DOS-Project part-2-

Adding Replication, Load Balancing and Caching

Distributed Operating Systems-10636456.

Haya Mikkawi
11819675
6 May 2022

Overall Program:

In this part of the project, we were asked to modify part 1 to implement: replication, load balancing and caching.

The system consists of:

1- Frontend server that is now not only responsible for forwarding requests to the other two servers, but also, in this version it implements load balancing: for this purpose I implemented round-robin algorithm.

It also has an in-memory cache to enhance the performance, for this I used a hash map for caching requests.

2- Two replicas of both Catalog server and order server, for this purpose, I ran 2 VMs, on each one I run the two servers and the database which I chose to be mongoDB, this is for replication, as for consistency, I used a push based method, in which any of the catalog servers notify the other whenever a patch request for (quantity or cost) is received or whenever a book is purchased so that the other server make the changes to its database, too and this way both servers are always up to date.

Also, whenever a change happen at any of the servers, it sends an invalidate request to the frontend server so that the frontend server removes the modified book from its cache in case it exists.

Used technologies: Node.js & express, mongoDB, Parallels (for the VMs) and each VM had Ubuntu running on it.

How It works:

The system works as the following:

1- The client only contacts with the frontend server and doesn't know about anything behind the scenes.

2- When the frontend server receives any get request, it first checks if it has the required information in the cache, if yes, then it doesn't need to contact

with any remote server. Else, it contacts one of the two replicas of the catalog servers and it chooses which server to contact using round robin algorithm.

3- When the frontend server receives a patch request, it forwards it to one of the replicas of the catalog server and it chooses which server to contact using round robin algorithm.

4- When the frontend server receives a request of purchasing a book, it forwards it to one of the replicas of the order server and it chooses which server to contact using round robin algorithm.

5- When any of the catalog server replicas receive a get request, it returns the result from its database.

6- When any of the catalog server replicas receive a patch request (either from the frontend server or from the order server when a book is purchased), it makes the required changes to the database and then sends two requests:

a. A request to notify the other replica to make the same changes.
b. A request to notify the frowned server to remove this book from the cache if it exists.

7- When any of the replicas of the catalog servers receives a notification message from the other replica, it applies the required changes.

Trade-Offs:

Like any microservices application:

The deployment part for each service independently was easy to do. Also, the client knows nothing about what happens behind the scene and only communicates with one server which is the frontend server.

However, there may be an overhead due to the communication between servers, moreover, the frontend server can become a bottleneck to the system.

Replication is good for availability and reliability, but it increased the communication overhead between the replicas.

Caching is a very good technique in terms of performance, I chose in memory caching (hash map) and not to have a separate caching server to reduce the communication overhead.

Using round-robin algorithm for choosing which replica to contact has the following trade-off: it's simpler than any other algorithm but other algorithms may give better results.

Possible Improvements:

All of the project's requirements were implemented and tested, but here some possible improvements: Better level of consistency, as I only used messaging between replicas for the simplicity and because I only have 2 replicas, however in a real system, it may be more complex, by using acknowledgments, timers, ...etc.

How to run the program:

1- You need to have a tool like Postman to send requests.

2- On your machine you have to install Node.js and npm, then you can run the frontend server code after running (npm install) to install all of the packages needed, the fronted server will run on port: 3002

3-You have to run two VMs, On both of them: you need to install MongoDB and get it running on port 27017, and fill it with books with the attributes mentioned in the requirements of the project, you'll also have to install Node.js, npm.

You may need to give the virtual machines the same IP addresses as mine: 10.211.55.3 and 10.211.55.4 otherwise, you'll need to change the IP address used in the code. Then you can run my catalog server code and my order server code (also after running npm install to install the packages needed). The catalog server will run on port 3000 and the order server will run on port 3001.

You need to set the following on your environment, or you can create a development environment name it(dev.env) and put it inside a directory called config and set the following attributes:

```
1 PORT=3000
2 REP_IP="10.211.55.4"
3 FRONT="192.168.1.17"
```

dev.env on Catalog server on VM 1 that has IP: 10.211.55.3

```
1 PORT=3001
2 REP_IP="10.211.55.4"
3 FRONT="192.168.1.17"
```

dev.env on order server on VM1 that has IP: 10.211.55.3

```
1 PORT=3000
2 REP_IP="10.211.55.3"
3 FRONT="192.168.1.17"
```

dev.env on Catalog server on VM 2 that has IP: 10.211.55.4

```
1 PORT=3001
2 REP_IP="10.211.55.3"
3 FRONT="192.168.1.17"
```

dev.env on order server on VM 2 that has IP: 10.211.55.4

**** Please note you need to change FRONT into your local machine IP address.**

4- Now you can run rpm run start for both servers on both replicas.

5- Now, you can start sending requests to localhost:3002