Universidad Nacional del Litoral

Proyecto Final de Carrera

Diseño de un sistema de detección de anomalías en redes de computadoras.

Informe de avance 3

Pineda Leandro
Córdoba
1 de marzo de 2018

1. Introducción

2. Comunicación entre componentes

La arquitectura de un sistema es la descripción de su estructura en términos de componentes específicos y sus interrelaciones. Mediante esta modularización, podemos asegurarnos que dicha estructura satisface las demandas actuales (es decir, resuelve el problema para el cual fue construido) y puede ser adaptada para satisfacer demandas futuras. Podemos definir a los sistemas distribuidos como aquellos compuestos por varios componentes que no comparten el mismo espacio de memoria[1]. Cuando se diseñan sistemas distribuidos es necesario considerar lo siguiente:

- ¿Cuales son las entidades que se comunican entre si?
- ¿Cómo van a comunicarse, o para ser mas específicos, que paradigma de comunicación va a usarse?

Estas preguntas son centrales para entender los sistemas distribuidos; qué se está comunicando y cómo esas entidades se comunican entre si, definen una gran cantidad de variables a ser consideradas por quienes construyen estos sistemas.

Las entidades que se comunican en un sistema distribuido son típicamente procesos, lo que nos permite entender a los sistemas distribuidos como procesos que se relacionan mediante los paradigmas de comunicación *entre procesos* apropiados. Podemos nombrar tres paradigmas de comunicación:

- Comunicación *entre procesos*.
- Invocación remota.
- Comunicación indirecta.

Comunicación entre procesos La comunicación *entre procesos* refiere al soporte de bajo nivel para la comunicación entre procesos en sistemas distribuidos, incluyendo primitivas para manejo de mensajes, acceso directo a las API provistas por protocolos de Internet (esto es, usando Sockets) y soporte para comunicación *multicast*.

Para comunicarse, un proceso envía un mensaje (una secuencia de bytes) a un receptor y un procesos ejecutandose allí recibe el mensaje. Esta actividad involucra el pasaje de datos de un proceso emisor a un proceso receptor y puede significar la sincronización de ambos procesos.

Invocación remota La invocación remota representa el paradigma de comunicación más común en sistemas distribuidos. El intercambio de mensajes entre las entidades comunicantes es bidireccional, de forma que operaciones remotas, procedimientos y métodos, pueden ser invocados como se define a continuación:

- Protocolos *request-reply*: estos protocolos involucran el intercambio de mensajes desde el cliente al servidor y luego del servidor al cliente, donde el primer mensaje representa la operación que será ejecutada en el servidor (con los parámetros necesarios) y el segundo contiene cualquier resultado de dicha operación. Este paradigma es mas bien primitivo, y es utilizado generalmente en sistemas embebidos donde la *performance* es de suma importancia.
- *Remote procedure calls* (RPC) o llamadas a procedimientos remotos: este concepto, atribuido inicialmente a Birrel and Nelson [1984], representó un gran cambio en los paradigmas de computación distribuida. En RPC, los procedimientos de los procesos ejecutandose en computadoras remotas pueden ser invocados como si se encontraran el espacio local de memoria. De esta manera, el sistema abstrae aspectos acerca de la distribución, como la codificación de los parámetros y resultados y mecanismo de pasaje mensajes. Este esquema soporta comunicación cliente-servidor pero depende de servidores que ofrezcan un conjunto de operaciones a través de una interfaz de servicio para que los clientes puedan llamar esas operaciones como si estuviesen disponibles localmente.

- *Remote method invocation* (RMI) o invocación remota de métodos: RMI es similar a RPC pero utiliza objetos distribuidos. Bajo este paradigma, un objeto cliente puede invocar métodos de un objeto remoto. De la misma forma que con RPC, ciertos detalles de como se implementa la comunicación quedan ocultos al usuario. Algunas implementaciones de RMI pueden incluir, además, soporte para darle a los objetos identidad y la habilidad de usar esos identificadores de objetos en llamadas remotas.

Comunicación indirecta Las técnicas discutidas hasta aquí tienen una cosa en común: la comunicación representa una relación en ambos sentidos entre el emisor y receptor, con los emisores enviando explícitamente mensajes/invocaciones a los receptores asociados. Los receptores generalmente deben saber sobre la identidad de los emisores y, en la mayoría de los casos, ambas partes deben existir al mismo tiempo para que la comunicación sea exitosa. Lo descrito anteriormente no puede garantizarse en ciertos escenarios. Por esto, surgieron numerosas técnicas donde la comunicación es indirecta a través de una tercera entidad, permitiendo un gran grado de desacople entre emisores y receptores. En particular:

- Los emisores no necesitan saber a quien le están enviando datos.
- Emisores y receptores no necesitan existir al mismo tiempo.

Las técnicas más usadas para comunicación indirecta incluyen:

- Sistemas *publish-suscribe*: en estos sistemas, un gran número de productores (o *publishers*) distribuyen eventos (elementos de información de interés) a un número similar de consumidores (o *suscribers*). Usar cualquier de los paradigmas discutidos anteriormente hubiera sido complejo e ineficiente y por lo tanto los sistemas *publish-suscribe* (a veces llamados sistemas basados en eventos) surgieron para cubrir esta demanda[1]. Todos los sistemas *publish-suscribe* comparten la característica crucial de proveer un servicio intermedio que asegura que la información generada por los productores es enrutada eficientemente a los consumidores que deseen dicha información.

- Colas de mensajes: de la misma forma que los sistemas *publish-suscribe* proveen un estilo de comunicación uno a muchos, las colas de mensajes ofrecen un servicio punto a punto mediante el cual los procesos de los productores pueden enviar mensajes a una cola especifica y los procesos consumidores pueden recibir los mensajes o ser notificados de la llegada de nuevos mensajes a la cola. Las colas, entonces, ofrecen una indirección entre los procesos productores y consumidores.

En la actualidad, existen numerosas tecnologías que permiten implementar comunicación indirecta en arquitecturas distribuidas. Se consideraron las siguientes alternativas:

- RabbitMQ: es un *message broker* de proposito general, sólido y maduro, que soporta la mayoría de los protocolos estándares como AMQP. Su diseño de centra en entrega consistente de mensaje a consumidores que leen mensajes a una velocidad similar a la que el *message broker* monitorea su estado.

- Kafka: está diseñado para procesar grandes volumenes de mensajes tipo *publish-suscribe* y *streams*. Pensado para ser durable, rápido y escalable. Provee persistencia de mensajes y se ejecuta en cluster de servidores que almacenan flujos de eventos en categorías llamadas *tópicos*.

- ActiveMQ: es similar a Kafka pero a diferencia de este, los *message brokers* son los responsables de mantener el mensaje hasta que los consumidores lo hayan procesado. Esto hace que los *brokers* sean un poco más complejos y degrada la perfomance a medida que los el número de consumidores aumenta.

- Mosquitto: es una implementación de alta perfomance para el protocolo MQTT. Si bien está pensado para aplicaciones IoT, sus thoughput para mensajes de poco tamaño es muy buena.

Actualmente existen 3 protocolos ampliamente aceptados (AMQP, MQTT y STOMP), que son soportados por las alternativas descritas anteriormente. Sin embargo, Mosquitto es el único diseñado exclusivamente como *message broker* para MQTT. Además, las otras soluciones brindan una serie de características que son muy buenas pero no son necesarias para esta implementación, lo cual agregaría innecesariamente complejidad y demanda de recursos de procesamiento.

Dada la introducción sobre los aspectos de la comunicación de los diferentes componentes en sistemas distribuidos, estamos en condiciones de analizar los cambios realizados en la arquitectura del sistema con el objetivo de incrementar sus prestaciones en cuanto a *performance*.

2.1. Implementación previa del sistema: REST API

En la implementación inicial se había optado por registrar la ocurrencia de eventos usando una interfaz REST API, mayormente por la sencillez de su implementación y el amplio soporte y adopción que tiene esta tecnología en el mundo de los *webservices*. Sin embargo, se observa que cuando se intentan procesar cantidades masivas de eventos en tiempo real, el sistema evidencia una *performance* pobre dada la naturaleza síncrona de la comunicación.

Los resultados de las pruebas que serán mostradas a continuación, se realizaron en un sistema con las siguientes característica: procesador AMD FX6300 a 2,5 GHz, 8GB de memoria RAM, sistema operativo *Ubuntu Linux 16.04*, kernel *GNU/Linux 4.13.0.32-generic*.

Para determinar el volumen real de eventos que el sistema puede manejar, se generaron datos sintéticos y se registraron en el sistema de manera consecutiva por un lapso de 10 minutos, sin espera entre la registración de cada evento. El gráfico 1 muestra la cantidad de eventos procesados a lo largo del tiempo.

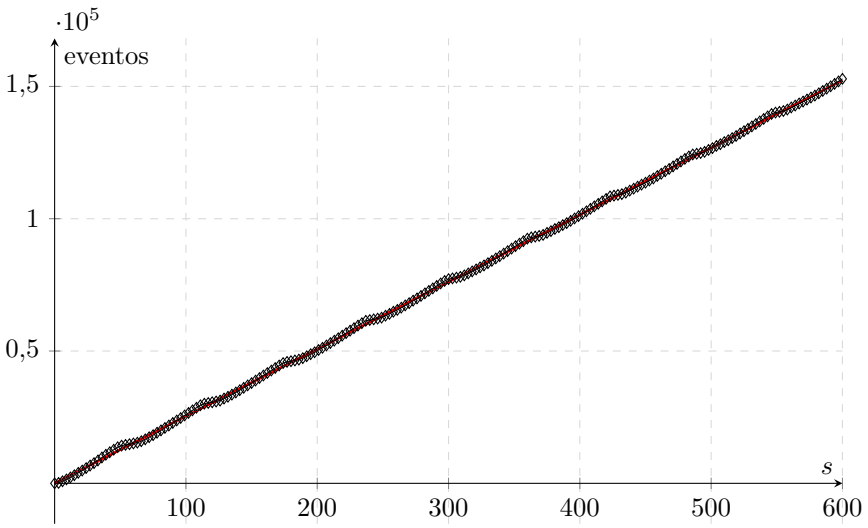

Figura 1: Eventos procesados a lo largo del experimento (REST API).

La recta que mejor ajusta los datos está dada por la ecuación $y = 253{,}71x + 32{,}1$. La pendiente de la recta sirve entonces para estimar la performance de la implementación REST del sistema en alrededor de 250 eventos por segundo.

2.1.1. Nueva implementación: Messaging Queue

Para esta nueva implementación es necesario un nuevo componente en la arquitectura del sistema. Se agregó una instancia de un message queue y se hicieron las modificaciones necesarias para que el webservice se suscriba a una cola determinada y procese los eventos que son allí publicados: esto es cambiar la interfaz para la creación de nuevos eventos e implementar los callbacks necesarios.

Las pruebas realizadas son similares a las presentadas anteriormente. Se generaron eventos aleatorios y se publicaron en un canal determinado de la cola de mensajes. Usando la cantidad de eventos que el sistema procesa a intervalos regulares de tiempo, se confecciono la gráfica 2. A

simple vista se puede observar un aumento pronunciado en la performance del sistema. La recta que ajusta a los datos está dada por la ecuación $y = 6907{,}2x - 48449$, por lo que podemos estimar un volumen de procesamiento de alrededor de 6900 eventos por segundo. La única desventaja de este método de envío de mensajes es que requiere que los clientes publiquen los mensajes en un canal dado, haciendo uso de las librerías correspondientes para su implementación.

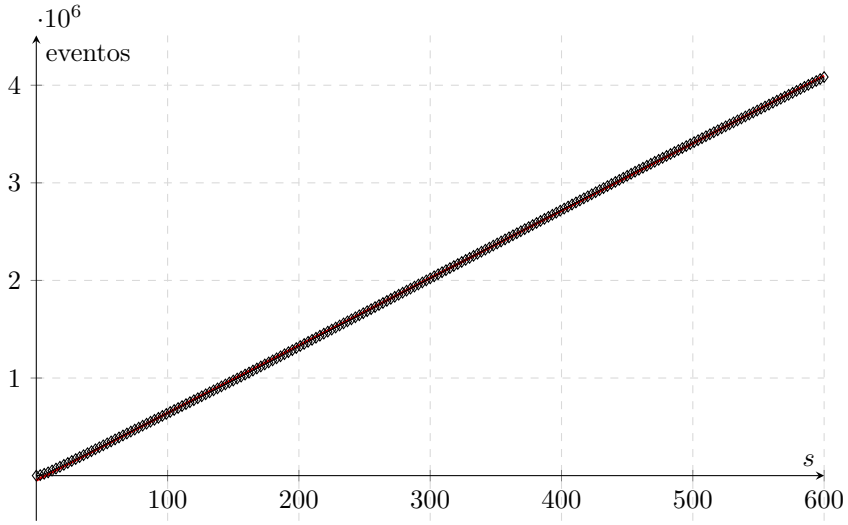

Figura 2: Eventos procesados a lo largo del experimento (Message Queue).

Con la implementación actual, podemos observar que la performance del sistema es 27 veces mejor que la implementación del segundo incremento. Aunque la primera implica el despliegue de un servicio de cola de mensajes y escribir clientes que publiquen mensajes en la misma, la ganancia en prestaciones es lo suficientemente buena como para que el beneficio justifique los costos de desarrollo.

3. Cambios en la arquitectura

Los cambios en el código del micro servicios que fueron realizados en este incremente, pueden agruparse en 3 categorías:

- Incrementar la performance del sistema.
- Mejorar la legibilidad del código (refactoring).
- Exponer la información necesaria para implementar una interfaz web.

3.1. Incremento en la performance del sistema

Para este cambio las modificaciones en el micro servicio no implicaron cambios muy grandes a el código del segundo incremento dado que la interfaz para registrar eventos estaban bien definidas. Como puede verse en el caso de uso descrito en el segundo informe (ver 3), los eventos nuevos son manejados por el componente *SketchManager*.

En lugar de ser enviados directamente por un actor, el componente está suscrito a un canal particular de la cola de mensajes. Mediante un *callback*, cada vez que un evento nuevo es publicado, el componente es notificado y obtiene el mensaje del canal. Aquí, podemos clasificar el mecanismo de lectura de mensajes como *push*, es decir, un actor empuja eventos al servicio, y *pull*, donde el servicio obtiene los eventos de un origen de datos dado (en este caso, una cola de mensajes).

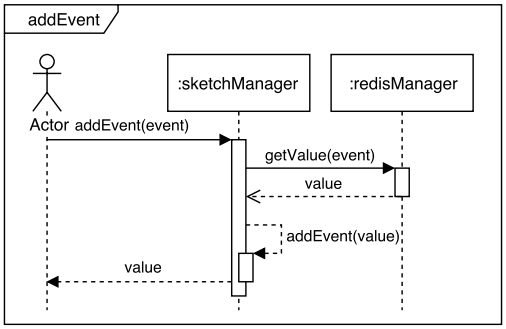

Figura 3: Diagrama de secuencia para nuevos eventos (segundo incremento)

Referencias

[1] G. Mühl, L. Fiege y P. Pietzuch, *Distributed Event-Based Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006, ISBN: 3540326510.

[2] G. Cormode y M. Hadjieleftheriou, «Finding Frequent Items in Data Streams», *Proc. VLDB Endow.*, vol. 1, n.º 2, págs. 1530-1541, ago. de 2008, ISSN: 2150-8097. DOI: 10.14778/1454159.1454225. dirección: http://dx.doi.org/10.14778/1454159.1454225.

[3] S. Muthukrishnan, «Data Streams: Algorithms and Applications», *Found. Trends Theor. Comput. Sci.*, vol. 1, n.º 2, págs. 117-236, ago. de 2005, ISSN: 1551-305X. DOI: 10.1561/0400000002. dirección: http://dx.doi.org/10.1561/0400000002.

[4] D. Terry, D. Goldberg, D. Nichols y B. Oki, «Continuous Queries over Append-only Databases», *SIGMOD Rec.*, vol. 21, n.º 2, págs. 321-330, jun. de 1992, ISSN: 0163-5808. DOI: 10.1145/141484.130333. dirección: http://doi.acm.org/10.1145/141484.130333.

[5] D. Tong y V. Prasanna, «High Throughput Sketch Based Online Heavy Hitter Detection on FPGA», *SIGARCH Comput. Archit. News*, vol. 43, n.º 4, págs. 70-75, abr. de 2016, ISSN: 0163-5964. DOI: 10.1145/2927964.2927977. dirección: http://doi.acm.org/10.1145/2927964.2927977.

[6] G. Cormode y M. Hadjieleftheriou, «Methods for Finding Frequent Items in Data Streams», *The VLDB Journal*, vol. 19, n.º 1, págs. 3-20, feb. de 2010, ISSN: 1066-8888. DOI: 10.1007/s00778-009-0172-z. dirección: http://dx.doi.org/10.1007/s00778-009-0172-z.

[7] L. J. Guibas, «Problems», *Journal of Algorithms*, vol. 2, n.º 2, págs. 208 -210, 1981, ISSN: 0196-6774. DOI: http://dx.doi.org/10.1016/0196-6774(81)90022-5. dirección: http://www.sciencedirect.com/science/article/pii/0196677481900225.

[8] R. M. Karp, S. Shenker y C. H. Papadimitriou, «A Simple Algorithm for Finding Frequent Elements in Streams and Bags», *ACM Trans. Database Syst.*, vol. 28, n.º 1, págs. 51-55, mar. de 2003, ISSN: 0362-5915. DOI: 10.1145/762471.762473. dirección: http://doi.acm.org/10.1145/762471.762473.

[9] E. Kranakis, P. Morin e Y. Tang, «Bounds for Frequency Estimation of Packet Streams», en *In SIROCCO*, 2003, págs. 33-42.

[10] E. M. Hutchins, M. J. Cloppert y R. M. Amin, «Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains», *Leading Issues in Information Warfare & Security Research*, vol. 1, pág. 80, 2011.

[11] A. Metwally, D. Agrawal y A. El Abbadi, «Efficient Computation of Frequent and Top-k Elements in Data Streams», en *Proceedings of the 10th International Conference on Database Theory*, ép. ICDT'05, Edinburgh, UK: Springer-Verlag, 2005, págs. 398-412, ISBN: 3-540-24288-0, 978-3-540-24288-8. DOI: 10.1007/978-3-540-30570-5_27. dirección: http://dx.doi.org/10.1007/978-3-540-30570-5_27.

[12] G. Cormode y S. Muthukrishnan, «An Improved Data Stream Summary: The Count-min Sketch and Its Applications», *J. Algorithms*, vol. 55, n.º 1, págs. 58-75, abr. de 2005, ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2003.12.001. dirección: http://dx.doi.org/10.1016/j.jalgor.2003.12.001.

[13] F. Putze, P. Sanders y J. Singler, «Cache-, Hash-, and Space-efficient Bloom Filters», *J. Exp. Algorithmics*, vol. 14, 4:4.4-4:4.18, ene. de 2010, ISSN: 1084-6654. DOI: 10.1145/1498698.1594230. dirección: http://doi.acm.org/10.1145/1498698.1594230.

[14] B. H. Bloom, «Space/Time Trade-offs in Hash Coding with Allowable Errors», *Commun. ACM*, vol. 13, n.º 7, págs. 422-426, jul. de 1970, ISSN: 0001-0782. DOI: 10.1145/362686.362692. dirección: http://doi.acm.org/10.1145/362686.362692.

[15] D. Crockford, *The application/json Media Type for JavaScript Object Notation (JSON)*, RFC 4627, jul. de 2006. DOI: 10.17487/RFC4627. dirección: https://rfc-editor.org/rfc/rfc4627.txt.