**Trabajo final de Licenciatura en Ciencias de la Computación**

**Streaming - Cola de mensajería basada en contenido**

****

**Universidad Nacional de San Luis**

**Facultad de Ciencias Físico Matemáticas y Naturales**

**Autor:** Jorge Federico Arito.

**Director:** Dr. Daniel Riesco.

**Co-Director:** Ms. Ariel Leiva.

Indice

1. **Introducción……………………………………………………………....4**

**1.1 Introducción…………………………………………………………...……….4**

**1.2 Generalidades de ingeniería de software………………………………....4**

**1.3 Generalidades de arquitectura de software……………………………....5**

**1.4 Streaming de datos…………………………………………….……………...5**

**1.5 Motivación……………………………………………………….……………...6**

**1.6 Objetivos……………………………………………………….………………..7**

1. **Conceptos Básicos…………………………………………………….10**

**2.1 Servicio Web………………………………………………….……………….10**

**2.2 REST………………………………………………………….………………...12**

**2.3 JSON………………………………………………………….………………...13**

**2.4 XML……………………………………………………………………………..15**

**2.5 APIs REST……………………………………………………….…………….18**

2.5.1 URIs………………………………………………………………………….....20

2.5.2 Tipos de peticiones HTTP……………………………………………….…...21

**2.6 Resiliencia……………………………………………………………………..23**

2.6.1 Factores que afectan a la resiliencia……………………………….…….…23

**2.7 Fault Tolerance………………………………………………………….…….24**

**2.8 High Availability**

1. **Arquitecturas de Software………………………………………..…..27**

**3.1 Patrones Arquitectónicos…………………………………………………..27**

**3.2 Arquitectura en Capas………………………………………………………28**

**3.3 Event-Driven Architecture………………………………………………….30**

3.3.1 Mediator Topology……………………………………………………....…….31

3.3.2 Broker Topology……………………………………………………………….34

**3.4 Consideraciones generales de Event-Driven…………………………...36**

**3.5 Sistema de colas basado en contenido………………………………….37**

3.5.1 Basado en Tópicos……………………………………………………....…...38

3.5.2 Basado en Contenido………………………………………………………...39

1. **Arquitectura y Aspectos del Sistema……………………………….41**

**4.1 Casos de uso funcionales………………………………………………….42**

**4.2 Casos de uso aspiracionales……………………………………………....53**

**4.3 Grails…………………………………………………………………………...55**

**4.4 Redis…………………………………………………………………………....56**

4.4.1 Redis Pub/Sub…………………………………………………………………..57

**4.5 Microservices Software Arq (UML)........................................................57**

4.5.1 Producer Layer API…………………..…………………..…………………..58

4.5.2 Consumer Layer …………………..…………………..……………………...60

4.5.3 Jobs Layer…………………..…………………..……………………………..62

4.5.4 Redis Layer…………………..…………………..……………………………62

4.5.5 Redis Retry Layer…………………..…………………..…………………….63

4.5.6 Subscribers Layer…………………..…………………..…………………….64

**4.6 Metrics………………………………………………………………………….65**

**4.7 Limits…………………………………………………………………………...68**

1. **Conclusión y Anexos…………………………………………………..70**

**5.1 Conclusion…………………………………………………………………….70**

**5.2 Anexos…………………………………………………………………………71**

5.2.1 Códigos de respuesta HTTP………………………………………………...71

5.2.2 Posibles mejoras que se pueden aplicar: brpoplpush…………………….72

5.2.3 Informacion para Analytics…………………………………………………...73

5.2.4 Chaos Engineering…………………………………………………………....74

1. **Bibliografia………………………………………………………………76**

**1. Introduccion**

**1.1 Introduccion**

Este proyecto de tesis comienza con una introducción a conceptos de la Ingeniería de Software [1] [2] como así también de las arquitecturas y patrones, para luego continuar con conceptos básicos que ayudarán, en conjunto con el entendimiento de distintos tipos de arquitecturas, a formular la tesis presentada como Streaming - Cola de mensajería basada en contenido.

**1.2 Generalidades de ingeniería de software**

La **ingeniería del software**, según la definición de la IEEE en 1993 [3], es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software. La ingeniería del software ofrece métodos o técnicas para desarrollar y mantener software de calidad que resuelven problemas de todo tipo (o lo intentan), y trata áreas muy diversas de la informática y de las ciencias computacionales.

Con el transcurso de los años se han desarrollado recursos que conforman la ingeniería del software, es decir, herramientas y técnicas de especificación, diseño e implementación del software: la programación estructurada, la programación orientada a objetos, las herramientas CASE, la documentación, los estándares, CORBA, los servicios web, el lenguaje UML, etc.

La ingeniería del software comprende:

* Proceso de desarrollo de software (especificación, implementación y diseño, etc…).
* Metodologías para el desarrollo de software (RUP, patrones, framework…).
* Herramientas de desarrollo de software.

Dentro de la ingeniería de software se abre un tema que es de vital importancia: la arquitectura del software.

**1.3 Generalidades de arquitectura de software**

La arquitectura de softwarepor otra parte consiste en el diseño de componentes de una aplicación (entidades del negocio), generalmente utilizando patrones de arquitectura. El diseño arquitectónico debe permitir visualizar la interacción entre las entidades del negocio y además poder ser validado, por ejemplo por medio de diagramas de secuencia. Un diseño arquitectónico describe en general cómo se construirá una aplicación de software. Para ello se documenta utilizando diagramas, por ejemplo:

* Diagrama de clases
* Diagrama de base de datos
* Diagrama de despliegue
* Diagrama de secuencia
* Diagrama de componentes

**1.4 Streaming de datos**

Dentro de las arquitecturas de software tenemos un vertical donde se va a enmarcar el contenido de esta tesis que es el **Streaming de datos de gran magnitud**.

Arquitecturas de software para Streaming de datos, son arquitecturas pensadas especialmente para transmisión masiva de datos. Una cantidad tan excesiva de datos que las aplicaciones de software que tradicionalmente usamos no son capaces de computar, tratar y poner en valor en un tiempo razonable.

**Streaming - Cola de mensajería basada en contenido**  surge como necesidad para resolver un problema puntual que es la comunicación de mensajes de diferentes tipos (topic queue) entre sistemas distribuidos permitiendo a los sistemas enviarlos (publicar) y a los interesados (suscriptores) recibirlos y además permitiendo a estos filtrar solo los que les pueden resultar relevantes (Content-Based) y todo mediante una interfaz REST.

*¿Qué es un sistema de streaming ?*

La retransmisión (en inglés *streaming)* es la distribución digital de contenido multimedia o texto a través de una red de computadoras, se refiere frecuentemente a una corriente continua que fluye sin interrupción, y habitualmente a la difusión de audio y video, en este sistema usado para transmitir mensajes en formato JSON (explicado más adelante).

*¿Qué sería un sistema de streaming basado en contenido?*

Un sistema de colas basado en contenidos es un sistema igual al descrito en el punto anterior, sumando la posibilidad de suscribirse a un tema particular o de interés, donde se pueda especificar ciertas condiciones que son las que a este le interesa que los mensajes cumplan. En cuanto un mensaje cumple las condiciones será enviado al sistema interesado, en caso de haber mensajes que no cumplen con las condiciones especificadas ese mensaje se descarta para el sistema en cuestión.

Esta mejora al sistema de colas tradicional, permite a los sistemas suscriptores evitar tener una arquitectura de software muy compleja dada que la carga que van a recibir es un porcentaje de la totalidad de mensajes publicados en un tópico particular. Otra mejora es que de esta manera al no enviar la totalidad de mensajes a todos los suscriptores, se reduce la carga que se le da a los recursos de networking.

**1.5 Motivacion**

Cuál fue la motivación de investigar e incursionar en la combinación de varias arquitecturas de software para lograr un sistema distribuido cuya responsabilidad es la propagación de mensajes a distintos destinos y de alta performance ?

Muchas veces estas motivaciones vienen dadas de distintos tipos de necesidades, ellas pueden venir desde los stakeholders del sistema, de esa manera un arquitecto tiene que empezar a evaluar en base al conocimiento que camino tomar, por otro lado algo que también influye mucho en la motivación, es el background de conocimiento con el que cuenta, que suele ser el que inclina la balanza por un camino o por el otro.

Más allá de lo antes dicho, la motivación más grande y que fue la determinante en la creación de este proyecto de tesis, es poder contar con un sistema único que permita recibir un mensaje en un tema particular, y que este mensaje pueda ser propagado a los interesados (suscriptores) en los mensajes de este tema (al estilo de un broadcast de red), además sumando la capacidad de que cada uno de los suscriptores pueda, de manera casi intuitiva, no solo recibir los mensajes del tema que le interesa sino también poder filtrar los mensajes que realmente le son de interés dentro de un tema en particular (puede estar interesado en un subconjunto más restrictivo) y siempre acompañando al posible crecimiento que puede tener el sistema, de manera que sea escalable a futuro y evitando generar inconsistencias en los datos que se transmiten a través de él. Esto último pensado en la responsabilidad de ser un “broadcast” de mensajes.

Obviamente este planteo como motivación luego se va a ir trabajando y ajustando más al detalle, ya que lograr esto, implica entender varios conceptos básicos de las arquitecturas de software, pilar donde se va a basar en gran parte los objetivos de este sistema, y por otro lado en conceptos de tecnologías donde a partir de determinados criterios se seleccionan las que son consideradas más adecuadas para acompañar la motivación de la tesis.

**1.6 Objetivos**

El objetivo principal del desarrollo de esta tesis está directamente relacionado con la motivación, al final del camino se quiere poder contar con un diseño de arquitectura de software que permita:

* Especificar un conjunto de tópicos donde se podrán publicar mensajes, estos tópicos serán colas de mensajes que deberán funcionar similar al modelo conocido como colas FIFO (primero en entrar, primero en salir). Y además,
* Permitir a las aplicaciones interesadas en un tema participar, suscribirse para empezar a recibir mensajes, con el agregado de poder especificar condiciones sobre ellos, que deberán cumplir para ser recibidos por el suscriptor.
* De los dos puntos anteriores se desprenden las acciones básicas que el sistema deberá permitir, que tiene que ver con la administración tanto de los tópicos como de los suscriptores, que son las acciones de : crear, borrar, consultar o modificar.

Además se quiere que el sistema final cuente con ciertas características de calidad, descritas a continuación:

* Agnóstico: es decir que el sistema no sepa qué mensajes o tópicos se transfieren a través de él, si que cumpla que el mensaje que llega se distribuya.
* Consistente: tanto desde el punto de vista de cantidad de mensajes como así también del contenido de los mensajes. De esta manera el sistema solo tiene que ser un “pasamano” de mensajes, y asegurando siempre que el mensaje que llega, es un mensaje que se transmite a los suscriptores interesados.
* Alta Disponibilidad: tiene que ser de alta disponibilidad, este es el pilar fundamental del sistema ya que el sistema en sí puede formar parte del camino crítico de las aplicaciones que lo utilizan.
* Alta performance: el sistema debe cumplir con los objetivos anteriores, pero debe hacerlo en tiempos aceptables, esto es para evitar que la mensajería no llegue a tiempo a destino.
* Agilidad: capacidad de responder rápidamente a un entorno en constante cambio. Dado que los componentes del sistema son de un solo propósito y están completamente desacoplados de otros componentes del sistema, los cambios generalmente están aislados para uno o varios procesadores de eventos y pueden realizarse rápidamente sin afectar a otros componentes.
* Facilidad de despliegue: esto tiene que ver con que por más complejo que termine siendo el sistema, este deberá ser de fácil administración en entornos productivos, seguramente esta característica va a influir mucho en las decisiones de arquitectura que se tomen.
* Testable: Si bien las pruebas de unidades individuales no son demasiado difíciles, sí requieren algún tipo de cliente de prueba especializado o herramienta de prueba para generar eventos.
* Escalable: junto con el objetivo de alta disponibilidad, es otro objetivo fundamental, a futuro el sistema debe comportarse de la misma manera tanto para grandes cargas de tráfico como para gran cantidad de tópicos y suscriptores. Seguramente también esta característica determinará gran porcentaje de las decisiones de arquitectura que se tomarán.
* Web: que el sistema sea pensado desde el momento cero para ser soportado en un entorno de web services.

Luego de conjugar los requerimientos externos y un poco del background propio, viene la etapa de análisis y pruebas de concepto. En este momento es cuando la investigación es clave. Para el caso, lo que se hizo fue pensar primero los casos de uso funcionales más relevantes como así también algunos casos de uso más aspiracionales en base a lo visto previamente.

**2. Conceptos Básicos**

Para poder continuar se debe establecer el contexto para algunos conceptos que son necesarios para el mejor entendimiento de la implementación que se aplicará.

Estos conceptos surgen de empezar a profundizar más en los objetivos y características deseables del sistema descritos en el punto anterior.

**2.1 Servicio Web**

El consorcio W3C define los Servicios Web como sistemas de software diseñados para soportar una interacción interoperable máquina a máquina sobre una red. Los Servicios Web suelen ser APIs Web que pueden ser accedidas dentro de una red (principalmente Internet) y son ejecutados en el sistema que los aloja.

La definición de Servicios Web propuesta alberga muchos tipos diferentes de sistemas, pero el caso común de uso se refiere a clientes y servidores que se comunican mediante mensajes XML que siguen el estándar SOAP.

*En los últimos años se ha popularizado un estilo de arquitectura Software conocido como REST (Representational State Transfer). Este nuevo estilo ha supuesto una nueva opción de forma de uso de los Servicios Web. A continuación se listan los tres estilos de usos más comunes:*

*•* Remote Procedure Calls (RPC, Llamadas a Procedimientos Remotos): Los Servicios Web basados en RPC presentan una interfaz de llamada a

procedimientos y funciones distribuidas, lo cual es familiar a muchos desarrolladores. Típicamente, la unidad básica de este tipo de servicios es la operación WSDL [4] (Web Services Description Language, es un formato del Extensible Markup Language que se utiliza para describir servicios web).

Las primeras herramientas para Servicios Web estaban centradas en esta visión. Algunos lo llaman la primera generación de Servicios Web. Esta es la razón por la que este estilo está muy extendido. Sin embargo, ha sido algunas veces criticado por no ser débilmente acoplado, ya que suele ser implementado por medio del mapeo de servicios directamente a funciones específicas del lenguaje o llamadas a métodos. Muchos especialistas creen que este estilo debe desaparecer.

*• Arquitectura Orientada a Servicios (Service-oriented Architecture, SOA).*

*La arquitectura orientada a servicios es un modelo arquitectónico de tecnología para soluciones orientadas a servicios con características distintas y con objetivos estratégicos asociados con la informática orientada a servicios. Pueden existir diferentes tipos de arquitectura orientada a servicios, dependiendo del alcance de su aplicación.*

Como forma de arquitectura tecnológica, una implementación de SOA puede consistir en una combinación de tecnologías, productos, APIs, y varias otras partes. La estructura real de una arquitectura orientada a servicios implementada, es única dentro de cada empresa; sin embargo, se caracteriza por la introducción de nuevas tecnologías y plataformas que apoyan específicamente la creación, ejecución y evolución de soluciones orientadas a servicios. Como resultado, la construcción de una arquitectura con el modelo SOA establece un entorno adecuado para la lógica de la solución que ha sido diseñado de acuerdo con los principios de diseño de orientación al servicio. SOA se define con la premisa de contar con características que deben existir para soportar la aplicación de principios de orientación a servicios, en lugar de restricciones formales como las del estilo de arquitectura REST [35].

• REST (REpresentational State Transfer - Transferencia de Estado Representacional). Los Servicios Web basados en REST intentan emular al protocolo HTTP o protocolos similares mediante la restricción de establecer la interfaz a un conjunto conocido de operaciones estándar (por ejemplo GET, PUT,...). Por tanto, este estilo se centra más en interactuar con recursos con estado, que con mensajes y operaciones.

REST [35] además proporciona el siguiente conjunto de "Propiedades arquitectónicas de interés" que ayudan a establecer los objetivos de diseño que se encuentran detrás de la aplicación REST:

* Performance
* Escalabilidad
* Sencillez
* Modificabilidad
* Visibilidad
* Portabilidad
* Fiabilidad

Ahora bien, una vez definido lo que es un servicio web y algunos de los posibles estilos para poder implementarlos, basándose en los objetivos que se plantean en el desarrollo de esta tesis, se decide optar para que el sistema sea basado en REST, porque a simple vista machea de manera casi trivial con los objetivos del mismo, sobre todo los relacionados a la administración de los tópicos y suscriptores. Veamos un poco más de detalle a continuación.

**2.2 REST**

La Transferencia de Estado Representacional(REST) es un estilo de arquitectura software para sistemas hipermedia (conjunto de métodos o procedimientos para escribir, diseñar o componer contenidos: texto, audio, video, etc.) distribuidos como la World Wide Web (www). El término se originó en el año 2000, en una tesis doctoral sobre la web, escrita por Roy Fielding, y ha pasado a ser ampliamente utilizado por la comunidad de desarrollo [4].

REST afirma que la web ha disfrutado de escalabilidad como resultado de una serie de diseños fundamentales:

* Un protocolo cliente/servidor sin estado: cada mensaje HTTP enviado por el cliente contiene toda la información necesaria para comprender la petición por parte del servidor. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes [35].
* Un conjunto de operaciones bien definidas que se aplican a todos los *recursos* de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son POST, GET, PUT y DELETE. Con frecuencia estas operaciones se equiparan a las operaciones CRUD (Crear, Leer, Actualizar y Borrar) en bases de datos, que se requieren para la persistencia de datos.
* Un conjunto de códigos de estados de respuesta universales que permiten rápidamente entender si la operación ejecutada fue exitosa o no, denominados ***http status codes***. Nosotros nos enfocaremos en los siguientes:
  + - 200: cuando la operación (por lo general GET o PUT) fue recibida, entendida y aceptada con éxito.
    - 201: cuando la operación está completa y se crea un nuevo recurso.
    - 204: cuando la operación proporciona un código de estado y un encabezado en la respuesta, pero no hay un cuerpo de entidad en la respuesta.
    - 400: cuando el servidor no entendió la solicitud.
    - 500: se presenta cuando la solicitud no fue completada. El servidor cumplió una condición inesperada (excepción).
* Una sintaxis universal para identificar los recursos. En un sistema REST, cada recurso es direccionable únicamente a través de su URI (Identificador de recursos uniforme).
* El uso de hipermedios, tanto para la información de la aplicación como para las transiciones de estado de la aplicación: la representación de este estado en un sistema REST son típicamente XML o JSON. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.

Ahora podemos establecer cuál será el protocolo del sistema, que será utilizado para el intercambio de información. Si bien ya se habló de protocolo, resta definir qué tipo de mensaje será utilizado en el mismo. En la actualidad en la industria del software, para servicios web basados en REST, los formatos de mensajes más utilizados son XML y JSON. A continuación se detallan las propiedades de cada uno de ellos.

**2.3 JSON**

El formato de datos JavaScript Object Notation [5] (JSON) permite que las aplicaciones se comuniquen a través de una red, generalmente a través de APIs REST. JSON es independiente de la tecnología, no propietario y portátil. Todos los lenguajes y plataformas modernos (por ejemplo, Java, JavaScript, Ruby, C#, PHP, Python y Groovy) y las plataformas brindan un excelente soporte para producir (serializar) y consumir (deserializar) datos JSON. JSON es simple: consta de construcciones amigables para el desarrollador como objetos, matrices y pares de nombre/valor. JSON no se limita a la transferencia de estado representacional (REST); también funciona con lo siguiente:

* Node.js (que almacena metadatos del proyecto en package.json)
* Bases de datos NoSQL como MongoDB
* Plataformas de mensajería como Kafka

En los primeros días, los detractores de REST se burlaban de los servicios web RESTful por no ser estándar, pero (al igual que HTTP) JSON es de hecho un estándar. Tanto el Grupo de Trabajo de Ingeniería de Internet (IETF) como Ecma International (antes Asociación Europea de Fabricantes de Computadoras, o ECMA) han reconocido a JSON como estándar. Douglas Crockford creó originalmente JSON en 2001 y lo estandarizó inicialmente en 2006 bajo RFC 4627 a través del IETF. En el otoño de 2013, Ecma International también estandarizó JSON bajo ECMA 404. Con el reconocimiento de Ecma (según Douglas Crockford), JSON ahora se considera un estándar de procesamiento de datos internacional formal.

Antes de continuar, veamos una pequeña muestra de JSON. El ejemplo a continuación muestra un documento JSON simple.

Ejemplo:

{ **"key"**: "Mi primer documento JSON" }

Un documento JSON válido puede ser cualquiera de los siguientes:

* Un objeto rodeado de llaves {y}
* Una matriz entre corchetes, [y]

El ejemplo anterior muestra un objeto que contiene un solo par clave / valor, donde la clave, "key", tiene como valor a "Mi primer documento JSON".

Aunque la estandarización a través de Ecma International y el IETF ha ayudado a JSON a ganar aceptación en la industria, otros factores han popularizado JSON:

* El crecimiento explosivo de las APIs RESTful (APIs que implementan como patrón arquitectónico a REST) basadas en JSON
* La simplicidad de las estructuras de datos básicas de JSON
* La creciente popularidad de JavaScript

Miles de API RESTful aprovechan JSON. Una lista de muestra de API RESTful basadas en JSON populares incluye lo siguiente:

* LinkedIn
* Twitter
* Facebook
* Salesforce
* GitHub
* DropBox
* Servicios web de Amazon (AWS)

Ahora que conocemos un poco más de JSON tenemos que hacer lo mismo con XML, y luego intentar establecer puntos de comparación para ver cuál será el formato de mensaje elegido para el sistema.

**2.4 XML**

XML (Extensible Markup Language) contiene, da una forma, etiqueta, estructura y protege la información [6]. Lo hace con símbolos incrustados en el texto, llamados marcador. El marcador mejora el significado de la información de determinadas formas, identificando las partes y cómo se relacionan entre sí. Por ejemplo, cuando lee un periódico, puede distinguir los artículos por su espacio y posición en la página y el uso de diferentes fuentes para títulos y encabezados. El marcador funciona de manera similar, excepto que en lugar de espacios y líneas, usa símbolos.

El marcador es importante para los documentos electrónicos porque son procesados ​​por programas de computadora. Si un documento no tiene etiquetas ni límites, entonces un programa no sabrá cómo distinguir un fragmento de texto de cualquier otro fragmento. Esencialmente, el programa tendría que trabajar con todo el documento como una unidad, limitando severamente las cosas interesantes que puede hacer con el contenido. Un periódico sin espacio entre artículos y con un solo estilo de texto sería una masa de texto enorme y poco interesante. Probablemente pueda averiguar dónde termina un artículo y comienza otro, pero sería mucho trabajo. Un programa de computadora no podría hacer ni siquiera eso, ya que carece de todas las habilidades para hacer coincidir patrones, excepto las más rudimentarias.

El marcador de XML divide un documento en contenedores de información separados llamados elementos. Sellan los datos por completo, los etiquetan y proporcionan un paquete conveniente para el procesamiento por computadora. Como las cajas, los elementos se anidan dentro de otros elementos. Un elemento grande puede contener un montón de elementos, que a su vez contienen otros elementos, y así sucesivamente hasta los datos. Esto crea una estructura jerárquica inequívoca que conserva todo tipo de información auxiliar: secuencia, propiedad, posición, descripción, asociación. Un documento XML consta de un elemento externo que contiene todos los demás elementos, además de información administrativa opcional en la parte superior.

Ejemplo:

**<key>**Mi primer documento XML**</key>**

¿Se nota la diferencia entre el marcador y los datos? Los símbolos de marcador están delimitados por corchetes angulares (<>). <key> es uno de esos símbolos, llamados etiquetas. Los datos, o el contenido, llenan el espacio entre estas etiquetas, en el ejemplo Mi primer documento XML. A medida que se acostumbre a mirar XML, usará las etiquetas como señales para navegar visualmente por los documentos.

En este ejemplo, se puede ver un patrón: las etiquetas funcionan como delimitadores, marcando el comienzo y el final de las regiones, mientras que otras marcan un lugar en el texto. Incluso el ejemplo anterior contiene bastante información:

* Límites: Un texto comienza en un lugar y termina en otro. Las etiquetas <key> y </key> definen el inicio y el final de una colección de texto y marcador.
* Roles: ¿Qué hace una región de texto en el documento? Aquí, las etiquetas <key> y </key> dan un propósito al contenido del elemento: un texto, a diferencia de cualquier otro tipo de texto en línea, como una fecha o énfasis.
* Posiciones: Los elementos conservan el orden de sus contenidos, lo que es especialmente importante en documentos en prosa como este.
* Contención: el software de procesamiento XML tiene en cuenta el anidamiento de elementos, que puede tratar el contenido de manera diferente según el lugar donde aparezca. Por ejemplo, un título puede tener un tamaño de fuente diferente dependiendo de si es el título de un periódico o un artículo.
* Relaciones: un fragmento de texto se puede vincular a un recurso en otro lugar.

Ahora que conocemos también el concepto de XML y sus bondades, podemos hacer una comparación rápida con JSON para determinar cuál será el formato utilizado para el sistema.

JSON es simple y está reemplazando gradualmente a XML como el formato principal de intercambio de datos en Internet. JSON es fácil de leer y sus estructuras se traducen fácilmente en conceptos bien entendidos por los desarrolladores de software: matrices, objetos y pares de nombre / valor. Ya no tenemos que rascarnos la cabeza ni discutir más sobre lo que debería ser un Elemento o un Atributo. Los objetos y sus miembros de datos se ajustan mucho mejor al diseño y desarrollo orientado a objetos (OO). Un documento formateado en JSON suele ser más pequeño que su equivalente XML, porque JSON tiene menos gastos generales y es más compacto. Esto se debe a la falta de etiquetas de inicio y finalización que rodean cada elemento de datos. Entonces, a nivel empresarial, JSON es más eficiente de procesar que XML, porque los documentos JSON pueden transmitirse a través de una red y procesarse más rápido que sus contrapartes XML.

La mayor diferencia entre ambos son:

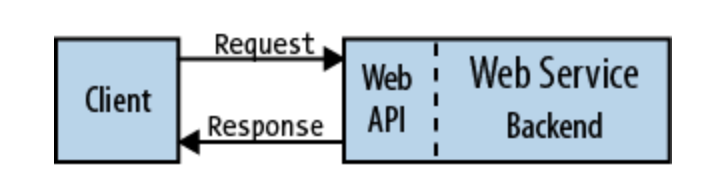
* XML tiene que ser analizado con un analizador XML. JSON se puede analizar mediante una función de JavaScript estándar.
* XML es mucho más difícil de analizar qué JSON.
* JSON se analiza en un objeto JavaScript lista para su uso.
* JSON solo admin texto o números como datos.
* Ambos permiten representar subestructuras, y pueden ser interpretados de manera nativa por los lenguajes de programación más utilizados
* JSON carece de propiedades para describir a los objetos.

Dadas las comparaciones anteriores y sumado a la gran aceptación que en los últimos años viene teniendo, es que se decide utilizar como mensajes el formato JSON para este sistema.

Ahora ya contamos con conceptos básicos definidos que nos permiten ir imaginando cómo se va a diseñar el sistema. Además de esto, se tiene que profundizar más en aspectos relacionados a los sistemas basados en APIs REST, y que también son de vital importancia para el mismo.

**2.5 APIs REST**

Los servicios web son servidores web especialmente diseñados que respaldan las necesidades de un sitio o cualquier otra aplicación [7]. Los programas cliente utilizan interfaces de programación de aplicaciones (API) para comunicarse con los servicios web. En términos generales, una API expone un conjunto de datos y funciones para facilitar las interacciones entre programas informáticos y permitirles intercambiar información. Como se muestra en la Figura a continuación, una API web es la cara de un servicio web, que escucha y responde directamente a las solicitudes de los clientes.



El estilo arquitectónico REST se aplica comúnmente al diseño de API para servicios web modernos. Una API web que se ajusta al estilo arquitectónico REST es una API REST. Tener una API REST hace que un servicio web sea "RESTful". Una API REST consta de un conjunto de recursos interconectados. Este conjunto de recursos se conoce como modelo de recursos de la API REST.

Las API REST bien diseñadas pueden atraer a los desarrolladores de clientes para que utilicen servicios web. En el mercado abierto de hoy, donde los servicios web rivales compiten por la atención, un diseño de API REST estéticamente agradable es una característica imprescindible.

Para muchos, diseñar una API REST a veces puede parecer más un arte que una ciencia. Algunas de las mejores prácticas para el diseño de API REST están implícitas en el estándar HTTP, mientras que en los últimos años han surgido otros enfoques pseudo estándar. Sin embargo, hoy debemos seguir buscando respuestas a una gran cantidad de preguntas, como:

* ¿Cuándo se deben nombrar los segmentos de ruta de URI con sustantivos en plural?
* ¿Qué método de solicitud se debe utilizar para actualizar el estado del recurso?
* ¿Cómo asigno operaciones que no son CRUD a mis URI?
* ¿Cuál es el código de estado de respuesta HTTP apropiado para un escenario determinado?
* ¿Cómo se logra que el sistema sea resiliente o bien tenga una alta disponibilidad, que criterio deben ser tenidos en cuenta?
* ¿Es necesario que nuestro servicio sea tolerante a fallos en alguna de las piezas?
* ¿Cómo debo estructurar un hipervínculo en JSON?

Como se ve, hay muchos interrogantes a la hora de diseñar una api rest, a continuación los temas y hasta finalizar el capítulo se intentarán ir dando conceptos que permitan responder la mayoría de las preguntas.

**2.5 APIs REST**

2.5.1 URIs

Las API REST utilizan identificadores uniformes de recursos (URI) para abordar los recursos. En la Web actual, los diseños de URI abarcan desde obras maestras que comunican claramente el modelo de recursos de la API como:

*http://api.example.restapi.org/france/paris/louvre/leonardo-da-vinci/mona-lisa*

a aquellos que son mucho más difíciles de entender para las personas, como:

http://api.example.restapi.org/68dd0-a9d3-11e0-9f1c-0800200c9a66

Tim Berners-Lee incluyó una nota sobre la opacidad de las URI en su lista "Axiomas de la arquitectura web" [7]:

*“Lo único para lo que puede utilizar un identificador es para hacer referencia a un objeto. Cuando no esté eliminando las referencias, no debe mirar el contenido de la cadena URI para obtener otra información.”*

El esquema básico de una url es el siguiente:

URI = scheme "://" authority "/" path [ "?" query ]

Cada segmento de ruta de URI, separado por barras diagonales (/), representa una oportunidad de diseño. Asignar valores significativos a cada segmento de ruta ayuda a comunicar claramente la estructura jerárquica del diseño del modelo de recursos de una API REST.

El componente de consulta de una URI contiene un conjunto de parámetros que se interpretarán como una variación o derivado del recurso que está identificado jerárquicamente por el componente de ruta.

El componente de consulta puede proporcionar a los clientes capacidades de interacción adicionales, como búsqueda y filtrado ad hoc. Por lo tanto, a diferencia de los otros elementos de una URI, la parte de la consulta puede ser transparente para el cliente de una API REST.

En este punto una vez que comprendemos el concepto de URI podemos avanzar con la definición de los métodos http utilizados en las APIs REST y de los posibles códigos de respuesta para cada petición que llegará al servidor.

**2.5 APIs REST**

2.5.2 Tipos de peticiones HTTP

Cada método HTTP tiene una semántica específica y bien definida dentro del contexto del modelo de recursos de una API REST. El propósito de GET es recuperar una representación del estado de un recurso. HEAD se utiliza para recuperar los metadatos asociados con el estado del recurso. PUT debe usarse para agregar un nuevo recurso a una tienda o actualizar un recurso. DELETE elimina un recurso de su padre. POST debe usarse para crear un nuevo recurso dentro de una colección y ejecutar controladores.

Además de estos métodos http existe una petición de OPTIONS, donde una API REST puede incluir un cuerpo que incluya más detalles sobre cada opción de interacción. Por ejemplo, el cuerpo de la respuesta podría contener una lista de formas de relación de enlace, que se analizan en la sección (debe usarse una forma coherente para representar las relaciones de enlace).

Veamos un poco más de detalle:

* DELETE: Método de solicitud HTTP utilizado para eliminar a su padre.
* GET: Método de solicitud HTTP utilizado para recuperar una representación del estado de un recurso.
* HEAD: Método de solicitud HTTP que se usa para recuperar los metadatos asociados con el estado del recurso.
* OPTIONS: Método de solicitud HTTP que se utiliza para recuperar metadatos que describen las interacciones disponibles de un recurso.
* POST: Método de solicitud HTTP que se utiliza para crear un nuevo recurso dentro de una colección o ejecutar un controlador.
* PUT: Método de solicitud HTTP utilizado para insertar un nuevo recurso en una tienda o actualizar un recurso mutable.

Además de las operaciones básicas soportadas por HTTP, recordar que asociadas a estas están los códigos de respuesta posibles, que fueron introducidos en la sección “Rest”, y para más detalle se puede continuar en uno de los anexos del documento.

Ya se puede empezar a visualizar el sistema haciendo uso de las definiciones de URIs como así también de los estados http que le darán una semántica determinada a cada acción que permitirá el sistema.

Ahora continúan temas relacionados a aspectos deseables del sistema, relacionados directamente con la definición y funcionamiento de una API basada en arquitectura REST, conceptos que van desde Resiliencia hasta la Alta Disponibilidad deseable en este tipo de sistemas, de esta manera se cerrará el capítulo 2 enfocado 100% en conceptos básicos para el posterior entendimiento del sistema que se propondrá.

**2.6 Resiliencia**

Generalmente, en las disciplinas de la ingeniería, la resiliencia se mide por la capacidad de un sistema para perseverar o superar una falla importante en una parte crítica del sistema. Es la capacidad de un sistema para "Recibir un golpe en un componente crítico" y recuperarse en un tiempo razonable, forma o duración aceptable. El software resiliente debe tener la capacidad de resistir una falla en un componente crítico pero aún así recuperarse de una manera y duración predefinidas aceptables. Las fallas en el software pueden surgir de actividades / ataques intencionales (ingeniería del caos) o fallas no intencionales. De cualquier manera, un sistema resiliente debe tener la capacidad para recuperarse después de tales fallas. Software que no se recupera por sí solo o aplicaciones que se recuperan después de una larga duración no son consideradas resilientes [8].

**2.6 Resiliencia**

2.6.1 Factores que afectan a la resiliencia

Hay varios factores que afectan la resiliencia del software. Los factores incluyen complejidad, globalización, interdependencia, cambios con rapidez, nivel de integración del sistema y factores humanos.

Las organizaciones de hoy tienen complejos sistemas en red que provienen de muchas interacciones e interdependencias. Por ejemplo, una empresa de telecomunicaciones que proporciona un servicio de dinero a través de una aplicación de terceros tiene un complejo sistema con múltiples puntos de fallo. Cuando un sistema crítico falla, toda la plataforma no puede entregar el servicio requerido.

Otro factor que afecta a la resiliencia es aquel Software que se basa en internet y es dinámico, siempre cambiante.

Las redes aumentan sus posibilidades de fracaso. La centralidad de la red puede introducir complejidades que conducen a mayores posibilidades de errores.

La combinación de sistemas introduce mayores posibilidades de falla debido a la complejidad. Esta complejidad hace que sea extremadamente difícil proporcionar una plataforma con niveles constantes de resiliencia. Cuanto mayor sea el número de sistemas integrados, mayores son las posibilidades de tener menor resiliencia.

Uno de los enfoques obvios para asegurar la resiliencia son pruebas adecuadas en cada fase dentro del desarrollo de software. Cuando el software está siendo completado, una serie de pruebas son necesarias para garantizar que todos los aspectos del software satisfagan los requisitos del diseño. Pruebas correctas pueden asegurar una mejor integridad y resiliencia en el nuevo software. Más específicamente, las pruebas claras y regulares deben concentrarse en la fase de codificación.

Las pruebas de regresión suelen ser un camino para lograr que el sistema sea pensado y adecuado para tener mayor resiliencia. Una rutina de error no probada es una causa importante de fallo del sistema y la incapacidad para recuperarse fácilmente después de una ruptura.

Como vemos hasta ahora lo importante para que el sistema sea resiliente no es solo poder recuperarse ante posibles fallas en las piezas del mismo, sino también tratar de que el sistema como un todo no sea un software complejo, esa complejidad pasa a ser determinante en virtud de que sea resiliente.

Vemos a continuación algunos conceptos básicos respecto de que se puede hacer para intentar lograr al final del camino un sistema que efectivamente sea resiliente tolerando las fallas que pueden llegar a presentarse en su funcionamiento

**2.7 Fault Tolerance**

Diseñar sistemas preparados para ser tolerantes a fallos es algo que está directamente relacionado con resiliencia.

Los diseños de sistemas tolerantes a fallas utilizan los siguientes conceptos básicos:

* Modularidad: Descomponer el sistema en módulos. La descomposición es típicamente jerárquica. Por ejemplo, una computadora puede tener un módulo de almacenamiento que a su vez tiene varias memorias.
* Servicio: El módulo proporciona una interfaz bien especificada para alguna función.
* Contención de fallas: si el módulo es defectuoso, el diseño evita que se contaminen otros.
* Reparación: cuando un módulo falla, se reemplaza por un nuevo módulo.
* Fallo rápido: cada módulo debe funcionar correctamente o debe detenerse inmediatamente.
* Modos de falla independientes: los módulos interconexiones deben diseñarse de modo que si un módulo falla, la falla no debería afectar también a otros módulos.
* Redundancia y reparación: tener módulos de repuesto ya instalados o configurados, cuando un módulo falla, el segundo puede reemplazarlo casi instantáneamente. El módulo fallido puede ser reparado y desconectado mientras el sistema sigue prestando servicio. Este es el primer punto directamente relacionado con la disponibilidad que se quiere lograr para el sistema, que se describe en la siguiente sección.

Estos principios se aplican a las fallas de hardware, fallas de diseño y fallas de software [9].

Lo importante de esta sección, es que ayuda a pensar la mejor manera para estructurar al sistema de Streaming, teniendo en cuenta puntos que serán claves.

**2.8 High Availability**

La alta disponibilidad se refiere a los sistemas que son duraderos y es probable que funcionen continuamente sin fallas durante mucho tiempo. El término implica que las partes de un sistema se han probado en su totalidad y, en muchos casos, hay adaptaciones para fallas en forma de componentes redundantes [9].

La alta disponibilidad requiere sistemas diseñados para tolerar fallas: para detectar la falla, informarla, enmascarar y luego continúe el servicio mientras el componente defectuoso se repara sin conexión. Más allá de lo prosaico, fallas de hardware y software, un sistema de alta disponibilidad debe tolerar entre otras las siguientes fallas o escenarios:

* Las actualizaciones o reparaciones de software generalmente requieren la interrupción del servicio al instalar nuevos software.
* Tareas de mantenimiento sobre base de datos para agregar nuevos tipos de información a la base de datos, para reorganizar los datos para que puedan procesarse de manera más eficiente, o para redistribuir los datos
* Fallos de operaciones: los operadores a veces cometen errores que llevan a interrupciones del sistema.

Como lo vemos, estos conceptos de alta disponibilidad están fuertemente relacionados a los de *Fault Tolerance* y *Resiliencia* ya que la combinatoria de ambos aspectos nos permiten de alguna manera asegurar la alta disponibilidad de nuestros sistemas, algo que ya vamos a ver, es deseable para el presentado en esta tesis.

**3. Arquitecturas de Software**

**3.1 Patrones Arquitectonicos**

Una vez definidos algunos conceptos o definiciones necesarios para el desarrollo de tesis de este sistema, se empezará a analizar las distintas arquitecturas de software o patrones arquitectónicos que son de interés para la misma y como luego a través de la conjunción de varias de ellas, o de determinados aspectos que algunas poseen, podemos empezar a pensar en la solución final que dará forma al sistema en cuestión.

Los patrones arquitectónicos expresan un esquema de organización estructural fundamental para los sistemas de software. Proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y pautas para organizar las relaciones entre ellos. Los patrones arquitectónicos son plantillas para arquitecturas de software concretas. Especifican las propiedades estructurales de todo el sistema de una aplicación y tienen un impacto en la arquitectura de sus subsistemas. La selección de un patrón arquitectónico es, por lo tanto, una decisión de diseño fundamental al desarrollar un sistema de software [10].

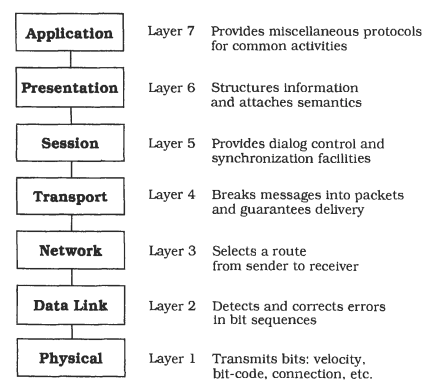
Del apartado motivacional de esta tesis, donde se establece lo que realmente se quiere lograr al final del camino, se describen funcionalidades básicas que se quieren tener en el sistema. Empezando a entender a partir de ellas, el primer paso para definir el manejo de los posibles temas que podrán formar parte del sistema, como así también la administración de los mismos, y por otro lado también la administración de los interesados en estos temas (suscriptores), se desprenden funcionalidades o escenarios básicos que el sistema debería permitir, y para los cuales tenemos que encontrar la arquitectura de software que mejor se ajuste a ellos, estas funcionalidades serán lo que se conoce como ABMs: poder crear, modificar, obtener o eliminar tanto tópicos como suscriptores.

En este escenario particular donde las funcionalidades son un tanto básicas, se puede imaginar como arquitectura de software algo bien simple y acotado, que también acompañe por un lado el posible crecimiento a futuro en el uso del sistema, como así también sea simple y de bajo costo el mantenimiento. La arquitectura de software más relacionada y utilizada por desarrolladores de software para estos escenarios es la conocida como arquitectura basada en capas, la que describiremos en el apartado siguiente [11].

**3.2 Arquitectura en Capas**

El patrón arquitectónico de capas ayuda a estructurar aplicaciones para que puedan descomponerse en grupos de subtareas en las que cada grupo de las subtareas se encuentran en un nivel particular de abstracción.

Como está especificado en [10], los protocolos de red son probablemente el ejemplo más conocido de arquitecturas en capas. Dicho protocolo consta de un conjunto de reglas y condiciones que describen cómo los programas de computadora se comunican a través de las piezas de la máquina. El formato, contenido y significado de todos los mensajes están definidos. Todos los escenarios se describen en detalle, generalmente por gráficos de secuencia. El protocolo especifica acuerdos en una variedad de niveles de abstracción, que van desde los detalles de la transmisión de bits hasta lógica de aplicación de alto nivel. Por lo tanto, los diseñadores utilizan varios sub-protocolos y son organizados en capas. Cada capa se ocupa de un aspecto de la comunicación y utiliza los servicios de la siguiente capa. La Organización Internacional de Normalización (ISO) definió el siguiente modelo arquitectónico:



Un enfoque en capas se considera una mejor práctica que implementar el protocolo como bloque monolítico. Desde que se implementan conceptualmente diferentes temas por separado trae varios beneficios, por ejemplo, ayudar al desarrollo por equipos y también a la codificación incremental y pruebas. El uso de piezas semi independientes también permite la exploración más fácil ante cambio de piezas individuales. Y además mejor implementación, adaptación a las nuevas tecnologías, nuevos lenguajes o algoritmos que pueden incorporarse reescribiendo una sección delimitada de código.

Es muy común dentro de los arquetipos para el comportamiento dinámico de aplicaciones en capas encontrarse con un escenario como es descrito a continuación:

Escenario: es probablemente el escenario más conocido por los desarrolladores. Un cliente emite una solicitud a la capa N, esta capa al no poder ejecutar la totalidad de la solicitud, ejecuta una llamada a la capa N-1 que ejecute sub-tareas, a su vez la capa N-1 tiene que llamar a la capa N-2 para que esta ejecuta otras subtareas, y así se repite el proceso hasta llegar a capas que ejecutan tareas, y responde a la misma a la capa anterior, y así hasta llegar nuevamente a la capa N que fue quien inicialmente recibió la solicitud del cliente. Este escenario es conocido como top-down, donde la capa superior transmite la responsabilidad de ejecutar tareas a capas inferiores, y en ocasiones aguarda el resultado para continuar con su procesamiento hasta poder generar una respuesta final a la solicitud inicial.

Además de esto, como se define en [36], la arquitectura en capas, es también conocida como estilo de arquitectura de N niveles, es uno de los estilos de arquitectura más comunes. Este estilo de arquitectura es el estándar de facto para la mayoría de las aplicaciones, principalmente debido a su simplicidad, familiaridad y bajo costo. También es una forma muy natural de desarrollar aplicaciones debido a la ley de Conway, que establece que las organizaciones que diseñan sistemas están obligadas a producir diseños que son copias de las estructuras de comunicación de estas organizaciones. En la mayoría de las organizaciones hay desarrolladores de interfaces de usuario (UI), desarrolladores de backend, desarrolladores de reglas y expertos en bases de datos (DBA). Estas capas organizativas encajan perfectamente en los niveles de una arquitectura en capas tradicionales, lo que la convierte en una opción natural para muchas aplicaciones comerciales. Si un desarrollador o arquitecto no está seguro de qué estilo de arquitectura está usando, o si un equipo de desarrollo ágil "simplemente comienza a codificar", es muy probable que sea el estilo de arquitectura en capas que está implementando.

Luego de esto, es momento de pensar cómo modelar la funcionalidad de poder recibir un mensaje en un tema y propagar el mismo a los suscriptores interesados, y que es la más compleja del sistema, vemos a continuación que arquitecturas de software son las más adecuadas para esto.

**3.3 Event-Driven Architecture**

Como lo muestra [11], el patrón de arquitectura basado en eventos es un patrón asíncrono y distribuido utilizado para desarrollar aplicaciones altamente escalables. También es altamente adaptable y se puede usar para aplicaciones pequeñas, grandes, y complejas. La arquitectura impulsada por eventos está formada por componentes de procesamiento de eventos de un solo propósito altamente desacoplados que reciben y procesan eventos de forma asíncrona.  
  
El patrón de arquitectura basado en eventos consta de dos tipologías principales, el mediador (hub) y el intermediario. La topología de mediador se usa comúnmente cuando necesita orquestar múltiples pasos dentro de un evento a través de un mediador central, mientras que la topología de intermediario se usa cuando desea encadenar eventos juntos sin el uso de un mediador central. Debido a que las características de la arquitectura y las estrategias de implementación difieren entre estas dos topologías, es importante comprender a cada una de ellas para saber cuál es la más adecuada para cada escenario.

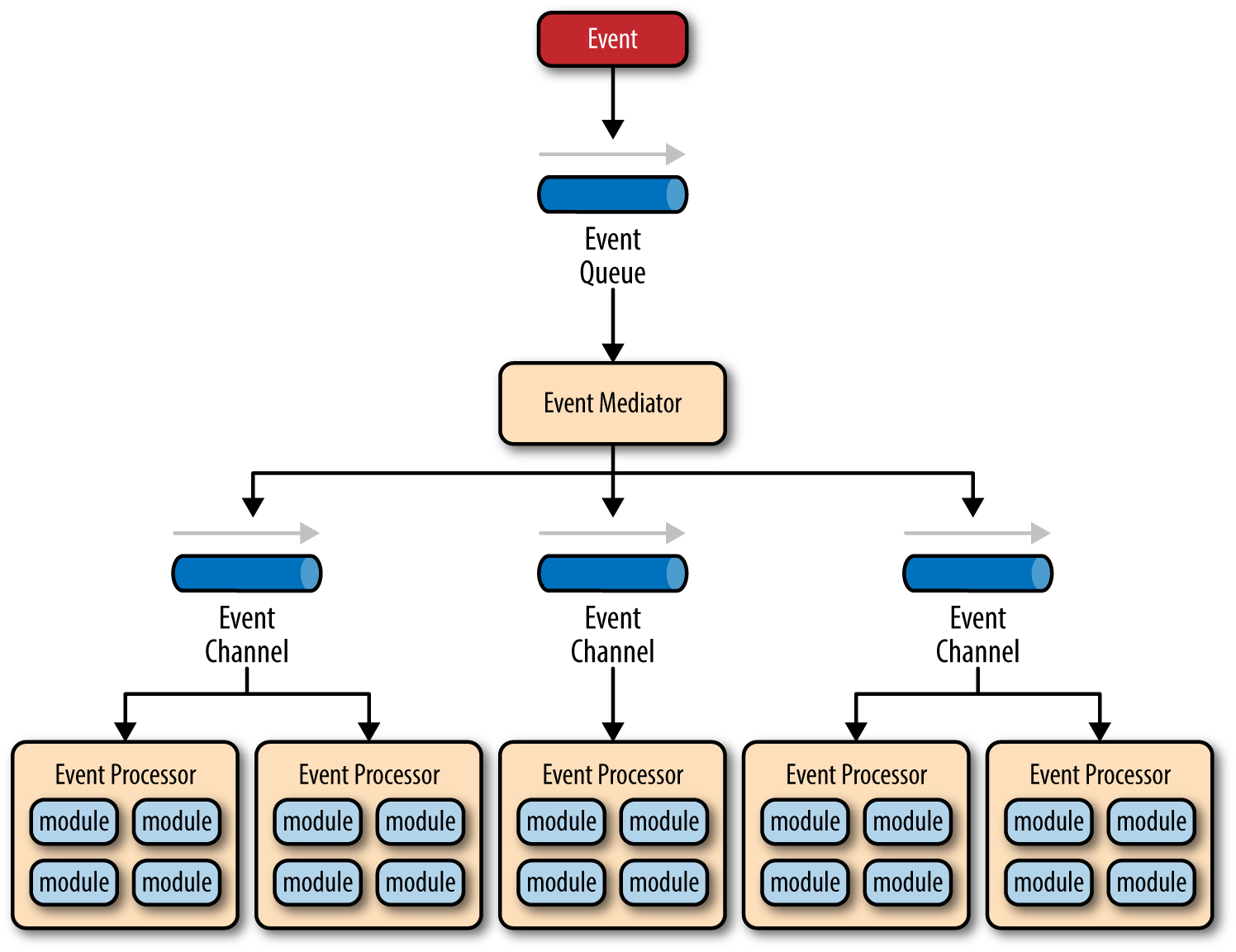
**3.3 Event-Driven Architecture**

3.3.1 Mediator Topology

También nombrado en [11], la topología con mediador es útil para eventos que tienen varios pasos y requieren algún nivel de orquestación para procesar el evento. Por ejemplo, un solo evento para colocar una operación bursátil puede requerir que primero valide la operación, luego verifique el cumplimiento de esa operación bursátil con varias reglas de cumplimiento, asigne la operación a un intermediario, calcule la comisión y finalmente realice el intercambio con ese corredor. Todos estos pasos requerirían algún nivel de orquestación para determinar el orden de los pasos y cuáles se pueden hacer en serie y en paralelo.

Hay cuatro tipos principales de componentes de arquitectura dentro de la topología de mediador:

* colas de eventos,
* un mediador de eventos,
* canales de eventos y
* procesadores de eventos.

El flujo de eventos comienza con un cliente que envía un evento a una cola, que se utiliza para transportar el evento al mediador. El mediador de eventos recibe el evento inicial y lo organiza enviando eventos asincrónicos adicionales a los canales de eventos para ejecutar cada paso del proceso. Los procesadores de eventos, que escuchan en los canales de eventos, reciben el evento del mediador de eventos y ejecutan una lógica específica para procesar el evento. La Figura a continuación ilustra la topología de mediador general del patrón de arquitectura dirigida por eventos.  


Es común tener entre una docena o varios cientos de colas de eventos en una arquitectura impulsada por eventos. El patrón no especifica la implementación del componente de cola de eventos; puede ser una cola de mensajes, un punto final de servicio web o cualquier combinación de los mismos.

Hay dos tipos de eventos dentro de este patrón: un evento inicial y un evento de procesamiento. El evento inicial es el evento original recibido por el mediador, mientras que los eventos de procesamiento son los generados por el mediador y recibidos por los componentes de procesamiento de eventos.

Los canales de eventos son utilizados por el mediador de eventos para pasar de forma asincrónica eventos de procesamiento específicos relacionados con cada paso en el evento inicial a los procesadores de eventos. Los canales de eventos pueden ser colas de mensajes o temas de mensajes, aunque los temas de mensajes se utilizan más ampliamente con la topología del mediador, de modo que los eventos de procesamiento pueden ser procesados ​​por varios procesadores de eventos (cada uno de ellos realiza una tarea diferente según el evento de procesamiento recibido).

Los componentes del procesador de eventos contienen la lógica de la aplicación necesaria para procesar el evento de procesamiento. Los procesadores de eventos son componentes de arquitectura autónomos, independientes y altamente desacoplados que realizan una tarea específica en la aplicación o el sistema. Si bien la granularidad del componente del procesador de eventos puede variar de detallada (por ejemplo, calcular el impuesto a las ventas en un pedido) a granularidad (por ejemplo, procesar una reclamación de seguro), es importante tener en cuenta que, en general, cada componente procesador de eventos debe realizar una sola tarea y no depender de otros procesadores de eventos para completar su tarea específica.

El componente mediador de eventos es responsable de orquestar los pasos contenidos en el evento inicial. Para cada paso del evento inicial, el mediador de eventos envía un evento de procesamiento específico a un canal de eventos, que luego es recibido y procesado por el procesador de eventos. Es importante tener en cuenta que el mediador de eventos en realidad no realiza la lógica necesaria para procesar el evento inicial; más bien, conoce los pasos necesarios para procesar el evento inicial.

**3.3 Event-Driven Architecture**

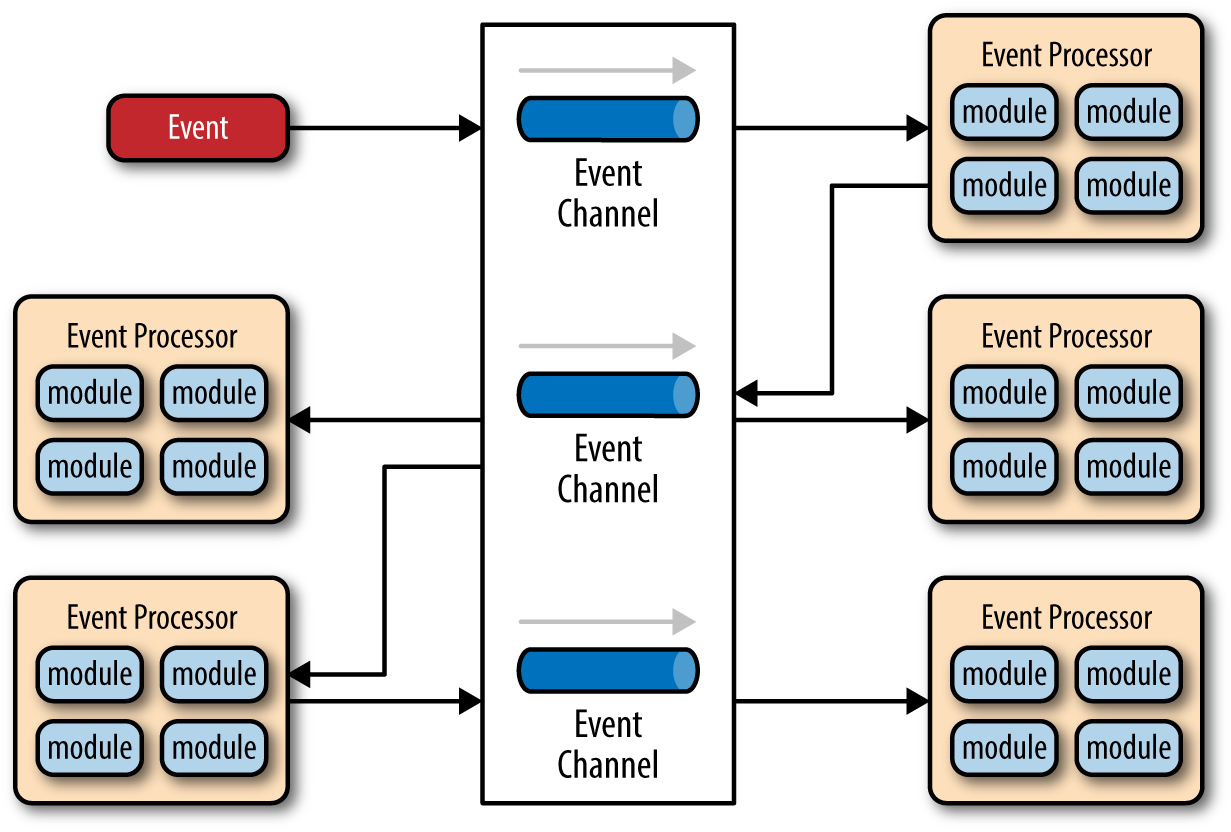
3.3.2 Broker Topology

La topología broker difiere de la topología mediator en que no existe un mediador central de eventos; el flujo de mensajes se distribuye a través de los componentes del procesador de eventos en forma de cadena a través de un intermediario de mensajes ligero (por ejemplo, ActiveMQ, HornetQ, etc.). Esta topología es útil cuando tiene un flujo de procesamiento de eventos relativamente simple y no desea (o necesita) orquestación central de eventos [11].

También como se define en [10], el patrón de arquitectura de Broker se puede utilizar para estructurar un sistema de software distribuido con componentes desacoplados que interactúan mediante invocaciones de servicios remotos. Un componente de intermediario es responsable de coordinar las comunicaciones, como el envío de solicitudes, así como de transmitir los resultados y las excepciones.

Hay dos tipos principales de componentes de arquitectura dentro de la topología broker (intermediario): un componente intermediario y un componente de procesador de eventos. El componente de intermediario se puede centralizar o federar y contiene todos los canales de eventos que se utilizan dentro del flujo de eventos. Los canales de eventos contenidos en el componente de intermediario pueden ser colas de mensajes, temas de mensajes o una combinación de ambos.

Esta topología se ilustra en la Figura siguiente.



Como se puede ver en el diagrama, no hay un componente central de evento-mediador que controle y orquesta el evento inicial; más bien, cada componente del procesador de eventos es responsable de procesar un evento y publicar un nuevo evento que indica la acción que acaba de realizar. Por ejemplo, un procesador de eventos que equilibra una cartera de acciones puede recibir un evento inicial llamado división de acciones. En función de ese evento inicial, el procesador de eventos puede realizar un reequilibrio de la cartera y luego publicar un nuevo evento para el corredor llamado cartera de reequilibrio, que luego sería recogido por un procesador de eventos diferente. Téngase en cuenta que puede haber ocasiones en las que un procesador de eventos publique un evento, pero ningún otro procesador de eventos lo recoja. Esto es común cuando se está desarrollando una aplicación o proporcionando funcionalidades y extensiones futuras [11].

**3.4 Consideraciones generales de Event-Driven**

El patrón de arquitectura impulsada por eventos es un patrón relativamente complejo de implementar, principalmente debido a su naturaleza distribuida y asincrónica. Al implementar este patrón, debe abordar varios problemas de arquitectura distribuida, como la disponibilidad de procesos remotos, la falta de capacidad de respuesta y la lógica de reconexión del intermediario en caso de falla de un intermediario o mediador [11].

Una consideración a tener en cuenta al elegir este patrón de arquitectura es la falta de transacciones atómicas para un único proceso. Debido a que los componentes del procesador de eventos están altamente desacoplados y distribuidos, es muy difícil mantener una unidad de trabajo transaccional entre ellos. Por esta razón, al diseñar la aplicación con este patrón, debe pensar continuamente en qué eventos pueden y no pueden ejecutarse de forma independiente y planificar la granularidad de los procesadores de eventos en consecuencia. Si se encuentra que necesita dividir una sola unidad de trabajo entre procesadores de eventos, es decir, si está usando procesadores separados para algo que debería ser una transacción no dividida, probablemente este no sea el patrón correcto para su aplicación.

Quizás uno de los aspectos más difíciles del patrón de arquitectura impulsado por eventos es la creación, el mantenimiento y la administración de los contratos de los componentes del procesador de eventos. Cada evento generalmente tiene un contrato específico asociado (por ejemplo, los valores de los datos y el formato de los datos se pasan al procesador de eventos). Al utilizar este patrón, es de vital importancia establecer un formato de datos estándar (por ejemplo, XML, JSON, Java Object, etc.) y establecer una política de versiones del contrato desde el principio.

**3.5 Sistema de colas basado en contenido**

En los últimos años, con el tremendo desarrollo de Internet y el rápido crecimiento de información, cada vez más aplicaciones de Internet requieren la difusión de información entre un gran número de entidades muy dispersas. En este entorno, miles o incluso millones de entidades se distribuyen globalmente y sus ubicaciones y comportamientos pueden variar. Las características a gran escala, dinámicas y geográficamente diseminadas del entorno requiere una técnica escalable, eficiente y confiable para la difusión de información. Las comunicaciones síncronas y punto a punto individuales rígidas y estáticas no pueden cumplir con los requisitos. Los sistemas de publicación / suscripción (pubsub) han estado recibiendo cada vez más atención por la forma de interacción débilmente acoplada que proporciona en entornos a gran escala, similar a los descritos en los patrones arquitectónicos de los puntos anteriores [12].

Un sistema pubsub se compone de muchos nodos distribuidos en una red de comunicación. En tal sistema, los clientes son entidades autónomas que intercambian información publicando eventos y suscribiéndose a las clases de eventos que son de interés. Los clientes no están obligados a comunicarse directamente entre ellos pero están bastante desacoplados: la interacción ocurre a través de los nodos del sistema que se coordinan para enviar información de los editores a los suscriptores. En el sistema, los editores producen información y los suscriptores consumen información. Específicamente, los editores publican información en forma de eventos y los suscriptores expresan su intereses en un evento o un patrón de eventos en forma de filtros de suscripción. Un dato evento especifica valores de un conjunto de atributos asociados con el evento. Las suscripciones pueden ser muy expresivas y especificar criterios de filtrado complejos mediante el uso de un conjunto de predicados sobre atributos de eventos. Cuando un sistema pubsub recibe un evento publicado por un editor, hace coincidir el evento con las suscripciones y entrega el evento a los suscriptores coincidentes. Un suscriptor instala y elimina una suscripción del sistema ejecutando las operaciones de suscripción y cancelación de suscripción respectivamente.

Los procesos en sistemas pubsub son clientes de un servicio de notificación subyacente y pueden actuar como productores y consumidores de mensajes, llamados notificaciones de eventos o notificaciones para abreviar. Una notificación es un mensaje que describe un evento. Las notificaciones se inyectan en el sistema de eventos a través de una llamada de publicación, en lugar de ser publicado hacia un receptor específico. Son transmitidos por el servicio de notificación subyacente a aquellos consumidores que han registrado una suscripción correspondiente. Las suscripciones describen el tipo de notificaciones que reciben los consumidores que están interesados. Se ha propuesto una variedad de sistemas de publicación / subs basados ​​en contenido. Estos sistemas de suscripciones se pueden clasificar en dos tipos:

* Basado en topicos
* Basado en contenido

**3.5 Sistema de colas basado en contenido**

3.5.1 Basado en Tópicos

En los sistemas de suscripciones basados ​​en tópicos, cada evento pertenece a un tema determinado (también denominado grupo, canal o tema). Los suscriptores expresan su interés en un tema en particular y reciben todos los eventos publicados dentro de ese tema en particular. Cada tema corresponde a un canal lógico que conecta idealmente a cada posible editor con todos los suscriptores interesados. Los árboles de difusión y multidifusión de red pueden utilizarse para difundir eventos a los suscriptores interesados. El modelo basado en temas ha sido la solución adoptada en todos los primeros sistemas de suscripciones.

Los sistemas de suscripciones basados ​​en tópicos solo aceptan suscripciones generales. El principal inconveniente del modelo basado en tópicos es la expresividad muy limitada que ofrece a los suscriptores. En consecuencia, un suscriptor tiene que recibir todos los eventos pertinentes a un tópico, aunque el suscriptor podría estar interesado solo en un subconjunto de los eventos. Además, los sistemas basados ​​en temas ofrecen opciones limitadas de suscripciones. Existen muchos problemas relacionados con la baja expresividad de los tópicos. Otro método para mejorar la expresividad del modelo basado en tópicos es la variante de tópico filtrado, donde se realiza una fase de filtrado adicional una vez que se recibe el mensaje en función del contenido del mensaje. Los mensajes que no cumplen con el filtro no se envían al suscriptor.

**3.5 Sistema de colas basado en contenido**

3.5.2 Basado en Contenido

A diferencia de los sistemas basados ​​en tópicos, los sistemas basados ​​en contenido permiten suscripciones detalladas al permitir restricciones en el contenido del evento. En el sistema basado en contenido, las notificaciones suelen constar de una serie de pares de atributo / valor. En un sistemas de suscripción basados ​​en contenido,

la inscripción puede incluir un número arbitrario de nombres de atributos y criterios de filtrado en sus valores. Solo aquellos eventos que satisfacen todos los predicados se entregan al interesado. Por lo tanto, los sistemas basados ​​en contenido aumentan la selectividad de las suscripciones al permitir que las suscripciones tengan múltiples dimensiones.

En sistemas basados en contenido, la correspondencia de suscripciones y publicaciones se basa en el contenido y no se necesitan conocimientos previos. Estos sistemas son más expresivos. Los suscriptores expresan su interés especificando condiciones sobre el contenido de los eventos que les interesan. En otras palabras, una suscripción es una solicitud formada por un conjunto de restricciones compuestas por disyunción o conjunción de operadores. Las posibles limitaciones dependen del tipo de atributo y del idioma de suscripción. La mayoría de los idiomas de suscripción comprenden operadores de igualdad y comparación como así también expresiones regulares. Por tanto, estos sistemas son más flexibles y útiles ya que los suscriptores pueden especificar sus intereses con mayor precisión utilizando un conjunto de predicados. El suscriptor no necesita aprender un conjunto de nombres de temas y su contenido antes de suscribirse. El principal desafío en la construcción de tales sistemas es desarrollar un algoritmo de coincidencia eficiente que se adapte a millones de publicaciones y suscripciones.

Como se observa en la definición de los patrones arquitectónicos se empieza a dilucidar de manera abstracta como se pueden empezar a unir las piezas del sistema, para que de alguna manera se termine cumpliendo con los objetivos y motivaciones del mismo.

A continuación, el capítulo siguiente, definirá la arquitectura pensada para el sistema presentado en esta tesis, haciendo uso de todos los conceptos vistos hasta el momento.

**4. Arquitectura y Aspectos del Sistema**

Para la situación planteada en este sistema, y haciendo referencia a parte de la motivación del mismo, la funcionalidad relacionada con el ABM tanto de temas como de suscriptores, es fácilmente moldeable con el patrón arquitectónico en capas, donde se pueden establecer responsabilidades claras en diferentes capas:

1. Capa 1 (webserver): es quien tiene la responsabilidad de recibir la solicitud con la acción a ejecutar (alta, baja, modificación o consulta), tal vez ejecutar validaciones previas, tomar alguna decisión de negocio y luego ejecutar una llamada a la capa inferior en este caso la capa de persistencia de información.
2. Capa 2 (Almacenamiento Clave-Valor): esta capa es la que ejecuta ante determinadas llamadas de la capa 1, las tareas en el almacenamiento elegido para el sistema, es quien tiene la responsabilidad sobre los datos del mismo.

Esta pieza del sistema quedaría modelado de la siguiente manera



Ahora bien, lo que se termina de definir en este caso es sin dudas la parte más simple de lo que planteamos como motivación del sistema, que sería la funcionalidad que permite la configuración y administración del mismo.

**4.1 Casos de uso funcionales**

Si bien tanto los casos funcionales como los aspiracionales ya fueron nombrados en el capítulo 1, ahora vamos a describir cada uno con un detalle más preciso.

1. Poder tener un ABM de tópicos

Esto consiste en poder tener los métodos básicos de una aplicación: alta, baja, modificación y consulta, en este caso al usar REST vamos a poder tener las operaciones.

* + **POST**: para crear un tópico.

| **REQUEST**  URI: /topics  METHOD: POST   JSON Body:   {  "description": "Nuevo topico",  "name": "topic\_test",  "max\_subscribers": 100  }  **RESPONSE**  Status Http Code: 201  JSON Body:  {  "name": "topic\_test",  "description": "Nuevo topico",  "max\_subscribers": 100,  "subscribers": []  } |
| --- |
|  |

* + **PUT**: para modificar un tópico.

| **REQUEST**  URI: /topics/:name  METHOD: PUT   JSON Body:   {  "description": "Topico modificado",  "max\_subscribers": 1000  }  **RESPONSE**  Status Http Code: 200  JSON Body:  {  "name": "topic\_test",  "description": "Topico modificado",  "max\_subscribers": 1000,  "subscribers": []  } |
| --- |

* + **DELETE**: para eliminar un tópico.

| **REQUEST**  URI: /topics/:name  METHOD: DELETE   JSON Body:   {}  **RESPONSE**  Status Http Code: 204  JSON Body:  {} |
| --- |
|  |

* + **GET**: para obtener un tópico.

| REQUEST  URI: /topics/:name  METHOD: GET   JSON Body:   {}  RESPONSE   Status Http Code: 200  JSON Body:  {  "name": "topic\_test",  "description": "Topico modificado",  "max\_subscribers": 1000,  "subscribers": []  } |
| --- |

1. Poder enviar un mensaje a un tópico

Para este caso de uso lo que queremos lograr es que dado que ya podemos crear un tópico nuevo, ahora queremos poder enviar un mensaje determinado a ese tópico previamente creado.

| REQUEST  URI: /publications  METHOD: POST   JSON Body:   {  "topic": "topic\_test",  "message": {  "key1": "value1",  "key2": "value2",  "key3": "value3"  }  }  RESPONSE   Status Http Code: 201  JSON Body:  {  "output": "Message published"  } |
| --- |

1. Poder tener un ABM de suscriptores.

Este caso de uso es similar a 1, es decir tener un set de operaciones básicas que en este caso permitan manipular lo que son los suscriptores de los mensajes.

**POST**: para crear un suscriptor.

| REQUEST  URI: /subscribers  METHOD: POST   JSON Body:   {  "callback": "http://server:8080/mock/subscriber",  "description": "Subscriber de prueba",  "topic": "topic\_test" }  RESPONSE   Status Http Code: 201  JSON Body:  {  "id": "6455bad00128806bb260fcf12fce7159",  "callback": "http://server:8080/mock/subscriber",  "description": "Subscriber de prueba",  "topic": "topic\_test" } |
| --- |
|  |

* + **PUT**: para modificar un suscriptor.

| REQUEST  URI: /subscribers/:id  METHOD: PUT   JSON Body:   {  "callback": "http://server:8080/mock/subscriber\_new",  "description": "Subscriber de prueba modificado" }  RESPONSE   Status Http Code: 200  JSON Body:  {  "id": "6455bad00128806bb260fcf12fce7159",  "callback": "http://server:8080/mock/subscriber\_new",  "description": "Subscriber de prueba modificado",  "topic": "topic\_test" } |
| --- |

* + **DELETE**: para eliminar un suscriptor.

| REQUEST  URI: /subscribers/:id  METHOD: PUT   JSON Body:   {}  RESPONSE   Status Http Code: 204  JSON Body:  {} |
| --- |
|  |

* + **GET**: para obtener un suscriptor.

| REQUEST  URI: /subscribers/:id  METHOD: PUT   JSON Body:   {}  RESPONSE   Status Http Code: 200  JSON Body:  {  "id": "6455bad00128806bb260fcf12fce7159",  "callback": "http://server:8080/mock/subscriber\_new",  "description": "Subscriber de prueba modificado",  "topic": "topic\_test"  } |
| --- |

1. Que un suscriptor pueda suscribirse o desuscribirse de uno o más tópicos

Este es un caso de uso que si bien está contemplado en el caso de uso anterior, forma parte de lo que quiero lograr como sistema que sea simple de usar, este caso lo que me obliga es a relajar una condición, de esta manera un mismo suscriptor puede querer escuchar más de un tema en el mismo endpoint (callback).

1. Que un suscriptor pueda aplicar una regla de filtrado de mensajes al momento de suscribirse.

Ya vimos en el capítulo introductorio el hecho de la motivación por un lado, pero además de eso dijimos que esa motivación preliminar se iba a ir modelando con más detalle, uno de esos detalles es lo que también se comenta en ese capítulo y tiene que ver con ser un sistema de streaming basado en contenido. Esto se ve reflejado directamente en este caso de uso, modificando un poco la especificación de la creación y modificación de un suscriptor de la siguiente manera:

* **POST**: para crear un suscriptor.

| REQUEST  URI: /subscribers  METHOD: POST   JSON Body:   {  "callback": "http://server:8080/mock/subscriber",  "description": "Subscriber de prueba",  "topic": "topic\_test",  "rules": "(key1 == value1)" }  RESPONSE   Status Http Code: 201  JSON Body:  {  "id": "6455bad00128806bb260fcf12fce7159",  "callback": "http://server:8080/mock/subscriber",  "description": "Subscriber de prueba",  "topic": "topic\_test",  "rules": "(key1 == value1)" } |
| --- |
|  |

* **PUT**: para modificar un suscriptor.

| REQUEST  URI: /subscribers/:id  METHOD: PUT   JSON Body:   {  "callback": "http://server:8080/mock/subscriber\_new",  "description": "Subscriber de prueba modificado",  "rules": "(key1 == value1 && key2.contains(\"value\")" }  RESPONSE   Status Http Code: 200  JSON Body:  {  "id": "6455bad00128806bb260fcf12fce7159",  "callback": "http://server:8080/mock/subscriber\_new",  "description": "Subscriber de prueba modificado",  "topic": "topic\_test",  "rules": "(key1 == value1 && key2.contains(\"value\")"  } |
| --- |

A modo explicativo, lo que permite el aplicar rules de esta manera, es que el sistema interprete el mensaje que le llega a un tópico particular, y antes de enviarlo a un suscriptor determinado valide si la regla establecida por el suscriptor aplica para los valores del mensaje que llega, de ser así, el sistema enviará el mensaje, sino, lo descarta para ese suscriptor.

Veamos un ejemplo paso a paso, para ser más concreto:

1. El tópico topic\_test tiene dos suscriptores:
   1. Suscriptor con callback http://server:8080/mock/suscriber\_1 (id: asdasd12313).
   2. Suscriptor con callback http://server:8080/mock/suscriber\_2 (id: asdasd43443).
2. El suscriptor con id asdasd12313 tiene la siguiente configuración:

| {  "id": "asdasd12313",  "callback": "http://server:8080/mock/suscriber\_1",  "description": "Subscriber de prueba 1",  "topic": "topic\_test",  "rules": "(key2.contains(\"value\"))"  } |
| --- |

1. El suscriptor con id asdasd43443 tiene la siguiente configuración:

| {  "id": "asdasd43443",  "callback": "http://server:8080/mock/suscriber\_2",  "description": "Subscriber de prueba 2",  "topic": "topic\_test",  "rules": "(key1 == value1)"  } |
| --- |

1. Llega un mensaje al tópico topic\_test con la siguiente información:

| {  "topic": "topic\_test",  "message": {  "key1": "value1",  "key2": "message"  }  } |
| --- |

El sistema lo que va a hacer es:   
 a- Validar que suscriptores tiene el tópico topic\_test, para el caso tiene 2.  
 b- Interpreta el mensaje y ejecuta la validación para ver si puede enviarlo al suscriptor 1 y 2:  
 El suscriptor 1 tiene la rule key2.contains(\"value\"), como se ve el mensaje NO cumple con esa condición, por lo que el suscriptor NO va a recibir el mensaje, se descarta.  
 El suscriptor 2 tiene la rule key1 == value1, como se ve el mensaje SI cumple con esa condición, por lo que el suscriptor en este caso SI va a recibir el mensaje.

Con este ejemplo particular se explica de manera clara como debería ser el funcionamiento en casos exitosos del sistema, pero viendo esto y sabiendo que estamos trabajando en un entorno REST, el sistema al intentar enviar un mensaje a un suscriptor, la probabilidad de que falle puede ser tan alta como problemas en la red puedan existir, por lo que tenemos que tener en cuenta un caso de uso nuevo, que consiste en poder procesar reintentos de mensajes que no pudieron ser enviados de manera satisfactoria a los suscriptores.

Este nuevo caso de uso consiste en poder tener un mecanismo sencillo que permita dadas ciertas condiciones ejecutar reintentos de aquellos mensajes que no se pueden enviar de manera real-time a los suscriptores. Para resolverlo lo que voy a hacer es contar con un mecanismo que se describe a continuación:

1- Si el mensaje al intentar enviarlo a un suscriptor falla, lo voy a encolar en un

storage (más adelante veremos cuál va a ser es storage).

2- Tener un proceso corriendo cada X tiempo, el cual va a buscar mensajes

encolados en este storage y luego va a re-intentar enviar el mensaje.

3- Si en el re-intento vuelve a fallar, vuelve a encolar el mensaje en el storage

destinado para el caso de uso.

Con todos estos casos de uso, algunos básicos y otros más avanzados ya vamos describiendo el sistema que queremos lograr, a continuación veremos algunos casos de uso o esperables que el sistema también debe tener en cuenta y contemplar para que su funcionamiento no solo tenga buenas prestaciones sino que además sea robusto en aspectos que ya vimos en el capítulo 2, como lo son High Availability, Resiliencia, etc.

**4.2 Casos de uso aspiracionales**

1. Que los tópicos se puedan escalar horizontalmente.

Con esto se refiere a la posibilidad de que la cantidad de tópicos no esté limitada, es decir que la arquitectura que decida usar sea adecuada para poder escalar de manera horizontal, que es la única manera en la que el sistema puede crecer y no tener un límite.

1. Que la cantidad de suscriptores por tópico no sea un problema.

En este sentido luego de evaluar el funcionamiento y dada la problemática y solución (que está pensada para un entorno controlado en cuanto a la cantidad de suscriptores), la cantidad de suscriptores no debe ser un problema, a esto se refiere a que si bien un límite vamos a tener (dado que un mensaje lo debe enviar a N suscriptores) no tiene que ser un problema. Para esto tengo que tener en cuenta varios factores como son:

* El tiempo desde que el mensaje llega al sistema y se propaga a los N suscriptores.
* El posible uso del storage para los reintentos.
* Que la solución propuesta para el sistema, se asemeja al concepto de Broadcast en el entorno de redes, y dado que es un entorno controlado, se espera que la cantidad de interesados por los mensajes de un tópico, sea un número finito y conocido.

1. Que sea un sistema de alta performance.

Este es uno de los pilares fundamentales, va en sintonía con la motivación y agrega detalles en cuanto a la funcionalidad del sistema. Lo que quiero lograr con el sistema es que la dependencia del tiempo de propagación de los mensajes sea mínima (en el orden de los milisegundos), y no se vea afectada ni por la cantidad de mensajes, ni por la cantidad de suscriptores de un tópico, ambas problemáticas deben estar resueltas con la arquitectura y tecnología elegida para la solución. Esto va a permitir que el sistema ante casos extremos de tráfico tanto de entrada (mensajes publicados) como de salida (mensajes a los N suscriptores) no se vea afectado en su desempeño.

1. Que sea un sistema con resiliencia.

Con este caso nos referimos a lo visto en el capítulo 2, sobre todo siempre pensando que si bien la arquitectura que yo vaya a proponer parezca infalible, lo que hace al sistema vulnerable a fallas es principalmente el hardware sobre el cual montamos nuestro software. Si en nuestro diseño siempre tenemos esto en cuenta, vamos a poder pensar la solución al sistema como subsistemas o capas interconectadas, dado que si sucede algún problema por ejemplo de CPU, memoria e incluso energía, el sistema pueda ser resiliente a ello. Veremos a continuación como algunas arquitecturas de software ya conocidas a lo largo de la historia pueden ir allanando el camino de nuestro sistema propuesto en la presente tesis. Tal vez no sea una solución trivial, si seguramente sea una solución híbrida que combine aspectos de varias arquitecturas de software.

1. Que sea un sistema HA.

También relacionado al capítulo 2, precisamente a la sección de High Availability (alta disponibilidad), y de alguna manera con el resto de los casos de uso aspiracionales, esta también es una condición necesaria para el correcto funcionamiento del sistema. Algo comentaba en el punto anterior, y tal vez pensar al sistema en capas interconectadas sea una posible solución, esa interconexión debería ser el punto clave para lograr la alta disponibilidad, amén de seguir algunos principios básicos aplicables a cualquier software. Al tener una interconexión fuerte entre las capas, sumado a que las partes de cada capa deberían ser todas sin lógica propia (stateless), me va a permitir resolver el problema de la disponibilidad de manera rápida y con una complejidad media.

**4.3 Grails**

Muchos frameworks web modernos en el espacio de Java son más complicados de lo necesario y no adoptan los principios Don’t Repeat Yourself (DRY) [13].

Los marcos dinámicos como Rails y Django ayudaron a allanar el camino hacia una forma más moderna de pensar sobre las aplicaciones web. Grails se basa en estos conceptos y reduce drásticamente la complejidad de crear aplicaciones web en la plataforma Java. Sin embargo, lo que lo hace diferente es que lo hace basándose en tecnologías Java ya establecidas como Spring y Hibernate.

Grails es un marco de trabajo de stack completo e intenta resolver la mayor cantidad de piezas del rompecabezas de desarrollo web a través de la tecnología central y sus complementos asociados.

Grails se ha desarrollado con una serie de objetivos en mente:

* Ofrecer un framework web de alta productividad para la plataforma Java.
* Reutilizar tecnologías Java ya probadas como Hibernate y Spring bajo una interfaz simple y consistente.
* Ofrecer un framework consistente que reduzca la confusión y que sea fácil de aprender.
* Ofrecer documentación para las partes del framework relevantes para sus usuarios.
* Proporcionar lo que los usuarios necesitan en áreas que a menudo son complejas e inconsistentes:
  + Framework de persistencia potente y consistente.
  + Patrones de visualización potentes y fáciles de usar con GSP ([Groovy Server Pages](https://es.wikipedia.org/w/index.php?title=Groovy_Server_Pages&action=edit&redlink=1)).
  + Bibliotecas de etiquetas dinámicas para crear fácilmente componentes web.
  + Buen soporte de Ajax que sea fácil de extender y personalizar.
* Proporcionar aplicaciones ejemplo que muestran la potencia del framework.
* Proporcionar un entorno de [desarrollo orientado a pruebas](https://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas).
* Proporciona un entorno completo de desarrollo, incluyendo un servidor web y recarga automática de recursos.

Grails se ha diseñado para ser fácil de aprender, fácil para desarrollar aplicaciones y extensible. Intenta ofrecer el equilibrio adecuado entre consistencia y funcionalidades potentes.

**4.4 Redis**

Redis [14] es un almacén de estructura de datos en memoria de código abierto (con licencia BSD), que se utiliza como base de datos, caché y agente de mensajes. Redis proporciona estructuras de datos como cadenas, hashes, listas, conjuntos, conjuntos ordenados con consultas de rango, mapas de bits, hyperlog logs, índices geoespaciales y streaming. Redis tiene replicación incorporada, secuencias de comandos Lua, transacciones y diferentes niveles de persistencia en disco, y proporciona alta disponibilidad a través de Redis Sentinel y particionamiento automático con Redis Cluster.

Puede ejecutar operaciones atómicas en estos tipos, como agregar a una cadena; incrementar el valor en un hash; empujar un elemento a una lista; cómputo de intersección, unión y diferencia de conjuntos; o conseguir al elemento con la clasificación más alta en un conjunto ordenado.

Para lograr el máximo rendimiento, Redis trabaja con un conjunto de datos en memoria. Dependiendo de su caso de uso, puede conservar sus datos volcando periódicamente el conjunto de datos al disco o agregando cada comando a un registro basado en disco. También puede deshabilitar la persistencia si solo necesita un caché en memoria, en red y con muchas funciones.

Redis también admite la replicación asincrónica, con una primera sincronización sin bloqueo muy rápida, reconexión automática con resincronización parcial en la red.

Otras características incluyen:

* Transactions [15]
* Pub/Sub [16]
* [Lua](https://redis.io/commands/eval) scripting [17]
* Keys with a limited time-to-live [18]
* LRU eviction of keys [19]

**4.4 Redis**

4.1.1 Redis Pub/Sub

SUBSCRIBE, UNSUBSCRIBE y PUBLISH [16] implementan el paradigma de mensajería Publicar / Suscribir donde los remitentes (editores) no están programados para enviar sus mensajes a receptores específicos (suscriptores). Más bien, los mensajes publicados se caracterizan en canales, sin saber qué suscriptores pueden haber. Los suscriptores expresan interés en uno o más canales, y solo reciben mensajes que son de su interés, sin saber qué editores hay. Esta asociación de editores y suscriptores puede permitir una mayor escalabilidad y una topología de red más dinámica.

**4.5 Microservices Software Arq (UML)**

Ya descritos los conceptos necesarios para poder concretar la motivación que llevó a la elaboración de esta tesis presentada en el capítulo 1, como así también habiendo detallado las tecnologías a utilizar para llegar a la solución final del sistema, es momento de empezar a describir la arquitectura del mismo. Donde se mostrará por un lado el diagramas de componentes necesarios para llegar a la solución y por otro se detalla la responsabilidad de cada uno de los componentes utilizados.

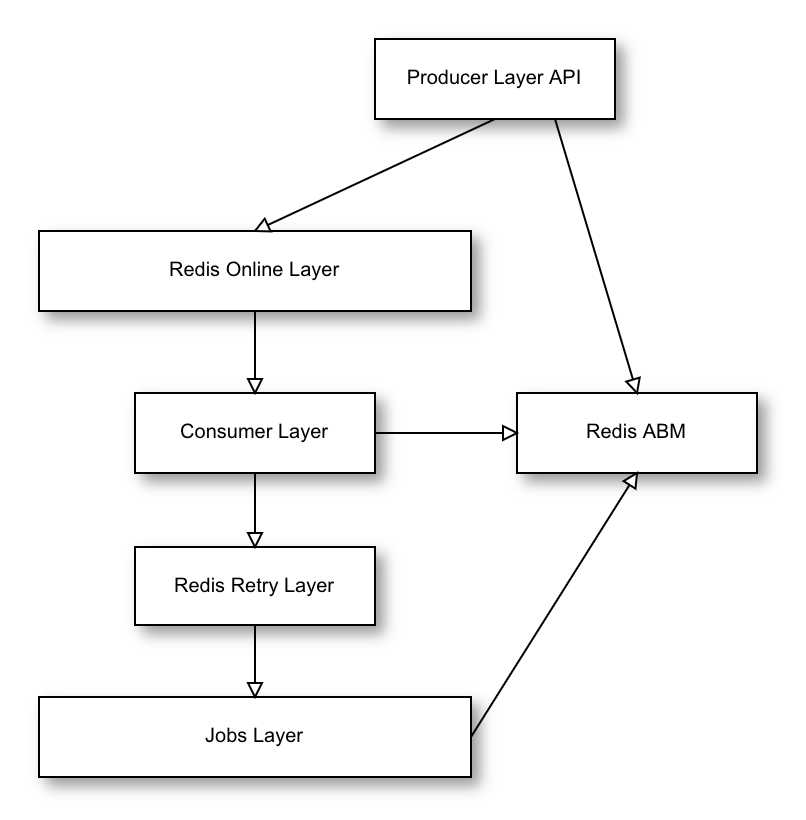


Diagrama de Componentes.

**4.5 Microservices Software Arq (UML)**

4.5.1 Producer Layer API

Primera capa funcional del sistema. Se desarrollará en Grails que fue descrito anteriormente, dadas las prestaciones de la tecnología y la posibilidad de poder tener un entorno Web exponiendo un servicio API Rest.

Esta capa tiene como primera responsabilidad exponer los servicios de administración del sistema, como fue descrito anteriormente. De manera resumida los servicios a exponer son:

* ABM de topicos: poder crear, borrar, modificar u obtener topicos.
* ABM de suscriptores: poder crear, borrar, modificar u obtener suscriptores.
* Poder publicar un mensaje en un tópico.
* Poder suscribirse a un tópico en particular con la posibilidad de aplicar también filtros dinámicos para recibir solo los mensajes que cumplan con una regla particular.

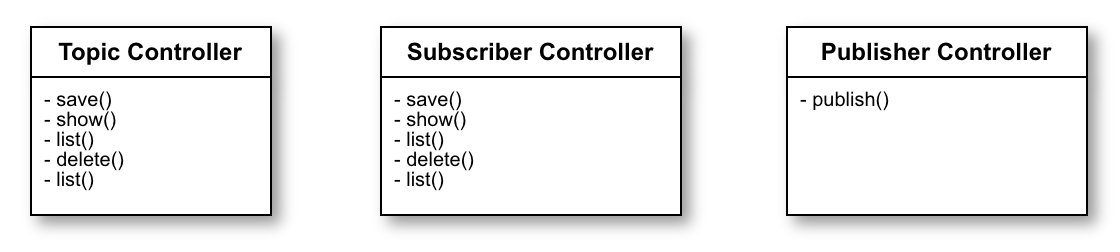
****

Diagrama de clases.

Esta capa es una capa Web, que tiene conexión directa con las capas de Redis Online y Redis ABM.

La capa Web en el caso de recibir una petición de los servicios de TopicController y SubscriberController va a tener interacción con operaciones primitivas de Redis en la capa Redis ABM.

Las operaciones primitivas de Redis para estas funcionalidades son:

* SET: Establecer la clave para contener el valor de la cadena. Si la clave ya tiene un valor, se sobrescribe, independientemente de su tipo. Cualquier tiempo de vida anterior asociado con la clave se descarta en la operación SET exitosa [20].
* GET: Obtiene el valor de la clave. Si la clave no existe, se devuelve el valor especial null. Se devuelve un error si el valor almacenado en la clave no es una cadena, porque GET solo maneja valores de cadena [21].
* DELETE: Elimina las claves especificadas. Una clave se ignora si no existe [22].
* KEYS: Devuelve todas las claves que coinciden con el patrón especificado como parámetro [23].

Para la funcionalidad de publicación de mensajes el servicio Web interactúa con la capa de Redis Online por medio del controller PublishController. Las operaciones primitivas de Redis para esta capa:

* RPUSH: Inserta todos los valores especificados al final de la lista asociada a la llave. Si la clave no existe, se crea como una lista vacía antes de realizar la operación de inserción. Cuando la clave contiene un valor que no es una lista, se devuelve un error [24].

Es posible enviar varios elementos usando una sola llamada de comando simplemente especificando múltiples argumentos al final del comando. Los elementos se insertan uno tras otro al final de la lista, desde el elemento más a la izquierda hasta el elemento más a la derecha. Entonces, por ejemplo, el comando *RPUSH mylist a b c* resultará en una lista que contiene a como primer elemento, b como segundo elemento y c como tercer elemento.

**4.5 Microservices Software Arq (UML)**

4.5.2 Consumer Layer

Segunda capa funcional del sistema, es también un web service desarrollado en Grails, es la que tiene la responsabilidad de interpretar por cada mensaje que llega a un tópico en particular conocer cuales son todos los suscriptores interesados en los mensajes, además tiene que tener control de las posibles reglas de filtrado que pueden tener los suscriptores para descartar un mensaje (filtrar) como así también en caso de no poder enviarlo a un determinado suscriptor, es responsable de encolarlo en la capa de Redis Retry, para un posterior intento de envío del mismo mensaje.

Desde el punto de vista funcional esta capa interactúa con las capas Redis Online y Redis Retry de la siguiente manera:

* En cuanto una instancia de web service de esta capa es iniciada, la primera tarea que tiene que ejecutar previo a marcar como disponible el servicio, es intentar conectarse con las instancias de la capa de Redis Online por medio de un comando de Redis, este comando es BLPOP [25], este comando es una primitiva que ejecuta un pop de lista con un bloqueo. Es la versión de bloqueo de LPOP [26] (otro comando) porque bloquea la conexión cuando no hay elementos para sacar de ninguna de las listas dadas. Cuando haya elementos, se extrae uno del encabezado de la primera lista que no está vacío y es retornado al proceso que está ejecutando el comando. Esto se diseña de esta manera para que el sistema funcione en el flujo online como un sistema NRT [27] (near real time) y siempre con estrategia de pushing para evitar cualquier tiempo muerto que podría tener un sistema de pooling .
* Una vez que el web service encuentra un nuevo mensaje, el mismo se tiene que conectar a la capa de Redis ABM para conocer cuales son los suscriptores que están interesados en el mensaje, una vez que tiene la lista de suscriptores comienza a enviarlos previo a validar si el suscriptor tiene alguna regla de filtrado, en caso de que tenga que enviarlo, intentar hacerlo, este intento lo hace por un determinado tiempo, en caso de que o bien este tiempo se termine (timeout) o que el suscriptor en cuestión esté inaccesible, lo que hace el web service es conectarse con la capa de Redis Retry para encolar el mensaje asociado a ese suscriptor para que a posteriori se intente enviar nuevamente.

Una particularidad de esta capa de web services, es que para cumplir con uno de los objetivos del sistema, que es la alta disponibilidad, en cuanto a cantidad de instancias en ejecución se van a tener al menos 2, y estas instancias se conectan a todas las instancias de la capa Redis Online, de esta manera siempre se va a tener al menos un web service de la capa de Consumers que van a estar intentando recibir mensajes nuevos.

Cabe aclarar que tanto esta capa de web service, como la de Producers y Jobs (descrita mas adelante), son capas consideradas stateless, cumpliendo también con otro de los objetivos del sistema de que sea agnóstico, de esta manera cualquier nueva instancia en cada uno de los web services, siempre cumple la misma función, esto permite que ante posibles problemas de infraestructura, los servicios puedan ser reemplazados sin ningún tipo de problema, como así también permitiendo escalar de manera horizontal esta capa.

**4.5 Microservices Software Arq (UML)**

4.5.3 Jobs Layer

Esta capa es la tercera capa funcional del sistema, también monta un servicio web sobre Grails, tiene como responsabilidad principal, poder reintentar los mensajes que bien puede ser por timeout del suscriptor, o por inaccesibilidad del mismo, no le llegaron en el flujo online.

Aca dependiendo de la particularidad del problema es la política de reintento que se puede aplicar, para este sistema lo que se optó es por reintentar por el periodo de 1 dia y cada 5 minutos, si se llegara a superar el periodo máximo de reintento el mensaje se perderá, esto es parte de lo que el sistema tiene como garantía de no pérdida de mensajes.

Otras funcionalidades que se tendrán también en esta capa, son todos aquellos procesos que pueden ser ejecutados en background para evitar sobrecarga del sistema. Una de ellas y que tiene un papel fundamental en la solubilidad del sistema, tiene que ver con el purgado de información vieja o que puede ser descartada dentro de los posibles storages del sistema, que son todas las capas de Redis que tiene el mismo. Los procesos relacionados a esta tarea, serán los encargados de ejecutar por ejemplo los comandos de blanqueo de los archivos de backup de Redis que por default serán configurados para persistir en disco local. De esta manera se evita un posible colapso o overquota del disco de cada Redis.

**4.5 Microservices Software Arq (UML)**

4.5.4 Redis Layer

Redis Layer, como lo especifica su nombre es una capa montada sobre Redis, pensada como un storage, pero con un tiempo muy reducido de permanencia de los datos en el mismo, está diseñada para que solo funcione como pasamano entre la capa de Producers y Consumers.

Amen de ser una capa de persistencia “volátil”, su criticidad para el sistema es alta, dado que en caso de no estar disponible el sistema deja de propagar mensajes entre productores y suscriptores.

Para que esta capa funcione con las prestaciones o premisas que se establecieron (Alta performance), lo que se decide es tener una configuración adecuada para cada una de las instancias que serán utilizadas, esta configuración es como sigue a continuación:

* Se comentarán las sentencias de configuración de persistencia en disco de la base propia de Redis (archivo .rdb propio de Redis), esto es para que funcione similar a una caché en memoria ram, de esta manera se logrará tener una performance óptima.
* Se activará el modo appendOnly [28], este modo permite a Redis, ir generando un archivo ordenada donde se van a ir escribiendo en secuencia todos los comandos de escritura que se ejecuten sobre el mismo, de esta manera ante una eventual caída del servidor, al restablecer el servicio se logra llegar exactamente al momento en que se cayó. Básicamente lo que aplica Redis en este caso es una lectura y ejecución de manera secuencial del archivo generado por el modo appendOnly, y una vez que termina, deja el servidor listo para ser usado con la misma foto que tenía en memoria ram al momento de caerse. Esta configuración es la recomendada por Redis cuando se quiere un sistema que sea Consistente (otra de las premisas de este sistema), porque asegura la NO pérdida de información ante una posible eventualidad con el servidor de datos.

**4.5 Microservices Software Arq (UML)**

4.5.5 Redis Retry Layer

Es la segunda capa de persistencia del sistema y también se basa en Redis. Esta capa es la encargada de persistir de manera semi permanente los mensajes que son encolados para reintentar su envío cuando sucede alguna eventualidad con un suscriptor (por ejemplo: timeout, caída del suscriptor, etc). La gran diferencia con respecto a la capa Redis Layer, es que en este caso las instancias serán usadas como un storage de tipo permanente, es decir que contendrán una base de datos local en cada instancia que se use.

Como particularidad de esta capa, se puede destacar que la configuración que se usará es para que por cada comando de escritura que se ejecute sobre la instancia, esto se guarde automáticamente en la base de datos local (archivo .rdb generado por Redis).

Dentro de las configuraciones de persistencia local de Redis hay muchas variantes, estas variantes son distintas combinaciones entre tiempo entre una persistencia en disco y otra, como así también dependiendo de la cantidad de ejecuciones de escritura que se ejecuten. Cabe aclarar que esta combinación juega un papel fundamental en la performance de la instancia de Redis. Para este sistema se evaluaron muchas variantes, y se optó por la que asegura consistencia y no pérdida de la información, esta configuración establece que cada un segundo o si se ejecuta un comando de escritura automáticamente persista en la base de datos local, de esta manera el sistema asegura el cumplimiento de las motivaciones que llevaron al diseño del mismo.

Hay dos puntos fundamentales a tener en cuenta en esta capa de persistencia:

1. El tamaño del disco está asociado a la instancia que se está usando: esto juega un papel fundamental en la capacidad que tendrá la base de datos a utilizar.
2. El tipo de disco que se esté utilizando en las instancias, en este caso va a ser un factor determinante en la performance de escritura en el mismo, para el sistema se aconseja utilizar disco de estado sólido o SSD, por tener mejores prestaciones en cuanto a performance de escritura como así también ante grandes cargas de información.

**4.5 Microservices Software Arq (UML)**

4.5.6 Subscribers Layer

Es la tercera y última capa de persistencia del sistema, también haciendo uso de Redis para seguir con la misma línea. A diferencia de las capas anteriores esta es una capa pensada para persistir información a largo plazo.

La información que el sistema almacenará aquí es la información asociada netamente a la funcionalidad de ABM del mismo. Esto es:

* Almacenará los tópicos creados en el sistema y toda la información asociada a los mismos.
* También almacenará los suscriptores que se dan de alta en el sistema, con las posibles configuraciones para especificar reglas de filtrado de mensajes.
* Y por último también almacenará la relación que se genera entre los tópicos y todos los posibles suscriptores. Esta información es la que será utilizada por la capa Consumers Layer para el envío de mensajes a los interesados.

Algo a tener en cuenta en esta capa es que tiene relación directa con dos piezas fundamentales del sistema:

* Con la capa Producers Layers, que será la encargada de brindar o exponer los servicios para el ABM tanto de tópicos como de suscriptores.
* Y con la capa Consumers Layer, que es la encargada de propagar los mensajes que llegan a los tópicos y enviarlos a los suscriptores.

En cuanto a la configuración particular que se tendrá en Redis, se optó por tener la misma configuración que en la capa Redis Retry (cada un segundo o si se ejecuta un comando de escritura automáticamente persista en la base de datos local), de esta manera se sigue cumpliendo con una de las motivaciones del sistema de que sea consistente y además asegure la NO pérdida de información.

De esta manera quedan descritas todas las capas del sistema, como así también como es la interacción entre ellas y la configuración adecuada para cada caso.

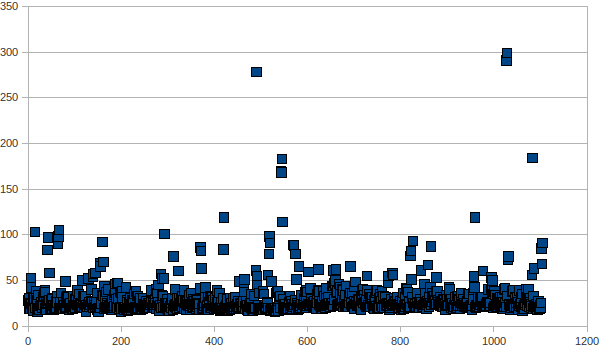
**4.6 Metrics**

Este tópico es fundamental en este tipo de sistemas, en este caso lo que se hizo fue dividir el concepto en dos.

Por un lado el sistema está preparado para poder enviar métricas real time a herramientas como NewRelic [29] o DataDog [30], que son herramientas disponibles en el mercado para poder almacenar datos en series de tiempo que posteriormente servirán para analizar y poder tomar decisiones.

Y por el otro también se decidió ejecutar pruebas que permitieran entender el comportamiento del sistema ante cargas recurrentes de tráfico, y de esta manera poder evaluar el comportamiento del mismo y ayudando a determinar si continúa cumpliendo con otra de las motivaciones del sistema, como lo son: alta performance, consistente y escalable, esta última como una de las principales pensando en el futuro uso del sistema.

El gráfico que se muestra a continuación muestra una prueba de 1100000 mensajes sobre 10 tópicos distintos, donde cada tópico tiene entre 1 y 1000 suscriptores (y asumiendo que los suscriptores están todos disponibles), y refleja el tiempo que demora un mensaje desde que ingresa en la capa de Producers Layer, hasta que sale de la capa Consumers Layer, es decir que pasa por todo el sistema, y demorando menos de 40 milisegundos en un percentil 99. Los puntos que se ven por fuera del percentil 99, son mensajes que demoraron en llegar a los suscriptores porque estos demoran en aceptarlos.



Lo que muestra la gráfica anterior es como haciendo un buen uso y configurando de manera correcta como se describió anteriormente cada pieza, se logra cumplir con las motivaciones iniciales del sistema.

Algo a tener en cuenta en este concepto de métricas son las operaciones que las instancias de las capas de Producers Layer y Suscribers Layer ejecutan sobre las instancias de la capa Redis Layer, ya que mucho del tiempo nombrado anteriormente depende de estas. A continuación se describe el funcionamiento de las operaciones que ejecutan tanto la capa que recibe el mensaje y lo almacena en Redis como también de quien lo consume de redis para enviarlo a los suscriptores interesados por este mensaje.

Operacion *RPUSH [24]:*

* Complejidad de tiempo: O (1) para cada elemento agregado.
* Descripción: Inserta el valor especificado al final de la lista. Cuando la clave contiene un valor que no es una lista, se devuelve un error.

Los elementos se insertan siempre al final de la lista, desde el elemento más a la izquierda hasta el elemento más a la derecha.

Operacion *BLPOP [25]*:

* Complejidad de tiempo: O (N) donde N es el número de claves proporcionadas, para el sistema descrito en esta tesis, se usará una única clave, por lo que la complejidad es O (1).
* Descripción: BLPOP [25] es una primitiva pop de una lista dada. Es la versión de bloqueo de LPOP [26] porque bloquea la conexión cuando no hay elementos para sacar de la lista. Se extrae un elemento del encabezado de la primera lista (cuando existe).
  + Comportamiento de bloqueo: si no existe ningún dato en la lista especificada, BLPOP [25] bloquea la conexión hasta que otro cliente realiza una operación LPUSH [31] o RPUSH [24] contra la lista.

Una vez que hay nuevos datos en la lista, el cliente regresa con el nombre de la clave que la desbloquea y el valor que aparece.

Cuando BLPOP [25] hace que un cliente se bloquee y se especifica un tiempo de espera distinto de cero, el cliente se desbloquea si no llega un mensaje antes de que expire ese tiempo, dando un error.

Se puede usar un tiempo de espera de cero para bloquear indefinidamente (como se hace en el sistema descrito aquí).

Las operaciones mencionadas como se ve, son de vital importancia para el sistema cuando intentamos hablar de performance, dado que la complejidad de tiempo que tiene es determinante. Estas operaciones aseguran que el sistema funcione con la misma performance para poca carga de trabajo como también para cargas excesivas, acompañando de esta manera a otro principio del sistema que es la escalabilidad.

**4.7 Limits**

Hasta el momento se viene hablando de las distintas decisiones que se han tomado para tener un sistema que cumpla con los puntos establecidos en el apartado de Objetivos del mismo, pero no hay que dejar de nombrar los posibles límites que puede tener, ya que al estar hablando de un sistema de software, sería absurdo pensar que el sistema no tiene límites duros.

Por el momento lo que se ha podido notar, luego de haber realizado pruebas exhaustivas y de carga en el sistema son límites que siempre están relacionados a temas de hardware:

* Los que están relacionados a temas de memoria física: en este sentido lo que se noto es que bajo las configuraciones especificadas en la description de capas donde se usarán como base de datos un Redis, hay que tener en cuenta en la definición, el tamaño del disco que tendrá cada instancia, dado que siempre (en cualquiera de las capas de donde se usa Redis) se estará escribiendo de una u otra manera en disco.
* Los límites de procesamiento: si bien las operaciones que ejecutan los web services sobre las instancias de Redis son de complejidad de tiempo lineal, hay un factor que puede afectar la performance del sistema y es el nivel de procesamiento o CPU que las instancias tengan, esto es porque por ejemplo en la capa Consumers Layer una vez que se recibe un mensaje (usando BLPOP) de una lista, este mensaje debe ser enviado a los suscriptores que están interesados en el mismo, este envío se ejecutará de modo recurrente usando hilos en Grails, estos hilos generan carga de CPU, por lo que muchos mensajes, enviados a muchos suscriptores es un tema importante a tener en cuenta.
* Y por último también se detectaron límites en el uso de memoria volátil o RAM: en este caso lo que generó problemas fue algo relacionado el punto anterior, que es la generación de hilos, esto también hace uso de una parte del stack de memoria, por lo que también que ser un parámetro a tener en cuenta cuando intentamos responder la pregunta, cuántos suscriptores tendrán cada tópico como máximo?

**5. Conclusión y Anexos**

**5.1 Conclusion**

A lo largo del presente informe se mostró cómo a partir de la aplicación de distintos patrones arquitectónicos se puede llegar a un sistema que cumple con muchas características que tal vez pensando en un único patrón nunca las podríamos lograr.

Para esto, fue importante enfocarse primero sobre las arquitecturas que desde el punto de vista del enfoque inicial del informe, podrían estar relacionadas, intentando describir de manera clara cada una de ellas, y tomando lo que, bajo el criterio de los objetivos del informe, era lo mejor de cada una en ese sentido.

Después fue momento de tomar decisiones que están relacionadas por un lado con la usabilidad del sistema y por el otro en las prestaciones del mismo desde el punto de vista de la Ingeniería del Software.

Para hacer al sistema fácilmente usable, se optó luego de entender las posibilidades que existen en el mercado, por un conjunto de servicios REST que al día de hoy, son los más usados mundialmente, dado que son autodescriptivos y con definiciones claras como el uso de URLs, verbos o acciones y códigos de respuesta.

En cuanto a las prestaciones del sistema, dada la experiencia en el desarrollo de aplicaciones para grandes cantidades de tráfico y procesamiento de información, la decisión fue inclinada a usar tecnologías que por un lado estuvieran alineadas con los objetivos del sistema como así también con lo más usado en el mercado.

Cabe aclarar que sistemas como el descrito en el presente informe, actualmente ayudan a resolver problemáticas de grandes compañías como Google o Facebook por nombrar algunas de las más importantes, en el afán de propagar información de manera rápida, eficiente y consistente.

Durante el desarrollo del informe, si bien no se comenta, o en algunos casos se nombran de manera resumida algunos, se investigaron temas que fueron super necesarios para el desarrollo del mismo: para poder ejecutar pruebas de carga se hizo investigación de herramientas que permitieran ejecutar de manera sostenida y bajo distintos contextos pruebas que nos permitieran asegurar que el sistema cumpliria con los objetivos planteados, tambien se participó de charlas y conferencias donde se profundiza en el buen uso y configuracion de Redis como base de datos que fue vital para poder tomar la decisión de storage.

Y por último, el sistema surge de la premisa de poder “replicar” un mensaje en varios, para poder de esta manera propagarlo a todos los interesados en el mismo, pero también pensando que estos mensajes pueden pertenecer a un centenar de temas diferentes y que los interesados puede ser miles por cada tema, haciendo que el sistema sea de un tamaño considerable y en constante crecimiento, todo lo descrito hasta acá permite asegurar que esto se cumple.

**5.2 Anexos**

5.2.1 Códigos de respuesta HTTP

HTTP define cuarenta códigos de estado estándar que se pueden utilizar para transmitir los resultados de la solicitud de un cliente. Los códigos de estado se dividen en las cinco categorías:

* 1xx: Informativo - Comunica información a nivel de protocolo de transferencia.
* 2xx: éxito - Indica que la solicitud del cliente se aceptó correctamente.
* 3xx: redirección - Indica que el cliente debe realizar alguna acción adicional para completar su solicitud.
* 4xx: Error del cliente - Esta categoría de códigos de estado de error apunta con el dedo a los clientes.
* 5xx: Error del servidor - El servidor asume la responsabilidad de estos códigos de estado de error.

Veamos un poco más de detalle de cada status http:

* 200: Indica un éxito inespecífico
* 201: Enviado principalmente por colecciones y tiendas, pero a veces también por controladores, para indicar que se ha creado un nuevo recurso.
* 202: Enviado por controladores para indicar el inicio de una acción asincrónica
* 204: Indica que el cuerpo se ha dejado en blanco intencionalmente
* 301: Indica que se ha asignado un nuevo URI permanente al recurso solicitado por el cliente.
* 303: Enviado por los controladores para devolver los resultados que considera opcionales
* 304: Enviado para preservar el ancho de banda (con GET condicional)
* 307: Indica que se ha asignado una URI temporal al recurso solicitado por el cliente.
* 400: Indica un error de cliente inespecífico
* 401: Se envía cuando el cliente proporcionó credenciales no válidas o se olvidó de enviarlas
* 402: Enviado para denegar el acceso a un recurso protegido
* 404: Enviado cuando el cliente intentó interactuar con un URI que la API REST no pudo asignar a un recurso
* 405: Enviado cuando el cliente intentó interactuar con un método HTTP no compatible
* 406: Enviado cuando el cliente intentó solicitar datos en un formato de tipo de medio no compatible
* 409: Indica que el cliente intentó violar el estado del recurso.
* 412: Le dice al cliente que no se cumplió una de sus condiciones previas
* 415: Enviado cuando el cliente envió datos en un formato de tipo de medio no compatible
* 500: Le dice al cliente que la API tiene sus propios problemas

**5.2 Anexos**

5.2.2 Posibles mejoras que se pueden aplicar: brpoplpush

Una posible mejora que se puede hacer al sistema es en la capa de Consumers Layer. Esto tiene que ver con la operación que se decidió usar para consumir los mensajes de Redis. La operación elegida fue BLPOP [25] (descrita en la sección “Consumers Layer” y “Metrics”).

La mejora consiste en cambiar de operación, y usar BRPOPLPUSH [32]. Esta operación se describe a continuación:

* Complejidad de tiempo: O (1) para cada elemento agregado.
* Descripción: este comando tiene la posibilidad primero de consumir de una lista un mensaje (de manera bloqueante igual que BLPOP [25]), y además dejar una copia de ese mensaje en una lista usada como backup. De esta manera se puede brindar aún más seguridad de que la información no se perderá, por ejemplo ante un evento de caída de instancias en la capa Consumers Layer. Para lograr esto último lo que se puede hacer es lo siguiente.
  1. Para que los mensajes no se acumulen de manera indefinida en la lista de backup, es recomendable que luego de procesar el mensaje de la lista original, se ejecute una operación LREM [33] de Redis, para eliminar el mensaje copiado en la lista de backup, de esta manera aseguramos que el mensaje no ocupará espacio en la base de datos, evitando tener problemas de límites.
  2. En la capa de Jobs Layer se puede agregar una tarea para ejecutar en background, esta tarea consistirá en recorrer la lista backup, analizar si el mensaje lleva ahí más de X cantidad de tiempo, y en caso de ser así, consumirlo y retroalimentar la lista original. Este mecanismo es el que permite recuperarse ante posibles eventos en la capa Consumers Layer, por ejemplo: una caída de una instancia por problemas de memoria o CPU.

**5.2 Anexos**

5.2.3 Informacion para Analytics

Algo que puede ser agregado al sistema, es una capa más de persistencia, donde se puede almacenar información de que mensajes llegan a que tópicos, y además a quienes se les envió o no ese mensaje, esto tiene dos posibles usos importantes:

1. Poder hacer troubleshooting ante problemas de envío o recibimiento de mensajes. Esto en un sistema de alta performance es un punto muy importante a tener en cuenta, dado que suele ser usado y hasta ha ayudado a resolver problemas de trazabilidad.
2. Para métricas, teniendo esta capa de persistencia, se puede agregar mucha más información, como por ejemplo: tiempo de respuesta de los suscriptores de mensajes, tamaño de los mensajes, respuestas de los suscriptores en el recibimiento de los mensajes. Esta información puede servir para tomar decisiones por ejemplo de: notificar al dueño de un suscriptor porque el mismo está procesando de manera lenta los mensajes, o porque los códigos de respuesta comenzaron a ser de error, etc.

**5.2 Anexos**

5.2.4 Chaos Engineering

Se destina un apartado para este tema ya que es importante tenerlo en cuenta en los días que corren, es una técnica ampliamente utilizada por grandes compañías como se manifiesta en [34] y merece dedicarle unos minutos dentro de este documento de tesis, y que sin dudas está relacionado al sistema que se quiere lograr.

Un aspecto que marca lo mucho o poco que puede estar trabajado un sistema en relación con los aspectos antes nombrados, y es la aplicación del concepto de *Chaos Engineering* (ingeniería del caos).

Este se define como el proceso de prueba de un sistema informático distribuido para garantizar que el sistema pueda soportar interrupciones inesperadas en su función. Se llama así porque se basa en conceptos de la teoría del caos, que se centra en el comportamiento aleatorio e impredecible en este ámbito, aplicado al desarrollo del software.

*¿Qué diferencia el Chaos Engineering del Testing tradicional?*

La gran diferencia es el hecho de que consiste en una práctica para generar información nueva en el momento que una pieza de software falla (intencionalmente), mientras que el testing tradicional consiste en probar una condición particular del sistema esperable.

Una forma de introducirnos en este tipo de pruebas, es pensar que el sistema por ejemplo puede presentar problemas de latencia, o comunicación entre las diferentes piezas del mismo.

*Fundamentos del Chaos Engineering*

Es la ejecución de experimentos para generar confianza en el comportamiento del sistema. Basado en los principios, se define un experimento:

1. Comience por definir el "estado estable" como un resultado medible de un sistema que indica comportamiento normal
2. Haga una hipótesis que este estado estable continuará tanto en el grupo de control como en el grupo experimental.
3. Introduzca variables que reflejan eventos del mundo real, como servidores que se bloquean, discos duros que funcionan mal, conexiones de red que se cortan.
4. Intente refutar la hipótesis buscando una diferencia en el estado estable entre el grupo de control y el grupo donde se produce el experimento.

Una vez definidos todos los conceptos necesarios para tener una base, a continuación se empezaran a tratar contenidos más enfocados en las arquitecturas de software que van a acompañar al desarrollo del sistema, ya sea de manera directa o a través de la combinación de algunas de ellas.

**6. Bibliografia**

[1] https://histinf.blogs.upv.es/2010/12/28/ingenieria-del-software/

[2] http://pegasus.javeriana.edu.co/~mad/Generalidades%20de%20las%20Arquitecturas%20de%20SW.pdf

[3] Institute of Electrical and Electronics Engineers (IEEE), IEEE Software Engineering

Standards Collection. IEEE Standard 610.12-1990, 1993.

[4] Rest vs WebServices Rafael Navarro Marset. ELP-DSIC-UPV

Modelado, Diseño e Implementación de Servicios Web 2006-07

[5] JSON at Work: Practical Data Integration for the Web - Tom Marrs - O’REILLY - 2017

[6] Learning XML, 2nd Edition - Erik T. Ray - O’REILLY - 2003

[7] REST API Design Rulebook - Mark Masse - O'Reilly - 2011

[8] https://csce.ucmss.com/books/LFS/CSREA2017/SER6068.pdf Resilience Methods within the Software Development Cycle (Acklyn Murray, Marlon Mejias, Peter Keiller) - 2017

[9] https://jimgray.azurewebsites.net/papers/ieee\_HA\_Swieorick.pdf

High Availability Computer Systems (Jim Gray Daniel P. Siewiorek) - 2018

[10] Pattern-Oriented Software Architecture - Volume 1 - Frank Bushman, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal - Wiley - 1996

[11] Software Architecture Patterns - Mark Richards - O’REILLY - 2015

[12] Content-based Publish/Subscribe Systems - Haiying Shen - 2010 - http://www.cs.virginia.edu/~hs6ms/publishedPaper/bookChapter/2009/sub-pub-Shen.pdf

[13] Grails - Introduction - https://docs.grails.org/4.0.6/guide/single.html#introduction

[14] Introduction to Redis - www.redis.io - https://redis.io/topics/introduction

[15] Redis Transactions - www.redis.io - https://redis.io/topics/transactions

[16] Redis Pub/Sub - www.redis.io - https://redis.io/topics/pubsub

[17] Lua scripting - www.redis.io - https://redis.io/commands/eval

[18] Redis Expiring - www.redis.io - https://redis.io/commands/expire

[19] LRU Cache - www.redis.io - https://redis.io/topics/lru-cache

[20] Redis Command SET - www.redis.io - https://redis.io/commands/set

[21] Redis Command GET - www.redis.io - https://redis.io/commands/get

[22] Redis Command DEL - www.redis.io - https://redis.io/commands/del

[23] Redis Command KEYS - www.redis.io - https://redis.io/commands/keys

[24] Redis Command RPUSH - www.redis.io - https://redis.io/commands/rpush

[25] Redis Command BLPOP - www.redis.io - https://redis.io/commands/blpop

[26] Redis Command LPOP - www.redis.io - https://redis.io/commands/lpop

[27] Practical Real-time Data Processing and Analytics - Shilpi Saxena, Saurabh Gupta - Packt Publishing - 2017

[28] Redis Persistence - www.redis.io - https://redis.io/topics/persistence

[29] NewRelic Official Web Site - www.newrelic.com

[30] Datadog: Cloud Monitoring as a Service - www.datadoghq.com

[31] Redis Command LPUSH - www.redis.io - https://redis.io/commands/lpush

[32] Redis Command BRPOPLPUSH - www.redis.io - https://redis.io/commands/brpoplpush

[33] Redis Command LREM - www.redis.io - https://redis.io/commands/lrem

[34] Chaos Engineering - Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones & Ali Basiri - O’reilly. Compliments of Netflix - 2015

[35] SOA with REST - Thomas Erl, Benjamin Carlyle, Cesare Pautasso, and Raj Balasubramanian - Prentice Hall - 2012

[36] Fundamentals of Software Architecture: An Engineering Approach - Mark Richards, Neal Ford - O'Reilly - 2020