a combinaciones específicas de eventos. Un motor de procesamiento CEP puede soportar reglas reactivas, i.e. cuando se detecta un patrón de eventos, una secuencia de código (acción) es ejecutada si se cumple una condición específica.

En términos generales, CEP es un método consistente en analizar flujos de información de cosas que ocurren (eventos) y actuar frente a ellos ya sea generando un nuevo evento o gatillando alarmas. Por otro lado, el Procesamiento de Eventos Complejos combina datos de múltiples fuentes y de estructura heterogénea para inferir patrones que sugieren circunstancias más complicadas. El objetivo es identificar eventos significativos (tales como oportunidades o amenazas) y responder frente a ellos lo antes posible.

Obwegel et al. [22] mencionan que CEP permite monitoreo en tiempo real de incidentes de la industrias y automatización de toma de decisiones para así responder ante amenazas o aprovechar oportunidades de negocios fugaces Estos eventos pueden ocurrir transversalmente en distintas capas de una organización o sistema. Hay múltiples aplicaciones de CEP, desde logística hasta detección de fraude y automatización de transacciones. Respecto de los productos comerciales disponibles, los autores mencionan que las soluciones pasaron de ser un *framework* a una consultoría, donde además de ofrecer el *framework*, capacitaban y entregaban los conocimientos expertos de cómo usarlo. Todo esto permite analizar un flujo de eventos en tiempo real y reaccionar oportunamente.

2.2 Mensajería y Colas

El *Middleware* es un componente de *software* diseñado para construir sistemas distribuidos a gran escala [13]. También permite conectar a múltiples procesos corriendo en varias máquinas a través de la red [3]. Además, soporta y simplifica la aplicaciones distribuidas. Esto incluye servidores *web*, aplicaciones, mensajería y herramientas para soportar el desarrollo y puesta en marcha de proyectos.

El *Middleware* permite interoperabilidad entre aplicaciones que corren en distintos ambientes, proveyendo servicios donde las aplicaciones pueden intercambiar datos de forma estándar. Finalmente, el *Middleware* se ubica en el medio entre aplicaciones que corren en

distintos ambientes o redes.

La mensajería consiste en una comunicación *peer-to-peer* entre aplicaciones [13]. Los sistemas de colas de mensajes cuentan con un agente externo al cual los clientes están conectados, por el que pueden enviar mensajes a cualquier cliente conectado o también recibirlos [3]. Sistemas distribuidos pueden comunicarse de dos formas: en primer lugar, 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 procesándolos sin esperar la respuesta de quien los envía [14].

Estos sistemas de mensajería usualmente apoyan dos tipos de patrones de mensajes: punto-a-punto (*p2p*) y Publicador-Subscriptor (*pub/sub*). Los dos estilos de mensajería son aplicables a 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 [28].

2.3 SEDA: *Staged Event-Driven Architecture*

Welsh et al. [34] proponen una arquitectura orientada a eventos por etapas. Abordan los siguientes problemas:

- Degradación del rendimiento bajo alta concurrencia,
- Traspaso de la complejidad del negocio al desarrollo del producto o aplicación,
- Baja capacidad de reacción de los sistemas ante problemas de servicio y alta carga.

Para esto se basan en las hebras y programación orientada a eventos (asíncrona) utilizando un conjunto de colas de las cuales consumir para aumentar la modularidad y facilitar el diseño de la aplicación. Cada etapa o “stage” es un procesador de eventos individualizado con un conjunto de hebras dentro de un flujo de eventos provenientes de una cola, los cuales son procesados en él y enviados a colas de otras etapas de forma asíncrona.

Con esto dan soporte para concurrencia masiva usando ejecución orientada a eventos donde sea posible. Además permite una construcción modular, depuramiento y análisis de

rendimiento. Demostraron con unos prototipos basados en colas de mensaje para procesamiento llamados *Sandstorm* y *Haboob* respectivamente, que la arquitectura utilizada es capaz de reaccionar ante un servicio degradado con alta carga y filtrar o priorizar eventos, ajustando de forma dinámica los recursos requeridos por las aplicaciones.

2.4 Advanced Message Queue Protocol

AMQP [1] de sus siglas en inglés, significa Protocolo Avanzado de Mensajería de Colas descrito por Vinoski en [30]. Es un estándar abierto para el paso de mensajes entre aplicaciones u organizaciones. Conecta sistemas, alimenta procesos de negocio con la información que requieren y transmite de manera confiable las instrucciones necesarias para lograr los objetivos definidos. Sus características principales a cumplir son: seguridad, confiabilidad, interoperatividad, estándar y abierto.

A pesar de que este protocolo es implementado por varias soluciones, cumple con muchas de las características más apetecidas y así como muchos proyectos basados en mensajería utilizan el *"broker"* que dirige, encamina y encola los mensajes. Esto resulta en un protocolo cliente/servidor o un conjunto de APIs sobre un protocolo no documentado que permite a las aplicaciones comunicarse con este *broker*. Los *brokers* reducen de buena forma la complejidad en redes grandes. Sin embargo, los productos de mensajería basados en un despachador centralizado significa agregar una caja negra grande y un punto de falla único adicional. Un *broker* se convierte rápidamente en un cuello de botella y un nuevo riesgo que manejar. Como alternativa, si es soportada, muchas veces se agregan varios *brokers* adicionales bajo un esquema de tolerancia a fallas. Esto significa crear más piezas, mayor complejidad y más cosas que puedan fallar.

2.5 RabbitMQ

Rostanski et al. [25] realizan una comparación utilizando RabbitMQ respecto de la alta disponibilidad y replicación de datos. RabbitMQ es un *broker* de mensajes de código abierto que adopta el estándar definido por AMQP, funciona como servidor de colas para compartir datos entre aplicaciones, encolar tareas o distribuir datos a varios procesos dispersos en la red. El principal elemento de RabbitMQ son las colas donde llegan y se almacenan (en memoria o en disco) los mensajes hasta que sean consumidos. Por otro lado, los

[1] www.amqp.org

intercambiadores son los encargados de distribuir los mensajes a las colas ya sea de forma directa o a través de una subscripción.

2.6 ZeroMQ

ZeroMQ [15] funciona como un *framework* de *sockets* que son las piezas claves que manejan las conexiones de forma automática. No se puede ver, trabajar con, abrir, cerrar o agregar conexiones. Alternativas como *send/receive*, *poll*, entre otros están disponibles. Toda comunicación es a través de los *sockets*. Las conexiones son privadas e invisibles, una de las claves de la escalabilidad de ZeroMQ.

Es una librería de código abierto que permite comunicación entre procesos de forma asíncrona, rápida y de bajo nivel utilizando conectores a través de *sockets*. Existen varios tipos de conexiones diseñados para distintos escenarios: (X)PUB/(X)SUB para el patrón publicador-subscriptor, REQ/RES para solicitud-respuesta, DEALER y ROUTER para comunicar entre clientes asíncronos con servidores asíncronos.

En este capítulo se abordó el estado del arte referente a patrones de arquitectura utlizados en Ingeniería de Software para dar solución a sistemas de grandes flujos de datos. Es así, como la Arquitectura Orientada a Eventos ha mostrado mejores resultados en estos ambientes y ha provocado un auge para la investigación y extensión de este tipo de patrones. En el capítulo siguiente, se describen casos de estudiio relacionados con el Procesamiento de Eventos Complejos, uno de las principales características descritas en ambientes orientados a eventos.

Capítulo 3

Detección de Eventos Complejos

3.1 Monitoreo en tiempo real

Li [20] 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. Aborda el problema de monitoreo de eventos en tiempo real y alerta temprana, para esto plantea lo siguiente: tener una base de datos en memoria para disminuir RTT (round-trip time), además extendiendo el paradigma *publish/subscribe* agregando un motor de reglas CEP especializado para procesamiento de eventos y finalmente el flujo de datos se almacena de forma permanente en forma paralela. Para esto, se utilizó un *hub* de mensajes conectado a una base de datos y varias fuentes de datos.

3.2 Control de tráfico

Dunkel et al. [8] proponen una arquitectura de referencia para un sistema de control de tráfico aplicado en Bilbao (España) donde se utiliza EDA y CEP argumentando que arquitecturas tradicionales o populares como SOA no son capaces o no están diseñadas para soportar gran cantidad de flujo de datos y procesar dependencias entre sí. Es así como la solución que plantean los autores posibilita el análisis y procesamiento de eventos en tiempo real. De forma general el proceso descrito se divide en las siguientes etapas: los datos se limpian en primera etapa; luego se agrupan por ubicación realizando agregaciones; a partir de los cuales se generan nuevos eventos correspondientes a alarmas de alta congestión u otros; con la nueva información se realiza un diagnóstico; para finalmente ejecutar acciones para combatir las anomalías encontradas. En la Figura 3.1 se observa el esquema que representa el flujo y procesamiento de eventos para un sistema de control de tráfico.

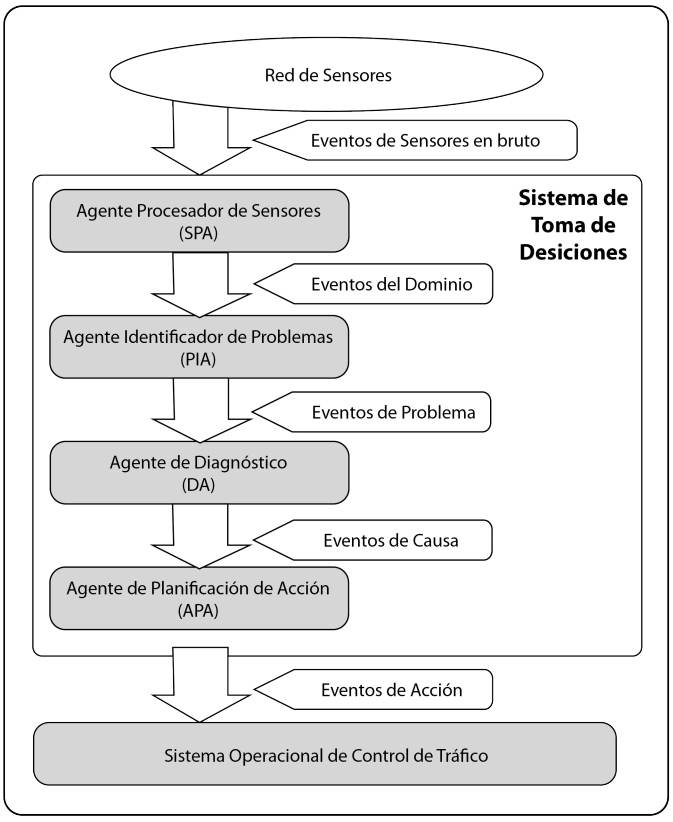


Figura 3.1: Esquema de un sistema para control de tráfico propuesto por Dunkel et al. [8]

3.3 M2M

Wan et al. [32] proponen una arquitectura orientada a eventos para comunicación máquina a máquina *M2M* y llevan a cabo un caso de estudio de control de vehículos. Presentan una arquitectura de alto nivel para SmartCities en donde se define los siguientes componentes: KBP (*Knowledge-Based Processors*) son responsables de producir (insertar/remover) y/o consumir (consultar/subscribir) notificaciones de eventos, cada notificación de un evento contiene la información individual, de la cual, por lo que la decisión de si publicar o no el evento es una de las funciones principales de los KBP. Por otro lado, los SIB (*Semantic Information Brokers*) están implementados para funcionar bajo el paradigma *publish/subscribe*, para que así los eventos sean transmitidos de los KBPs a los componentes suscritos a los eventos. En conjunto con estrategias adecuadas de enrutamiento conforman la plataforma de interoperatividad. En la Figura 3.2 se presenta la arquitectura propuesta por estos autores, la cual especifica el flujo de eventos y los componentes para su procesamiento, generación y reacción ante eventos.

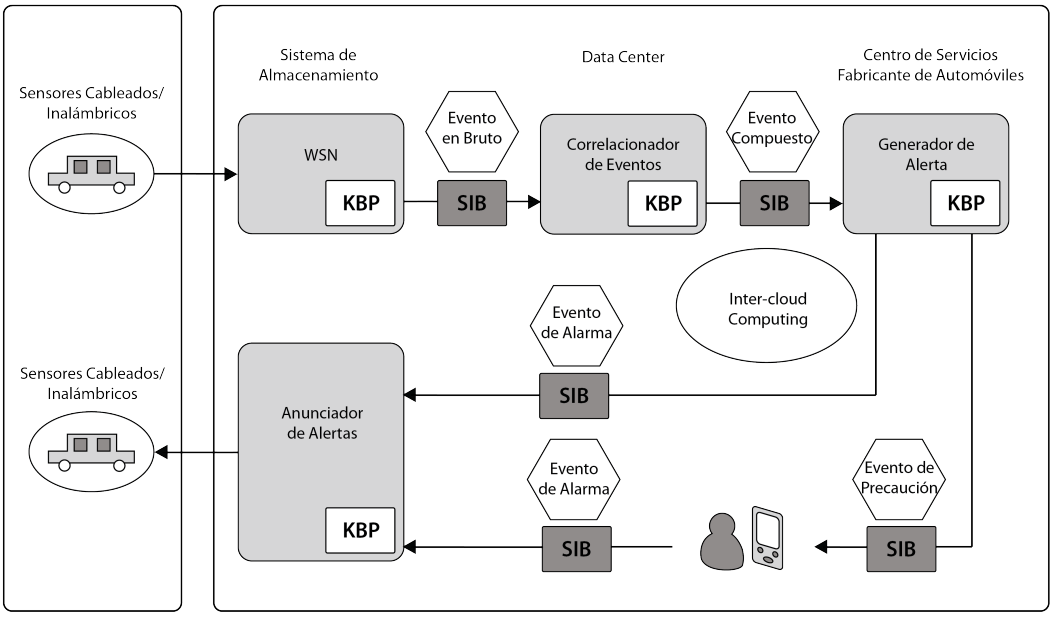


Figura 3.2: Arquitectura para sistema para control de tráfico propuesto por Wan et al. [32]

Este capítulo presenta casos de estudios relacionados con Arquitecturas Orientadas a Eventos y Procesamiento de Eventos Complejos. Basado en el estado del arte y la experiencia previa descrita en el siguiente capítulo se propone una arquitectura de referencia para sistema basado en sensores.

Capítulo 4

Propuesta para sistemas basados en sensores

En este capítulo se define una arquitectura orientada eventos diseñada para sistemas basados en sensores y sus objetivos.

4.1 Arquitectura de Referencia

Se ha propuesto algunas alternativas considerando EDA como un patrón adecuado para flujo de eventos; sin embargo, en el ámbito de integración organizacional, la mayoría de las herramientas existentes son propietarias o difíciles de implementar a medida (se utilizan como caja negra). Es por esto que a continuación se describe una arquitectura de referencia orientada a eventos para sistemas basados en sensores.

Por otro lado, con los experimentos realizados descritos en la Sección 5, se toma la decisión de utilizar ZeroMQ como sistema de colas de mensajes para implementar un caso de estudio enmarcado en el control de tráfico.

Para el diseño de la arquitectura se considera en primer lugar los 5 principios descritos en la Sección 2: Individualidad, *Push*, Inmediatez, *One-way* y Libre de comandos los cuales fueron parte de la definición del patrón de arquitectura orientada a eventos por parte de Chandy y Schulte [6].

En el caso del procesamiento de eventos complejos (CEP), la arquitectura contempla los tres patrones de diseño referentes a eventos descritos por Bruns y Dunkel [4]: Reducción de eventos, Transformación de eventos y Síntesis de eventos.

Finalmente, se diseña la arquitectura para funcionar sobre un sistema de colas de mensaje, beneficiándose del patrón de mensajería Publicador-Suscriptor y todas las características intrínsecas del mismo, tales como: direccionamiento de mensaje, suscripción según tópico, procesamiento en tiempo real, múltiples productores y consumidores y el mensaje (evento) como actor principal.

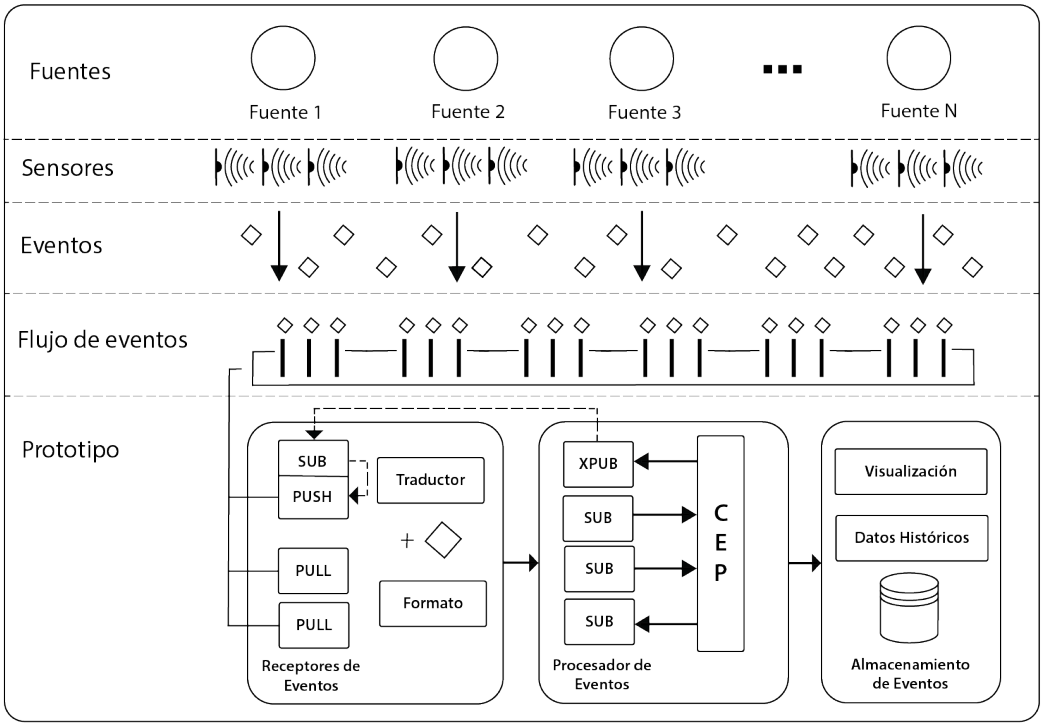


Figura 4.1: Arquitectura de Referencia para sistemas basados en sensores (control de tráfico)

En la Figura 4.1 se puede observar una arquitectura de referencia orientada a eventos mostrando sus componentes y cómo se comunican entre sí, indicando los tipos de conectores (*sockets*) que se utilizan. La idea detrás de esta arquitectura se concentra en los siguientes aspectos fundamentales:

- **Sistema *sensor-based***: múltiples clientes deben ser capaces de conectarse al sistema y enviar eventos identificados según su origen. Para esto, se diseña el "Receptor de eventos" quien se encarga de recibir los eventos de distintas fuentes, realizar tareas de traducción, limpieza y validación en caso de ser necesario, para finalmente ser enviados a otro componente para su posterior procesamiento.

- **Procesamiento de eventos complejos**: esto se lleva a cabo en el módulo CEP, en donde se procesa el flujo entrante de eventos en búsqueda de patrones valiosos para el negocio y generar notificaciones de sucesos claves.

- **Procesamiento en tiempo real**: la arquitectura permite procesar eventos a medida que llegan y de escalar fácilmente de forma horizontal. Así es como el procesamiento de eventos complejos ocurre de forma paralela al almacenamiento de datos históricos y la capa de visualización.

- **Conectores por tópico**: los conectores *SUB* que aparecen en la Figura 4.1 tienen la principal característica de recibir eventos según el tópico, es decir, el Receptor de Eventos en este caso está suscrito a eventos provenientes de los sensores que tiene asignado.

- **Generación de eventos**: a medida que los agentes CEP procesan eventos a medida que llegan, generan eventos enriquecidos de gran utilidad para generar alarmas, tomar medidas o verificar el estado del tráfico que reportan los sensores a nivel general, no individual.

En el Capítulo 6 se implementa un caso de estudio utilizando la arquitectura de referencia propuesta aplicado a un sistema de control de tráfico para finalmente en el Capítulo 7 realizar una simulación con datos reales.

4.2 Trabajos Previos

En el Capítulo 3 se presentan trabajos previos relacionados con detección de patrones de eventos complejos. Uno de los aspectos fundamentales a abordar es el procesamiento de eventos en tiempo real y para ello, la arquitectura que se propone desacopla el almacenamiento del procesamiento de eventos como procesos independientes. A diferencia de los trabajos realizados por Dunkel et al. [8], Wan et al. [32] y Li [20], la arquitectura de referencia que se propone en este capítulo, considera el procesamiento de eventos complejos como un proceso paralelo al de almacenamiento sin hacer uso de ello.

4.2.1 Aspectos comunes

Los trabajos previos mencionados y la presente propuesta tienen en común las siguientes características:

1. Arquitectura Orientada a Eventos: todas las propuestas utilizan un enfoque orientado a eventos debido a la naturaleza del problema y la idoneidad de este patrón arquitectónico para sistemas basados en sensores.
2. Recepción de eventos: en todas las propuestas, hay un componente inicial que se encarga de recibir los eventos de distintas fuentes de datos, encargándose de validar y traducir el contenido para luego ser procesado.
3. Almacenamiento de eventos: los eventos son almacenados en repositorios de datos persistentes para tener datos históricos y hacer consultas sobre ellos.
4. *Publish-Subscribe*: un factor común de los trabajos previos es el patrón publicador-subscriptor utilizado, el cual permite despachar eventos de acuerdo a los consumidores que estén interesados (a través de una subscripción).

4.2.2 Aspectos diferenciadores

La arquitectura propuesta difiere respecto de los trabajos previos, a continuación se presentan los principales elementos:

- Almacenamiento desacoplado: el proceso de almacenamiento como se menciona anteriormente, se realiza de forma paralela y el procesamiento de eventos no depende de los repositorios de datos ni hace consultas sobre ellos. Esto conlleva una ventaja en rendimiento y respecto de tolerancia de fallos, ya que el procesamiento de eventos puede quedar en segundo plano con tal de asegurar el almacenamiento permanente de los eventos en caso de que el sistema presente alta carga o incluso, pueden ocurrir en distintos ambientes de ejecución.
- Sistema de toma de decisiones: Dunkel [8] propone un sistema de toma de decisiones que se comunica con el sistema operacional de control de tránsito, así, todo el procesamiento de eventos ocurre dentro del primer sistema. Una de las desventajas de este enfoque, es que es poco flexible, sin embargo, permite desacoplar el procesamiento de eventos.

Capítulo 5

Verificación de escalabilidad de EDA

En esta sección se describe las pruebas a realizar para comparar dos sistemas de colas de mensaje, RabbitMQ versus ZeroMQ. Luego, se muestran los resultados obtenidos y un breve análisis de cuál de los dos sistemas será utilizado para implementar el caso de estudio aplicado a control de tráfico.

El propósito de este estudio, es verificar las capacidades intrínsecas de sistemas de colas de mensaje para implementar arquitecturas orientadas a eventos y su rendimiento frente a un alto flujo de datos.

5.1 Modelos a evaluar

ZeroMQ broker: Se implementa un *broker* como se aprecia en la Figura 5.1 y en la Figura 5.2. 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.

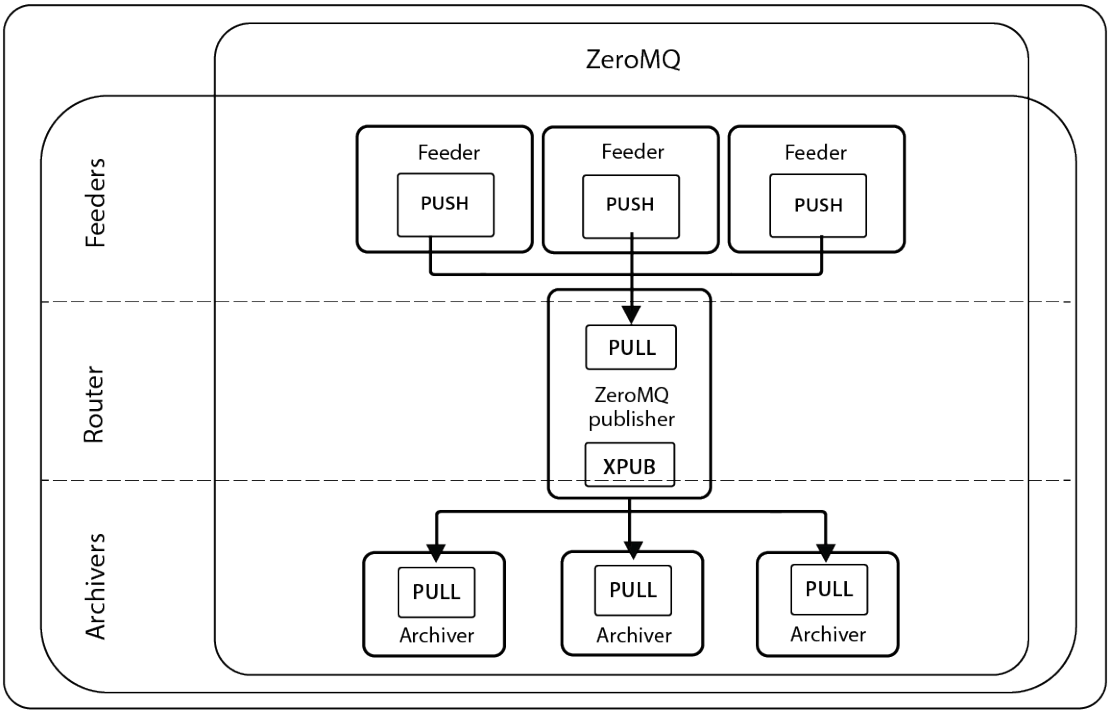


Figura 5.1: Casos de estudio implementados: ZeroMQ

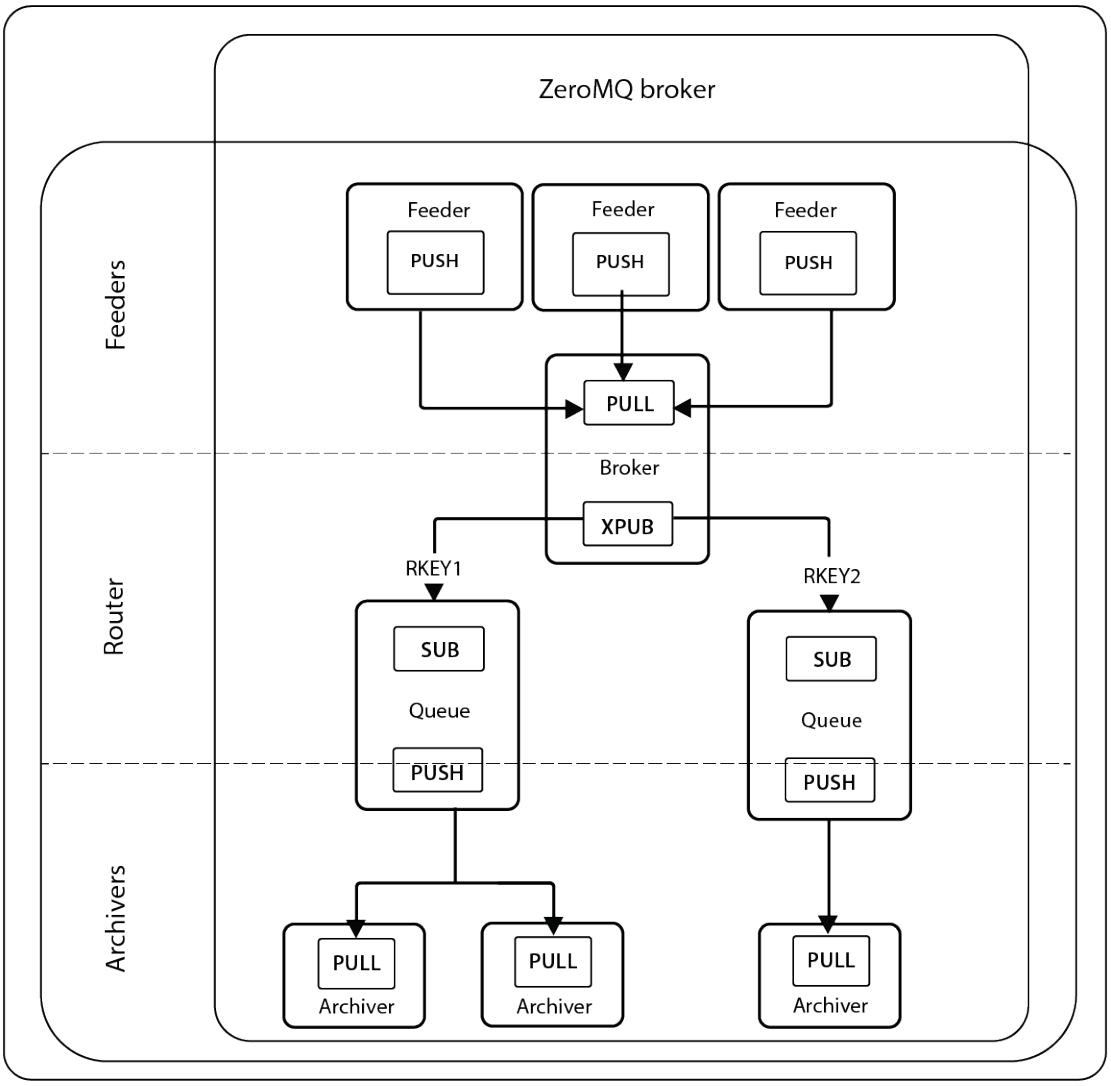


Figura 5.2: Casos de estudio implementados: ZeroMQ Broker

RabbitMQ: Se configura RabbitMQ (Figura 5.3) 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.

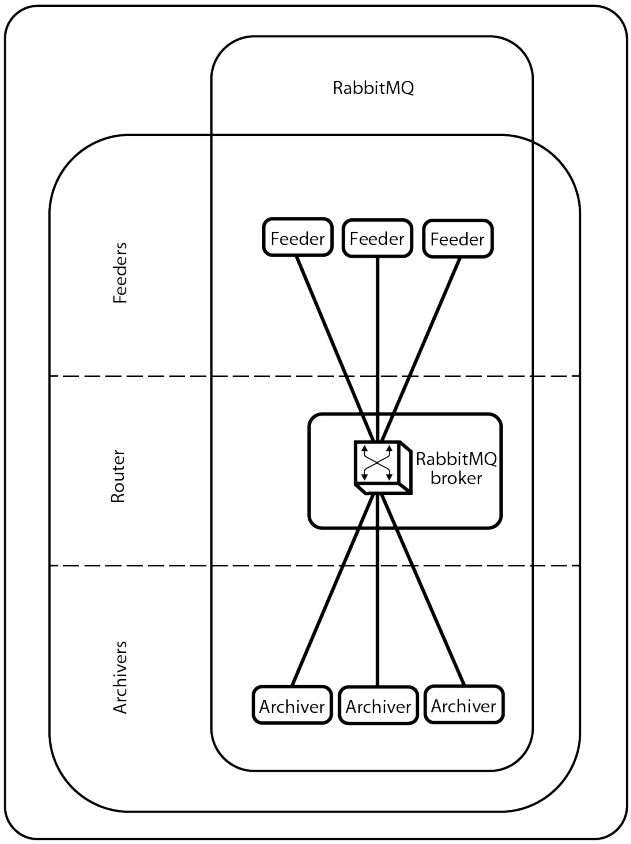


Figura 5.3: Casos de estudio implementados: RabbitMQ

5.2 Parámetros y entornos de prueba

En la Tabla 5.1 se observa las máquinas empleadas para las pruebas de rendimiento y el entorno de desarrollo utlizado.

Tabla 5.1: Especificación de Máquinas y Entorno utilizado

Característica	***Feeders***	***Broker***	***Archivers***
Sist. Operativo	GNU/Linux 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
Lenguaje	Python 2.7.8		
Control de Versiones **Código Fuente** ***Branch***	Git (Bitbucket) https://bitbucket.org/nestrada/zmq_rmq_eda/overview master + traffic/zmq-eda/master		
Principales Librerías	pyzmq=14.7.0, kombu=3.0.21, numpy=1.9.2, amqp=1.4.5		

En la Figura 5.4 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 un servidor que contiene los *archivers* (consumidor).

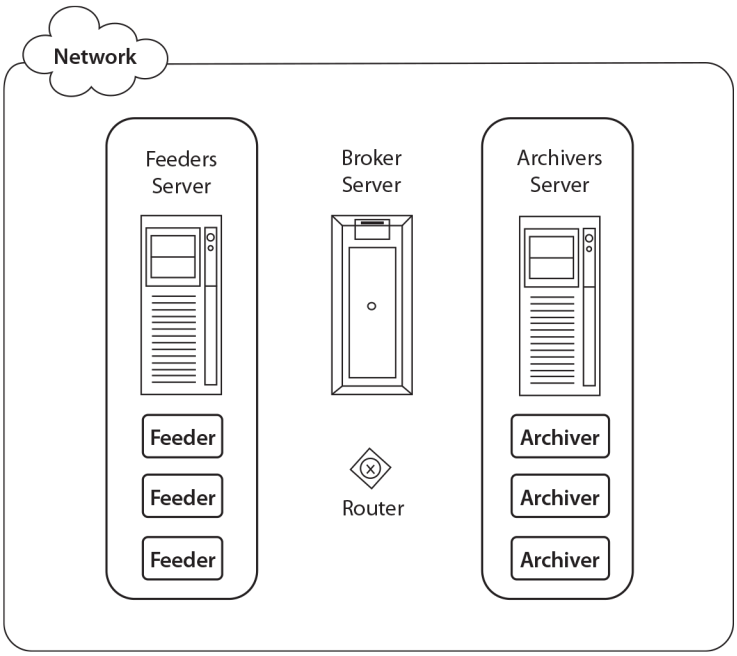


Figura 5.4: Diagrama de servidores

5.3 Pruebas de rendimiento

1. Pruebas de I/O:

   La prueba consiste en productores que envían mensajes a un servidor (*broker*) que es capaz de dirigir éstos 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.

   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.

2. 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.

5.4 Pruebas de escalabilidad

Una vez determinado el límite de I/O de un consumidor, se realizaron pruebas de desempeño de ambas tecnologías para estudiar cómo 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 utilizó 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).

5.5 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 utilizar otras tecnologías lo que sirve como referencia para sistemas de mensajería y arquitecturas orientadas a eventos.

5.6 Eligiendo un sistema de colas de mensaje

Se implementaron los casos de prueba [1], a continuación se pueden observar los resultados obtenidos.

Para cada prueba definida, los resultados se muestran a continuación.

5.6.1 Determinando el límite de operaciones sobre disco por consumidor

Para ambas tecnologías, se corrieron las pruebas descritas en la Sección 2. Los resultados obtenidos se pueden observar en la Figura 5.5. 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 muestra para ambas tecnologías el mismo tope en cuanto a rendimiento pero en distintas magnitudes, debido a que se trata del máximo de operaciones en disco que puede realizar cada proceso consumidor de forma individual y que en este caso, ZeroMQ permite procesamiento de mensajes más veloz que utilizando RabbitMQ.

Para este primer modelo implementado, es decir, ZeroMQ (ver Figura 5.1) se determinó el límite de operaciones en disco por consumidor. Si se observa la Figura 5.5, el límite cuando las operaciones por disco se estancan es prácticamente el mismo para ambas

[1] `https://bitbucket.org/nestrada/zmq_rmq_eda/src/`

tecnologías (límite 10.000 mensajes enviados por segundo), sin embargo, ZeroMQ procesa mayor cantidad de mensajes por segundo, un 51% superior. Esto significa que ante un flujo entrante de mensajes, ZeroMQ es más rápido procesándolos.

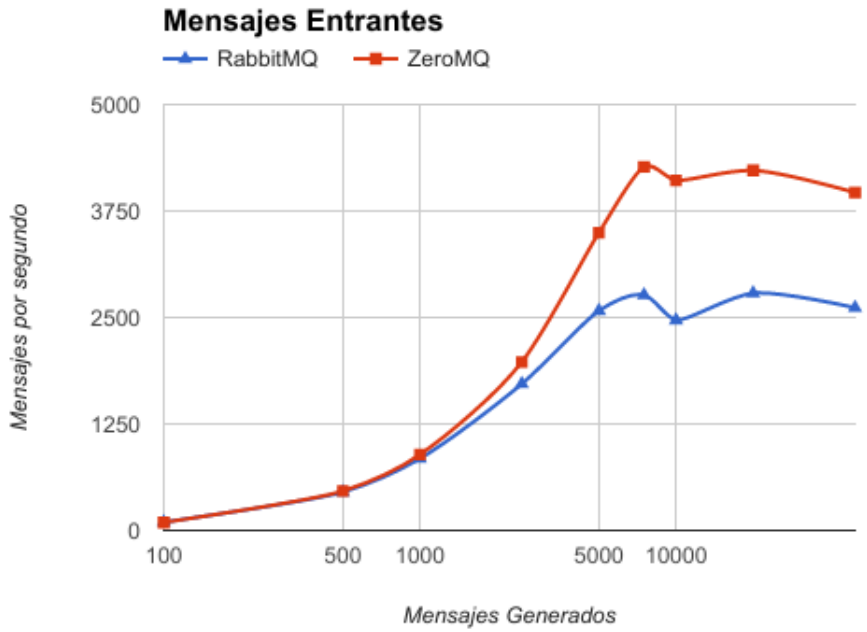


Figura 5.5: Límite de operaciones en disco por consumidor

5.6.2 Determinando el mejor desempeño según cantidad de consumidores

Una vez determinado el límite de operaciones por proceso consumidor, se utilizó el óptimo para realizar pruebas con varios consumidores. En este caso se midió con un productor, un *broker* y como límite 20.000 mensajes por segundo, variando la cantidad de consumidores como se define en la Sección 1.

Por otro lado, de forma de visualizar el rendimiento del prototipo frente a la variación de cantidad de consumidores, se utilizó la siguiente definición:

$$R(c) = \frac{r_c}{r_{ref} * c}$$

Donde el rendimiento relativo paralelo está dado por el cuociente entre el rendimiento obetenido para cierta cantidad de consumidores (r_c) y el producto entre el mejor rendimiento (r_{ref}) y la cantidad de consumidores (c). Es decir, este indicador es la fracción del rendimiento ideal esperado.

En la Figura 5.6 se observan los resultados obtenidos. En el caso de ZeroMQ, el rendimiento (msg/s) alcanzado es superior al obtenido por RabbitMQ en cuanto a cantidad de mensajes procesados, sin embargo, el rendimiento relativo evoluciona de la misma manera para ambas tecnologías.

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 5.7, ZeroMQ tienen un mayor desempeño y se estanca a partir de los 5 consumidores, en cambio RabbitMQ se estanca a partir de los 4 consumidores. Independiente de las capacidades de carga de cada sistema de colas, el umbral de degradación es mayor para ZeroMQ.

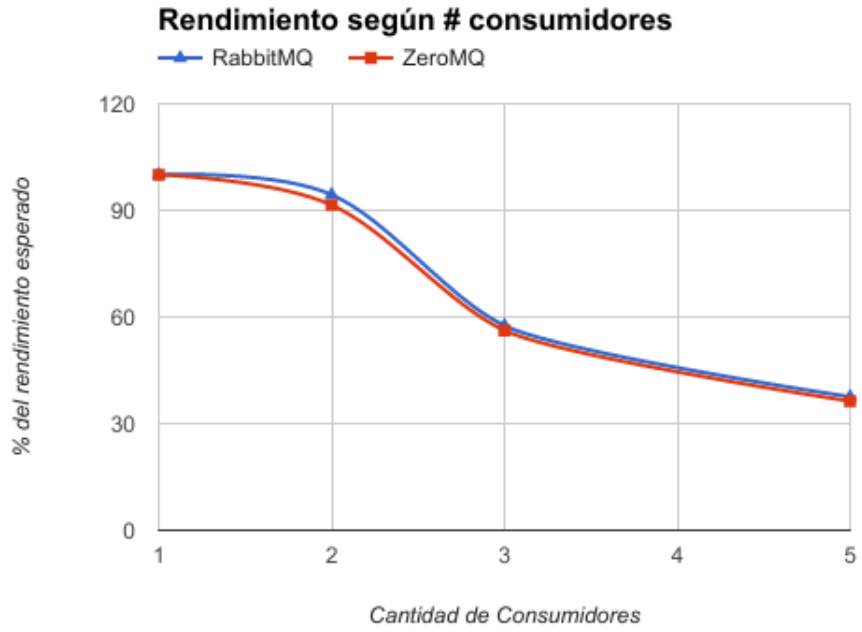


Figura 5.6: Gráfico de rendimiento con flujo fijo según cantidad de consumidores

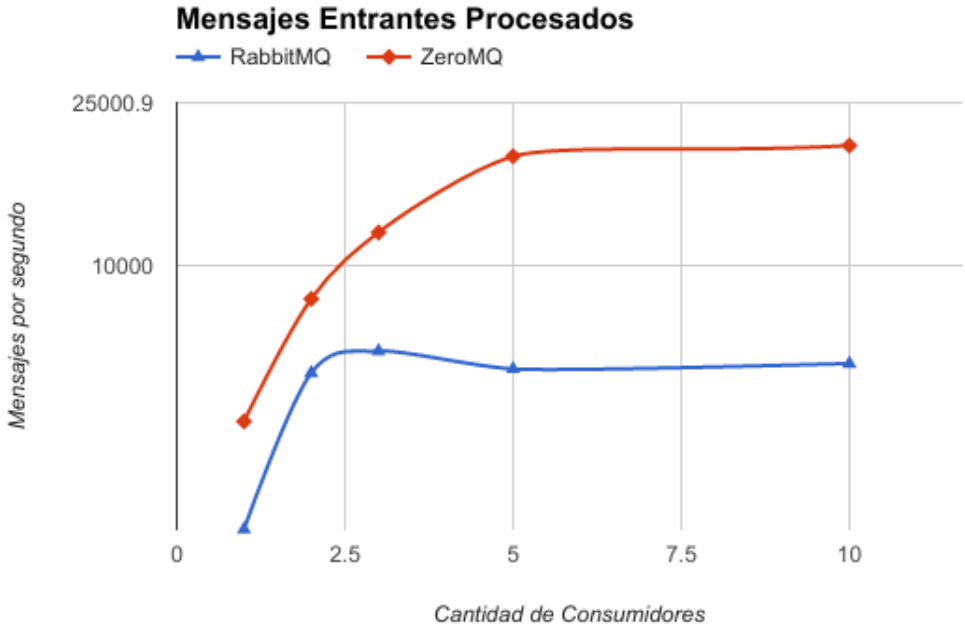


Figura 5.7: Gráfico de eventos procesados según cantidad de consumidores

Adicionalmente, en la Figura 5.6 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 5.7 donde el rendimiento se degrada en puntos distintos para ambas implementaciones.

Por otro lado, analizando el desempeño de ambos sistemas en la Figura 5.8 la diferencia entre ambos sistemas , siendo un 14% más eficiente el modelo implementado con ZeroMQ respecto del rendimiento obtenido con RabbitMQ.

En la Figura 5.8 se observa la diferencia de rendimiento relativo entre ambas tecnologías, siendo ZeroMQ en promedio un 22.2% superior, es decir, considerando todas las mediciones obtenidas está más cerca del rendimiento paralelo. El rendimiento de referencia corresponde al mayor obtenido dentro de los puntos de medición, es así como para ZeroMQ esto ocurre cuando se tienen un consumidor y para RabbitMQ con dos. Por otro lado, se observa que la primera se comporta de mejor forma hasta los 5 consumidores, tras lo cual decae de forma más abrupta (41% versus 21% de RabbitMQ) al pasar de 5 a 10 consumidores.

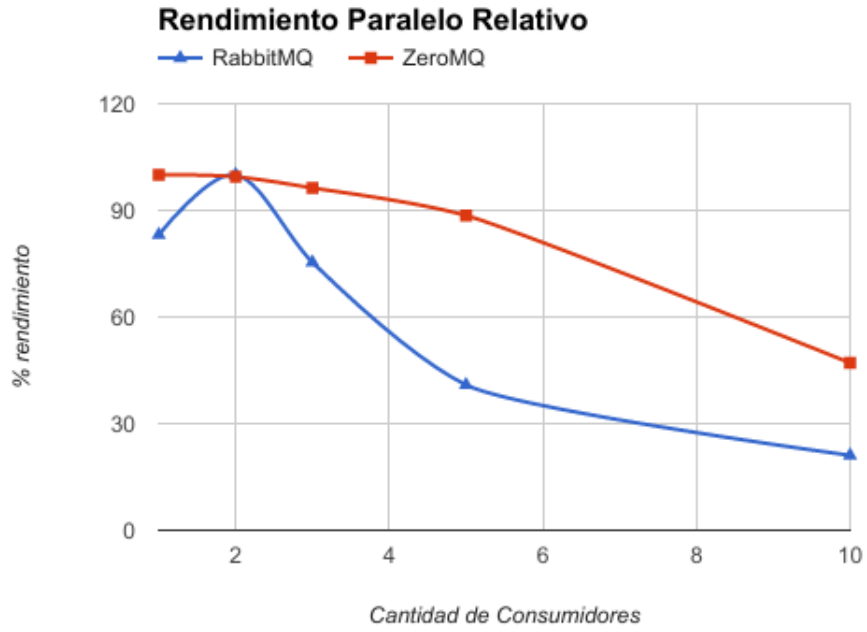


Figura 5.8: Gráfico de rendimiento según cantidad de consumidores

5.6.3 Determinando el rendimiento del *broker* con múltiples consumidores

Con las pruebas definidas en la Sección 2 y utilizando los resultados anteriores, se pone a prueba el *broker*. Se usó un productor, un *broker* y tres consumidores, variando los límites de mensajes a enviar por segundo.
Los resultados obtenidos se observan en la Figura 5.9 donde se puede observar que el rendimiento utilizando ZeroMQ es superior al logrado por RabbitMQ.

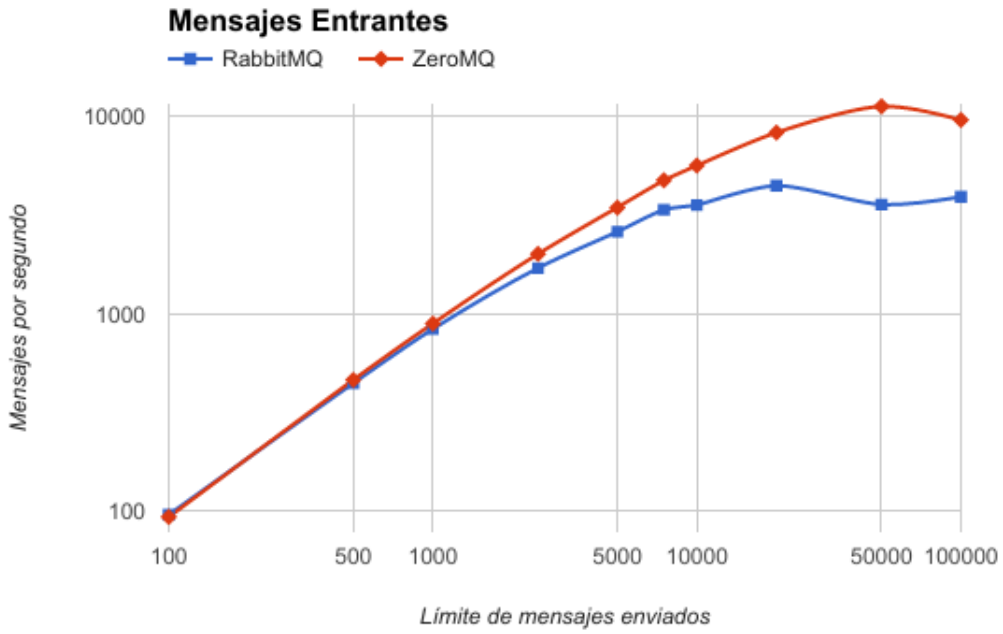


Figura 5.9: Gráfico de rendimiento del *broker* con 3 consumidores logarítmico

Finalmente, con 3 consumidores, se observa el rendimiento frente a un flujo de eventos creciente. Es así como en la Figura 5.9 se observa como el umbral de degradación para ZeroMQ es 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 5.10.

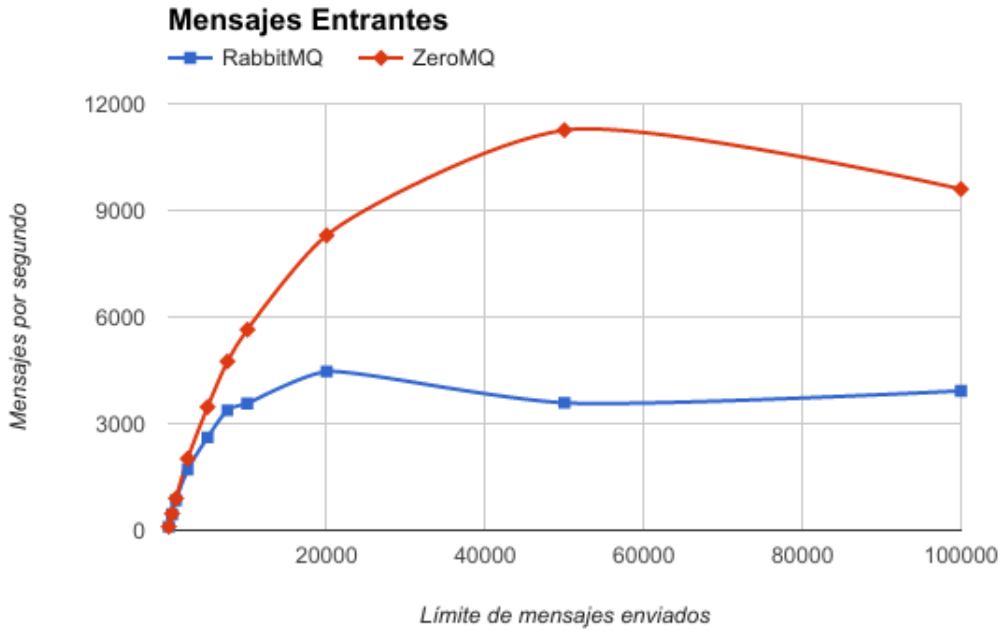


Figura 5.10: Gráfico de rendimiento del *broker* con 3 consumidores no logarítmico

5.6.4 Conclusiones de sistemas de colas de mensaje

A continuación se presenta las conclusiones respecto de las pruebas realizadas para elegir una tecnología de colas de mensaje y las razones para ser utilizada en la implementación del caso de prueba que se detallará en este trabajo.

- ZeroMQ es personalizable y liviano, funciona como una librería de *sockets* por sobre la cual operar, por lo que entrega mayor flexibilidad a la hora de implementar una solución o de diseñar una arquitectura. En cambio RabbitMQ, es una plataforma por sobre la cual operar sin posibilidad de modificar, sirve para casos en el que su topología y funcionalidades calcen con las características inherentes del producto de mensajería mencionado.
- Respecto del rendimiento en una máquina aislada, ZeroMQ rinde de mejor forma, aproximadamente un 31% mejor que RabbitMQ, lo que refleja la sobrecarga que conlleva un producto como RabbitMQ y por oto lado, lo modular de ZeroMQ que se traduce mejor rendimiento.
- De la escalablidad, la prueba utilizando ZeroMQ resultó un 28% más eficiente (cantidad de mensajes procesados por segundo) que el modelo utilizando RabbitMQ, considerando que en *throughput* fue un 35% más eficiente y para la razón de cambio fue un 22% mejor, ambos aspectos considerados igual de importantes, por eso se utilizó un promedio aritmético entre ambos.
- El rendimiento relativo paralelo fue un 22.2% mayor en el caso de ZeroMQ, lo que significa que a medida que se adicionan consumidores es capaz de mantenerse más cerca del rendimiento ideal respecto del punto de referencia (mejor rendimiento individual).

Finalmente, para el caso de estudio que se lleva a cabo en este documento, se decide utilizar ZeroMQ como tecnología de cola de mensajes por los resultados y conclusiones expuestos en este capítulo.

En este capítulo se realizaron pruebas de rendimientos comparando RabbitMQ y ZeroMQ, obteniendo como principal resultado un superior rendimiento del segundo sistema de colas utlizado. Tras los resultados obtenidos, se escoge ZeroMQ para implementar un prototipo aplicado a un caso de estudio que se define en el siguiente capítulo.

Capítulo 6

Prototipo para Procesamiento de Eventos Complejos

6.1 Diseño

Se implementó un prototipo basado en la arquitectura de referencia que se describe en la Figura 4.1 y considerando además, una simulación con datos reales.

El prototipo implementado considera los siguientes componentes:

1. **Generador de eventos *trace-driven***: obtiene los datos desde un archivo *csv*[1] y genera eventos de forma secuencial para cada sensor y en forma paralela de modo general, es decir, cada uno de los sensores está enviando señales como si estuviesen ocurriendo en tiempo real simulando el tránsito de vehículos. Dependiendo de la velocidad y la cantidad de mediciones se generan los eventos en tiempo de ejecución.

2. **Sensor**: cada uno de los sensores recibirá una notificación del generador de eventos simulando el comportamiento de los sensores para luego enviarlos a un receptor especializado. Pueden existir varios conectados.

3. **Receptor de Eventos**: recibe los eventos desde los sensores para ser procesados y enviados posteriormente al controlador.

[1] csv: sigla del inglés de valores separados por coma

4. **Controlador**: es el intermediario entre dos etapas: la primera consiste en el almacenamiento de los eventos; y la segunda en el procesamiento de eventos complejos.

5. **CEP**: componente que procesa el flujo de eventos entrantes en búsqueda de patrones, detección de anomalías y alerta frente límites definidos.

6. **Datos**: en esta parte, los datos son almacenados para su posterior uso o análisis histórico.

6.2 Implementación

La Figura 6.1 describe la estructura del procesador de eventos complejos. Por un lado se tiene el controlador (*controller*) que se encarga de recibir todo el tráfico de eventos, para luego ser enviado de forma paralela al proceso encargado del almacenamiento y al procesador de eventos complejos. Los agentes CEP reciben los eventos indistintamente utilizando el mecanismo *Round-Robin*. Producto de esta etapa, se generan eventos adicionales enviados al proceso agregador, proceso que recibe eventos según los niveles definidos en la Tabla 6.1.

De esta forma, el procesamiento de eventos complejos ocurre en dos fases representadas por los siguientes componentes:

1. **Agentes procesadores CEP**: cada agente procesador es un proceso independiente conectado al controlador del cual recibe los eventos. Por cada evento, el agente observa la velocidad verificando que no supere los límites (superiores e inferiores) y que la variación no supere los rangos definidos, de otra forma, genera nuevos eventos (alarmas) dependiendo el caso, las cuales son enviadas usando un tópico afín que identifique cada una hacia el agregador.

2. **Agregador**: el agregador recibe los eventos generados por los agentes CEP que corresponden a alarmas de diversos grados. A medida que va recibiendo loe eventos, verifica su correlación (temporal y espacial). En el caso de la correlación temporal, se utiliza una ventana de tiempo previamente definida y a su vez para la correlación espacial, el identificador numérico del sensor. Finalmente, el agregador busca patrones de ocurrencia de eventos inusuales concurrentes.

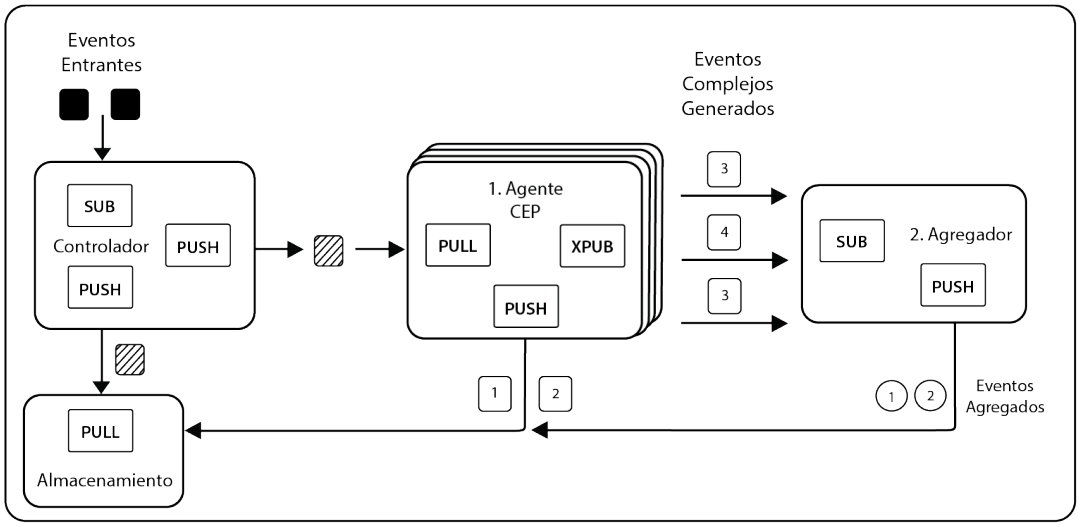


Figura 6.1: Implementación de CEP

6.2.1 Definiciones de alarmas

La Tabla 6.1 describe los niveles de alarmas especificando para cada uno: la palabra clave, el identificador, el tópico (*routing key*), las acciones necesarias a tomar en caso de su ocurrencia, el límite superior de velocidad en caso que aplique.

Tabla 6.1: Definición de Alarmas

Nivel	Id	Key	Acciones	Umbral
RECOVERY	-1	recovery.avg	send_event	10
OK	0	ignore.avg	-	10
WARNING	1	warning.avg	send_event	20
CRITICAL	2	critical.avg	send_event, cep_agg	50
EXCEPTION	3	exception.avg	send_avent, cep_agg	100
EXCEPTION_MIN	4	exception.min	send_event, cep_agg	5
EXCEPTOIN_MAX	5	exception.max	send_event	150
EXCEPTION_AGG	10	exception.agg	semd_event	-

6.2.2 Consideraciones de Procesamiento de Eventos Complejos

El Algoritmo 1 se tienen dos escenarios: (1) si la cantidad de mediciones almacenadas es menor al tamaño de la ventana deslizante se utiliza un promedio simple con los valores disponibles, y (2) si la cantidad de mediciones almacenadas supera el tamaño de la ventana, donde efectivamente la variación calculada se realiza tomando el valor de promedio móvil. Finalmente, se genera un evento acorde a los parámetros definidos (ver Tabla 6.1). En la Figura 6.2 se muestra un trozo de código que corresponde a la función *get_level(speed, variation)* donde está toda la lógica de los niveles de alarmas y eventos generados según su severidad. Se utiliza *yield* para retornar eventos, de tal forma que para una combinación velocidad-variación se pueden generar más de un evento según las definiciones que se establezcan previamente. En este caso, se pueden generar hasta 2 eventos, uno por límites de velocidad y otro por variación respecto de la media móvil.

Algoritmo 1 Algoritmo agentes CEP

```
1:  procedure CHECK(speed, values)
2:      offset ← length(values)
3:      if offset < WINDOW_SIZE then
4:          percent_variance ← 100 * (simple_average − speed)/simple_average
5:      else
6:          percent_variance ← 100 * (moving_average − speed)/moving_average
7:      end if
8:      cep_event ← get_level(speed, percent_variance)
9:      return cep_event
10: end procedure
```

En el Algoritmo 2 se observa la tercera forma de generar eventos complejos, la primera y la segunda descritas en el Algoritmo 1. En este caso, se utiliza ventanas deslizantes, las cuales son ventanas de tiempo en las cuales se analiza los eventos ocurridos en ellas que van variando de forma dinámica a medida que pasa el tiempo. En primer lugar se verifica que la ventana de tiempo correspondiente al indicador de fecha-hora exista, para luego obtener los eventos geográficamente consecutivos (dados por el identificador numérico de cada sensor) y verificar que exista un patrón de interés. En caso de existir se retorna todos los patrones encontrados.

```
def get_level(self, speed, variation):

    # evaluating threshold
    if speed < MIN_THRESHOLD:
        yield Notification.EXCEPTION_MIN

    elif speed > MAX_THESHOLD:
        yield Notification.EXCEPTION_MAX

    # evaluating moving average variation
    if variation < self.IGNORE['threshold']:
        # if recovered or still under traffic issues
        if (not self._last_notification_id
            and self.RECOVERY['notify_id']
                not in self._lasts_notifications_ids
            and self._lasts_notifications_ids.count(
                    self.IGNORE['notify_id']) >= WARMUP):

            yield Notification.RECOVERY

        else:
            yield Notification.IGNORE

    elif variation < self.WARNING['threshold']:
        yield Notification.WARNING

    elif variation < self.CRITICAL['threshold']:
        yield Notification.CRITICAL

    elif variation < self.EXCEPTION['threshold']:
        yield Notification.EXCEPTION

    else:
        yield Notification.EXCEPTION
```

Figura 6.2: Generación de Eventos Complejos

Algoritmo 2 Algoritmo del agregador

```
1:  procedure GET_AGGREGATED
2:      pattern_matched ← []
3:      for ts in available sliding_windows do
4:          delete ← false
5:          for events in get_consecutive(events in ts) do
6:              if length(events) > 1 then
7:                  pattern ← repr(events)
8:                  pattern_matched.append(pattern)
9:                  delete ← true
10:             end if
11:         end for
12:         if delete then
13:             events.delete(ts)
14:             delete ← false
15:         end if
16:     end for
        return pattern_matched
17: end procedure
```

Algoritmo 3 Algoritmo de detección de patrones

```
1:  procedure AGGREGATOR(events_queue, notifier)
2:      ts_base ← None
3:      while True do
4:          rkey,message ← events_queue.receive()
5:          ts_key ← timestamp − (timestamp % WINDOW_SIZE)
6:          register_event(message.sensor_id, message._timestamp)
7:          if ts_base is None then
8:              ts_base ← ts_key
9:          else
10:             if ts_key > ts_base then
11:                 for agg_event in get_aggregated(ts_base, WINDOW_SIZE) do
12:                     rkey ← get_rkey(message)
13:                     notifier.send_notification(rkey, message)
14:                 end for
15:                 ts_base ← ts_key
16:             else
17:                 pass
18:             end if
19:         end if
20:     end while
21: end procedure
```

En el Algoritmo 3 se observa el mecanismo utilizado para detectar patrones. Bajo el constante control mediante suscripción a eventos provenientes del agente CEP, se analizan a medida que llegan los eventos dentro de una ventana de tiempo, para así buscar patrones anómalos entre sensores adyacentes. En primer lugar, el detector de patrones escucha eventos de interés (eventos anómalos enviados por los agentes CEP) para luego registrarlos y procesarlos. A medida que una ventana de tiempo queda en el pasado, se busca los patrones para ser notificados. De esta forma, se tiene un detector de patrones en tiempo real con una ventana de tiempo determinada.

Finalmente, la Figura 6.3 muestra el mecanismo utilizado por el agregador y su funcionamiento de forma gráfica. Es importante notar que los agentes CEP procesan eventos de forma paralela enviando eventos al agregador según corresponda (definición de alarmas y búsqueda de patrones). De esta forma, para los niveles de alarma definidos y la búsqueda de patrones se considera tres aspectos:

1. Nivel de alarma: el agregador se suscribe a eventos según su nivel de severidad.
2. Temporal: el agregador realiza la búsqueda de patrones dentro de una ventana de tiempo acotada y previamente definida.
3. Espacial: la búsqueda de notificaciones de alarma se realiza entre sensores con identificadores consecutivos.

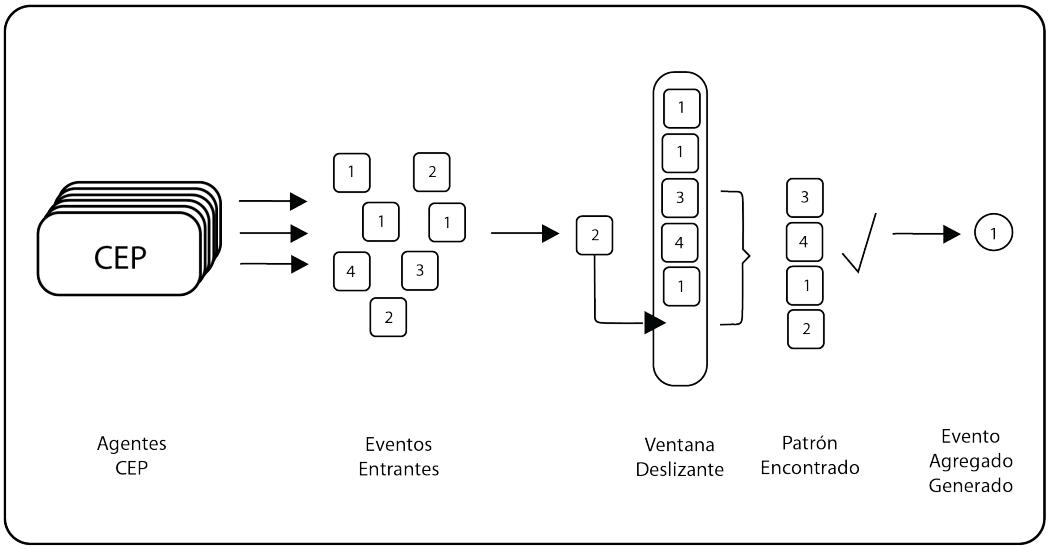


Figura 6.3: Funcionamiento del Agregador

Capítulo 7

Mediciones y Resultados

La Figura 7.1 basada en la arquitectura propuesta en la Sección 4.1, muestra un caso de estudio de control de tránsito implementado según lo descrito en la Sección 6. El Procesador de Eventos Complejos fue implementado considerando tres mecanismos de detección de anomalías: media móvil de la velocidad sobre una ventana de tiempo; límites mínimos y máximos de velocidad; y correlación de eventos (búsqueda de patrones). Otro punto a tomar en cuenta, es que el generador de eventos envía los mensajes basado en el identificador del sensor que lo originó. Así, los mecanismos utilizados en esta implementación tales como: *sliding-windows*, *granularity shift*, *semantic shift* (calce de patrones) y *content-based routing* (subscripción basado en el identificador del sensor) son mencionados por Bruns y Dunkel [4].

7.1 Datos para Simulación

Los datos utilizados para la simulación *trace-driven* fueron proporcionados por SECTRA [1] (datos para el 14 de Mayo del 2015).

Los datos de entrada están dados en intervalos de 15 minutos y contienen datos resumidos (cantidad de mediciones, fecha e identificador de sección). Cada registro contiene la cantidad de vehículos registrados, su velocidad promedio y tiempo de viaje. Estos datos fueron utilizados para validar y poner a prueba la arquitectura diseñada.

[1] http://www.sectra.gob.cl/ (Secretaría de Planificación de Transporte)

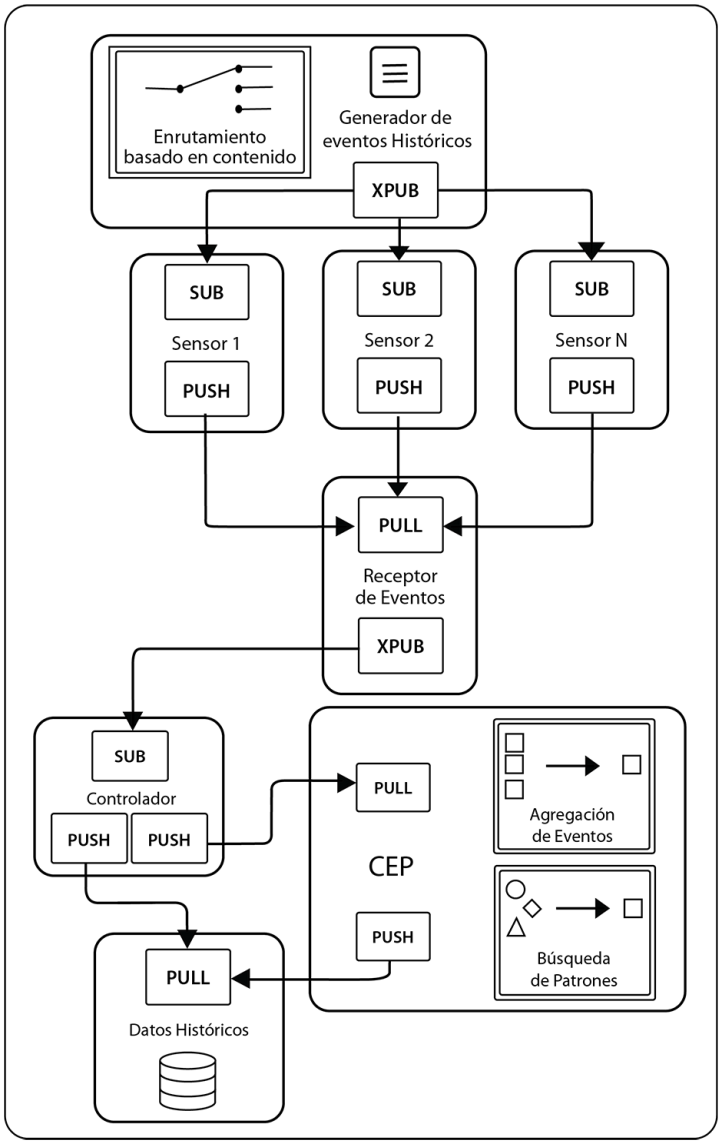


Figura 7.1: Caso de estudio implementado en un escenario de control de tráfico

7.2 Resultados del prototipo de la arquitectura propuesta

La arquitectura de referencia orientada a eventos para sistemas basado en sensores (Figura 7.1) considera lo siguiente:

- **Generación de eventos simulado**: los datos de entrada consideran intervalos de 15 minutos. Esto significa que el generador de eventos etiqueta cada evento con una fecha base más una diferencia definida por el desfase del evento dentro del intervalo. Además, como la velocidad indicada es un promedio, un generador aleatorio con una distribución normal [2] es utilizado con una desviación estándar de 7 km/h (considerando la varianza de velocidad en vehículos).
- **Enrutamiento a todo nivel**: un identificador de sensor es utilizado por el generador de eventos para comunicarse con los sensores. Después, una cadena de caracteres indica el tipo de evento como la llave por la que es enviado, e.g.: *event*, *warning*, *critical*, *exception*. Más aún, algunos componentes se comunican usando *sockets* PUSH/PULL cuando ni el enrutamiento ni el filtrado es necesario.
- **Media móvil y comparación de velocidades**: para determinar si la velocidad medida es razón para lanzar una alarma o no, se usa una función de media móvil (basada en *numpy.convolve* [3]). Para lograr detección de eventos complejos, esta función es combinada con la variación de velocidad y límites inferiores y superiores de velocidad.
- **CEP**: el procesamiento de eventos complejos ocurre en dos etapas: síntesis de alarmas y procesamiento con ventana de tiempo usando las técnicas aritméticas mencionadas anteriormente.
- **Topología**: la red de sensores está distribuida geográficamente en la Región Metropolitana de Chile, donde los identificadores numéricos de cada sensor constituyen su posición y relación con los demás sensores. Son un total de 24 sensores que envían eventos a medida que éstos ocurren.

[2] http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.normal.html
[3] http://docs.scipy.org/doc/numpy/reference/generated/numpy.convolve.html

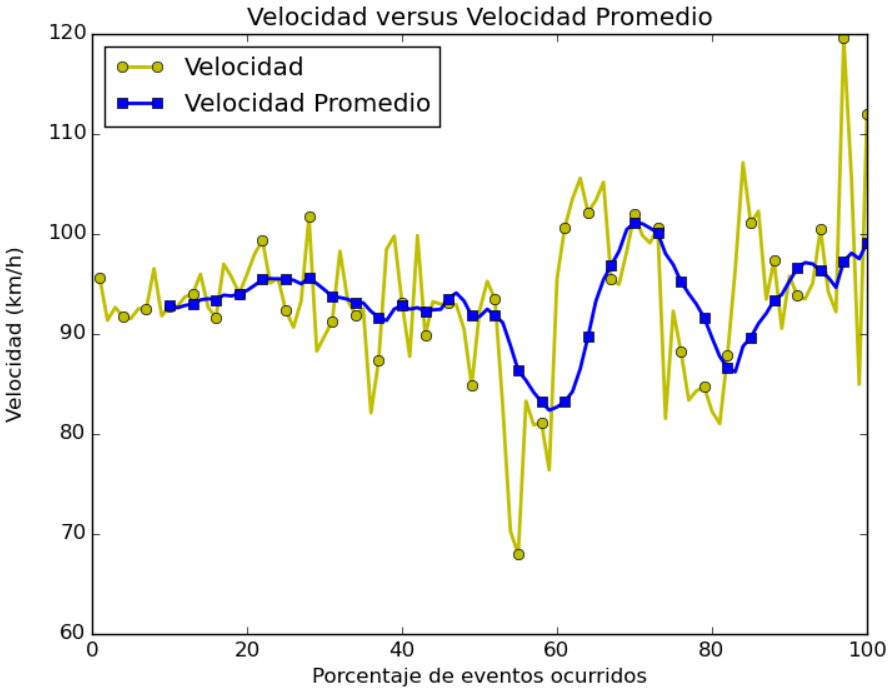


Figura 7.2: Gráfico de la velocidad de la totalidad de eventos procesados

En la Figura 7.2 un total de 639.114 eventos procesados muestran una comparación entre la velocidad medida y la media móvil (usando ventanas de tiempo de tamaño igual a 500 eventos). Es esperable que alarmas y eventos complejos se generen ante alzas o bajas notorias de la velocidad respecto de la media móvil debido al mecanismo de detección definido. De acuerdo a los niveles de severidad definidos para las alarmas en la Tabla 6.1, una alarma normal se considera entre un 10% y un 20% de variación respecto de la media móvil, lo que se traduce en las puntas cercanas a la línea azul, por otro lado, las alarmas de nivel crítico (entre un 20% y un 50% de variación) se observan como la cima al 40% de eventos ocurridos en el gráfico, y finalmente, las alarmas generadas con un nivel de severidad excepcional (por sobre 50% de variación), se visualizan como las cumbres al 60% y 100% de los eventos ocurridos, donde la velocidad registrada difiere mucho de la media móvil para la ventana de tiempo considerada. Estos eventos complejos son procesados por los agentes CEP como se muestra en la Figura 6.1.

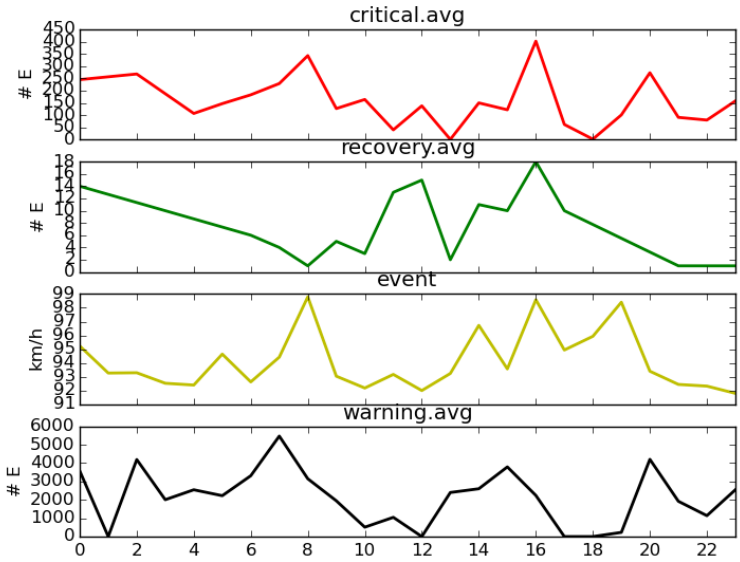


Figura 7.3: Notificaciones de alarmas según variación de velocidad para el sensor 1 (de arriba a abajo: 1. critical, 2. recovery, 3. velocidad, 4. warning)

Así como se menciona anteriormente, en la Figura 7.3, se observa la consistencia de los eventos generados (para el sensor número 1). Un alza en los eventos de alerta, significa un cambio considerable de la velocidad de los vehículos, lo que fue detectado en tiempo real usando el controlador CEP como se aprecia en la Figura 7.1.

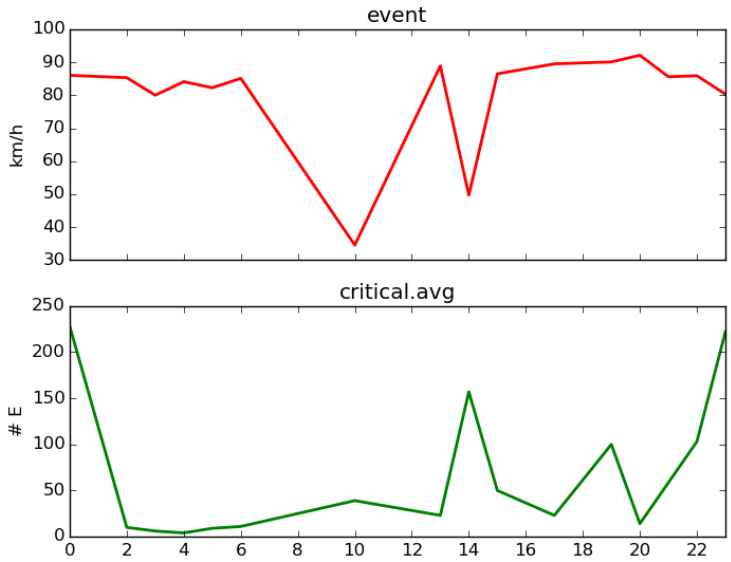


Figura 7.4: Notificaciones de alarmas según velocidad para el sensor 17 (alarma nivel bajo descartadas) (de arriba a abajo: 1. velocidad, 2. critical)

En la Figura 7.4 los niveles bajos de alarma para el sensor 17 fueron descartados para visualizar mejor lo que acontece, mostrando la relación entre la variación de velocidad y la cantidad de notificaciones de alarmas. Así, si la variación es baja, la cantidad de notificaciones también y viceversa.

Finalmente, la Figura 7.5 muestra la media móvil como un punto de comparación extra. Considerando que todos los casos se comportan de forma similar al total (Figura 7.2), este gráfico en particular muestra como el procesador CEP fue capaz de detectar problemas de tránsito bajo valores promedio normales (dentro de una ventana de tiempo).

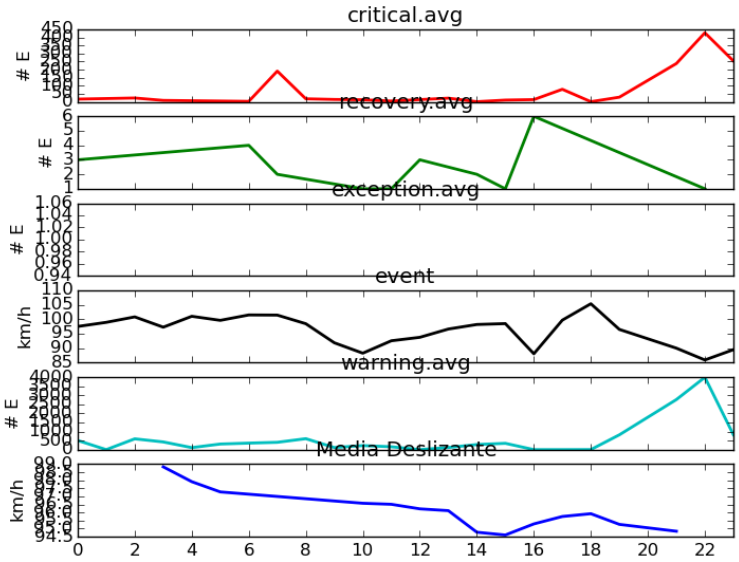


Figura 7.5: Notificaciones de eventos según velocidad para el sensor 2 (de arriba a abajo: 1. critical, 2. recovery, 3.exception, 4. velocidad, 5. warning, 6. media móvil)

Tal como se aprecia en el Gráfico 7.5 para el sensor 2, en el Gráfico 7.6 se observa las notificaciones generadas para el sensor número 3. Es más, la relación entre la variación de la velocidad y la cantidad de alarmas generadas se nota de forma más clara, es así como en los intervalos de horarios de las 7:00 - 10:00, 13:00 - 15:00, 17:00 - 19:00 y finalmente 20:00 - 21:00 se observa que la velocidad reportada por el sensor varía a la baja coincidiendo con la cantidad de alarmas de nivel *warning* que se generan, lo que deja ver la efectividad del prototipo implementado procesando los eventos a medida que llegan y generando nuevos eventos en tiempo real acorde a lo que sucede dentro de la ventana de tiempo implementada (ver Algortimo 1). Los mecanismos de procesamiento de eventos complejos utilizados, como la media móvil y la ventana de tiempo deslizante generan alertas frente a anomalías en el caso de estudio validado.

Por otro lado, los horarios en que se generan mayores cantidades de alarmas coinciden con los horarios punta del transporte y reflejan la efectividad de los mecanismos de detección implementados.

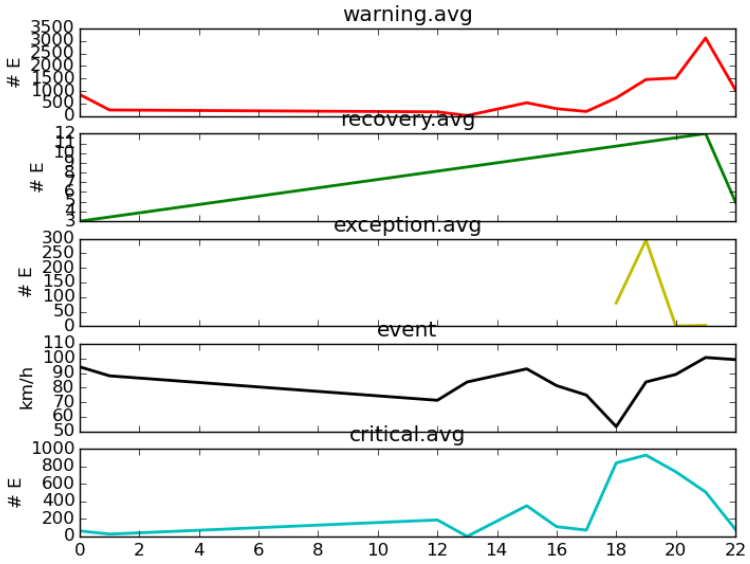


Figura 7.6: Notificaciones de eventos según velocidad para el sensor 3 (de arriba a abajo: 1. warning, 2. recovery, 3. exception, 4. velocidad, 5. critical)

7.3 Reconocimiento de patrones de eventos complejos

La Figura 7.7 muestra en primer lugar los eventos generados por el agregador, tal como se describe en la Figura 6.3 donde se observa la correlación entre las alarmas y los eventos complejos generados. De esta forma, se observa para el rango de tiempo entre las 7:00 y las 10:00 gran cantidad de alarmas generadas y en promedio entre 3 y 4 eventos agregados que consideran los criterios de correlación física, temporal y de anomalías de tráfico registradas. De noche (entre las 23:00 y las 7:00 horas) no hay alarmas generadas.

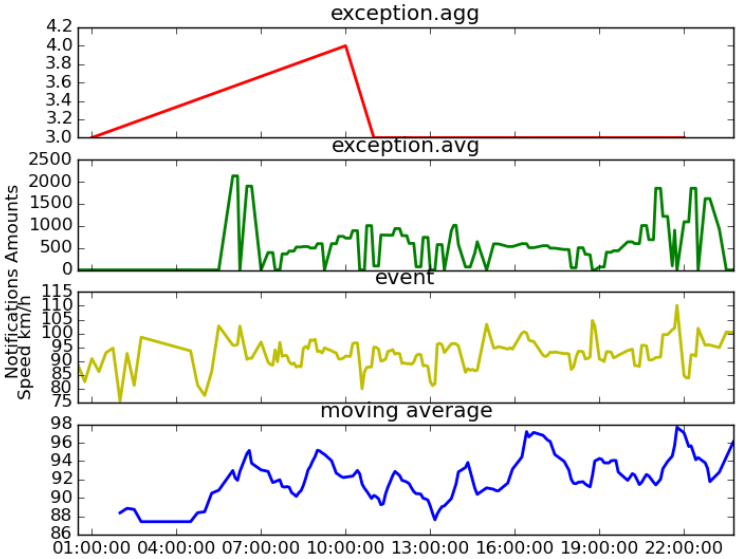


Figura 7.7: Eventos agregados versus alarmas para el Sensor 1 (de arriba a abajo: 1. aggregated, 2. exception, 3. speed, 4. media móvil)

Es importante considerar que todos los eventos que se muestran en la Figura 7.7 fueron emitidos en tiempo real, es decir, a medida que se recibían los eventos desde los sensores, los agregadores y el mecanismo de búsqueda de patrones implementado a través del enrutamiento de los mensajes, fueron capaces de detectar anomalías.

En la Figura 7.8 se observa el comportamiento oscilante respecto de las excepciones para el sensor 18. Las alarmas de las anomalías son detectadas en tiempo real, es decir, cuando un evento gatilla un evento agregado de acuerdo a los niveles de severidad definidos, el procesador CEP recibe la alarma y utiliza la media móvil y la ventana de tiempo deslizante para determinar si es necesario generar una alarma.

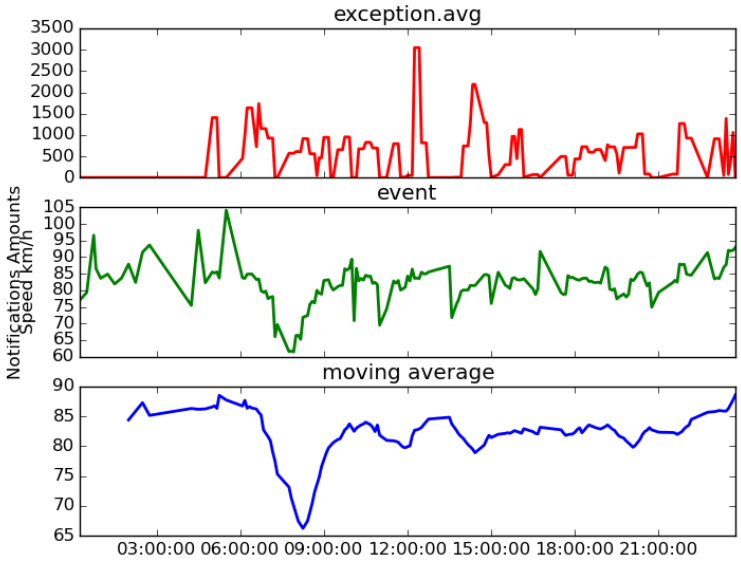


Figura 7.8: Eventos agregados versus alarmas para el Sensor 18 (de arriba a abajo: 1. exception, 2. speed, 3. media móvil)

Los eventos agregados que se muestran en las Figuras 7.7 y 7.8 son resultados del procesamiento de los eventos provenientes en primer lugar del controlador implementado que luego pasan a los agentes CEP, cada uno de estos agentes procesa los eventos a medida que van llegando (ver Figura 6.1) y los clasifica según su severidad de acuerdo a los niveles descritos en la Tabla 6.1, así, si dentro de las acciones está incluido el agregador, entonces el agente CEP envía un evento procesado, agregando datos adicionales como el nivel de severidad, las acciones a seguir y las estampillas de tiempo. Finalmente, cuando son recibidos por el agregador, los eventos ya clasificados son sometidos a evaluación dentro de una ventana de tiempo deslizante que implementa el agregador buscando un patrón, en este caso, correlación de eventos con sensores tal como se observa en la Figura 6.3.

Los eventos complejos generados por los Agentes CEP indican alarmas sobre niveles de velocidad respecto de la media móvil y de parámetros mínimos y máximos a medida que reciben los eventos y el agregador, evalúa sobre una ventana de tiempo buscando patrones anómalos. Ambos utilizan los algoritmos descritos en la Sección 6.2.2.

Finalmente, el agregador es un componente adicional que permite buscar patrones de eventos e integrar con notificaciones o eventos de fuentes externas para su correlación definiendo los patrones, niveles de severidad y reglas adecuadas dependiendo el escenario a enfrentar. En este estudio en particular, los parámetros se definieron para detectar anomalías en conductas viales como lo son congestión y excesos de velocidad.

Capítulo 8

Conclusiones

Comparación entre sistemas de colas de mensaje

Ambas tecnologías comparadas en la Sección 5.6 muestran un alto rendimiento, además, son capaces de procesar miles de mensaje por segundo y cuentan con la capacidad de filtrado de éstos. Es posible configurar distintas topologías con las tecnologías evaluadas, siendo fácilmente aplicable a un sistema basado en sensores como el principal caso de estudio en esta tesis.

RabbitMQ tiene incorporado el filtrado y la capacidad de canalizar mensajes, por otro lado, para ZeroMQ fue necesario implementar una cola intermedia para filtrar y dirigir mensajes.

Las pruebas para medir los límites de operaciones sobre disco descritas en la Sección 5, resultaron iguales para ambas tecnologías (con el mismo límite), lo cual es lógico, ya que ambos casos se corrieron sobre el mismo ambiente de desarrollo.

El umbral de degradación de ZeroMQ es más alto que el experimentado con RabbitMQ (rendimiento), lo que implica que tiene un mejor rendimiento y es rápido. Además, ZeroMQ resulta ser un 22.2% más eficiente en cuanto a rendimiento, es decir, se acerca un 22% más al rendimiento óptimo respecto de RabbitMQ.

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

Arquitectura para sistema basado en sensores

La arquitectura de referencia propuesta en la Sección 4.1 en esta tesis cumple con los principios fundamentales de las Arquitecturas Orientadas a Eventos, usando patrones de diseños de Bruns y Dunkel [4]. La Figura 7.1 muestra la arquitectura de referencia usando ZeroMQ.

Un sistema basado en sensores de tráfico fue implementado como caso de estudio procesando un flujo de eventos provenientes de múltiples clientes generando a su vez eventos complejos usando "Agregación de Eventos" y "Búsqueda de patrones", permitiendo la detección de incidentes en ruta, congestión y alarma cuando vehículos sobrepasan límites considerados seguros o incluso vehículos detenidos por avería u otros motivos.

Las capacidades intrínsecas de un sistema de colas de mensaje como ZeroMQ se ven puestas a prueba en un ambiente orientado a eventos, donde facilita el diseño e implementación de la arquitectura.

La Figura 7.5 muestra cómo el sistema es capaz de detectar condiciones de tránsito anómalas usando patrones de eventos. Es más, las notificaciones generadas fueron calculadas en tiempo real durante la ejecución a través del procesador de eventos complejos (CEP), tomando las ventajas de una comunicación asíncrona, el calce de patrones, la persistencia de datos y el controlador fueron ejecutadas de forma separada y en paralelo, compartiendo el canal de comunicación.

Verificación de Hipótesis

A continuación se detalla los resultados respecto de las dos hipótesis planteadas en este trabajo:

1. Como se comprueba en el Capítulo 5, la arquitectura orientada a eventos utilizando el patrón publicador-subscriptor implementado con ZeroMQ es un 22.2% más eficiente en cuanto a rendimiento (utilizando rendimiento relativo) y respecto de escalabilidad, el nivel de degradación del sistema es un 20% superior lo que significa que tiene mayor rendimiento (*throughput*, una de las características de escalabilidad).
2. El caso de estudio descrito en este trabajo considera detección de patrones de eventos complejos utilizando los mecanismos de enrutamiento y ventana de tiempo deslizante mediante un procesador de eventos complejos subscrito a las alarmas generadas en tiempo real tal como se comprueba en la Sección 7.3. El prototipo implementado de la arquitectura propuesta en el Capítulo 4 fue capaz de detectar anomalías viales en tiempo real sin acceder a los datos históricos, es más, el controlador paraleliza la tarea de almacenar los datos históricos y el procesamiento de eventos en tiempo real despachando los eventos entrantes en primer lugar al componente de almacenamiento y por otro lado, a los agentes CEP conectados.

Contribución y Trabajo Futuro

Esta tesis presenta las siguientes contribuciones dentro de los sistemas orientados a eventos:

1. Resultados de simulación y comparación entre dos sistemas de colas de mensajes en un entorno orientado a eventos.
2. Entrega una arquitectura de referencia orientada a eventos para sistemas basados en sensores.
3. Arquitectura propuesta aplicada en un caso de estudio real referente a un sistema de control de tráfico.
4. Amplía el conocimiento en el campo de Arquitecturas Orientadas a Eventos con un caso de estudio implementado.

Como trabajo futuro, se propone realizar pruebas de latencia (escalabilidad) bajo múltiples clientes conectados, considerando las pruebas de *throughput* realizadas en este trabajo.

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.

[2] Yahya Asadollahi et al. "A formal framework to model and validate event-based software architecture". In: *Procedia Computer Science* 3.0 (2011). World Conference on Information Technology, pp. 961 –966. ISSN: 1877-0509. DOI: 10.1016/j.procs.2010.12.157.

[3] 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.

[4] 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.

[5] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Chichester, UK: Wiley, 2007. ISBN: 978-0-470-05902-9.

[6] 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.

[7] Min Chen. "Towards smart city: M2M communications with software agent intelligence". English. In: *Multimedia Tools and Applications* 67.1 (2013), pp. 167–178. ISSN: 1380-7501. DOI: 10.1007/s11042-012-1013-4.

[8] Jürgen Dunkel et al. "Event-driven architecture for decision support in traffic management systems". In: *Expert Systems with Applications* 38.6 (2011), pp. 6530 –6539. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2010.11.087.

[9] AnnMarie Ericsson and Mikael Berndtsson. “REX, the Rule and Event eXplorer”. In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*. DEBS ’07. ACM, 2007, pp. 71–74. ISBN: 978-1-59593-665-3. DOI: 10.1145/1266894.1266906.

[10] N. Estrada and H. Astudillo. “Comparing scalability of message queue system: ZeroMQ vs RabbitMQ”. In: *Computing Conference (CLEI), 2015 Latin American*. 2015, pp. 1–6. DOI: 10.1109/CLEI.2015.7360036.

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

[12] Alessandro Di Giorgio and Laura Pimpinella. “An event driven Smart Home Controller enabling consumer economic saving and automated Demand Side Management”. In: *Applied Energy* 96.0 (2012). Smart Grids, pp. 92 –103. ISSN: 0306-2619. DOI: 10.1016/j.apenergy.2012.02.024.

[13] 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.

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

[15] Pieter Hintjens. *0MQ - The Guide - 0MQ - The Guide*. zguide.zeromq.org. (Accessed on 03/06/2016).

[16] Gregor Hohpe. *Programming Without a Call Stack – Event-driven Architectures*. 2006.

[17] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683.

[18] Chien-Chih Hsu and I.-Chen Wu. “An event-driven framework for inter-user communication applications”. In: *Information and Software Technology* 48.7 (2006), pp. 471 –483. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2005.05.005.

[19] Matjaz B. Juric. “WSDL and BPEL extensions for Event Driven Architecture.” In: *Information and Software Technology* 52.10 (2010), pp. 1023–1043.

[20] 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.

[21] David C. Luckham. *Event processing for business : organizing the real-time enterprise*. Hoboken, N.J. John Wiley & Sons, 2012. ISBN: 978-0-470-53485-4.

[22] H. Obweger et al. “Complex Event Processing off the shelf - Rapid development of event-driven applications with solution templates”. In: *Control Automation (MED), 2011 19th Mediterranean Conference on*. 2011, pp. 631–638. DOI: 10.1109/MED.2011.5983141.

[23] Nikos Papageorgiou et al. “Event-driven adaptive collaboration using semantically-enriched patterns”. In: *Expert Systems with Applications* 38.12 (2011), pp. 15409 –15424. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2011.06.020.

[24] Ioannis Patiniotakis et al. “Dynamic event subscriptions in distributed event based architectures”. In: *Expert Systems with Applications* 40.6 (2013), pp. 1935 –1946. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2012.10.005.

[25] M. Rostanski, K. Grochla, and A. Seman. “Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ”. In: *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*. 2014, pp. 879–884. DOI: 10.15439/2014F48.

[26] Jeffrey M. Squyres. “The Archicture of Open Source Applications”. In: vol. ii. Self published, 2012. Chap. 1.

[27] 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.

[28] 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.

[29] Thomas Usländer et al. “Designing environmental software applications based upon an open sensor service architecture”. In: *Environmental Modelling & Software* 25.9 (2010). Thematic issue on Sensors and the Environment – Modelling & {ICT} challenges, pp. 977 –987. ISSN: 1364-8152. DOI: 10.1016/j.envsoft.2010.03.013.

[30] S. Vinoski. “Advanced Message Queuing Protocol”. In: *IEEE Internet Computing* 10.6 (2006), pp. 87–89. ISSN: 1089-7801. DOI: 10.1109/MIC.2006.116.

[31] Agnès Voisard and Holger Ziekow. “ARCHITECT: A layered framework for classifying technologies of event-based systems”. In: *Information Systems* 36.6 (2011), pp. 937 –957. ISSN: 0306-4379. DOI: 10.1016/j.is.2011.03.006.

[32] 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.

[33] Di Wang et al. “Active Complex Event Processing infrastructure: Monitoring and reacting to event streams”. In: *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*. 2011, pp. 249–254. DOI: 10.1109/ICDEW.2011.5767635.

[34] Matt Welsh, David Culler, and Eric Brewer. “SEDA: an architecture for well-conditioned, scalable internet services”. In: *SIGOPS Oper. Syst. Rev.* 35.5 (Oct. 2001), pp. 230–243. ISSN: 0163-5980. DOI: 10.1145/502059.502057.

[35] I. Zappia, D. Parlanti, and F. Paganelli. “LiSEP: A Lightweight and Extensible Tool for Complex Event Processing”. In: *Services Computing (SCC), 2011 IEEE International Conference on*. 2011, pp. 701–708. DOI: 10.1109/SCC.2011.63.