UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Integración de aplicaciones

Laboratorio 3

Contenido

1. Introducción

En este laboratorio pondremos en práctica algunos de los conceptos estudiados en teoría sobre sistemas de mensajería. La idea es identificar aquellas operaciones que por sus especiales condiciones sean susceptibles de ser convertidas en asíncronas, extraerlas de la API RESTful y servirlas por medio de paso de mensajes.

2. La API

Como ya se ha visto en clase, las comunicaciones por paso de mensajes poseen ciertas ventajas indiscutibles, pero también conllevan ciertas complicaciones nada desdeñables, de manera que en general no es recomendable portar cualquier tipo de API a comunicación por medio de mensajes.

En particular, las operaciones más apropiadas para adoptar este estilo de comunicación son:

- Operaciones solicitadas con mucha frecuencia, que pueden ser procesadas por un número variable de consumidores, en función de la carga de trabajo.
- Operaciones costosas, que pueden ser procesadas a distintos ritmos en función de la carga de trabajo.
- Operaciones de actualización, que no requieran obtener respuesta inmediata.

Si analizamos la API de nuestros servicios podemos identificar algunos candidatos perfectos:

- POST /twitter/users: crea un usuario en el sistema. Esta operación es potencialmente costosa, ya que podría requerir efectuar comprobaciones y notificaciones varias, incluso enviando un e-mail al usuario recién creado.
- PUT /twitter/users/XXX: actualiza un usuario. Es una operación que no requiere un feedback inmediato, de manera que podría implementarse fácilmente como una operación asíncrona.
- DELETE /twitter/users/XXX: elimina un usuario del sistema. Puede ser una operación costosa, si además del usuario es necesario eliminar todos sus tweets y retweets basados en éste.
- POST /twitter/users/XXX/following: añade un usuario a la lista de usuarios seguidos. Podría resultar una operación costosa y no requiere feedback inmediato.
- DELETE /twitter/users/XXX/following: elimina un usuario de la lista de usuarios seguidos. Podría resultar una operación costosa y no requiere feedback inmediato.
- POST /twitter/tweets: añade un tweet. Es una operación muy frecuente, y puede recibir un número muy alto de peticiones concurrentes.
- POST /twitter/tweets/XXX/retweets: añade un retweet. Al igual que en el caso anterior, es una operación muy frecuente.
- POST /twitter/tweets/XXX/likes: añade un like a un tweet. Es una operación muy frecuente, y puede recibir un número muy alto de peticiones concurrentes.
- POST /twitter/tweets/XXX/dislikes: añade un dislike a un tweet. Es una operación muy frecuente, y puede recibir un número muy alto de peticiones concurrentes.

En las siguientes secciones estudiaremos cómo implementar mensajería en nuestro servicio utilizando ZeroMQ. Pero primero será necesario determinar cómo va a cambiar nuestra API para acomodar estas nuevas operaciones por paso de mensajes. Para ello debemos definir el formato de mensaje que vamos a utilizar. Adoptaremos las siguientes convenciones:

- Un mensaje será un objeto en formato JSON.
- El mensaje dispondrá de un campo *.type* de tipo *String* que determinará el tipo de operación solicitada. Podemos identificar los siguientes tipos de mensajes: “addUser”, “updateUser”, “removeUser”, “addFollowing”, “removeFollowing”, “addTweet”, “addRetweet”, “addLike”, “addDislike”.
- Cualquier parámetro necesario para solicitar la operación será un campo adicional del mensaje. Por ejemplo, el token será almacenado en el campo *.token* del mensaje. En el

caso de eliminar un usuario, el identificador del usuario será almacenado en el campo *.id*, etc.
- Si la operación requiere datos, serán almacenados en el campo *.data*. Por ejemplo, en el caso de crear un nuevo usuario, los datos del usuario serán almacenados en el campo *.data*.

A continuación se muestra un ejemplo de mensaje solicitando la creación de un nuevo usuario:

{“type”: “addUser”,
“data”: {“name”:”pepe”,”surname”:”pepe”,”email”:”pepe@upv.es”,”nick”:”pepe”,”password”:”xxx”}}

A continuación se muestra un ejemplo de un mensaje solicitando la actualización de un usuario:

{“type”: “addUser”,“token”:”yyy”,”id”:”xxx”,
“data”: {“name”:”pepe”,”surname”:”pepe”,”email”:”pepe@upv.es”,”nick”:”pepe”,”password”:”xxx”}}

3. Mensajería con ZeroMQ

En esta sección integraremos en nuestro servicio mensajería por medio de ZeroMQ. El primer paso consiste en instalar la librería *zeromq@5*.

> npm install zeromq@5

Comenzaremos con las operaciones relacionadas con los usuarios. En este caso, debemos de migrar a mensajería las siguientes operaciones:
- addUser
- updateUser
- removeUser
- addFollowing
- removeFollowing

Los cambios más relevantes se llevarán a cabo en el módulo *server.js*, en el que se implementa la API RESTful. En este mismo módulo añadiremos un socket ZeroMQ que atenderá las operaciones identificadas anteriormente. Para ello, primero crearemos el socket. Haremos que escuche por ejemplo en 1000 puertos más arriba que el servicio RESTful.

```
var model = require('./model_mongo');
var express = require('express');
var bodyParser = require('body-parser');
var app = express();
const zmq = require('zeromq');
const sock = zmq.socket('pull');

sock.bind("tcp://127.0.0.1:9090", (err) => {
    if (err) console.log(err.stack);
    else console.log('Listening async on 9090');
});
```

A continuación recibiremos mensajes y los procesaremos, de acuerdo a su tipo. La estructura del código sería similar a la siguiente:

```
sock.on('message', (msg) => {
    console.log('Received message: ' + msg.toString());
    msg = JSON.parse(msg.toString());
    switch (msg.type) {
      case 'addUser': ...
      case 'updateUser': ...
      case 'removeUser': ...
      case 'addFollowing': ...
```

```
      case 'removeFollowing': ...
      default: ...
    }
});
```

El siguiente paso consiste en ir reemplazando paulatinamente las operaciones que ahora mismo sirve el servicio RESTful por sus contrapartidas orientadas a mensajes. Proporcionamos los ejemplos de las operaciones “addUser” y “updateUser”.

```
sock.on('message', (msg) => {
    console.log('Received message: ' + msg.toString());
    msg = JSON.parse(msg.toString());
    switch (msg.type) {
      case 'addUser':
            console.log('add user ' + JSON.stringify(msg.data));
            model.addUser(msg.data, (err, user) => {
                if (err) console.log('add user ERROR: ' + err.stack);
                else console.log('add user SUCCESS');
            });
            break;
        case 'updateUser':
            console.log('update user ' + JSON.stringify(msg.data));
            model.updateUser(msg.token, req.data, (err, user) => {
                if (err) console.log('uppdate user ERROR: ' + err.stack);
                else console.log('update user SUCCESS');
            });
            break;
      case 'updateUser': ...
      case 'removeUser': ...
      case 'addFollowing': ...
      case 'removeFollowing': ...
      default: ...
    }
});
```

Este mismo proceso deberá seguirse con el resto de operaciones del servicio.

4. El cliente

Al modificar la API, deberemos de modificar convenientemente cualquier consumidor de dicha API. Por lo tanto, deberemos modificar la librería *model_rest.js,* eliminando las operaciones que antes se solicitaban a través de una interfaz RESTful y que ahora serán solicitadas con mensajería. Además, incluiremos un nuevo módulo *model_mq.js*, que incluirá la implementación de estas operaciones, que básicamente utilizarán el paquete *zeromq* para comunicarse con el servicio. Esta nueva librería *model_mq.js* deberá ser importada desde cli.js.

El resultado final se muestra en la siguiente figura.