Análisis tecnologías de brokers de mensajería

Apache Kafka

Características: Apache Kafka es una plataforma de transmisión de eventos diseñada para manejar flujos de datos de alta tasa y baja latencia. Se distingue por su capacidad para almacenar y procesar grandes volúmenes de datos en tiempo real, lo que lo hace ideal para aplicaciones de análisis y procesamiento de eventos. Algunas de sus principales características son: [1]

Mensajería Persistente: En el contexto del voluminoso big data, la pérdida de información no es una opción aceptable. Por ende, Apache Kafka se estructura con discos que aseguran un rendimiento constante, incluso cuando se manejan grandes volúmenes de mensajes, del orden de terabytes.

Alto Rendimiento: Considerando la magnitud del big data, Kafka está diseñado para operar eficientemente en hardware básico y tiene la capacidad de soportar millones de mensajes por segundo.

Distribuido: Kafka abraza de manera explícita la partición de mensajes en servidores y la distribución del consumo en un grupo de máquinas de consumo. Este enfoque se lleva a cabo sin comprometer la semántica de ordenación por partición.

Soporte para Múltiples Clientes: El sistema Apache Kafka facilita la integración de clientes provenientes de diversas plataformas, tales como Java, .NET, PHP, Ruby y Python.

Tiempo Real: Es esencial que los mensajes producidos por los hilos productores sean inmediatamente visibles para los hilos consumidores. Esta

[1]KMEME, Thein. Apache Kafka: Next Generation Distributed Messaging System. En: *Internatinoal Journal of Scientific Engineering and Technology Research*. 2014, vol 03, issue 47, p.9479. ISSN 2319-8885

característica resulta crítica para sistemas basados en eventos, como los sistemas de procesamiento de eventos complejos (CEP).

Modelo de Mensajería: Kafka emplea un modelo de mensajería basado en registros distribuidos, utiliza un protocolo binario sobre TCP para la transmisión de eventos y la retención de mensajes. Además de las estructuras de productor y consumidor definidas sobre todos los sistemas que hacen uso de brokers en la arquitectura general de estos, internamente cuenta con las siguientes estructuras lógicas: [2]

Topic: Se trata de una categoría donde se publican los mensajes enviados al broker. Los topics son particionados para permitir la paralelización y la distribución del procesamiento de datos. Cada partición es una secuencia ordenada y replicada de registros y se almacena en un broker. Los mensajes dentro de una partición se identifican por un número secuencial llamado offset.

Partitions: Las Particiones son subdivisiones de un topic. Cada partición se almacena en un broker específico y permite que los datos se distribuyan y procesen en paralelo. La partición también facilita la replicación y la tolerancia a fallos, ya que cada partición puede tener múltiples réplicas en diferentes brokers. Un único mensaje se escribe en una sola partición, y el orden de los mensajes se garantiza solo dentro de una partición.

ZooKeeper: ZooKeeper es utilizado por Kafka para la coordinación y el manejo de la configuración. ZooKeeper almacena metadatos del clúster, como la información sobre los brokers, los topics, las particiones y sus réplicas. Además, ZooKeeper se utiliza para la gestión de la elección del líder en el clúster, asegurando que haya una sincronización adecuada entre los nodos.

[2]MANISH, Kumar y CHANCHAL, Singh. Building Data Streaming Applications with Apache Kafka Birmingham: Packt Publishing, 2017. p. 26-28. ISBN 978-1-78728-398-5

Escalabilidad: Apache Kafka está diseñado para una escalabilidad masiva y es capaz de manejar flujos de datos extremadamente grandes: [3]

- **Clústeres**: Los clústeres de Kafka están compuestos de múltiples brokers que contienen decenas o incluso cientos de nodos en total, los cuales forman un sistema que garantiza la alta disponibilidad y un manejo eficiente de grandes volúmenes de datos.
- **Particiones**: Las particiones de Kafka juegan un papel fundamental en la escalabilidad del sistema al encargarse de distribuir los datos en particiones permitiendo que los mensajes se procesen en paralelo. La escalabilidad horizontal se logra aumentando el número de particiones y repartiendo estas particiones entre múltiples brokers.
- **Replicación**: Kafka cuenta con un proceso de replicación de las particiones entre múltiples nodos para asegurar la alta disponibilidad y la tolerancia a fallos.

Rendimiento: Cuando se evalúan los factores de rendimiento de brokers de mensajería se hacen uso de tres propiedades que identifican cómo es el rendimiento de estos en los sistemas. Estas características son latencia, que se refiere al tiempo que transcurre entre la publicación y el consumo del mensaje; el throughput, que se trata de la cantidad de mensajes procesados por unidad de tiempo; y la persistencia, que habla sobre la garantía de que los mensajes no se pierdan durante fallos. Kafka está optimizado para el rendimiento en el procesamiento de grandes volúmenes de datos de manera eficiente. Se pueden describir sus características según: [4]

---

[3]CONFLUENT. [What Is an Apache Kafka Cluster? (And Why You Should Care): CONFLUENT. [Consulta: 10 de junio 2024]. Disponible en: https://www.confluent.io/blog/what-is-an-apache-kafka-cluster/

[4]AMAZON WEB SERVICES. ¿Cuál es la diferencia entre Kafka y RabbitMQ? AWS. [Consulta: 10 de junio 2024]. Disponible en https://aws.amazon.com/es/compare/the-difference-between-rabbitmq-and-kafka/

Latencia: Kafka se distingue por ofrecer baja latencia en la entrega de mensajes, al implementar una configuración adecuada y bajo condiciones óptimas, Kafka puede proporcionar tiempos de respuesta inferiores a milisegundos. La arquitectura interna de Kafka, basada en el uso de logs secuenciales y un protocolo de comunicación altamente eficiente, minimiza la sobrecarga de procesamiento y reduce los tiempos de latencia. Además, las técnicas avanzadas de agrupación de mensajes y el uso de memoria para el almacenamiento temporal contribuyen a mantener esta baja latencia, incluso bajo cargas de trabajo intensivas.

Throughput: Kafka es capaz de manejar un throughput extremadamente alto, procesando millones de mensajes por segundo, lo que lo convierte en una solución ideal para aplicaciones de big data y análisis en tiempo real. La arquitectura distribuida de Kafka permite la escalabilidad horizontal, donde se pueden agregar más brokers para distribuir la carga de trabajo y aumentar la capacidad de procesamiento sin degradar el rendimiento. Además, la capacidad de Kafka para manejar particiones múltiples en cada topic facilita el procesamiento paralelo de datos, maximizando el throughput y permitiendo el manejo de picos de tráfico sin interrupciones.

Eficiencia de almacenamiento: Kafka almacena los mensajes de forma eficiente en logs distribuidos, lo que permite una alta durabilidad sin comprometer significativamente el rendimiento.

RabbitMQ

Características: RabbitMQ es un broker de mensajes ampliamente utilizado basado en el protocolo Advanced Message Queuing Protocol (AMQP). Es conocido por su fiabilidad, flexibilidad y facilidad de uso. RabbitMQ admite múltiples protocolos de mensajería, lo que lo convierte en una opción versátil para diversas

aplicaciones. Algunas de las características que convierten a RabbitMQ en una opción destacada son: [5]

Código abierto: RabbitMQ es actualmente propiedad de Pivotal Software Inc. y se distribuye bajo la Licencia Pública de Mozilla. Este proyecto de código abierto, redactado en Erlang, se beneficia de su libertad y flexibilidad, mientras aprovecha la solidez respaldada por Pivotal como producto.

Plataforma y proveedor neutral: En su función como intermediario de mensajes que implementa la especificación del Protocolo Avanzado de Cola de Mensajes (AMQP), la cual es neutral en cuanto a plataforma y proveedor, se encuentran disponibles clientes para prácticamente cualquier lenguaje de programación y en todas las principales plataformas informáticas.

Liviano: Este sistema requiere menos de 40 MB de RAM para ejecutar la aplicación principal de RabbitMQ junto con sus complementos, como la interfaz de gestión. Es importante señalar que la adición de mensajes a las colas puede incrementar su uso de memoria.

Bibliotecas de clientes para la mayoría de los lenguajes modernos: Con bibliotecas de clientes diseñadas para la mayoría de los lenguajes de programación modernos en diversas plataformas, se presenta como un intermediario convincente para la programación que proporciona un puente útil que permite a lenguajes como Java, Ruby, Python, PHP, JavaScript y C# compartir datos en diferentes sistemas operativos y entornos.

Flexibilidad en el control de compensaciones de mensajes: RabbitMQ brinda flexibilidad en el control de compensaciones de mensajería confiable en términos de rendimiento y eficacia. Algunas de las características que respaldan este hecho se encuentran en los mensajes que pueden designar si deben persistir en disco antes de la entrega y en configuraciones de clúster, donde las colas pueden ajustarse para ser altamente disponibles,

[5] ROY, Gabin. RabbitMQ in Deep. Nueva York: Manning Publications, 2017. ISBN 9781638353225

abarcando varios servidores para garantizar que los mensajes no se pierdan en caso de fallo del servidor.

Complementos para entornos de mayor latencia: Dada la diversidad de topologías y arquitecturas de red, se facilita la mensajería en entornos de baja latencia, así como complementos para entornos de mayor latencia, como Internet. Esto permite que RabbitMQ se agrupe en la misma red local y comparta mensajes federados en diversos centros de datos.

Complementos de terceros: Como núcleo para integraciones de aplicaciones, el aplicativo presenta un sistema de complementos flexible. Por ejemplo, existen complementos de terceros para almacenar mensajes directamente en bases de datos, utilizando RabbitMQ directamente para escrituras en bases de datos.

Capas de seguridad: Las conexiones de clientes pueden asegurarse mediante la aplicación exclusiva de comunicación SSL y la validación de certificados de clientes. El acceso de usuario puede gestionarse a nivel de host virtual, proporcionando aislamiento de mensajes y recursos a un nivel elevado. Además, el acceso a las capacidades de configuración, la lectura de colas y la escritura en intercambios se gestiona mediante coincidencia de patrones de expresiones regulares (regex). Por último, los complementos pueden emplearse para integrarse con sistemas de autenticación externos como LDAP.

Modelo de Mensajería: El modelo de mensajería de RabbitMQ se basa en AMQP, un protocolo de capa de aplicación diseñado para proporcionar mensajería confiable. AMQP soporta varios patrones de mensajería, incluyendo punto a punto, publicación/suscripción y colas de trabajo. RabbitMQ permite la definición de intercambios (exchanges), colas y enlaces (bindings) que determinan cómo se

enrutan los mensajes a través del sistema. Cada uno cumpliendo con funciones específicas como: [6]

Intercambio/Exchange: En el sistema de mensajería, un exchange actúa como un intermediario que recibe mensajes de los productores y determina cómo deben ser enrutados hacia las colas apropiadas. Los exchanges son configurables y pueden seguir diferentes reglas de enrutamiento, como enrutamiento directo, basado en temas o enrutamiento con comodines. Esta flexibilidad permite que los mensajes sean distribuidos eficientemente a los consumidores correctos, según las necesidades de la aplicación.

Enlace/bindings: Los bindings en sistemas de mensajería son configuraciones que determinan cómo los mensajes se enrutan desde los exchanges hacia las queues. Los bindings pueden utilizar diversas reglas de enrutamiento basadas en patrones o criterios específicos, como claves de enrutamiento. Esto permite una distribución eficiente y lógica de los mensajes, asegurando que cada mensaje llegue a la cola adecuada para su procesamiento posterior.

Cola/Queue: Una queue en el contexto de mensajería es un almacenamiento intermedio donde los mensajes se acumulan en espera de ser procesados por los consumidores. Las queues aseguran que los mensajes se manejen de manera ordenada y confiable, proporcionando un mecanismo de buffering que desacopla la producción de mensajes de su consumo. Esto permite que los sistemas manejen cargas variables y aseguren la entrega de mensajes incluso en condiciones de alta demanda.

Escalabilidad: RabbitMQ ofrece varias características para escalar horizontalmente y manejar cargas de trabajo crecientes. Entre sus principales mecanismos de escalabilidad se encuentran:

---

[6] ROY, Gabin. RabbitMQ in Deep. Nueva York: Manning Publications, 2017. ISBN 9781638353225

Clustering: RabbitMQ permite la creación de clústeres de múltiples nodos, el cual proporciona al sistema alta disponibilidad y balancea la carga en este, permitiendo que las colas y los intercambios se repliquen y gestionen entre varios nodos, los cuales realizan diferentes procesos en paralelo. Los nodos en un clúster de RabbitMQ comparten datos como configuraciones, colas e intercambios, lo que facilita la tolerancia a fallos y la escalabilidad del sistema. [7]

Adicionalmente, la biblioteca de RabbitMQ cuenta con opciones de personalización de procesos mediante el uso de plugins que permiten mejorar la escalabilidad de la aplicación. Algunos de estos son:

Shovel: Una herramienta que facilita la transferencia de mensajes entre brokers o clústeres RabbitMQ de manera persistente y fiable. [8]

Rendimiento: RabbitMQ es conocido por su rendimiento fiable en entornos de producción. Su capacidad de manejar altos volúmenes de mensajes depende de varios factores: [9]

Latencia: RabbitMQ presenta un sistema caracterizado por la baja latencia en la entrega de mensajes,ofreciendo una entrega de mensajes eficiente y fiable. Sin embargo, en escenarios de alta concurrencia, donde hay un gran número de mensajes siendo enviados y recibidos simultáneamente, RabbitMQ puede experimentar un aumento en la latencia en comparación

[7] RABBITMQ TM Documentation. Clustering Guide. Broadcom Inc. [Consulta: 10 de junio 2024]. Disponible en https://www.rabbitmq.com/docs/clustering

[8] RABBITMQ TM Documentation. Plugins. Broadcom Inc. [Consulta: 10 de junio 2024]. Disponible en https://www.rabbitmq.com/docs/plugins

[9] Amazon Web Services. ¿Cuál es la diferencia entre Kafka y RabbitMQ? AWS. [Consulta: 10 de junio 2024]. Disponible en https://aws.amazon.com/es/compare/the-difference-between-rabbitmq-and-kafka/

con sistemas como Apache Kafka. Esto se debe en parte a las diferencias en la arquitectura y los mecanismos de almacenamiento. Mientras que Kafka está optimizado para manejar grandes volúmenes de datos en flujo continuo mediante una estructura de registro en disco secuencial, RabbitMQ maneja mensajes en colas que pueden requerir más operaciones de entrada/salida en disco.

Throughput: RabbitMQ destaca al poder realizar un alto manejo de throughput, el cual puede ser significativamente mejorado mediante la configuración adecuada y el uso de múltiples nodos en un clúster. En un entorno de clúster, RabbitMQ distribuye la carga de trabajo entre varios nodos, lo que permite que cada nodo maneje una parte del tráfico total. Esto no solo mejora el throughput sino que también ofrece redundancia y alta disponibilidad

Persistencia: RabbitMQ permite la persistencia de mensajes, aunque esta característica mejora la robustez del sistema, puede tener un impacto significativo en el rendimiento, esto debido a que el proceso de escribir mensajes en disco es inherentemente más lento que mantenerlos en memoria, lo que puede aumentar la latencia y reducir el throughput. No obstante, RabbitMQ ofrece varias opciones para gestionar la persistencia y minimizar su impacto en el rendimiento del sistema.

ActiveMQ

Características: ActiveMQ es un broker de mensajería desarrollado por Apache que soporta el protocolo Java Message Service (JMS) y otros protocolos, lo que le permite integrarse bien en entornos Java y sistemas heterogéneos. Entre las principales características que este broker aporta están: [10]

[10] BRUCE, Snyder; DEJAN Bosanac y ROB Davies. ActiveMQ in Action. Nueva York: Manning Publications, 2011. ISBN 1638357021

Cumplimiento de JMS: La implementación de ActiveMQ cumple con la especificación JMS 1.1. Esta normativa no solo certifica la adhesión a estándares, sino que también ofrece una serie de beneficios y garantías significativas. Entre ellas, se destaca la posibilidad de entregar mensajes de manera sincrónica o asincrónica, la garantía de entrega de mensajes en una única ocasión, así como la preservación de la durabilidad de los mensajes para los suscriptores, entre otros aspectos de relevancia.

Conectividad: La plataforma dispone de una amplia variedad de opciones de conectividad, abarcando la compatibilidad con diversos protocolos como HTTP/S, multidifusión, SSL, Stomp, TCP, UDP, XMPP, entre otros. La capacidad de ser compatible con una extensa gama de protocolos otorga una mayor flexibilidad a la infraestructura. Dado que muchos sistemas preexistentes se encuentran vinculados a un protocolo específico sin posibilidad de modificación, la presencia de una plataforma de mensajería que admite múltiples protocolos resulta clave para reducir las barreras de adopción.

Persistencia y seguridad conectables: ActiveMQ ofrece diversas opciones de persistencia y permite una personalización total en cuanto a la seguridad. En este sentido, la seguridad en ActiveMQ puede ajustarse completamente a los requisitos particulares de autenticación y autorización.

API de cliente: Se proporciona una API de cliente que abarca diversos lenguajes, como C/C++, .NET, Perl, PHP, Python, Ruby, entre otros. Esta diversidad de soporte lingüístico amplía significativamente las posibilidades de implementación de ActiveMQ más allá del ámbito Java. Las diversas API de cliente posibilitan el acceso a todas las funciones y ventajas proporcionadas por ActiveMQ en distintos entornos de programación.

Agrupación de Brokers: Numerosos Brokers de ActiveMQ pueden colaborar en forma de red para mejorar la escalabilidad, conformando lo que se conoce como red de Brokers. Esta estructura es compatible con diversas topologías, ofreciendo así flexibilidad en su implementación.

- **Funciones avanzadas del Broker y opciones del cliente:** Se dispone de numerosas funciones avanzadas tanto para el Broker como para los clientes que se conectan a él, proporcionando un entorno sofisticado y completo para la gestión de mensajes.

Modelo de Mensajería: El modelo de mensajería de ActiveMQ se basa en JMS, proporcionando una API estándar para el envío de mensajes entre dos o más clientes. ActiveMQ admite varios patrones de mensajería, incluyendo punto a punto y publicación/suscripción, permitiendo a su vez una mayor flexibilidad en el envío de mensajes. Debido a esto, el broker cuenta con dos entidades lógicas, las cuales son: [11]

- **Colas:** Se definen como canalizaciones FIFO (first-in, first-out) destinadas a la gestión de mensajes producidos y consumidos por intermediarios y clientes. Los productores generan mensajes y los direccionan hacia estas colas. Posteriormente, las aplicaciones de consumo exploran y adquieren dichos mensajes, de manera secuencial y uno a la vez.
- **Temas:** Se constituyen como canales de transmisión de mensajes basados en suscripción. Cuando una aplicación productora emite un mensaje, varios destinatarios suscritos a ese tema reciben la transmisión correspondiente del mensaje.

De igual manera, ActiveMQ también soporta otros protocolos como AMQP, MQTT y STOMP, ofreciendo flexibilidad para diferentes escenarios de uso.

[11] Active MQ Documentation. Messaging Concepts. Apache Software Foundation. [Consulta: 10 de junio 2024]. Disponible en https://activemq.apache.org/components/artemis/documentation/1.1.0/messaging-concepts.html

Escalabilidad: ActiveMQ ofrece diversas opciones para escalar en entornos de producción: [12]

- **Clustering:** ActiveMQ permite el balanceo de carga eficiente y confiable de mensajes en una cola entre múltiples consumidores para distribuir la carga de trabajo mediante clusters de brokers.
- **Red de Brokers:** Las redes permiten la configuración de topologías de cliente/servidor o hub/spoke con múltiples brokers y clientes. Esto soluciona el problema de acumulación de mensajes en brokers sin consumidores, permitiendo mover mensajes de brokers con productores a brokers con consumidores, soportando así colas y temas distribuidos a través de una red de brokers.
- **Master/Slave:** Este modelo aborda el problema de los brokers independientes donde los mensajes son propiedad de un único broker físico en un momento dado. Si ese broker falla, los mensajes no pueden ser entregados hasta que el broker se reinicie. El modelo Maestro-Esclavo replica los mensajes a un broker esclavo, asegurando la conmutación por error inmediata al esclavo sin pérdida de mensajes en caso de un fallo del maestro
- **Almacenamiento de mensajes replicados**: Como alternativa al modelo Maestro-Esclavo, los almacenes de mensajes replicados permiten compartir los archivos de un broker utilizando un SAN o una unidad de red compartida. Si un broker falla, otro broker puede tomar el control de inmediato.

Rendimiento: El rendimiento de ActiveMQ varía según la configuración y la versión del broker que se usa, presentando una versión llamada Classic con las

[12] Active MQ Documentation. Clustering. Apache Software Foundation. [Consulta: 10 de junio 2024]. Disponible en https://activemq.apache.org/components/classic/documentation/clustering

configuraciones básicas del sistema y una versión bajo el nombre de Artemis que promete brindar un mejor rendimiento: [13]

- **Latencia:** Con cargas de trabajo medianas, ActiveMQ mantiene una baja latencia debido a su diseño interno que optimiza el flujo de mensajes a través de la red y los recursos del sistema.
- **Throughput:** Con cargas de trabajo medianas, ActiveMQ puede alcanzar niveles altos de throughput gracias a su capacidad para manejar múltiples productores y consumidores en paralelo. La implementación de características como la agrupación de mensajes, el balance de carga dinámico y la utilización de colas y topics distribuidos permite que el sistema procese miles de mensajes por segundo de manera eficiente en cargas de trabajo medianas.
- **Persistencia:** La persistencia que presentan los sistemas de ActiveMQ puede variar según la versión usada, contando con una base de datos compatible con JDBC para la conservación de datos cuando se usa la versión Classic. La persistencia en disco es más lenta comparada con la mensajería en memoria, pero garantiza que los mensajes no se pierdan en caso de fallos del sistema.

Mosquitto MQTT

Características: Mosquitto es un broker de mensajería ligero que implementa el protocolo Message Queuing Telemetry Transport (MQTT). Está diseñado para conexiones persistentes y es ideal para dispositivos IoT debido a su bajo consumo

[13] Quix Analytincs. ActiveMQ vs. Kafka: A comparison of differences and use cases. Quix. [Consulta: 10 de junio 2024]. Disponible en https://quix.io/blog/activemq-vs-kafka-comparison

de ancho de banda y simplicidad. Entre sus principales características se encuentran: [14]

Ligero y Eficiente: La implementación de MQTT demanda recursos mínimos, permitiendo su aplicación incluso en microcontroladores de dimensiones reducidas. Los encabezados de los mensajes MQTT también se caracterizan por su modesto tamaño, contribuyendo así a optimizar el ancho de banda de la red.

Escalable: La implementación de MQTT se distingue por requerir una cantidad mínima de código, lo cual resulta en un consumo ínfimo de energía durante sus operaciones. Este protocolo también incorpora funciones integradas para respaldar la comunicación con un extenso número de dispositivos.

Fiable: MQTT incorpora funciones integradas que reducen el tiempo necesario para que el dispositivo se vuelva a conectar con la nube.

Seguro: MQTT facilita a los desarrolladores la tarea de cifrar mensajes y autenticar dispositivos y usuarios mediante protocolos de autenticación modernos, como OAuth, TLS1.3 y certificados administrados por el cliente, entre otros.

Admitido: Diversos lenguajes de programación, como Python, brindan un sólido respaldo para la implementación del protocolo MQTT. Por ende, los desarrolladores pueden incorporar de manera sencilla una codificación mínima en cualquier tipo de aplicación.

Modelo de Mensajería y Protocolos: El protocolo MQTT, utilizado por Mosquitto, se centra en la publicación/suscripción, facilitando la comunicación eficiente entre dispositivos con recursos limitados. [15]

---

[14] Amazon Web Services. ¿Qué es MQTT? AWS. [Consulta: 10 de junio 2024]. Disponible en https://aws.amazon.com/es/what-is/mqtt/

[15] eG Innovations Inc. What is Mosquitto MQTT? eG Innovatinos [Consulta: 10 de junio 2024]. Disponible en https://www.eginnovations.com/documentation/Mosquitto-MQTT/What-is-Mosquitto-MQTT.htm

Tema: Hace alusión a las palabras clave empleadas por el agente MQTT para filtrar mensajes destinados a los clientes de MQTT. Estos temas se estructuran de manera jerárquica, siguiendo una lógica similar a la de un directorio de archivos o carpetas.

Publicación: Los clientes MQTT llevan a cabo la publicación de mensajes que incorporan tanto el tema como los datos en formato de bytes. El formato de los datos, como texto, binario, XML o JSON, queda a discreción del cliente.

Suscripción: Los clientes MQTT envían un mensaje SUBSCRIBE (SUSCRIBIRSE) al agente MQTT con el propósito de recibir mensajes relacionados con temas de su interés. Este mensaje contiene un identificador único y una lista de suscripciones.

MQTT es especialmente adecuado para aplicaciones de IoT debido a su eficiencia y capacidad para funcionar en entornos con ancho de banda limitado y alta latencia. **Escalabilidad**: **--**Mosquitto MQTT es ideal para aplicaciones IoT, y su escalabilidad se maneja de la siguiente manera: [16]

Clustering: Mosquitto, por su naturaleza de ser usado principalmente para desarrollos IoT, no cuenta con una robustez tal como las presentadas por otros brokers. Sin embargo, destaca en su flexibilidad de uso de cluster donde cuenta con dos modos que pueden adaptarse según la situación deseada:

Full Sync: Este modo de clúster consiste en varios nodos que operan en un modelo activo-pasivo. En este proceso, cuando el nodo activo falla, los nodos pasivos asumen su función de manera rápida y eficiente.

Dynamic Security Sync: En contraste, este modo utiliza nodos activo-activo, lo que permite que todos los nodos del clúster estén disponibles simultáneamente para atender conexiones de clientes.

[16] CEDALO. MQTT High Availability. CEDALO [Consulta: 10 de junio 2024]. Disponible en https://cedalo.com/mqtt-broker-pro-mosquitto/high-availability/

Rendimiento: Mosquitto está optimizado para entornos IoT, donde el rendimiento debe ser eficiente y ligero: [17]

Latencia: El protocolo del que hace uso Mosquitto ofrece baja latencia, ideal para dispositivos IoT que requieren tiempos de respuesta rápidos, con una configuración adecuada, comparado con otras tecnologías, este tiene una latencia más alta, lo cual indica un mayor tiempo de respuesta en la comunicación desempeñado de manera eficiente cargas ligeras.

Throughput: El uso de protocolo MQTT muestra un rendimiento aceptable para aplicaciones con cargas pequeñas y en escenarios de red simples. Sin embargo, su rendimiento decae significativamente con cargas más grandes y en entornos de red más complejos o distribuidos este factor no debe ser una desventaja puesto que es más que suficiente para la mayoría de las aplicaciones IoT, donde el volumen de datos manejados es significativamente menor.

[17] Wen-Yew Liang, Yuyuan Yuan, Hsiang-Jui Lin. *A Performance Study on the Throughput and Latency of Zenoh, MQTT, Kafka, and DDS*. 2023, p.9-15. arXiv:2303.09419