Propuesta de arquitectura de comunicación entre sistemas heterogéneos para la APPM

Joaquín Roberto Lima[1,2], Francisco Ezequiel Paez[1], Marcos Fernando Tidona[1,2]

[1] Depto. de Informática - Facultad de Ingeniería - Universidad Nacional de la Patagonia San Juan Bosco Sede Puerto Madryn, Chubut, Argentina
[2] Administración Portuaria de Puerto Madryn, Muelle Almirante Storni Ruta Prov. Nro. 1 S/N, Puerto Madryn, Chubut, Argentina
http://www.appm.com.ar

jlima@appm.com.ar
fep@ing.unp.edu.ar
ftidona@ing.unp.edu.ar

Resumen Este trabajo presenta el proyecto final para la aprobación de la cursada de la materia Sistemas Distribuidos de la carrera Licenciatura en Informática, dictada en la Facultad de Ingeniería en la Sede Puerto Madryn de la Universidad Nacional de la Patagonia San Juan Bosco. La cátedra propuso desarrollar como trabajo final una propuesta para mejorar o resolver defectos en alguno de los servicios a cargo del área de sistemas de la Administración Portuaria de Puerto Madryn (APPM), lugar de trabajo del autor principal. En base a esto, se propone una nueva arquitectura para el servicio de estación meteorológica en el Puerto Almirante Storni de la ciudad, que adolece de un problema cuya solución esta dentro del marco de los sistemas distribuidos. Este sistema es el de menor complejidad entre los actualmente operativos en el puerto y es el que generaría un menor impacto operativo en el caso de decidir actualizar la arquitectura actual. La arquitectura propuesta hace uso de Apache Kafka para mejorar la disponibilidad y la tolerancia a fallos, entre otros requerimientos, del sistema actual.

Keywords: Tolerancia a fallas · Kafka · APPM

1. Introducción

El proyecto surge de la propuesta de la cátedra de la asignatura Sistemas Distribuidos de la carrera Licenciatura en Informática de realizar un trabajo final relacionado con el ambiente laboral del autor.

Entre los trabajos del Área de Sistemas de la Administración Portuaria de Puerto Madryn[3] (APPM), la estación meteorológica presenta un buen caso de estudio por su simplicidad y porque adolece de un problema que tiene tratamiento

[3] https://www.appm.com.ar/

dentro del marco de los sistemas distribuidos. De los sistemas críticos que funcionan en el Puerto la estaciónmeteorológica es la que generará menor impacto en la operativa portuaria en el caso de que se decida migrar a la arquitectura propuesta en el presente trabajo.

1.1. Caso de estudio

La APPM brinda diversos servicios a los buques que llegan al puerto y a la comunidad de Puerto Madryn. Uno de estos servicios es el reporte de las condiciones climáticas en el muelle Almirante Storni. Se dispone para ello de una estación meteorológica ubicada en el muelle. Los datos del clima pueden ser consultados en la web[4] y son utilizados para determinar la operativa portuaria del día.

El proceso completo comienza con la captura de datos y termina con la presentación de los mismos en una página web, cuya arquitectura simplificada se presenta en la Figura 1.

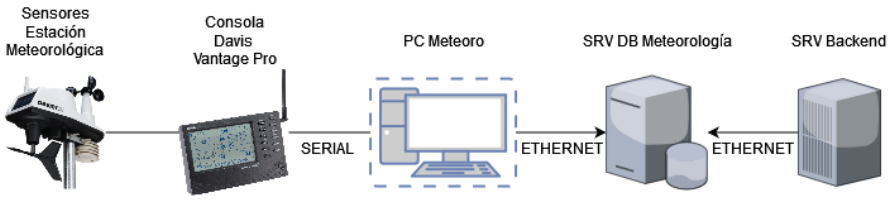

Figura 1. Arquitectura actual del sistema de meteorología de la APPM.

Los datos meteorológicos son generados por la estación meteorológica y recopilados en una consola *Davis Vantage Pro*. Una PC (denominada Meteoro) consulta estos datos cada 5 minutos mediante un *script* Perl y los inserta en un registro histórico en la base de datos de meteorología. Esta base de datos posee dos tablas, una para datos históricos (*clima_historico*) y otra para los datos más recientes (*clima_actual*, con una sola fila).

La base de datos meteorológica es consultada por una aplicación PHP en otro servidor, que procesa y transforma los datos. El manejo de la información y cómo llega al sitio web de la APPM no es de interés para el presente trabajo.

1.2. El problema

El principal defecto de la arquitectura actual es su falta de tolerancia a fallas. Ante la caída de cualquier equipo o enlace, la pérdida de información es inevitable. Aunque en el caso de la APPM los enlaces de comunicación no suelen

[4] `https://meteorologia.appm.com.ar`

fallar, si es normal que ocurran cortes de energía eléctrica totales o parciales que afectan a los equipos. Otros inconvenientes son:

- Alta acoplamiento entre cada componente. Por ejemplo, actualizar o cambiar el motor de base de datos obliga a readecuar cada componente.
- La seguridad, autenticación y control de acceso a los datos no depende de la arquitectura.
- No hay una API estándar para que futuros componentes software accedan a los datos, cada uno debe implementar su propio mecanismo de consulta (por ejemplo, usando SQL).
- No es eficiente que la aplicación PHP realice consultas continuas a la base de datos para conocer los datos de la estación.
- Se usa la base de datos como medio de comunicación entre dos sistemas, cuando solo debería almacenar información.

1.3. Objetivos específicos

Los objetivos específicos del presente trabajo son:

1. Investigar herramientas que permitan solucionar los problemas identificados.
2. Comparar herramientas para determinar cuál aplicar en el escenario planteado.
3. Desarrollar una arquitectura que solucione los inconvenientes detectados.
4. Evaluar en un entorno de prueba la nueva arquitectura frente a la ausencia de uno o varios nodos.

Dado que el caso de estudio es un entorno operativo del puerto, no se puede realizar pruebas que involucren la caída del mismo. Por lo tanto, la evaluación de caídas de nodos/servidores se realizo en un entorno de pruebas con un generador de datos meteorológicos simulado y un programa que sirve como receptor de datos.

2. Desarrollo

2.1. Selección de herramienta

Para satisfacer un cierto grado de tolerancia a fallas se investigaron herramientas de mensajería que permitan interconectar sistemas mediante un patrón de publicación-subscripción (*pub/sub*) [1]. Teniendo en cuenta las tendencias en empresas modernas (Linkedin, Facebook, Twitter, Tinder, Yahoo, Adobe, Hulu, Amazon, etc) se estudiaron 3 plataformas: RabbitMQ[5], Redis[6] y Kafka[7].

Para seleccionar la herramienta más adecuada se realizó una comparación, resumida en la Tabla 1.

Aunque cualquiera de las tres plataformas satisface los requerimientos para resolver el problema planteado, se valoraron los siguientes aspectos:

[5] `https://www.rabbitmq.com/`
[6] `https://redis.io/`
[7] `https://kafka.apache.org/`

Cuadro 1. Comparación de plataformas

	RabbitMQ	Apache Kafka	Redis Pub/Sub
Tasa (msg/sec)	50.000	1.000.000	1.000.000
Persistencia	soportada	soportada	no por defecto
Tamaño máximo de mensaje	2GB	1GB	1MB
Seguridad	SSL	TSL, JAAS	TSL, ACL
Disponibilidad	cluster y replicación	cluster y replicación	cluster

1. **Desempeño:** Kafka y Redis son mejores en este aspecto.
2. **Persistencia:** RabbitMQ y Kafka lo hacen de forma natural.
3. **Alta disponibilidad:** Todos lo contemplan, pero en Redis recién en las últimas versiones.
4. **Seguridad y gestión de errores:** Todos lo manejan a un nivel similar.

Se seleccionó Kafka como la herramienta para el desarrollo de la nueva arquitectura. Si bien Redis es una buena alternativa, se entiende que está más orientada al envío ágil de mensajes de corta duración, con tolerancia a pérdidas esporádicas.

2.2. Aquitectura propuesta

A fin de lograr una arquitectura tolerante a fallos, se armó un clúster Kafka compuesto por nodos redundantes con datos replicados, cada uno alojado en servidores físicamente separados.

Dado que Kafka es un *middleware* de mensajería que emplea el patrón *pub/sub*, la arquitectura planteada puede incorporar fácilmente otra estación meteorológica (en el vecino muelle Piedrabuena de la misma ciudad) y comunicar otros sistemas externos con el sistema principal del puerto.

Clúster Kafka El clúster de Kafka propuesto está compuesto por tres servidores ubicados físicamente en distintos lugares. A su vez, cada servidor tiene un nodo controlador (*controller*) y un nodo agente (*broker*), ilustrado en la Figura 2.

De entre el grupo de controladores (*controller quorum*) se elige un líder (*leader*) quien administra los metadatos del clúster y la comunicación con los *brokers*. El resto de los controladores replican los datos escritos en el lider y actúan como resguardo (*hot standby*) ante cualquier fallo. El lider también se conoce como *active controller* y sólo puede haber uno dentro de un clúster Kafka. En versiones recientes, Kafka utiliza *Apache Kafka Raft* (*KRaft*) como protocolo de consenso, en reemplazo de *Apache ZooKeeper*. Cada *broker* funciona de manera similar, pero empleando particiones (*partitions*) y tópicos (*topics*) [3].

Los parámetros de configuración más importantes del clúster son:

- Tres particiones por topico (*num partition*)

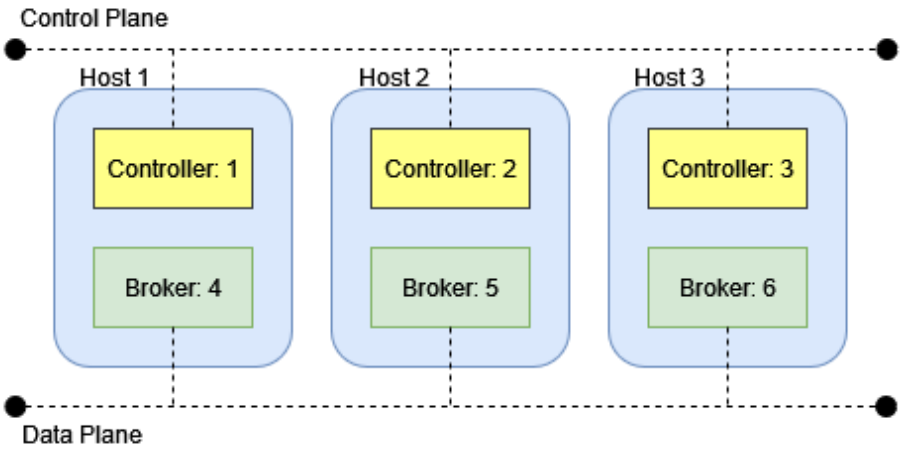

Figura 2. Vista de tres servidores con controladores y *brokers*.

- Tres réplicas por topico (*replication factor*)
- El mínimo de réplicas en sincronía es uno (*min insync replicas*)
- Tres réplicas por topico *__consumer_offsets* (*offsets topic replication factor*)

En esta configuración en todo momento hay tres particiones por topico y a su vez, tres réplicas de cada partición distribuidas en los *brokers* (la partición que es lider también se considera réplica a fines de contar la cantidad de réplicas por *broker*). Una de las réplicas actúa como líder de esa partición y las demás quedan en sincronía con el lider (ISR, *In-Sync Replicas*).

Si se pierde comunicación con el líder de una partición, una réplica de la misma partición en otro nodo puede tomar el rol de líder (siempre que sea ISR).

Disponibilidad Dado N (*replication factor*) y M (*min insync replicas*) se puede expresar:

Topic Data Durability: tolera hasta $N - 1$ *brokers* caídos

Topic Data Availability: tolera hasta $N - M$ *brokers* caídos

Aplicado a la configuración propuesta, se tolera la caída de hasta dos *brokers*. Vale aclarar que dejar el mínimo de réplicas en sincronía en uno implica que un topico va a estar disponible para escritura si al menos hay una réplica en ISR. Esto quiere decir que está garantizada la escritura en al menos una partición [2].

Esta decisión de diseño fue tomada por considerar más aceptable la disponibilidad del sistema frente a las caídas de componentes, que la completitud de los datos. Cómo se garantiza sólo una escritura (la del lider), puede ocurrir que la caída de un nodo ocurra justo antes de que los datos de la partición se hayan replicado, por lo que cuando otro asuma como líder, se pierda ese dato. Esto afectaría una lectura de la estación, pero se puede considerar una pérdida aceptable.

Entorno de desarrollo Para armar el entorno de pruebas, se usaron cuatro máquinas virtuales (VM):

- Tres VM para armar el clúster Kafka.
- Una VM para ejecutar el programa que simula la lectura y envio de datos, y su posterior procesamiento.

El despliegue de Kafka en cada VM se hace mediante Docker, que permite agilizar la configuración y administración. Se armó un archivo *docker-compose.yml* para cada *host* del clúster (Figura 2).

- **Host 1:** Con IP 172.16.200.13 y configuración del controlador 1 y *broker* 4.
- **Host 2:** Con IP 172.16.200.14 y configuración del controlador 2 y *broker* 5.
- **Host 3:** Con IP 172.16.200.15 y configuración del controlador 3 y *broker* 6.

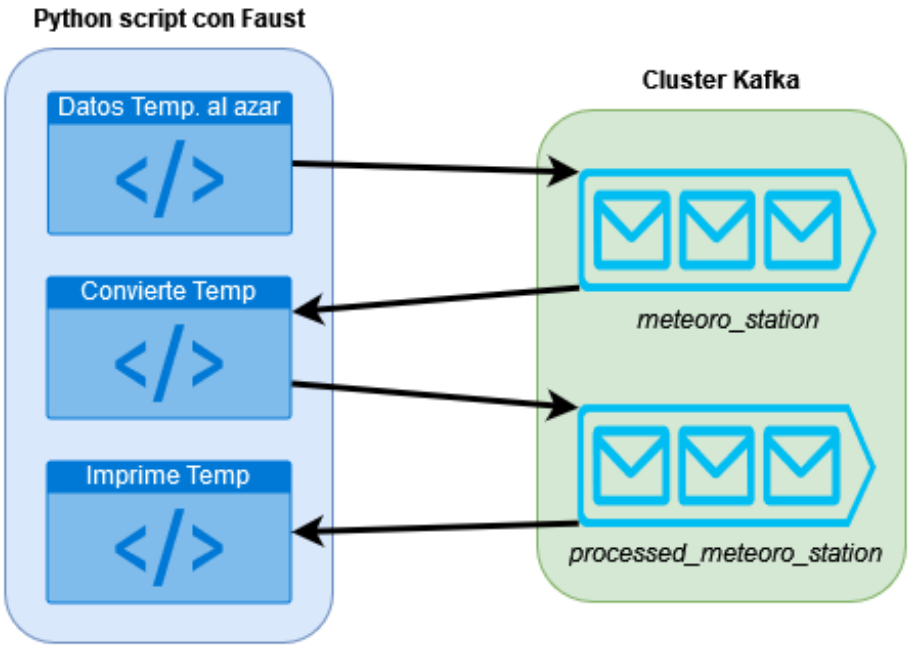

Figura 3. Programa en Python para probar el clúster Kafka

La cuarta VM ejecuta un simple programa en Python (Figura 3). El programa utiliza Faust[8] y consta de una tarea y dos agentes que se comunican con dos tópicos del clúster:

[8] `https://faust.readthedocs.io/`

- ***meteoro_station***: Para los datos generados de la estación meteorológica, sólo se tiene el valor de la temperatura medida en grados Fahrenheit.
- ***processed_meteoro_station***: Se usa para almacenar otros valores de temperatura (Celsius, Kelvin, etc). Se hace la conversión tomando como base el valor en Fahrenheit.

Si bien el segundo tópico no es esencialmente necesario para probar el clúster, fue útil para probar el procesamiento de los datos en tiempo real que ofrece Faust, similar a Kafka Streams.

Una tarea periódica genera cada 5 segundos datos al azar de temperatura y luego lo envía al tópico *meteoro_station*. Un agente que está suscrito a dicho tópico consume el mensaje, lo procesa (convierte a otras unidades de medida) y luego lo envía al tópico *processed_meteoro_station*. Un segundo agente suscrito a este tópico, consume los datos de temperatura y los imprime por la salida estándar.

```
Topic: meteoro_station  TopicId: lu1vHkb8QMajQOCzmRsVZA PartitionCount: 3       ReplicationFactor: 3
        Topic: meteoro_station  Partition: 0    Leader: 5       Replicas: 5,6,4 Isr: 5,6,4
        Topic: meteoro_station  Partition: 1    Leader: 6       Replicas: 6,4,5 Isr: 6,4,5
        Topic: meteoro_station  Partition: 2    Leader: 4       Replicas: 4,5,6 Isr: 4,5,6
```

Figura 4. Estado del topic *meteoro_station* en el clúster

```
Topic: processed_meteoro_station        TopicId: veDbv1eBSYaWMK0Hq27QLA PartitionCount: 3       ReplicationFactor: 3
        Topic: processed_meteoro_station        Partition: 0    Leader: 5       Replicas: 5,6,4 Isr: 5,6,4
        Topic: processed_meteoro_station        Partition: 1    Leader: 6       Replicas: 6,4,5 Isr: 6,4,5
        Topic: processed_meteoro_station        Partition: 2    Leader: 4       Replicas: 4,5,6 Isr: 4,5,6
```

Figura 5. Estado del topic *processed_meteoro_station* en el clúster

3. Resultados

Con el clúster ejecutando, se crearon los dos tópicos necesarios y luego se ejecutó el programa en Python. La captura en la Figura 6 muestra como se genera un valor al azar de temperatura para un muelle en cada instante[9]. Hay dos salidas por cada "lectura del sensor", una antes y otra después del procesamiento de la temperatura (conversión). Luego se procedió a realizar la prueba que simula las caídas de los nodos del clúster. El orden fue el siguiente:

1. El nodo 1 (172.16.200.13) con el controlador 1 y *broker* 4.
2. El nodo 2 (172.16.200.14) con el controlador 2 y *broker* 5.

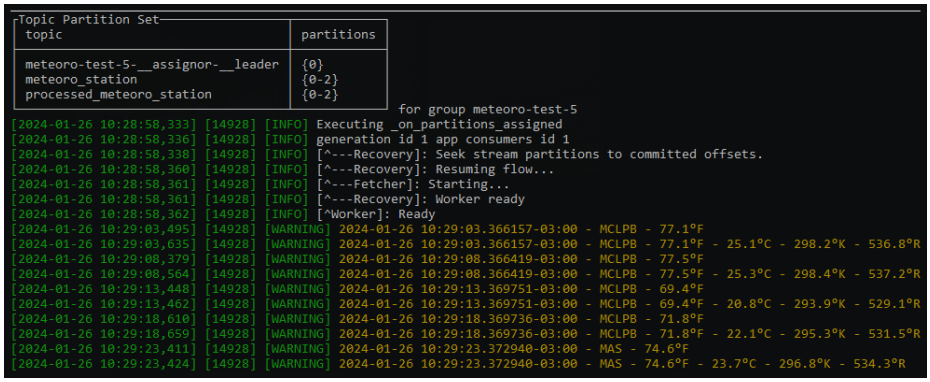

Figura 6. Visualización de los datos de temperatura generados

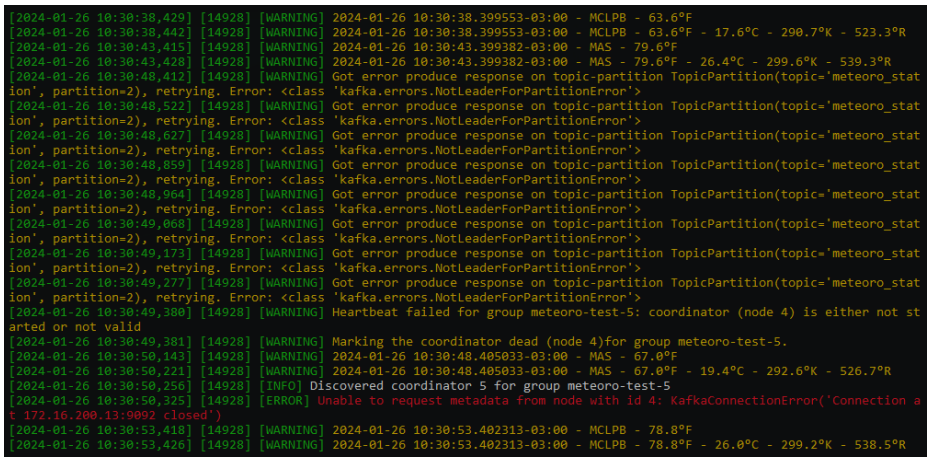

Figura 7. Visualización de los datos generados frente a la primera caída

3.1. Primera caída

En la Figura 7 se puede apreciar que a pesar de la desconexión del nodo 1, el sistema continúa enviando datos de temperatura. Observando las marcas de tiempo (*timestamp*), no hay pérdida de información por la caída del nodo 1.

[9] MAS, Muelle Almirante Storni
MCLPB, Muelle Comandante Luis Piedra Buena

3.2. Segunda caída

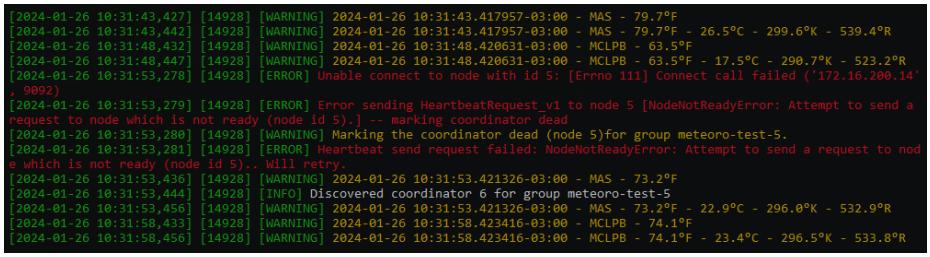

Figura 8. Visualización de los datos generados frente a la segunda caída

Igual que antes, en la Figura 8 se presenta el error por perder la conexión con el nodo 2 pero aún así el sistema continúa funcionando y no se pierden datos.

3.3. Tópico meteoro_station

En la Figura 9 se presenta una captura que muestra el estado del tópico *meteoro_station* en las siguientes situaciones:

- Funcionamiento normal (ningun nodo caído).
- Falla de comunicación con el nodo 1.
- Falla de comunicación con el nodo 1 y 2.

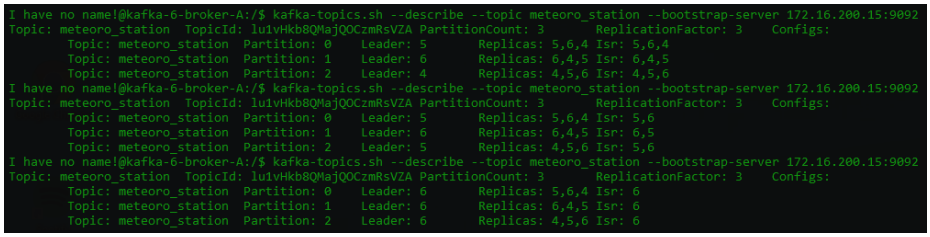

Figura 9. Estado del topíc frente a las caídas simuladas

Ante cada caída el líder de cada partición cambia en función de las réplicas ISR disponibles en otros nodos. En la última situación, todas las particiones del *broker* 6 son líderes. Esto es lo que se busco lograr al armar el clúster de Kafka con la configuración propuesta: un sistema ***tolerante a fallas***.

4. Conclusiones

La arquitectura propuesta mitiga el problema de caídas de nodos en las evaluaciones realizadas. Este era el inconveniente principal a resolver y si bien la solución propuesta es satisfactoria a nivel de equipos (caída de servidores o fallas en su comunicación), es vulnerable a otros tipos de caídas como fallas de comunicación o cortes de energía generales o parciales en las instalaciones del muelle.

Por lo tanto, alcanzar un alto nivel de robustez implica no sólo pensar la arquitectura a nivel de componentes software, sino también incluir otras vistas que afectan al sistema. La nueva arquitectura con el clúster Kafka es útil para la comunicación lógica de los sistemas, pero sólo con eso no es suficiente en un entorno real.

Luego de hacer varias pruebas en el entorno de desarrollo, se llegó a la conclusión de que también es posible ajustar la variable *min_insync_replicas* e incrementar su valor a dos, para garantizar así la doble escritura de los datos. Este cambio es beneficioso si se prioriza la integridad de los datos y se garantiza que las caídas de los nodos son a lo sumo de uno a la vez. Está modificación no supone un mayor problema y puede ser interesante probarla en otros casos de uso.

La sencilla arquitectura propuesta no aprovecha todas las posibilidades que ofrece Kafka. Es una configuración simple que responde a la problemática concreta planteada. Se puede realizar diversas mejoras aplicando otras herramientas como ksqlDB, Kafka Connect, Kafka Streams, API Kafka, Schema Registry, etc.

Como conclusión final, el trabajo realizado ofrece una solución factible para mejorar la tolerancia a fallas de uno de los servicios que brinda la APPM. Aunque por cuestiones operativas esta solución no será aplicada en el futuro cercano, la misma fue presentada exitosamente como futura alternativa en el área de sistemas. Por lo tanto, este trabajo no sólo cumplió su objetivo académico (aprobación de la cursada de la asignatura Sistemas Distribuidos) si no que también aportó al desarrollo profesional del autor principal en su ambiente laboral.

Referencias

1. Jiang, Y., Liu, Q., Qin, C., Su, J., Liu, Q.: Message-oriented middleware: A review. In: 5th International Conference on Big Data Computing and Communications, BIGCOM 2019, QingDao, China, August 9-11, 2019. pp. 88–97. IEEE (2019). https://doi.org/10.1109/BIGCOM.2019.00023
2. Narkhede, N., Shapira, G., Palino, T.: Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale. O'Reilly Media, Inc., 1st edn. (2017)
3. Raptis, T.P., Passarella, A.: On efficiently partitioning a topic in apache kafka. In: Tsihrintzis, G.A., Virvou, M., Hsiao, K., Nicopolitidis, P., Guo, Y. (eds.) International Conference on Computer, Information and Telecommunication Systems, CITS 2022, Piraeus, Greece, July 13-15, 2022. pp. 1–8. IEEE (2022). https://doi.org/10.1109/CITS55221.2022.9832981