Descripción del Sistema FaaS

Arquitectura de microservicios

La arquitectura del sistema FaaS se basa en una estructura modular donde cada componente tiene una responsabilidad específica:

Proxy inverso (APISIX)

Actúa como punto de entrada para todas las solicitudes. Maneja la autenticación utilizando autenticación básica de usuario contraseña, mediante el manejo de consumidores de APISIX. Realiza el balanceo de carga para redirigir las peticiones a las instancias disponibles utilizando el algoritmo roundrobin. Está configurado para permitir un mayor tiempo de espera a las peticiones de activación para que no ocurra un error 408 Request Timeout mientras se espera a la ejecución de la función, dicho tiempo está configurado a 11 minutos, dando más tiempo que el establecido por defecto (60 segundos), sin dejar de establecer un límite fijo para el usuario.

Servidor API

El servidor HTTPS se desarrolló usando Node.js como entorno de ejecución, Express.js como entorno de trabajo y Typescript como lenguaje de programación. Al estar implementado en TypeScript, se beneficia de un ecosistema robusto y de la tipificación estática, lo cual mejora la mantenibilidad del código. Express se utilizó para acelerar el proceso de desarrollo utilizando las funcionalidades de enrutamiento, controladores y otras herramientas para la construcción de APIs que esta librería proporciona. La arquitectura sigue un patrón de controladores y servicios, donde las rutas HTTP están desacopladas de la lógica de negocio. Los servicios interactúan directamente con el sistema NATS a través de las clases `NatsService` y `FunctionService`.

Cola de Mensajes (NATS)

El sistema hace uso de NATS para coordinar la ejecución de funciones entre las diferentes instancias del servicio. Se utilizan distintos tópicos y el Key-Value Store para manejar las activaciones de funciones, su ejecución y la notificación de resultados. Es crucial para desacoplar los componentes y permitir una comunicación asíncrona. NATS ofrece alta disponibilidad y persistencia opcional, lo cual es esencial para mantener la integridad de las solicitudes de activación. A continuación se destacan algunos detalles de su implementación.

Tópicos

Activations

Este tópico se utiliza para gestionar las solicitudes de activación de funciones. Cuando un usuario solicita ejecutar una función, el servicio `FunctionService` publica un mensaje en este tema con los detalles de la tarea, como el identificador único de la solicitud (`taskId`), la imagen que debe ejecutarse, los parámetros de la función y la instancia que la solicita.

El mismo servicio también se suscribe a este tema para procesar las activaciones. Al recibir un mensaje, se encarga de ejecutar la función utilizando un proceso hijo. Una vez completada la ejecución, publica el resultado en el tema correspondiente a la instancia de origen.

Completed.<instance_id>

Cada instancia del servicio tiene su propio sub-tópico dentro de `Completed` , identificado por su ID. Este tópico se utiliza para notificar que una función previamente activada ha sido ejecutada. Los resultados de la función, o errores si la ejecución falló, se publican en este tópico.

El servicio `FunctionService` se suscribe a este tema en cada instancia utilizando su identificador. Al recibir un mensaje, verifica si el resultado corresponde a una tarea que fue solicitada en esa instancia. Si es así, resuelve la promesa asociada a la solicitud original, entregando el resultado al usuario.

Key-Value Store (kvStore)

Este almacén se utiliza para llevar un registro del número de funciones activas por usuario. Cada usuario tiene una clave específica que representa su conteo actual de activaciones activas. El servicio `FunctionService` lee y actualiza este valor para garantizar que no se supere el límite máximo de funciones concurrentes por usuario.

Antes de publicar una activación, el sistema consulta este almacén para determinar cuántas funciones activas tiene el usuario. Si el límite se ha alcanzado, la activación es rechazada. Cuando una función comienza o termina, el sistema actualiza este valor para reflejar los cambios en el estado de activación.

Flujo de comunicación

El proceso comienza cuando un usuario solicita la ejecución de una función. Esta solicitud se registra publicando un mensaje en `Activations` . Una instancia del servicio, suscrita a este tema, procesa el mensaje ejecutando la función en un proceso hijo (Node Child Process). Durante la ejecución, el sistema indica cada 30 segundos al servidor NATS que el mensaje está siendo procesado para evitar re-entregas innecesarias (a menos que se supere el tiempo de ejecución permitido para cada función).

Una vez que la ejecución finaliza, el resultado o el error correspondiente se publica en el subtema `Completed.<instance_id>` de la instancia que solicitó la ejecución. El servicio, que escucha este subtema, procesa el mensaje y resuelve la promesa asociada a la solicitud original, entregando el resultado al usuario.

Base de Datos

Las bases de datos utilizadas son 3 en total, cada una cumpliendo una responsabilidad única y relacionada a una tecnología diferente.

etcd

Es un almacén de clave-valor open source, distribuido y uniforme que permite la configuración compartida, la detección de servicios y la coordinación del programador de clústeres o sistemas distribuidos de máquinas. Se utiliza como centro de configuración para APISIX, que funciona como proxy inverso cumpliendo el rol de balanceador de carga y centro de autenticación.

Jetstream KV

El KV Store permite a las aplicaciones cliente crear contenedores de datos y utilizarlos como matrices (o mapas) asociativos persistentes y consistentes de forma inmediata (en lugar de eventualmente). Se utiliza en el proyecto para mantener registro de la cantidad de activaciones en curso de cada usuario.

Para mantener simplicidad, el diseño se mantuvo simple, relacionando a cada registro con el usuario mediante la convención de Jetstream de utilizar subjects que identifiquen el registro deseado, como a continuación:

En este ejemplo, un usuario con nombre de usuario **jorgeSD** tiene 4 activaciones en curso:

Key	Value
jorgeSD.activations.active	4

Se utilizó esta convención para tener la posibilidad de agregar otras funcionalidades como mantener cuenta de las activaciones completadas, activaciones con error, u otras propiedades relacionadas al usuario.

Mongo DB

MongoDB es un sistema de gestión de bases de datos no relacionales y de código abierto, que utiliza documentos flexibles en lugar de tablas y filas para procesar y almacenar varias formas

de datos. En el proyecto es utilizado para mantener registro de las funciones de cada usuario y las características que identifican a la función (nombre, referencia docker, etc).

Workers

Los workers se implementan utilizando el módulo `child_process` de Node.js, el módulo `child_process` proporciona la capacidad de generar nuevos procesos en lugar de hilos. Está diseñado para trabajar con tareas de larga duración e interacción con el sistema operativo. Al utilizar este módulo para crear un proceso secundario, se inicia un nuevo proceso independiente del proceso principal de Node.js, aunque ambos pueden estar conectados y comunicarse entre sí.

Las referencias que se hagan a workers en este proyecto, aunque pudiera ser confundido con los worker threads de Node.js, realmente se refieren al rol que cumplen estos procesos secundarios en el proyecto. Se eligió utilizar los procesos secundarios debido a que proveen mayor escalabilidad, aislamiento y robustez en la aplicación de este proyecto.

Cómo trabajan los Workers

A continuación se detalla la ejecución de los workers de forma resumida:

1. Los workers descargan la imagen y ejecutan el contenedor a partir de la imagen, comunicando al proceso principal el resultado de la ejecución.
2. Cada worker recibe la referencia a una imagen de Docker y el parámetro de entrada codificado en un string.
3. El worker ejecuta el contenedor de Docker correspondiente, pasa el string como parámetro y recoge el resultado desde stdout.
4. Los logs generados durante la ejecución se recopilan mediante stderr.

Escalado de los Workers

Escalado Horizontal

Un proceso secundario en Node.js hace referencia a una instancia completamente independiente del entorno de ejecución de Node.js. Se pueden crear procesos secundarios para ejecutar tareas en paralelo con el bucle de eventos principal. Aunque Node.js en sí es de un solo subproceso, se pueden generar procesos separados que se ejecuten en paralelo, lo que permite que el sistema realice operaciones de múltiples subprocesos a nivel del sistema operativo.

Los workers se activan automáticamente cada que se recibe una solicitud de activación, y estos se ejecutan concurrentemente, lo que permite escalar horizontalmente la activación de

funciones sin interrumpir la ejecución principal de la API REST de Node.js.

Configuración del sistema para adaptarse a cambios en la carga

Balanceo de carga

APISIX distribuye las solicitudes entre múltiples instancias del servidor API.

Cola dinámica

La cola NATS asegura que las tareas no se pierdan y permite manejar picos de carga temporalmente almacenando las solicitudes en espera.

Límites de concurrencia:

Configuración para rechazar solicitudes cuando se alcance el máximo número de activaciones concurrentes. Se configuró un límite fijo de activaciones concurrentes por usuario.

Cómo debe configurarse todo el sistema para adaptarse a los cambios de carga

Debido a el diseño del sistema, un cambio de carga excesivo se vería reflejado mayormente en la carga del CPU y memoria dispuestos para atender los procesos que ejecuten la activación de las funciones. Al ser la comunicación de las activaciones asíncrona y no bloqueante, no se vería demasiado afectado el flujo de comunicación con la API y el servidor NATS.

Lo ideal sería monitorear los recursos del sistema mediante las herramientas apropiadas (si se despliega en el cloud, se podrían establecer protocolos de autoescalado) para realizar un escalado horizontal (para evitar la interrupción de las funciones en curso) agregando más hosts que puedan atender la demanda de ejecución de funciones durante el pico de demanda.

Así mismo, se espera que todas las instancias del servicio API se comuniquen con el servidor NATS delimitado dentro del mismo entorno, sin embargo el diseño podría cambiar para que múltiples hosts puedan comunicarse con la misma instancia del servidor NATS para lograr una mejor resilencia, incluso se pueden configurar clusters del servidor NATS para conseguir una mejor distribución y replicación, sin embargo estos detalles se dejaron fuera de la implementación por temas de tiempo.

Manejo del acceso a servicios externos por parte de las funciones

Las funciones pueden acceder a servicios externos mediante un acceso libre a internet siempre que el host que ejecuta los servicios tenga configurados los puertos necesarios para su salida a internet, cualquier necesidad de identificar las funciones mediante políticas de roles queda a la responsabilidad del usuario en la implementación de las funciones.

Virtudes y limitaciones

Virtudes

Flexibilidad y modularidad

Las funciones pueden conectarse dinámicamente a múltiples servicios externos según sea necesario, permitiendo aplicaciones escalables y reutilizables.

Independencia de implementación

Cada función puede ser independiente y contener las configuraciones específicas necesarias para acceder a servicios externos, como credenciales o puntos de acceso.

Escalabilidad

Gracias al uso de contenedores Docker, cada función puede escalarse de forma horizontal sin interferir con otras funciones o servicios externos.

Limitaciones

Latencia adicional

El acceso a servicios externos desde funciones dentro de contenedores puede introducir latencias, especialmente si el servicio está alojado en una red diferente o distante.

Gestión de credenciales

Proveer credenciales seguras a cada función sin comprometer su seguridad es complejo.

Conexiones simultáneas

Funciones concurrentes pueden sobrecargar servicios externos, lo que requiere mecanismos de limitación de solicitudes o gestión de picos.

Dependencia de red

Si la función depende críticamente de un servicio externo y este falla, la ejecución de la función también fallará.

Datos adicionales que deberían incluirse en las cadenas codificadas

Para que las funciones puedan interactuar eficientemente con servicios externos, deben incluirse algunos metadatos importantes al codificar las cadenas de parámetros o configuraciones:

Credenciales de acceso seguras

Información como claves API o tokens JWT para autenticar la función con el servicio externo. Deberían ser efímeras y gestionarse mediante un sistema seguro como AWS Secrets Manager, HashiCorp Vault, o Kubernetes Secrets.

Registros de servicio

Direcciones URL, nombres de host o endpoints de los servicios externos que la función necesita.

Configuraciones personalizadas

Parámetros específicos para el servicio, como regiones geográficas (ej., `us-east-1`), puertos, o configuraciones de red.

Políticas de reintento

Información sobre cuántos intentos realizar si falla la conexión con el servicio externo.

Tiempos de espera

Límites de tiempo para las solicitudes hacia el servicio externo, para evitar bloqueos.

Identificador de solicitud (traceId)

Para propósitos de trazabilidad en sistemas distribuidos, cada solicitud puede llevar un identificador único que permita rastrear su ejecución a través de múltiples servicios.

Otros aspectos importantes

Seguridad en el acceso

El acceso a servicios externos debe estar estrictamente controlado. Los siguientes mecanismos son recomendables:

- Uso de roles y políticas de IAM (Identity and Access Management) para limitar el acceso.

- Conexiones cifradas (por ejemplo, TLS) entre las funciones y los servicios externos.
- Reducción del ámbito de acceso de las funciones mediante redes privadas virtuales (VPN) o redes internas como VPC en AWS.

Gestión de la configuración dinámica:

Usar herramientas como ConfigMaps o Environment Variables para proveer configuraciones dinámicas a las funciones sin necesidad de modificar el código fuente.

Monitoreo y auditoría

Implementar un sistema de monitoreo que registre todas las solicitudes hacia servicios externos. Esto puede ayudar a identificar problemas, mitigar riesgos de abuso, y cumplir con normativas de compliance.

Manejo de fallos

Incorporar un diseño resistente que permita manejar fallos en los servicios externos mediante patrones como:

- Circuit Breaker (interrumpir solicitudes tras múltiples fallos).
- Fallback (usar resultados predeterminados si el servicio no responde).
- Cacheo (almacenar temporalmente respuestas de servicios para reducir dependencias).

Con la implementación de estas estrategias, el sistema FaaS podría manejar eficientemente el acceso a servicios externos, asegurando su seguridad, confiabilidad y escalabilidad.

Comparación con otros sistemas FaaS open source

Nuestra implementación de FaaS basada en contenedores Docker tiene características que la distinguen de alternativas open source como OpenFaaS, Apache OpenWhisk, Knative y Kubeless. A continuación, se describe cómo se compara en términos de arquitectura, flexibilidad, escalabilidad, facilidad de uso y mantenimiento, rendimiento, y ecosistema.

Arquitectura

El sistema es compacto y minimalista, enfocado específicamente en ejecutar funciones dentro de contenedores Docker utilizando un sistema de mensajería como NATS para orquestar la ejecución. Esto permite un control fino sobre cada etapa del proceso, desde la cola de activaciones hasta la ejecución y la recolección de resultados. Alternativas como OpenFaaS y Knative, por otro lado, proporcionan arquitecturas más amplias y modulares basadas en Kubernetes. Estas plataformas abstraen gran parte de la complejidad operativa, gestionando

automáticamente el escalado y la infraestructura subyacente, pero a costa de ser más complejas y menos controlables a nivel individual.

Flexibilidad

Nuestra implementación ofrece un alto nivel de personalización en cómo se definen y ejecutan las funciones, ya que todo está bajo el control del usuario. Sin embargo, las alternativas open source están diseñadas para soportar un rango más amplio de casos de uso. Por ejemplo, Apache OpenWhisk admite múltiples lenguajes de programación y puede integrarse con herramientas avanzadas como sistemas de análisis de eventos o APIs serverless. Knative, por su parte, se basa en Kubernetes, lo que lo hace altamente extensible para entornos híbridos o en la nube.

Escalabilidad

Esta solución está optimizada para entornos controlados con patrones de carga específicos. El uso de NATS para la gestión de colas es eficiente, pero puede requerir ajustes manuales para manejar picos de tráfico o fallos de nodo. Las alternativas como Knative o OpenFaaS aprovechan Kubernetes para escalar automáticamente las instancias de función basándose en métricas como solicitudes concurrentes o consumo de CPU, lo que facilita su operación en entornos dinámicos o con cargas impredecibles.

Facilidad de uso

La facilidad de uso y mantenimiento es otro factor importante. Nuestra implementación, aunque más sencilla en términos de componentes, podría ser más difícil de adoptar para usuarios no familiarizados con Docker y las configuraciones específicas del sistema. OpenFaaS y Apache OpenWhisk ofrecen interfaces más amigables, con portales web o CLI diseñados para facilitar la creación y despliegue de funciones sin necesidad de entender los detalles internos del sistema.

Rendimiento

En cuanto al rendimiento, nuestra implementación probablemente tenga una ventaja en casos donde se requiere baja latencia en la activación de funciones, ya que utiliza un diseño ligero con pocos niveles de abstracción. Las plataformas como Knative o Kubeless, al estar basadas en Kubernetes, suelen tener una mayor latencia inicial debido al tiempo de arranque de pods, aunque esto se puede mitigar mediante configuraciones avanzadas como el "pod autoscaling".

Ecosistema

Finalmente, el ecosistema y soporte comunitario es un área donde las alternativas open source tienen una ventaja significativa. Estas plataformas cuentan con una amplia comunidad de

desarrolladores, documentación extensa, y plugins para integrarse con servicios populares como bases de datos, colas de mensajes, y herramientas de monitoreo. Nuestra implementación, aunque específica y controlada, requeriría esfuerzos adicionales para crear un ecosistema similar, especialmente si se busca integrar nuevas funcionalidades o soportar más casos de uso.

Conclusión de la comparación

En resumen, nuestra implementación destaca por ser ligera, específica y altamente controlable, lo que la hace ideal para proyectos donde se conoce bien la carga y el entorno. Sin embargo, las soluciones open source suelen ser más versátiles, escalables y fáciles de usar para un rango más amplio de aplicaciones, a costa de una mayor complejidad inicial. La elección entre tu sistema y una plataforma open source dependerá de los requisitos específicos del proyecto y las prioridades del equipo de desarrollo.