Grupo 4

Diseño e Implementación de un Middleware que Implemente un Servicio de Mensajería Asincrónica entre Aplicaciones

Integrantes

- Manuela Castaño - mcastanof1@eafit.edu.co
- Miguel Vásquez - mvasquezb@eafit.edu.co
- Manuel Villegas - mvillegas6@eafit.edu.co

Repositorio

- https://github.com/VillegasMich/middleware-message-mom

Credenciales AWS

- **Usuario:** *mvillegas6@eafit.edu.co*
- **Contraseña:** *Gemelo-1*

Requisitos

Funcionales

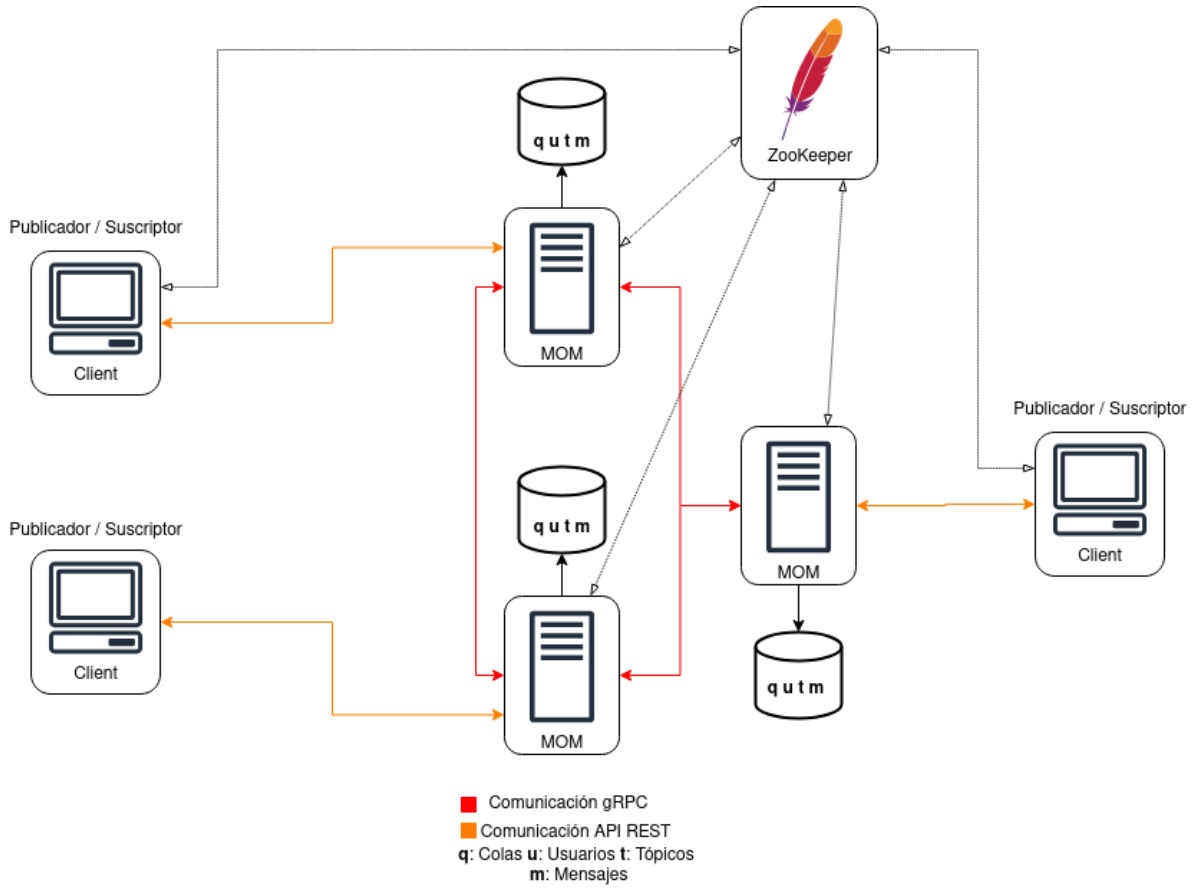
1. **Gestión de la conexión de clientes:** El sistema debe permitir a múltiples clientes autenticarse, conectarse y desconectarse del MOM, ya sea de forma persistente (con estado) o sin mantener una conexión constante (sin estado).
2. **Autenticación y Autorización:** Los usuarios deben autenticarse antes de interactuar con el sistema.
3. **Gestión del Ciclo de Vida de Tópicos:** El sistema debe permitir la creación de tópicos, la eliminación de tópicos únicamente por parte del usuario que los creó, y listar todos los tópicos disponibles.
4. **Gestión del Ciclo de Vida de Colas:** El sistema debe permitir la creación de colas, la eliminación de colas únicamente por parte del usuario que las creó, y listar todas las colas disponibles.
5. **Envío y Recepción de Mensajes:** El sistema debe permitir enviar mensajes a una cola o tópico en específico, así como también recibir mensajes desde una cola o tópico.
6. **Modelo de Suscripción:** El sistema debe ofrecer un modelo de suscripción a las colas y tópicos tanto por tipo push como tipo pull.
7. **API REST y gRPC:** El sistema debe exponer servicios a los clientes a través de API REST, mientras que la comunicación entre servidores MOM es por medio de gRPC.
8. **Persistencia de Datos:** El sistema debe implementar un mecanismo de persistencia para asegurar que los mensajes y configuraciones no se pierdan ante una falla.

No Funcionales

1. **Seguridad**: Las credenciales de autenticación deben ser transmitidas mediante mecanismos de cifrado.
2. **Tolerancia a fallos**: El sistema debe ser capaz de recuperarse de fallos de algún nodo mediante replicación de datos y mecanismos de respaldo.
3. **Escalabilidad**: La arquitectura del sistema debe permitir la incorporación de nuevos nodos al cluster sin afectar la funcionalidad existente.
4. **Particionamiento y Replicación:** El sistema debe distribuir las colas y tópicos, además de otra información, entre diferentes nodos del cluster, además de replicar esta información.
5. **Transparencia**: El sistema debe ser transparente, es decir, los clientes no deben saber los detalles internos del cluster, como la ubicación de los tópicos o colas. Además, el uso del sistema debe ser homogéneo, independientemente del nodo al que se conecte el cliente.
6. **Desempeño y Eficiencia**: El sistema debe responder en tiempos aceptables ante operaciones de envío y recepción de mensajes.
7. **Mantenibilidad**: La arquitectura debe permitir la inclusión de nuevas funcionalidades sin necesidad de una reestructuración completa.

8. **Multiusuario**: El sistema debe soportar la conexión concurrente de múltiples usuarios, garantizando la correcta identificación y separación de sus recursos y mensajes.
9. **Modelo de Comunicación Distribuida**: El sistema debe soportar interacciones sincrónicas y asincrónicas.

Diseño de Arquitectura



Documentación Cliente

El cliente es el encargado del uso de los servidores **MOM** para el envío y recepción de mensajes tanto por colas como por tópicos, este cliente mantiene una comunicación API REST con los servidores donde se envían y/o reciben las peticiones pertinentes. El cliente mantiene un mecanismo de **pull** hacia el servidor donde le pide constantemente los mensajes al servidor.

Nuestro programa cliente está escrito en Python, utilizando distintos paquetes como:

- **annotated-types**: Proporciona tipos anotados adicionales para usar con validadores y frameworks de tipado como `pydantic`.

- **anyio**: Librería de concurrencia que unifica `asyncio` y `trio`, utilizada por frameworks asincrónicos como FastAPI y Starlette.

- **bcrypt**: Implementación de hashing de contraseñas utilizando el algoritmo bcrypt, ampliamente usado para autenticar usuarios.

- **certifi**: Incluye certificados raíz actualizados para verificar la seguridad de las conexiones HTTPS.

- **charset-normalizer**: Intenta detectar la codificación de texto en datos binarios; alternativa a `chardet`.

- **ecdsa**: Implementa firmas digitales con curvas elípticas (ECDSA), usadas en criptografía y autenticación.

- **fastapi**: Framework moderno para construir APIs web rápidas y con tipado fuerte usando Python y ASGI.

- **idna**: Soporte para nombres de dominio internacionalizados (IDN), necesarios en URLs no ASCII.

- **kazoo**: Cliente de alto nivel para Apache ZooKeeper, usado para coordinación y servicios distribuidos.

- **markdown-it-py**: Parser de Markdown altamente compatible y extensible, basado en JavaScript `markdown-it`.

- **mdurl**: Parser/encoder de URLs compatible con Markdown y `markdown-it`.

- **passlib**: Biblioteca completa para hash de contraseñas, compatible con múltiples algoritmos.

- **pyasn1**: Implementación de ASN.1 (Abstract Syntax Notation One), usada para decodificar certificados y estructuras de red.

- **pydantic**: Validación y serialización de datos con tipado en Python; base de FastAPI.

- **pydantic_core**: Núcleo escrito en Rust para acelerar la validación de datos en `pydantic`.

- **Pygments**: Resaltador de sintaxis para mostrar código fuente con colores (usado en `rich` o documentación).

- **PyJWT**: Librería para codificar y decodificar JSON Web Tokens (JWT), útil en autenticación.

- **python-dotenv**: Carga variables de entorno desde un archivo `.env` al entorno de ejecución.
- **python-jose**: Implementación de JOSE (JWT, JWE, JWS, JWK) en Python, para seguridad y autenticación.
- **requests**: Librería HTTP sencilla y poderosa para hacer peticiones a APIs.
- **rich**: Librería para imprimir texto en la terminal con formato, colores, tablas, barras de progreso, etc.
- **rsa**: Implementación del algoritmo RSA para cifrado y firmas digitales.
- **ruff**: Linter extremadamente rápido para Python, ayuda a mantener estilo y calidad de código.
- **six**: Compatibilidad entre Python 2 y 3, aunque actualmente se usa menos con la desaparición de Python 2.
- **sniffio**: Detecta automáticamente el entorno asincrónico en uso (`asyncio`, `trio`), requerido por `anyio`.
- **starlette**: Toolkit ASGI ligero sobre el cual se construye FastAPI, proporciona routing, middleware y más.
- **typing_extensions**: Proporciona tipos de anotación que no están aún en el módulo estándar `typing`.
- **urllib3**: Cliente HTTP robusto y con muchas características, base de `requests`.

Estos paquetes nos entregan todas las funcionalidades necesarias para que el cliente pueda cumplir su función de manera pertinente. Dentro del mismo cliente podemos encontrar la siguiente estructura de los archivos más importantes.

```
core/
├── Listener.py
├── Queue.py
├── Topic.py
├── User.py
Util.py
config.py
main.py
requirements.txt
zookeeper.py
```

El cliente va a tener la posibilidad de realizar distintas tareas, entre las más importantes tenemos el registro y logueo de nuevos usuarios, creación, envío y recepción de mensajes tanto de tópicos como de colas, esto a través de una interfaz por consola interactiva para el usuario.

Toda interacción que el cliente tiene con los distintos servidores se hace de manera transparente gracias a la implementación de un **ZooKeeper** y la correcta aplicación de los principios de los middleware.

Finalmente, el cliente presenta las siguientes funcionalidades:

Registro de usuario

- Permite a un nuevo usuario registrarse en el sistema.

Inicio de sesión

- Permite a un usuario autenticarse y obtener un token de acceso.

Listar todas las colas

- Muestra una lista de todas las colas disponibles en el servidor.

Crear una cola

- Permite crear una nueva cola de mensajes.

Eliminar una cola

- Elimina una cola específica del sistema.

Enviar mensaje a una cola

- Envia un mensaje a una cola seleccionada.

Recibir mensaje de una cola

- Recupera un mensaje de una cola, si hay mensajes disponibles.

Suscribirse a una cola

- Permite al usuario suscribirse a una cola para recibir mensajes automáticamente.

Cancelar suscripción a una cola

- Detiene la recepción automática de mensajes desde una cola suscrita.

Listar todos los tópicos

- Muestra una lista de todos los tópicos disponibles en el sistema.

Crear un tópico

- Permite crear un nuevo tópico de publicación.

Eliminar un tópico

- Elimina un tópico específico del sistema.

Ver mensajes recolectados de un tópico

- Muestra los mensajes almacenados que fueron publicados en un tópico.

Enviar mensaje a un tópico

- Publica un mensaje en un tópico específico.

Suscribirse a un tópico

- Permite al usuario suscribirse a un tópico para recibir mensajes publicados.

Salir del cliente

- Finaliza la ejecución del cliente y cierra la conexión con el middleware.

Documentación Servidor

El servidor MOM (Message Oriented Middleware) es el encargado de la recepción de las peticiones para el registro y autenticación de usuarios, creación de colas y tópicos, envío y recepción de mensajes, entre otras muchas funcionalidades. Como base es un servidor web construido en FasApi de tal manera que los clientes se comuniquen a través de peticiones HTTP, mantiene una persistencia de datos en una base de datos MySQL y el uso de comunicación gRPC para comunicarse con los otros servidores existentes.

El programa utiliza distintos paquetes como:

- **FastAPI** (`fastapi==0.115.11`)
  Framework moderno, rápido y tipado para construir APIs web asincrónicas.

- **Starlette** (`starlette==0.46.1`)
  Toolkit ASGI ligero sobre el que se construye FastAPI. Proporciona el motor de routing, middleware, etc.

- **Uvicorn** (`uvicorn==0.34.0`)
  Servidor ASGI rápido y ligero ideal para ejecutar aplicaciones FastAPI en producción.

- **Typer** (`typer==0.15.2`)
  Framework para construir CLIs modernas basado en anotaciones de tipo, usado en

herramientas como `fastapi-cli`.

- **SQLAlchemy** (`SQLAlchemy==2.0.39`)
  ORM robusto y flexible para definir y consultar bases de datos relacionales en Python.

- **Alembic** (`alembic==1.15.1`)
  Herramienta de migraciones para bases de datos que funciona junto con SQLAlchemy.

- **Databases** (`databases==0.9.0`)
  Interfaz asincrónica para trabajar con SQLAlchemy y otras bases de datos de manera no bloqueante.

- **PyMySQL** (`PyMySQL==1.1.1`)
  Cliente MySQL escrito en Python puro.

- **aiomysql** (`aiomysql==0.2.0`)
  Cliente asincrónico para MySQL compatible con asyncio.

- **bcrypt** (`bcrypt==4.3.0`)
  Algoritmo seguro de hashing para contraseñas.

- **passlib** (`passlib==1.7.4`)
  Biblioteca de hashing de contraseñas con soporte para múltiples algoritmos.

- **PyJWT** (`PyJWT==2.10.1`)
  Soporte para la codificación y decodificación de JSON Web Tokens.

- **python-jose** (`python-jose==3.4.0`)
  Implementación completa de JOSE (JWS, JWE, JWT, JWK).

- **email-validator** (`email_validator==2.2.0`)
  Valida y normaliza direcciones de correo electrónico de forma segura.
- **httpx** (`httpx==0.28.1`)
  Cliente HTTP asincrónico moderno compatible con HTTP/1.1 y HTTP/2.

- **requests** (`no explícito, pero útil si se usa`)
  Alternativa sincrónica a `httpx` para llamadas HTTP.

- **websockets** (`websockets==15.0.1`)
  Implementación del protocolo WebSocket para conexiones en tiempo real.

- **kazoo** (`kazoo==2.10.0`)
  Cliente de alto nivel para Apache ZooKeeper, útil para coordinación en sistemas

distribuidos.

- **python-dotenv** (`python-dotenv==1.0.1`)
  Carga variables de entorno desde archivos `.env`.

- **rich** (`rich==13.9.4`)
  Salida colorida y estilizada en la terminal: tablas, logs, errores, barras de progreso, etc.

- **rich-toolkit** (`rich-toolkit==0.13.2`)
  Utilidades extra que extienden `rich`, ideales para desarrollo de interfaces en terminal.

- **fastapi-cli** (`fastapi-cli==0.0.7`)
  Herramienta para ejecutar y gestionar proyectos FastAPI desde la línea de comandos.

Estos paquetes nos entregan todas las funcionalidades necesarias para que el servidor pueda cumplir su función de manera pertinente. Dentro del mismo servidor podemos encontrar la siguiente estructura de los archivos más importantes.

```
alembic/
app/
├── core/
│   ├── Listener.py
│   ├── Queue.py
│   ├── Topic.py
│   └── User.py
├── grpc/
├── models/
├── protobufs/
├── repository/
├── routes/
├── RoundRobinManager.py
migrations/
alembic.ini
docker-compose.yml
main.py
requirements.txt
zookeeper.py
```

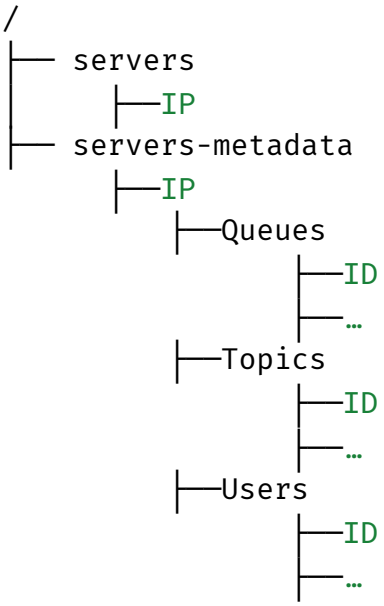
El servidor es quien aplica las distintas tareas que el cliente solicita a través de la consola, realiza el correcto manejo de los mensajes, colas y tópicos entre el y los distintos otros servidores existentes, esto con la gran ayuda del ZooKeeper del cual se hablara más adelante.

El sistema de servidores implementa varias características clave que garantizan un rendimiento eficiente, disponibilidad y robustez. En primer lugar, **la escalabilidad** permite agregar nuevos servidores de forma transparente, sin afectar la operación del sistema. Además, se implementa **particionamiento de datos**, lo que distribuye la información entre múltiples servidores para evitar cuellos de botella y mejorar la eficiencia. También se cuenta con **replicación**, asegurando que los datos estén disponibles en al menos dos servidores activos, lo que reduce el riesgo de pérdida de información. Por último, el sistema está diseñado con **tolerancia a fallos**, de modo que si un servidor se cae, el cliente no percibe interrupciones. En ese caso, el sistema replica y redistribuye automáticamente los datos, y cuenta con mecanismos para restaurar el estado del servidor al momento de la falla.

Documentación del ZooKeeper

Como se dijo anteriormente se implementó el uso de un ZooKeeper (Apache) para el manejo del clúster de servidores, este software nos entrega muchas funcionalidades que ayudan para el correcto funcionamiento de los sistemas y de todas las características de los servidores mencionadas anteriormente. Entre sus principales ventajas se encuentran: la **coordinación distribuida**, que permite mantener una visión consistente del estado del sistema entre todos los nodos; la **detección de fallos**, que permite identificar rápidamente servidores inactivos y redirigir la carga de forma automática; y el **descubrimiento de servicios**, gracias al cual los clientes pueden conectarse siempre al servidor más adecuado sin intervención manual. Además, ZooKeeper facilita la **gestión de la configuración centralizada**, la **sincronización de procesos distribuidos** y la **elección de líderes**, aspectos fundamentales para garantizar alta disponibilidad, tolerancia a fallos y balanceo de carga en entornos distribuidos con múltiples servidores.

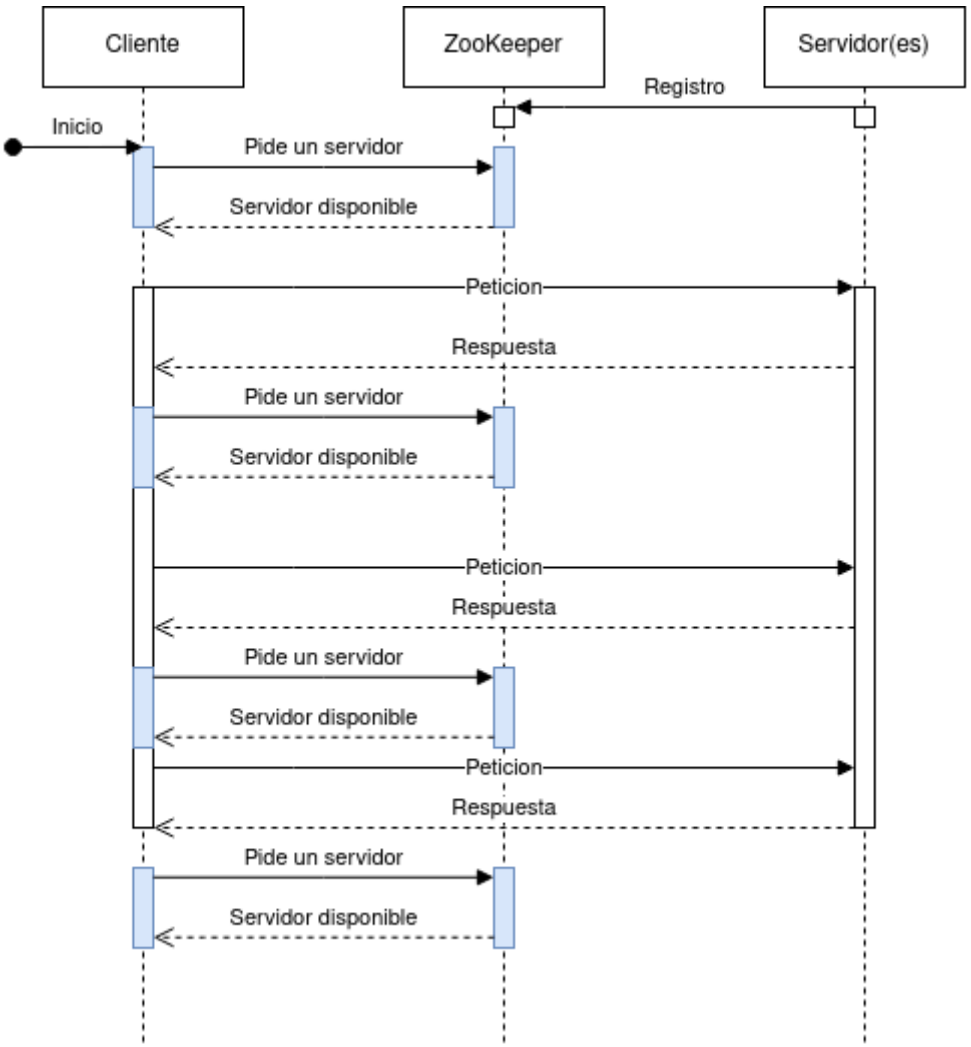
Nuestro ZooKeeper tiene la siguiente estructura:



Encontramos la ruta **servers** donde se encuentran las direcciones IP de los servidores disponibles actualmente, son nodos efímeros los cuales desaparecen al momento que el server se desconecta, y es de esta lista de servers que se eligen para interactuar con el

cliente. También encontramos la ruta de **servers-metadata** donde también se encuentran las IP de los, pero estos no desaparecen si el servidor llega a caer de forma inesperada, cada una de estas direcciones IP tiene a su vez por dentro las colas, los tópicos y los usuarios locales de cada servidor (los ID de cada elemento almacenado).

Conociendo la estructura entonces podemos explicar el funcionamiento del sistema con el ZooKeeper:

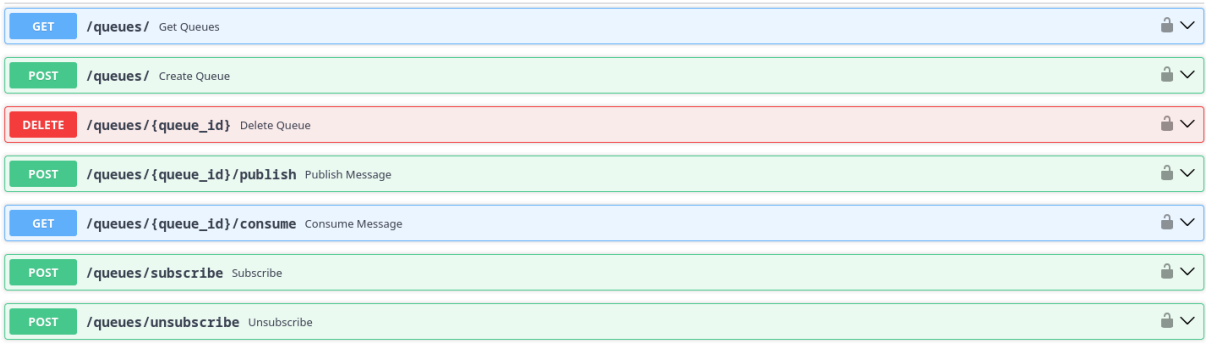


Después de cada petición, el ZooKeeper le entrega un nuevo server disponible a cliente a manera de round robin.

Documentación API

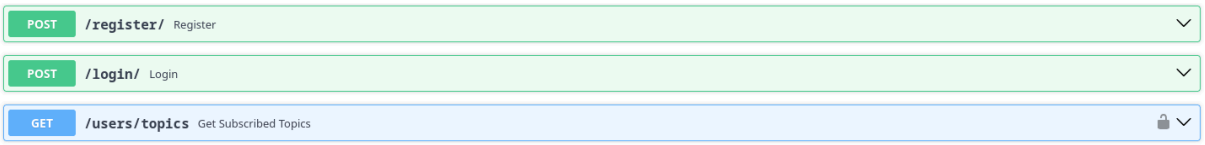
Con el servidor iniciado la documentación se puede encontrar en la ruta http://{IP}:8000/docs

Colas



Tópicos

Usuarios



Justificación de las tecnologías usadas

Para la implementación del middleware MOM se seleccionaron tecnologías que permiten construir un sistema distribuido robusto, eficiente y escalable. **FastAPI** fue elegida como framework principal del servidor por su velocidad, soporte asincrónico nativo y facilidad para definir APIs REST, facilitando la interacción con los clientes. Para la comunicación entre servidores, se optó por **gRPC** debido a su eficiencia, bajo consumo de ancho de banda y soporte para transmisión binaria mediante protocolos Protobuf, lo cual es ideal para ambientes distribuidos de alto rendimiento. La persistencia de usuarios, colas, tópicos y mensajes se gestiona mediante **MySQL**, una base de datos relacional madura, confiable y ampliamente adoptada, lo cual garantiza integridad de datos y soporte para consultas complejas. Finalmente, se utilizó **Apache ZooKeeper** como mecanismo de coordinación y

gestión del clúster de servidores, permitiendo detección de fallos, balanceo de carga, descubrimiento de servicios y almacenamiento distribuido de metadatos, funcionalidades esenciales para garantizar la disponibilidad y consistencia del sistema MOM.

Funcionamiento del sistema

El objetivo del sistema es permitir la **comunicación asincrónica de mensajes** entre clientes, los cuales pueden actuar como **publicadores** y/o **suscriptores**, a través de un clúster de servidores MOM diseñados para ofrecer dichas funcionalidades. Como se mencionó anteriormente, la meta no es solo facilitar el intercambio de mensajes, sino hacerlo aplicando principios clave de los sistemas distribuidos, tales como la **replicación**, el **particionamiento** y la **tolerancia a fallos**. Por esta razón, resulta fundamental detallar las estrategias empleadas en la implementación de cada uno de estos mecanismos dentro del sistema.

Replicación

Para la replicación de datos, cada servidor implementa una lógica que replica la petición hacia otro servidor aleatorio actualmente conectado. Esta selección se realiza utilizando la información de nodos disponibles en **ZooKeeper**. El servidor que recibe la petición principal se encarga de notificar al servidor replicador, mediante comunicación **gRPC**, para que este ejecute la misma operación (como la creación o eliminación de colas y tópicos). Todos los elementos creados (colas y tópicos) se registran en ZooKeeper con el fin de que los demás servidores puedan conocer su existencia y ubicación, permitiendo así una coordinación eficiente. La replicación solo se lleva a cabo si hay al menos **dos servidores activos**; en caso contrario, la operación se realiza únicamente en el servidor que recibe la petición.

Particionamiento

Para el particionamiento de datos se implementó un mecanismo basado en **Round Robin**, coordinado a través de **ZooKeeper**. Al iniciar la aplicación, el cliente se conecta automáticamente a uno de los servidores disponibles de forma aleatoria. A partir de ese momento, cada acción ejecutada por el cliente se distribuye de manera cíclica entre los servidores del clúster, siguiendo el orden determinado por el Round Robin. Esta estrategia permite balancear la carga de trabajo y lograr una **distribución equitativa de los datos** entre los servidores, ya que cada operación (como la creación de colas, tópicos o envío de mensajes) se atiende por un servidor diferente, lo que naturalmente produce un **particionamiento horizontal** del almacenamiento y procesamiento dentro del sistema.

Tolerancia a fallos

La **tolerancia a fallos** es un aspecto fundamental para garantizar la disponibilidad y correcto funcionamiento del servicio. Gracias a la coordinación proporcionada por **ZooKeeper**, el sistema mantiene un monitoreo constante del estado de los servidores del clúster. Esto permite que, en caso de que un servidor falle o se desconecte inesperadamente, los clientes sean redirigidos automáticamente a otro servidor disponible,

sin necesidad de intervención manual. La estrategia de selección dinámica del servidor, combinada con el modelo de Round Robin, asegura que el cliente siempre se comunique con un nodo funcional, sin percibir interrupciones en el servicio. De esta manera, se logra una experiencia transparente para el usuario, incluso ante la presencia de fallos en alguno de los componentes del sistema.