UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

MEMORIA LABORATORIO SAD

QUEUE SERVICE IN THE CLOUD

Eduardo Gómez Lorente

Oliver Gutiérrez Saucedo

María Lina Riera Ferrer

2021

Resumen

El presente proyecto tiene como objetivo ofrecer un servicio clould siguiendo el modelo de negocio FaaS (Function as a Service).

La funcionalidad que ofrece este cloud service es la de calcular la integral de pi entre una cota inferior y superior y con un margen de precisión que dependerá de la cantidad de iteraciones que haya seleccionado el cliente.

El cliente se comunica con el servicio a través del protocolo http realizando una petición post con un payload en los que especifica los tres parámetros mencionados anteriormente. Esta petición es atendida por un conjunto de microservicios REST replicados que actúan como Frontend para recibir la petición y encaminarla hacia otro microservicio donde se almacenan las peticiones llamado Queue, este componente también replicado se encarga de distribuir el trabajo entre los Workers que son los que implementan la solución del problema. Cuando el problema está resuelto, envían la respuesta a las Queues y luego estas serán las encargadas de responder al cliente concreto que hizo la petición.

En el proyecto se abordan cuestiones como la alta disponibilidad a través de la replicación de cada componente y, por otra parte, la tolerancia a fallos por medio del patrón de envíos *Exactly once*.

Tabla de contenidos

1. Componentes

Para el desarrollo de este proyecto, se han tenido que implementar diferentes componentes con el objetivo de cumplir toda la funcionalidad requerida. Como ya sabemos, los servicios Cloud como este, deben implementarse combinando diferentes microservicios, como pueden ser en nuestro caso las queues, los workers, el service registry y los frontends. En este apartado vamos a exponer todos estos componentes que hemos diseñado para nuestro proyecto y explicar cómo se han implementado.

1.2 Service Registry

Cuando trabajamos con microservicios replicados como es nuestro caso, necesitamos tener un registro de ellos, es decir, conocer las instancias de los microservicios que están corriendo en cada momento para saber, por ejemplo, el puerto y la dirección IP donde están funcionando y que esto facilite nuestra tarea de balancear la carga.

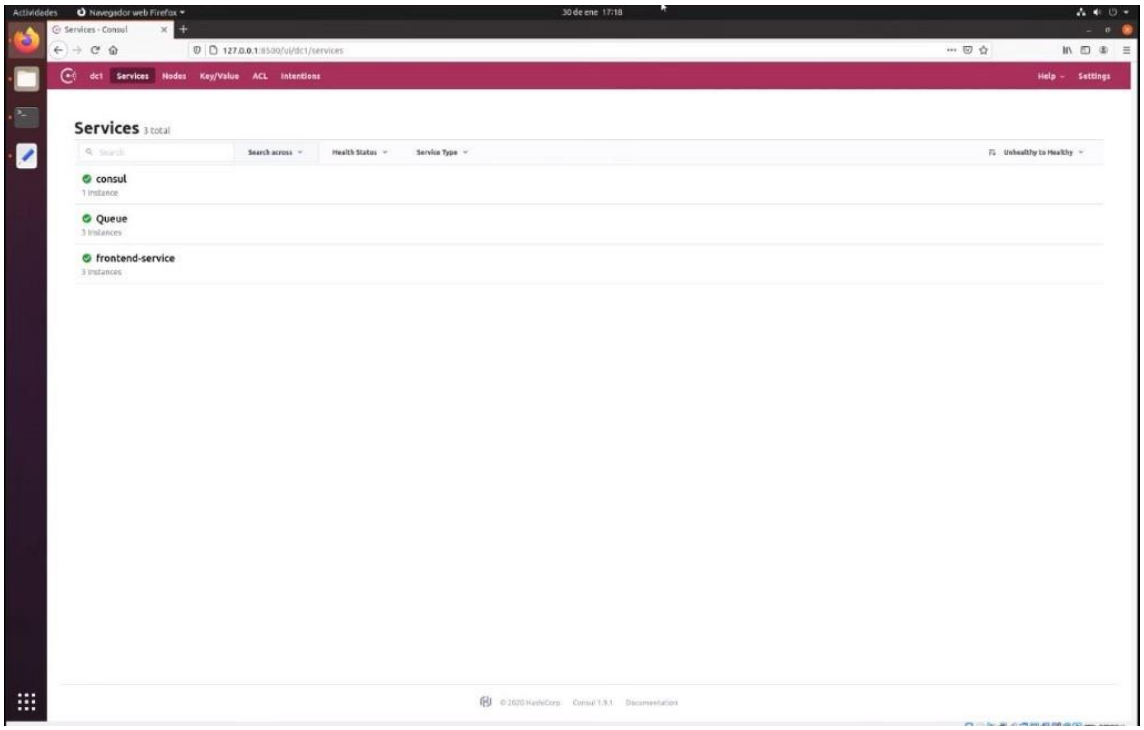
Para resolver esto, utilizamos el patrón Service Registry que nos propone crear un servidor centralizado donde todos nuestros microservicios se registren cuando se pongan en funcionamiento. De esta forma dentro de este servidor tendremos todas las instancias de los microservicios activas con sus direcciones IP y puerto asignadas a la hora del arranque.

Como se ha comentado, esto nos sirve para saber qué microservicios están activos en todo momento, ya que aparte de registrarse, los microservicios están constantemente interactuando con el Service Registry para que este se asegure de que ese microservicio sigue activo o si por el contrario no responde, eliminarlo del registro después de un tiempo determinado.

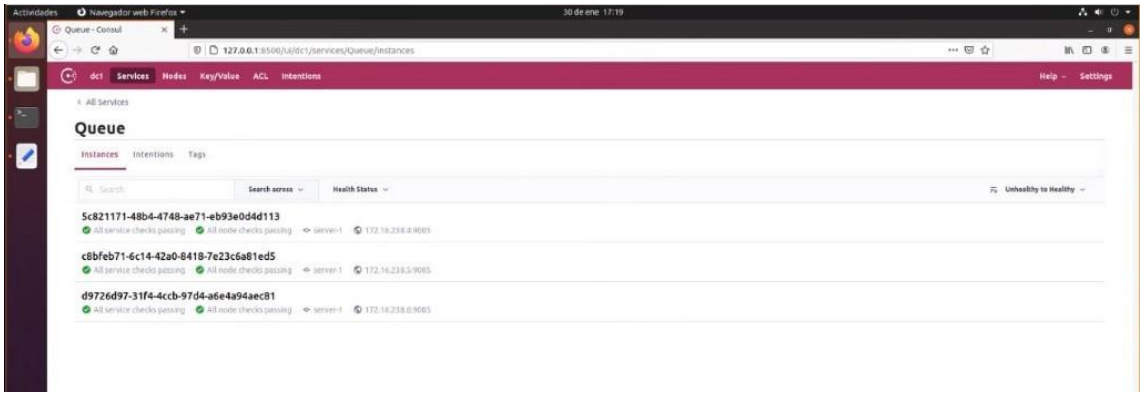
En nuestro caso, se decidió utilizar el servicio de Consul que es una herramienta que nos ofrece HashiCorp, donde todos nuestros microservicios se registrarán y podrán consultar información sobre el resto de los microservicios activos.

Las 3 características principales que nos ofrece Consul y por las que hemos seleccionado esta herramienta son las siguientes:

- En primer lugar, nos ofrece un servicio de Service Discovery, es decir, descubrimiento de servicios centralizado, el cual es bastante sencillo de gestionar teniendo en cuenta que la alternativa sería implementar un protocolo de búsqueda descentralizado como los de P2P.
- Por otro lado, nos ofrece Health Checking ya que los clientes de Consul pueden proporcionar cualquier número de comprobaciones de estado y nosotros podemos utilizar esta información para, por ejemplo, monitorear el estado del clúster, en nuestro caso sería para tener monitoreado el estado de todos nuestros microservicios, para poder detectar cualquier problema en uno de ellos de manera instantánea.
- Consul despliega un servidor web en local que nos ofrece una página web a modo de UI donde podemos ver toda la información de forma intuitiva y cómoda. (se despliega en localhost:8500).



Como podemos observar en la imagen, accediendo a la *url* de nuestro servicio podemos ver todas las instancias de los microservicios que tenemos activos y registrados. Dentro de cada microservicio podemos ver sus diferentes instancias:



En este caso se muestran las 3 instancias del microservicio de colas activas en ese momento.

Por último, aquí podemos observar cómo hemos realizado el registro de un microservicio a Consul.

```
consul.agent.service.register(check1, function(err) {
    if (err) throw err;
});
```

Este caso es el registro del componente *Queue*. Se llama a la función anterior para realizar un registro de ese microservicio. Para ello definimos la variable check con toda la información a registrar:

```
var check1 = {
    id: SERVICE_ID,
    name: SERVICE_NAME,
    addres: HOST,
    port: PORT1,
    check: {
        tcp: HOST+':'+PORT1,
        ttl: '5s',
        interval: '5s',
        timeout: '5s',
        deregistercriticalserviceafter: '1m'
    }
};
```

Por tanto, esta será la información que luego tendremos disponible a través de Consul y accesible desde cualquier otro microservicio. También se declaran algunos valores para por ejemplo el intervalo de comprobación de que el servicio sigue activo o el tiempo que debe pasar para que cuando no responda se elimine ese microservicio del *Service Registry*.

1.2 Load Balancer

Un aspecto muy importante dentro de nuestro proyecto ya que tenemos múltiples réplicas de los microservicios es el balanceo de carga. Con esto queremos que conseguir que, en primer lugar, todas las peticiones que le lleguen al Front desde el exterior se balanceen en las diferentes instancias de este, es decir, si tenemos 3 instancias del microservicio Front y 3 peticiones, cada petición irá a una instancia distinta. Lo mismo ocurre con las peticiones desde el Front a las diferentes instancias de las colas y desde los *worker*s a estas mismas colas.
El primer balanceador de carga, el del cliente al *Frontend* está desarollado con *haproxy* y el otro haciendo uso de las posibilidades que nos ofrece zmq. Por un lado, el *fron*t tendrá un socket *dealer* que repartirá las peticiones a las colas siguiendo el patrón (RR) y los *workers* tendrán un socket de tipo *req* para la misma función.

1.3 Frontend

El componente *Frontend* es el encargado de en primer lugar, utilizarse como entrypoint para poder recibir peticiones y, por otro lado, debe enviar las peticiones recibidas al componente *Queue* para que este pueda almacenar los trabajos y una vez hayan sido resueltos por un *Worker* el componente *Queue* le devuelva el resultado al *Frontend* que recibió esa petición.
En nuestro caso, el entrypoint lo hemos implementado haciendo uso de *express*, que se utiliza básicamente para diseñar y crear aplicaciones web y API de una forma rápida y sencilla, nos ayuda a administrar el enrutamiento, las solicitudes HTTP, etc.

```
service.post('/frontEnd/integraPI', async (req, res, next) => {
  isDone = true;
  var infoerror = validateInput(req);
  if(infoerror != 'OK'){
      let reponse = {
          code: 502,
          message: infoerror,
          responded: `Responded by: ${SERVICE_ID}`
      };
      return res.json(reponse);
  }

  let mensaje = {
      inf: req.body.inf,
      sup: req.body.sup,
      ite: req.body.ite
  };
  log.debug('Sending message: ' + JSON.stringify(mensaje));
  dealer.send(['','','', JSON.stringify(mensaje), 'peticion']);
  while(isDone){
      await sleep(300);
  }
  return res.json({code: 200, message: `${resp}`, responded: `Responded by: ${SERVICE_ID}`});
});

function validateInput(req){
```

En esta imagen podemos observar cómo hacemos uso de *express* en nuestro componente Frontend.
La variable service hace referencia a express y con esta función podemos recibir las peticiones en la dirección especificada y obtendríamos los datos necesarios para resolver la petición que posteriormente ejecutará un Worker, como en este caso son el límite inferior, el superior y el número de iteraciones.
Por otro lado, una vez ha recibido un mensaje de petición, lo que hace es reenviarlo mediante el balanceo de carga que se ha mencionado con anterioridad a una de las colas que estén disponibles en ese momento para que resuelva la petición y por último le devuelva el resultado correspondiente a dicha petición.

1.4 Worker

El componente worker es el que se hará cargo de la resolución de las peticiones del frontend. En nuestro caso concreto, dados unos parámetros realizará el cálculo de la integral.
Por otro lado, el worker en nuestro caso se comunicará con las colas, mediante el balanceo de carga mencionado anteriormente de worker-queue para poder obtener el trabajo a realizar ya que en ningún momento el worker tiene interacción directa con el frontend.
Como todos los anteriores, este componente ha sido implementado con *zmq*. El trabajo que hemos definido que realice es el siguiente:

```
function integra(inf,sup,n){
    let infl = Number.parseFloat(inf);
    let supl = Number.parseFloat(sup);
    let nl = Number.parseInt(n);
    let integral= 0.0;
    let i = 0;
    let inc = 0.0;
    inc=(supl-infl)/nl;
    area= 0.0;
    x= infl;
    while (i<n){
        area += Math.sin(x);
        x = x+inc;
        i++;
    }
    integral = area*inc;
    return integral;
}
```

Donde inf hace referencia al límite inferior, sup al límite superior y n al número de iteraciones.

1.5 Queue

Otro componente que nos encontramos es el componente Queue, este componente es básicamente un buffer que almacena las peticiones que se realizan a través del Frontend.
Este componente, implementado también con zmq, recibe las peticiones del frontend a través del balanceo de carga y las almacena a la espera de que algún worker se comunique con ella y de esta forma el worker pueda realizar el trabajo. Una vez resuelto el trabajo, el worker le envía el resultado a este componente que será el encargado de devolver al mismo cliente que le hizo la petición el resultado correspondiente.
Un aspecto importante que este componente tiene que cumplir es que pueda comunicarse con otras instancias de sí mismo ya que en esta arquitectura todos los componentes mencionados están replicados. Este aspecto está explicado con detalle en el apartado de comunicación entre colas replicadas.
Por otro lado, hemos realizado un diseño de las colas a prueba de fallos, para que el sistema no deje de funcionar si un worker falla y no dejar ningún trabajo sin responder.

2. Propiedades del Cloud Service

En este apartado explicaremos las características y propiedades de la arquitectura de nuestro *Cloud Service.*

2.1 Búsqueda entre agentes

Tal como habíamos mencionado en apartados anteriores, la búsqueda de agentes en el sistema se hace de forma centralizada; el agente service registry actúa como servidor para los 3 componentes: el *frontend*, las *queues* y los *workers*.

Cuando un microservicio se pone en marcha, necesita conocer qué *queues* están disponibles en cada momento, ya que, en algunos casos podría aumentar. Para ello, al iniciarse se conecta el servidor de registro para obtener las ip de las *queues* que están funcionando en ese momento y poder conectarse a dichas *queues*. Puede darse el caso de que se levanten nuevas instancias de la *queue*, por lo que periódicamente se debe preguntar al servidor por la cantidad total y por si hace falta establecer una nueva conexión. En nuestro código esto se comprueba llamando a una función *heartbeats* cada 8 segundos.

En el caso de las queues sucede algo parecido, estos agentes tampoco conocen las direcciones del resto de *queues* por lo que será necesario que lo consulten con la autoridad central. Es decir, cada *queue* se registra en el service registry para que el resto de *queues* puedan encontrarla y comunicarse con entre ella en caso de ser necesario como veremos con más detalle en apartados siguientes.

2.2 Manejo de recursos o información

En este apartado vamos a definir un conjunto de situaciones que tienen un resultado en el estado del servicio:

- Si el agente frontend ha encargado la realización de un trabajo y por alguna razón dicho agente falla (shutdown) antes de obtener la respuesta de la queue entonces la solicitud del cliente se perderá y no será respondido.

- Si alguna queue sufre un cierre abrupto, entonces toda la pila de trabajos que almacena se perderá y, por tanto, la petición del cliente no será respondida.

- Si la queue envía un trabajo a un worker y este no le responde en menos de 2000ms entonces se considera que el worker ha sufrido un fallo y por tanto el trabajo vuelve a la pila de trabajos pendientes para intentarlo luego con otro worker, de esta manera garantizamos que el mensaje se intenta enviar a los consumidores *exactly once*.
  Para que se cumpla este patrón, debe garantizarse primero que las *queues* no fallan. Si se da esta condición, entonces, cada vez que el FrontEnd envíe trabajo, la *queue* que lo reciba intentará enviar la petición al *worker* que sepa que está disponible (las *queues* almacenan un array de *workers* disponibles por si se da el caso de que un worker pida trabajo, pero la *queue* no tenga peticiones pendientes, en tal caso deberá preguntar al resto de *queues* por trabajos y si ninguna tiene, pues debe almacenar la *id* del worker por si más adelante llegan peticiones). Una vez enviado el trabajo, la *queue* pone en

marcha un temporizador para que dentro de 2000ms pregunte al resto de *queues* si alguna ha recibido la respuesta del *worker*. Cada paquete que se envía al *worker* se le añade una *header* que consiste en un *timestamp* que identifica inequívocamente cada mensaje (el *timestamp* se genera marcando la fecha de envío la cual tiene una precisión del orden de nanosegundos). Cuando la *queue* que envió el paquete pregunta al resto, les envía también el header o *id* del paquete, para que, en caso de que el resto de colas lo reciban más tarde, este mensaje sea ignorado y por tanto no se dé el caso de un reenvío duplicado. Si alguna *queue* contesta que ya recibió la respuesta del *worker* y por tanto ya la puso a disposición del Front, entonces el envío se considera exitoso. Pero, si ninguna *queue* ha recibido el mensaje de respuesta del *worker*, entonces la *queue* que envió el mensaje y solo ella es la responsable de reenviar el mensaje a otro *worker*, el resto de *queues* ignoran cualquier respuesta posterior del *worker*.
Por todo ello, podemos decir que nuestro servicio cumple con el patrón de envió *exactly once*.

- Si un worker notifica su disponibilidad para recibir trabajo a una *queue* y dicha *queue* no tiene trabajo disponible para enviarle, entonces apilara la identidad del workers a la espera de un trabajo o a la espera de que otra *queue* que si tenga una petición se comunique con ella para poder enviarle esa petición al worker disponible.

2.3 Alta disponibilidad

Es uno de los puntos fuertes del FaaS, la alta disponibilidad está garantizada con la replicación de agentes. Para cumplir con este principio deben cumplirse las siguientes condiciones:

- **Replicación del agente Frontend:**

El agente Frontend debe orquestarse de forma replicada, es decir, se debe crear más de una instancia del microservicio para que la carga de peticiones se reparta. Si un agente Frontend falla, el servicio puede seguir disponible ya que habrá otros agentes que puedan recibir la petición.

- **Replicación del agente Queue:**

La replicación de las queues nos ayudan también en la alta disponibilidad, ya que mientras más queues repliquemos, más capacidad de almacenamiento de tareas podrá aceptar el servicio en su conjunto.

- **Replicación del agente Worker:**

Para el correcto funcionamiento del servicio es necesario la replicación del worker ya que si disponemos de pocos workers y un número reducido de queues el servicio podría colapsar a causa de una apilación masiva de tareas que no son tratadas con la velocidad que necesitan.

2.4 Tolerancia a fallos

Para que el sistema pueda funcionar correctamente, se asume que los siguientes agentes no pueden fallar:

1. ***El Service Resgistry***

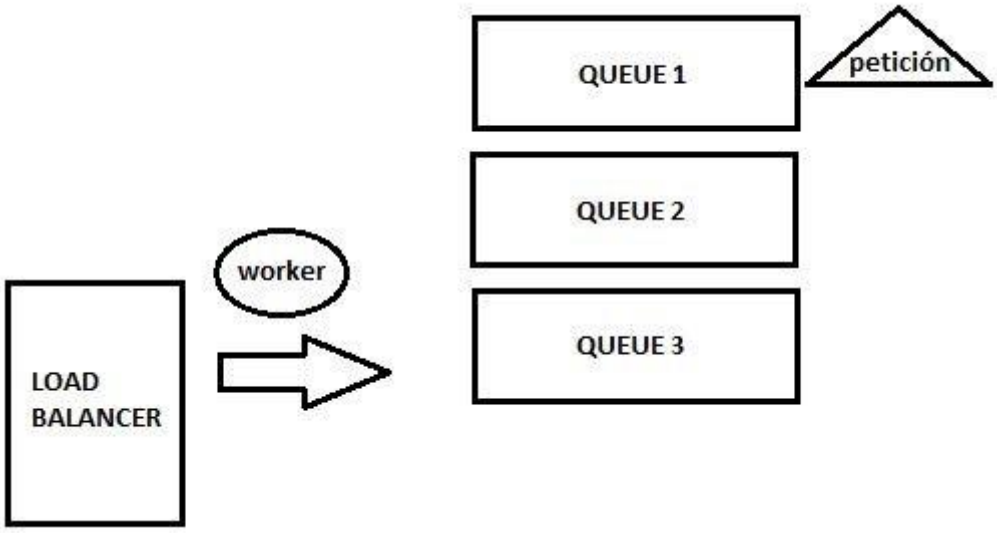
Es un elemento central en el descubrimiento de agentes, si el service registry sufre un fallo catastrófico que provoque el shutdown del servicio, entonces el FaaS dejará de funcionar.

2. ***Las queues***

Las queues no pueden tener un fallo catastrófico ya que, de haberlo, se perdería la información de trabajos almacenados en dicha queue.

3. Funcionamiento de colas replicadas

En este apartado, se va a explicar cómo se ha llevado a cabo uno de los grandes problemas que surgen a la hora de realizar este proyecto y es cómo realizar la comunicación entre distintas colas replicadas.



Para ponernos en contexto sobre el problema a resolver, imaginemos que la Queue 1 es quien mediante el balanceador de carga recibe la petición del frontend y, por otro lado, es la Queue 3 la que, a través del balanceador de carga recibe la información de que un worker que está disponible para trabajar. En esta situación tenemos un problema ya que la Queue 1 tiene una petición que debe ser resuelta, pero es la Queue 3 quien tiene la información de que hay un worker disponible para trabajar, por lo que el trabajo nunca llegaría a realizase debido a que al worker nunca se le asignaría ese trabajo.

Por este motivo es tan importante en nuestro sistema la comunicación entre colas. El primer aspecto a tener en cuenta es que en este proyecto estamos asumiendo que las colas no van a fallar nunca. Una vez tenemos esto en consideración, la solución a este problema se planteó de la siguiente manera:

En primer lugar, la cola que reciba la información de que hay un worker disponible para realizar un trabajo, deberá comprobar que no tiene ningún trabajo almacenado por realizar, y posteriormente si no tiene ningún trabajo disponible, apilará el worker.

En segundo lugar, cada cola que reciba una petición del frontend y no tenga ningún worker, se comunicará con el resto de las colas para comprobar si alguna cola tiene un worker reportado y si es así, le mandaríamos la petición a esa cola para que a su vez se la mande al worker con el que ha establecido la conexión.

Como se ha comentado en apartados anteriores, nuestro sistema utiliza el servicio consul que será el que utilizará cada cola para así conocer la identidad de las demás instancias de colas que hay disponibles y así poder establecer esta comunicación con ellas mediante el socket.

```
/*************************************/
//En esta sección se añaden las direcciones del resto del queues
//excluyendose a sí misma. Se comprueba de forma periodica ya que en
//cualquier momento podria añadirse una nueva queue
/*************************************/
setInterval(() => {
consul.agent.service.list(function(err, result){
    if (err) throw err;
    for(let i in result)
    {
        if(result[i].Service.startsWith(SERVICE_NAME) && (result[i].Address != HOST))
            queues.add(result[i].Address);
    }
});
}, registryInterval);
```

Otro aspecto a tener en cuenta es qué hacer si en el momento que un worker se reporta como disponible ninguna de las colas tenga peticiones apiladas.

En este caso, las colas pasarían a apilar en un vector la identidad del worker que está a la espera de poder realizar un trabajo. Pasados 3000ms, la *queue* que recibió la petición de trabajo del *worker* preguntará al resto de *queues* por trabajo pendiente (obviamente la *queue* comprobará primero que ella no haya recibido ya peticiones en ese intervalo). Con esto conseguimos que nuestro sistema no se quede "tonto" y con esto queremos decir que todas las peticiones que llegan van a ser resueltas siempre y cuando haya un worker disponible, y que, si hay un worker disponible y hay una petición por resolver, el worker siempre acabará realizando esta petición y no se quedará parado sin hacer (al menos no más de 3000ms).

```
73
74  /*************************************/
75  //En esta función queremos preguntar periodicamente
76  //si se da el caso de que tengamos workers apilados
77  //pero aún no tengamos trabajos pendientes.
78  //Se trata de no secuestrar workers que podrían estar
79  //trabajando.
80  /*************************************/
81
82  setInterval(() => {
83      if((workers.length > 0) && (who.length == 0)){
84          let queues_copy = [...queues];
85          askForJob(queues_copy, workers.shift());
86      }
87  }, askForJobInterval);
88
```

4. Sistema de comunicación

En este apartado quisiéramos centrarnos en cómo hacemos posible la comunicación entre los diferentes agentes, y cómo estas interacciones hacen variar los estados del sistema por lo eventos que les llegan.

- ***Comunicación Frontend-Queue***

En este camino de la comunicación establecemos una comunicación de mensajes siguiendo el patrón dealer-router.

En el Frontend creamos un socket tipo 'dealer' para hacer el envío de tareas a las queues, la conexión se hace estableciendo un connect a las direcciones de las colas que obtenemos gracias al service registry. Debido a la naturaleza del propio socket, los envíos de trabajos a las *queues* se hará de forma de forma rotatoria (round robin).

Por el lado de las queues, tenemos un socket tipo 'router', los cuales hace un bind para escuchar y responder peticiones que lleguen a su socket. Cuando llegue un mensaje, deberá revisar en primero si dispone de algún worker que esté disponible para que le pueda enviar el trabajo, si no, entonces apila el trabajo pendiente guardando el mensaje y la identidad del cliente que hizo la petición. Cuando el trabajo está realizado, el worker responde el mensaje adjuntado la identidad del cliente para que la queue pueda responder directamente al frontend que envió el mensaje.

- ***Comunicación Queue-Worker***

Para esta comunicación hemos decidido implementar el patrón router-req.

En el primer tipo de comunicación, el worker desea reportar su disponibilidad para asumir trabajo, por lo que se crea un socket tipo 'req' para que redireccione la petición a una queue al azar, en la parte de la queue, se dispone de un socket tipo 'router' en el que escucha los mensajes de disponibilidad del worker. Si tiene trabajo pendiente de enviar, entonces envía el trabajo y la IP de la propia queue por el mismo canal en el que el worker hizo la req.

- ***Comunicación Queue-Queue***

El funcionamiento de este camino de comunicación ya se explicó anteriormente en el apartado "4. Funcionamiento de las colas replicadas". Utilizamos un patrón tipo req-rep para realizar esta comunicación.

En resumen, tenemos que decir que hemos creado un canal de comunicación especialmente diseñado para la comunicación entre las queues. Recordemos que las queues necesitan estar comunicadas con el resto para que en caso de disponer de un worker libre y no tener trabajo pendiente de asignarle, pueda preguntarle al resto por trabajo para poder hacer uso del worker disponible, si no, entonces lo apila a la espera de recibir trabajo o volver a preguntar al resto dentro de 3000ms.

5. Tolerancia a fallos en el componente worker

En nuestro sistema, se asume que las colas no van a fallar nunca, pero sí se tiene en cuenta un fallo de los workers a la hora de realizar el trabajo. Con esto, lo que queremos conseguir es que no haya ninguna petición del front que se quede sin contestar. Para esto hemos implementado distintas funciones dentro del componente Queue.

En primer lugar, cuando una cola envía un trabajo a un worker, se añade un timestamp preciso, del orden de nanosegundos al mensaje original para que este valor nos sirva como identificador único del mensaje.

Como sabemos, aunque sea la queue1 la que le haya enviado un trabajo x al worker1, la respuesta que dará este no tendrá que ser obligatoriamente la queue1 ya que realiza (RR), por lo que hemos diseñado un sistema para que las colas que envíen un trabajo a un workers, sean capaces de asegurar que el trabajo se ha realizado o, por el contrario, si el worker ha fallado, reenviar el trabajo a otro worker.

Dentro de cada cola, tendremos un vector de trabajos respondidos donde iremos apilando el identificador único de los trabajos a los que nos haya respondido un worker. Por tanto, si la queue1 es la que manda el trabajo al worker y es la queue2 la que ha recibido la respuesta, será esta quien en el vector de respondidos tendrá la identificación de la petición que realizó la queue1 que a su vez tendrá otro vector con la identificación de los trabajos que ha enviado.

```
/************************************************/
/*2 segundos despues del envio, es responsabilidad de la queue emisora
/*preguntar al resto si alguna ha recibido y por tanto reenviado ya la respuesta
/*del worker, si ninguna ha recibido la respuesta en 2s desde envio,
/*suponemos que ha fallo y por tanto hay que reenviar
/************************************************/
function askForJobCompleted(freeQueues, result){
    if(!responded.includes(result.header)){         //Comprobamos primero que no
        if(freeQueues.length == 0){
            var aux = {};
            aux.inf = result.inf;
            aux.sup = result.sup;
            aux.ite = result.ite;
            var msg = JSON.stringify(aux);
            sendToWorker(result.idusu, msg);        //En caso de que ninguna queu
            return;
        }
        else{
            let qqReq = zmq.socket('req'); // socket de comunicacion entre colas
            qqReq.connect('tcp://'+freeQueues[0]+':'+PORT2);
            var x = result.header.toString();
            qqReq.send(['job_responded', x]);       //Le preguntamos al resto de
            qqReq.on('message', (msg, sep) =>{
                if(msg == 'NO'){
                    freeQueues.shift();
                    askForJobCompleted(freeQueues, result); //Si no han enviado e
                }
            });
        }
    }
}
```

Aquí podemos observar cómo dentro de la cola tenemos una función que se encarga de verificar si un trabajo ha sido respondido. Lo primero de todo, comprobamos que no sea la propia cola la que haya respondido. Si la respuesta no ha llegado a ninguna cola, entonces asumimos que el worker ha fallado y realizamos un reenvío del trabajo. Preguntamos a todas las colas disponibles por el identificador de nuestro trabajo para comprobar si a alguna le ha llegado una respuesta con dicho identificador.

Esta función se ejecuta dos segundos después de que una cola le haya enviado un trabajo a un worker.

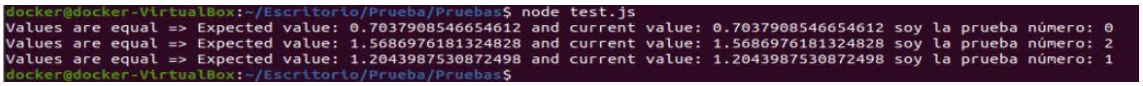
Otro aspecto importante, es que si por lo que fuera el worker respondiera 2000ms después, como nosotros hemos asumido que ese worker ha fallado, descartaríamos esa respuesta ya que ya habríamos realizado el reenvió a otro worker y seguramente esa petición ya haya sido respondida al frontend. Para ello contamos con una variable *blacklist* en cada cola donde incluimos los identificadores de las peticiones que hemos asumido que han fallado para ignorar su respuesta.

6. Pruebas realizadas

Como en todo proyecto software, para comprobar el correcto funcionamiento de todos los componentes del sistema se deben realizar varios tipos de pruebas para poder confirmar que el sistema actúa como debería.

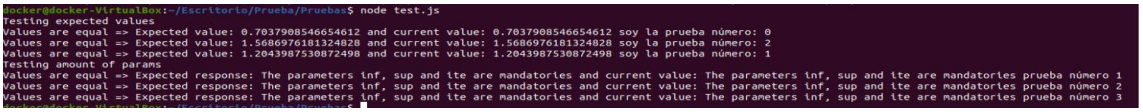
En nuestro caso, las pruebas realizadas mediante código han sido las siguientes:

En primer lugar, como nuestro sistema se basa en realizar el cálculo de un integral dado unos valores, se realizaron pruebas para que dados unos datos de entrada devuelva el valor correcto de la integral que previamente nosotros habíamos calculado para poder realizar este test. Todos los resultados fueron positivos y siempre devolvía el valor correcto para cada petición.



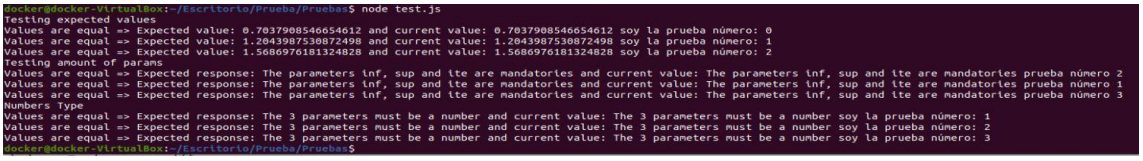
Como podemos observar, la ejecución del código para esta prueba es satisfactoria, el sistema nos devuelve el valor esperado de la integral dados unos parámetros de entrada.

En segundo lugar, nuestro sistema tiene que ser capaz de controlar la cantidad de parámetros de entrada que acepta, ya que el sistema no funcionaría si algún parámetro necesario no lo tiene o si tiene algún parámetro de más que no conoce, por lo que al ejecutar una petición se comprueba que tenemos los parámetros que exactamente necesitamos.



Al igual que en el anterior caso, ejecutando la prueba, obtenemos la respuesta esperada que nos indica que los tres parámetros de entrada tienen que ser inf, sup y ite. En este caso hicimos las pruebas lanzando peticiones sin alguno de los tres parámetros de entrada.

Por último, es importante que a parte de comprobar que tenemos el número de parámetros de entrada correctos, que estos sean del tipo que le corresponde, en este caso, serían valores numéricos.



Aquí podemos observar que la última salida de la prueba, donde le dimos valores de entradas no numéricos para obtener la salida esperada del sistema.

7. Conclusiones y trabajo futuro

Una vez finalizado nuestro proyecto, podemos destacar que la tarea que consideramos ha sido la más importante y a su vez la más difícil de desarrollar. El principal problema a la hora de desarrollar nuestro Cloud Service era tener en cuenta el funcionamiento de las colas replicadas. Para resolver este problema se tuvieron en cuenta muchos aspectos, pero todas las soluciones que se plantearon fueron asumiendo que las colas nunca fallaban. Surgieron problemas debido a la comunicación con el Worker de la cola y la propia comunicación entre colas replicadas que se resolvieron como hemos indicado en apartados anteriores.

Una vez hemos acabado la implementación, podemos concluir que se ha conseguido un sistema funcional que funciona a través de la interacción de unos microservicios replicados (worker, queue, frontend) que se comunican a través de balanceadores de carga (Haproxy y zmq) los cuales consiguen su propósito utilizando como resolvedor de direcciones nuestro Service Registry (Consul).

Cuando una petición llega a través del primer balanceador de carga al frontend, este se encarga de reenviar esta petición por el siguiente balanceador de carga a una cola. Esta cola se encarga de almacenar el trabajo y esperar a que se reporte un worker como disponible. Una vez tiene la identidad de un worker que está listo para trabajar, le envía la petición y recibe de este la respuesta, por último, le reenvía al frontend el resultado de su petición. En nuestro caso, el sistema se encarga de calcular la integral dado unos parámetros de entrada como se ha mostrado anteriormente.

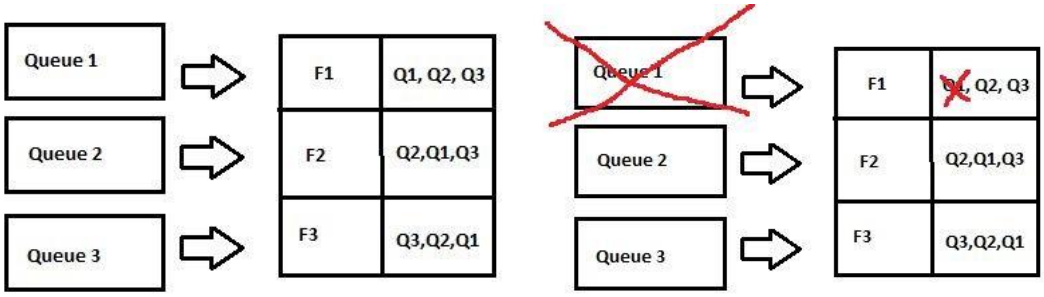
Ahora bien, hemos conseguido tener un sistema funcional, pero ¿Cómo podríamos mejorar este sistema? La verdad es que hay muchos frentes de mejoras abiertos y por eso a continuación vamos a mostrar los que consideramos más importantes.

En primer lugar, una de las vías de mejora que se nos abren a la hora de seguir con el desarrollo de nuestro Cloud Service es la de tener más de un Service Registry en el sistema. Como ya sabemos, el Service Registry tiene en nuestro sistema un papel fundamental, al resolver y almacenar para su consulta todas las direcciones de nuestros microservicios activos en el sistema, por lo que, ¿Qué pasaría si este componente nos falla? Lo cierto es que el sistema no podría seguir funcionando ya que los balanceadores de carga no podrían seguir realizando su trabajo al no conocer las direcciones ni las colas podrían comunicarse entre sí al no conocer la identidad de otras colas. Por tanto, se ha pensado que este componente debería estar replicado, para así tener una mayor tolerancia a fallos en caso de que uno de los Service Registry falle por cualquier motivo y nuestro sistema sea capaz de seguir funcionando por tanto también tendríamos una mayor disponibilidad.

Por otro lado, de una manera muy parecida a la del Service Registry, deberíamos tener en cuenta que el balanceador de carga también podría fallar. Por tanto, la idea es replicar también los balanceadores de carga. En este caso, no serviría tener x replicas activas de los balanceadores de carga en el sistema pues eso no funcionaría, debemos tener réplicas de estos balanceadores de carga que sólo se activen si el balanceador de carga que está activo al inicio falla. Una vez el

que esté activo falla, entonces las peticiones dejarán de pasar por ese primer balanceador de carga y pasarán por el otro, quien asumirá el rol de balanceador de carga principal.

En tercer lugar, pensando en realizar un sistema más real, deberíamos dejar de asumir que las colas no pueden fallar nunca como se ha hecho en este proyecto. Para resolver este problema que nos puede surgir deberíamos de cambiar gran parte de la lógica de las colas. En primer lugar, ahora mismo nuestro sistema está diseñado para que una petición que llegue a través del frontend sólo se almacene en una cola, por lo que, si esa cola fallara, ¿Perderíamos la petición y se quedaría sin responder para siempre? La respuesta es que en nuestro sistema actual sí se perdería para siempre. La solución a esto podría ser que en cada cola tuviéramos un vector de trabajos, con esto nos referimos a que si tenemos en un momento dado dos colas disponibles y la primera cola recibe un trabajo a través del balanceador de carga, está se almacene el trabajo y a parte le comunique a la otra cola que hay un trabajo a realizar para que también se lo almacene. Una vez que ese trabajo sea realizado por un worker, la cola que haya recibido la respuesta deberá comunicarle a la otra que esa petición ha sido resuelta por lo que las dos la eliminarían del vector. Dentro de este vector del que estamos hablando, para lograr que un trabajo no se realice dos veces por dos workers que se conecten a dos colas diferentes, ya que hemos comentado que todas las colas tendrán los mismos trabajos almacenados, por tanto, para resolver esto, dentro del vector aparte de guardarnos la petición, guardaríamos el orden de responsables de dicha petición. Para que se entienda, por ejemplo, el trabajo 1 tiene asignado como primer responsable la cola 1 para responderle a la petición y como subresponsables las demás colas activas, por lo que las colas sólo servirían a los workers los trabajos de los cuales son responsables. Para que todo esto tenga sentido, las colas periódicamente tienen que saber sepan que instancias de colas a parte de ella misma están activas para así revisar la cola de trabajos y comprobar que el responsable del trabajo sigue activo, para en caso contrario, asignarle ese trabajo a un subresponsable para que ese trabajo no se quede sin resolver.



Aquí tenemos ejemplificado como funcionaría, las 3 colas tendrían el vector donde cada petición (F1, F2, F3) tendría asignado un orden de responsables (Q1, Q2, Q3) para resolver la petición. Una vez, por ejemplo, la cola 1 se cae, al trabajo que tenía como primer responsable la cola1, en este caso F1, se le asigna su siguiente responsable.

Para acabar, tal y como se comentaba en la actividad, también podríamos tener workers que estén especializados para realizar un trabajo en concreto para que, en resumidas cuentas, ampliar las funcionalidades que ofrece nuestro Cloud Service. También conseguiríamos un proyecto mucho más elástico si cada vez que las colas detecten que tienen mucho trabajo almacenado durante x segundos y no hay workers suficientes para satisfacer a todas las peticiones en un tiempo razonable, el sistema sea quien levante más instancias de workers para realizar el trabajo en un menor tiempo.