MiniFaaS en Kumori

Universidad Politécnica de Valencia
Máster Universitario en Computación en la Nube y de Altas Prestaciones
Asignatura Cloud Computing

Autor:
Juan González Caminero

Contenidos

1 Introducción **2**

2 Diseño **2**

2.1 Breve descripción de la funcionalidad de cada componente 2

3 Uso del sistema **5**

3.1 API REST 5

3.2 Funcionalidades básicas 6

3.2.1 Especificación de una función 6

3.2.2 Registro de una función en el sistema 6

3.2.3 Envío de una petición de ejecución y recepción del resultado 6

3.2.4 Autoescalado 7

3.2.5 Registro del uso del sistema 7

4 Ejemplo de uso **7**

5 Problemas conocidos **9**

1 Introducción

Este documento describe el proceso de diseño e implementación de un sistema Function as a Service (FaaS) en la plataforma Kumori, con la siguiente estructura: La sección 2 explica el diseño del sistema y muestra un diagrama del mismo, la sección 3 presenta la API REST, y describe cómo se realizan una serie de tareas básicas. La sección 4 propone un ejemplo de uso del sistema, y por último la sección 5 habla de algunos problemas conocidos del sistema en su estado actual.

2 Diseño

El sistema, que se puede ver en el diagrama de la figura 1, consta de 5 componentes principales:

- Frontend: Recibe peticiones REST de los usuarios.
- Database: Base de datos CockroachDB.
- NATS: Cola de mensajes que gestiona las comunicaciones entre el frontend y los Workers.
- Worker: Recibe peticiones de ejecución de funciones y las realiza.
- Autoscaler: Monitoriza el sistema y añade o quita trabajadores en función del estado de los mismos.

Además, se usan varios componentes de Kumori. Para recibir peticiones https del exterior se usa un http-inbound conectado con el fronted a través de un balanceador de carga. Entre los distintos componentes del sistema se utilizan conectores "fullconnector".

Tanto el Frontend como los Workers están implementados en JavaScript, y se ejecutan mediante Node.js. El sistema sólo puede ejecutar funciones descritas en este lenguaje. El Autoscaler está implementado mediante un script Bash.

2.1 Breve descripción de la funcionalidad de cada componente

Frontend:
El Frontend recibe peticiones REST de los usuarios. A excepción de las peticiones de ejecución de funciones, que se envían a los Workers a través de la cola NATS, el resto de peticiones se ejecutan en el propio frontend, ya que la carga computacional para el nodo es muy pequeña, y la base de datos hace parte de las operaciones. Por ejemplo, el registro de un usuario en la base de datos, o extraer el historial de uso de recursos de un usuario.

Imágen en Docker Hub:
https://hub.docker.com/repository/docker/juangonzalezcaminero/faas_frontend

Database:
La base de datos contiene tres tablas: La tabla de usuarios, que sólo contiene una columna, el nombre de usuario. La tabla de funciones, que contiene el nombre de la función, el propietario, y el código de la función. Por último, la tabla de uso, que almacena, para cada ejecución, el nombre de la función, el usuario, y el tiempo de ejecución.

NATS:
La cola de mensajes NATS actúa como un balanceador de carga entre el frontend y los trabajadores. Además de esto, los trabajadores envían el resultado de las ejecuciones al frontend a través de NATS.

Worker:
Los trabajadores escuchan peticiones de ejecución de funciones desde NATS. Cuando reciben una petición, extraen el código de la base de datos y lo ejecutan con los argumentos proporcionados. Cuando la ejecución termina, envían el resultado al frontend a través de la cola NATS.

Imágen en Docker Hub:
https://hub.docker.com/repository/docker/juangonzalezcaminero/faas_worker

Autoscaler:
El Autoscaler monitoriza el estado del sistema mediante el comando kumorictl describe deployment, y cuando es necesario, genera un manifiesto de despliegue que usa para actualizar el número de trabajadores usando el comando kumorictl update deployment.

Imágen en Docker Hub:
https://hub.docker.com/repository/docker/juangonzalezcaminero/faas_autoscaler

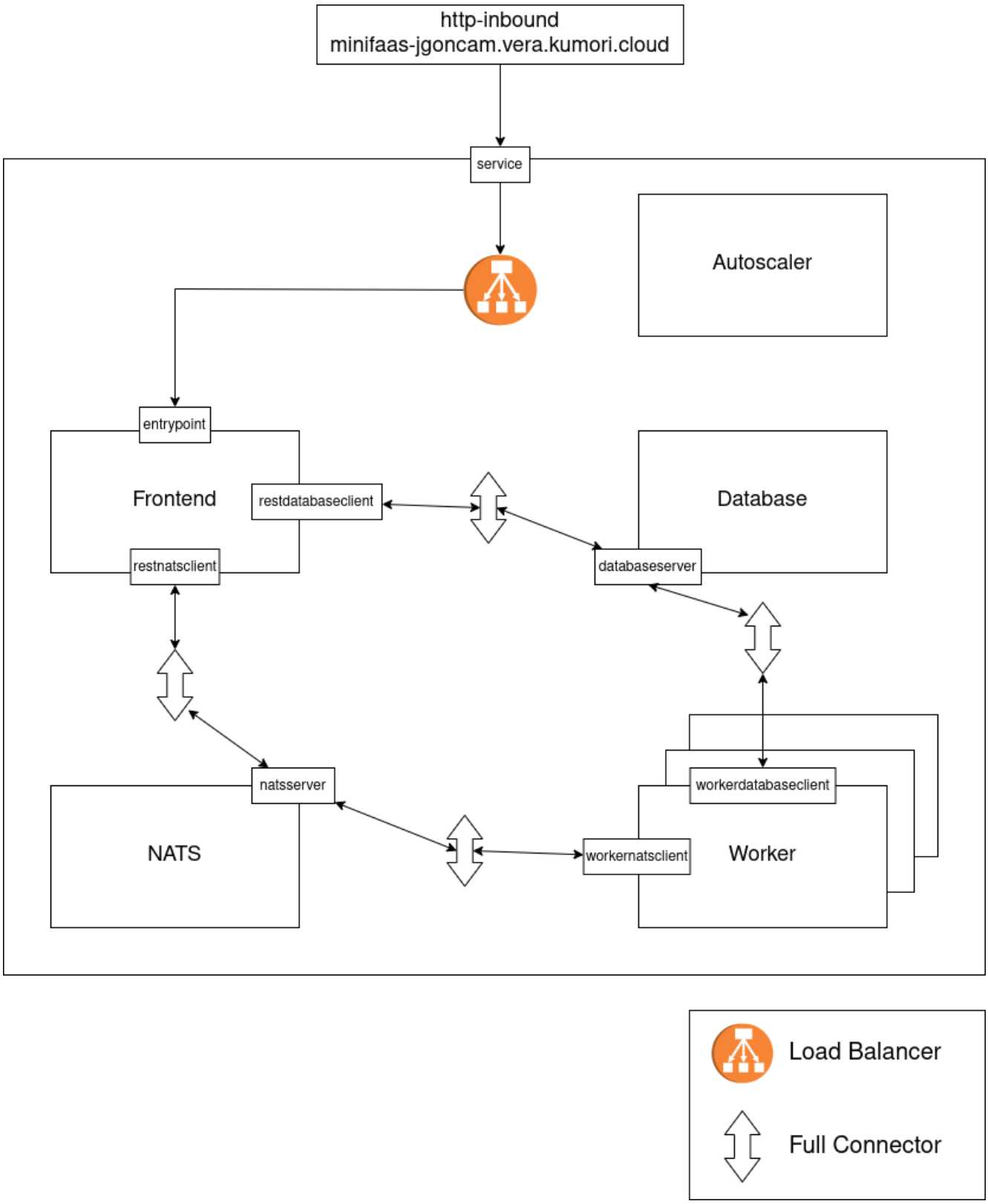

Figure 1: Diseño del sistema

3 Uso del sistema

3.1 API REST

La API del sistema acepta las siguentes peticiones:

- **GET /**
  Si el sistema está en funcionamiento, devuelve un mensaje que lo indica.

  Ejemplo: curl -X GET https://minifaas-myuser.vera.kumori.cloud/

- **GET /user**
  Devuelve una lista con los usuarios registrados en el sistema.

  Ejemplo: curl -X GET https://minifaas-myuser.vera.kumori.cloud/user

- **GET /user/<username>/usage**
  Devuelve el historial de ejecución para el usuario especificado, y el tiempo total de uso del sistema.

  Ejemplo: curl -X GET https://minifaas-myuser.vera.kumori.cloud/user/faasuser/usage

- **GET /user/<username>/function**
  Devuelve la lista de funciones registradas para el usuario especificado.

  Ejemplo: curl -X GET https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function

- **POST /user/<username>**
  Registra el nombre de usuario especificado en el sistema.

  Ejemplo: curl -X POST https://minifaas-myuser.vera.kumori.cloud/user/faasuser

- **DELETE /user/<username>**
  Elimina el usuario especificado del sistema.

  Ejemplo: curl -X DELETE https://minifaas-myuser.vera.kumori.cloud/user/faasuser

- **POST /user/<username>/function/<functionname>**
  Registra la función con el nombre especificado para el usuario especificado. El código de la función debe enviarse en el cuerpo del mensaje.

  Ejemplo: curl –data-binary '@suma.js' -H "Content-Type: text/plain"
  https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/suma

- **GET /user/<username>/function/<functionname>**
  Ejecuta la función especificada y devuelve el resultado.

  Ejemplo: curl -X GET -d '"args":[1, 2]' -H "Content-Type: text/plain"
  https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/suma

- **DELETE /user/<username>/function/<functionname>**
  Elimina la función especificada.

  Ejemplo: curl -X DELETE https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/suma

3.2 Funcionalidades básicas

3.2.1 Especificación de una función

Las funciones se deben especificar en lenguaje JavaScript. Hay que tener en cuenta las siguientes restricciones:

- El fichero que se envía debe contener la definición de una función JavaScript con el mismo nombre con el que se registra en el sistema.
- El fichero que se envía no puede contener código fuera de la función.
- No es posible por el momento la instalación de módulos externos.
- El resultado devuelto al usuario es la salida de la función. La salida debe ser serializable en formato JSON.

Esta es la definición de dos de las funciones de ejemplo del sistema:
suma.js

```
function suma(a, b)
{
        return a+b;
}
```

random_array.js

```
function random_array(size)
{
        var arr=[];
        for(let i=0; i<size; i++){
            arr.push(Math.random());
        }
        return arr;
}
```

3.2.2 Registro de una función en el sistema

El registro de una función en el sistema se realiza a través de la petición:

POST /user/<username>/function/<functionname>

Si el usuario está registrado en el sistema, y la función no existe, se registrará en la base de datos. Es necesario enviar la función en el cuerpo del mensaje, e indicar que el contenido es texto plano. Con curl, asumiendo que la función está en el fichero funcion.js, esto se haría con las opciones:

–data-binary '@funcion.js' -H "Content-Type: text/plain"

3.2.3 Envío de una petición de ejecución y recepción del resultado

Para enviar una petición de ejecución al sistema, se utiliza la petición:

GET /user/<username>/function/<functionname>

Es necesario enviar los argumentos como texto plano en el cuerpo del mensaje, con el formato:

"args":[arg1, arg2, ...]

Con curl esto se puede hacer con los argumentos:

-d '"args":[arg1, arg2, ...]' -H "Content-Type: text/plain"

Las peticiones de ejecución son síncronas, el usuario que realiza la petición queda a la espera del resultado, hasta que la ejecución termina y se recibe la respuesta desde Frontend.

Internamente, cuando Frontend recibe una petición de ejecución, le asigna una ID y la registra en la cola NATS, y se registra en un asunto de la cola específico para esa ejecución. La cola NATS asigna el trabajo a un Worker al azar, y este ejecuta la función y devuelve el resultado a Frontend a través de NATS. Finalmente, Frontend envía la salida al usuario.

3.2.4 Autoescalado

El componente Autoscaler monitoriza el estado del sistema a través de kumorictl describe deployment. El objetivo del autoescalado es que haya siempre tres Workers por debajo de un 80% de uso de CPU.

Cada minuto, el script comprueba que este sea el caso. Si hay menos de tres trabajadores por debajo del 80% de carga, añade uno utilizando kumorictl update deployment. Esto se repite cada minuto hasta alcanzar el número objetivo. Si hay más de tres trabajadores por debajo de esa carga, se elimina un trabajador usando el mismo procedimiento.

3.2.5 Registro del uso del sistema

El sistema mide el tiempo de uso por parte de cada usuario. Cuando se ejecuta una función, se mide el tiempo con el contador "process.hrtime" de JavaScript, que ofrece una precisión de nanosegundos. Al terminar la ejecución, se almacena el tiempo en la base de datos asociado al usuario y función correspondientes. Es posible obtener el historial de ejecuciones de un usuario junto al tiempo total de uso del sistema.

4 Ejemplo de uso

Esta sección presenta un ejemplo de uso del sistema con algunas funciones de ejemplo.
Ejecutamos los siguientes comandos en el directorio "minifaas":

Descargamos las dependencias:

```
kumorictl fetch-dependencies
```

Configuramos la URL de admisión:

```
kumorictl config --admission admission-ccmaster.vera.kumori.cloud
```

Hacemos login en la plataforma con nuestro usuario:

```
kumorictl login myuser
```

Nota: El autoscaler utiliza mis credenciales para realizar la monitorización (jgoncam), por lo que esta funcionalidad no estará disponible si el despliegue se realiza con otro usuario.
Establecemos el dominio por defecto:

```
kumorictl config --user-domain myuser
```

Registramos un certificado:

```
kumorictl register certificate faascert.wildcard \
  --domain *.vera.kumori.cloud \
  --cert-file cert/wildcard.vera.kumori.cloud.crt.wildcard \
  --key-file cert/wildcard.vera.kumori.cloud.key.wildcard
```

Regitramos el inbound:

```
kumorictl register http-inbound minifaasinb \
  --domain minifaas-myuser.vera.kumori.cloud \
  --cert faascert.wildcard
```

El inbound no funcionará hasta pasado un tiempo debido a los tiempos de propagación de DNS:

```
curl https://minifaas-myuser.vera.kumori.cloud
curl: (6) Could not resolve host: minifaas-myuser.vera.kumori.cloud
```

Registramos el despliegue:

```
kumorictl register deployment minifaasdep \
--deployment ./cue-manifests/deployment
```

Nota: Es importante llamar al despliegue "minifaasdep", ya que el autoscaler utiliza ese nombre para monitorizar el sistema.
Enlazamos el inbound con el despliegue:

```
kumorictl link minifaasdep:service minifaasinb
```

Ahora podemos ejecutar:

```
kumorictl describe deployment minifaasdep
```

Para visualizar el estado del sistema.

A continuación, vamos al directorio "Ejemplo".
Comenzamos por comprobar que el sistema está activo:

```
curl -X GET https://minifaas-myuser.vera.kumori.cloud/
The MiniFaaS frontend is up!
```

A continuación, añadimos un usuario:

```
curl -X GET https://minifaas-myuser.vera.kumori.cloud/user
There are no users registered in the system
curl -X POST https://minifaas-myuser.vera.kumori.cloud/user/faasuser
faasuser added to the system
```

Registramos dos funciones de ejemplo para el usuario:

```
curl --data-binary '@suma.js' -H "Content-Type: text/plain" \
https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/suma
suma added to the system for user faasuser

curl --data-binary '@random_array.js' -H "Content-Type: text/plain" \
https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/\
random_array
random_array added to the system for user faasuser
```

Y realizamos algunas ejecuciones de prueba:

```
curl -X GET -d '{"args":[1, 2]}' -H "Content-Type: text/plain" \
https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/suma
```

```
3

curl -X GET -d '{"args":[50, 40]}' -H "Content-Type: text/plain" \
https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/suma
90

curl -X GET -d '{"args":[4]}' -H "Content-Type: text/plain" \
https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/\
random_array
[0.437667661747563,0.01856356190768338,0.5870984797426209,0.4329093550785834]

curl -X GET -d '{"args":[2]}' -H "Content-Type: text/plain" \
https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/\
random_array
[0.9527953579139501,0.23508150908274783]
```

Ahora podemos comprobar el tiempo de uso para este usuario:

```
curl -k https://minifaas-myuser.vera.kumori.cloud/user/faasuser/usage
Execution history for faasuser
[["suma",0.000150758],["suma",0.000221502],
["random_array",0.000213538],["random_array",0.000173587]]
Total usage in seconds: 0.000759385
```

Para ver el efecto del autoescalado, podemos usar la función bucle, que tarda en ejecutarse unos 8 minutos en un worker en Kumori. Durante la ejecución, podemos ir comprobando mediante el comando kumorictl describe deployment, cómo aumenta el uso de CPU para uno de los workers, y cómo se despliega automáticamente uno nuevo cuando se supera un 80% de uso de la CPU. Una vez termina la ejecución, podemos observar cómo se elimina el Worker adicional.
Es posible que el escalado no se produzca inmediatamente al superar el 80% de carga, y tarde algo más. Esto se debe a que el Autoscaler comprueba la carga cada minuto, y una vez detecta que se ha superado el límite hace login en kumori y despliega el nuevo manifiesto, lo cual lleva un tiempo. En todas las pruebas hasta ahora, el escalado se ha producido mucho antes de que termine esta ejecución de 8 minutos.

```
curl -k --data-binary '@bucle.js' -H "Content-Type: text/plain" \
https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/bucle
bucle added to the system for user faasuser

curl -k -X GET -d '{"args":[]}' -H "Content-Type: text/plain" \
https://minifaas-myuser.vera.kumori.cloud/user/faasuser/function/bucle
```

5 Problemas conocidos

En esta sección se describen algunos problemas del sistema en su estado actual.

- **Ejecuciones largas:** Cuando llega una petición de ejecución de una función que tarda mucho en ejecutarse, el Worker al que se le asigna no puede atender otras peticiones. Debido a que NATS no ofrece una forma de enviar las siguientes peticiones a los trabajadores que estén libres (Por ejemplo, si el modo de la cola fuera Pull en lugar de Push), y a que envía las peticiones de forma aleatoria, es posible que una petición llegue al Worker que está ejecutando esta función, y que no se ejecute hasta que la anterior termina.
- **Timeout del HTTP-Inbound:** Parece que el HTTP-Inbound termina la conexión con el cliente pasados unos 50 segundos desde la petición, lo que implica que si la ejecución tarda más que esto,

no se recibirá el resultado. Esto tiene fácil solución aumentando el tiempo de timeout del inbound, pero parece que por el momento no existe la opción.

- **Certificado caducado:** El certificado que tenemos los alumnos está caducado, por lo que es necesario desactivar la verificación SSL al usar curl mediante la opción -k.