Second Project Report

Kushal Samir Mehta | Shree Hari

The aim of this project is to provide load balancing and implement an efficient service discovery algorithm between multiple clients and servers.
Our proposed architecture included, clients, a registry that maintains the locations of the servers (dynamically keeps track of the load count on every server), a copy of the registry and finally the servers.

Initial Architecture

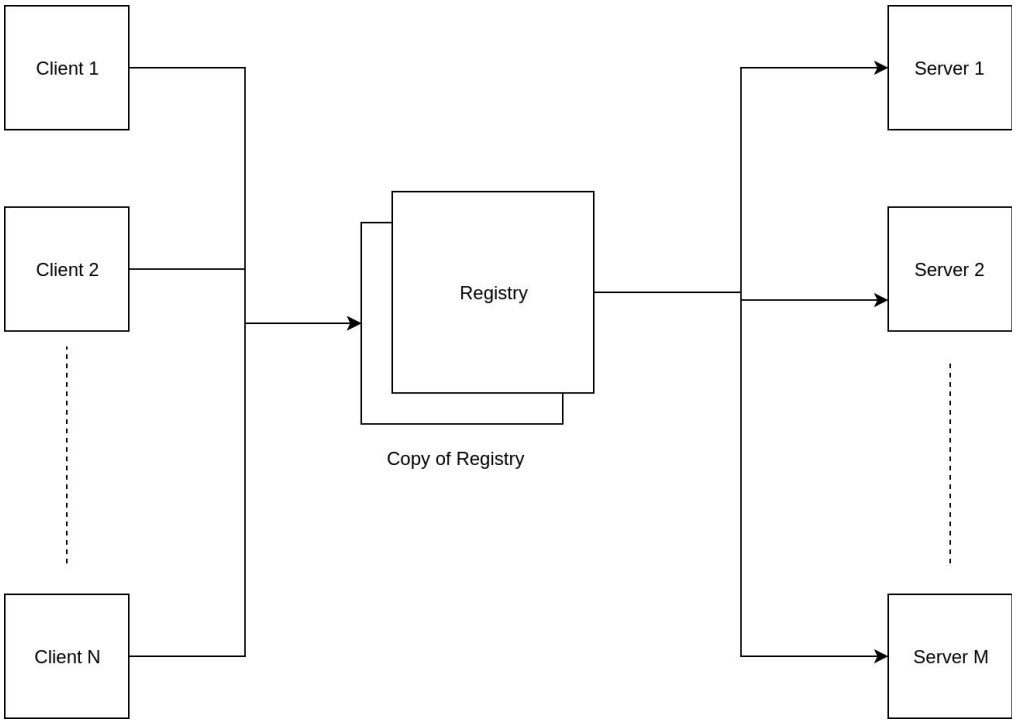

Figure 1. **The proposed architecture**. Client 1 requests for a service, addition of 2 numbers. Server 1 and Server 'M' are able to provide this service. The registry containing the list of servers and the load count would direct the request to the server that has the least load count and can provide the service.

Updated Architecture

Technologies and Frameworks used

- Spring Boot - For SOAP Service
- Java
- NodeJS - For Registry
- Python - For Registry updation and as a testing platform
- Postman - For documentation and testing the services

The updated architecture includes

- Multiple clients
- Registry server
- Registry JSON file
- Heartbeat checker
- Four servers running on four different ports

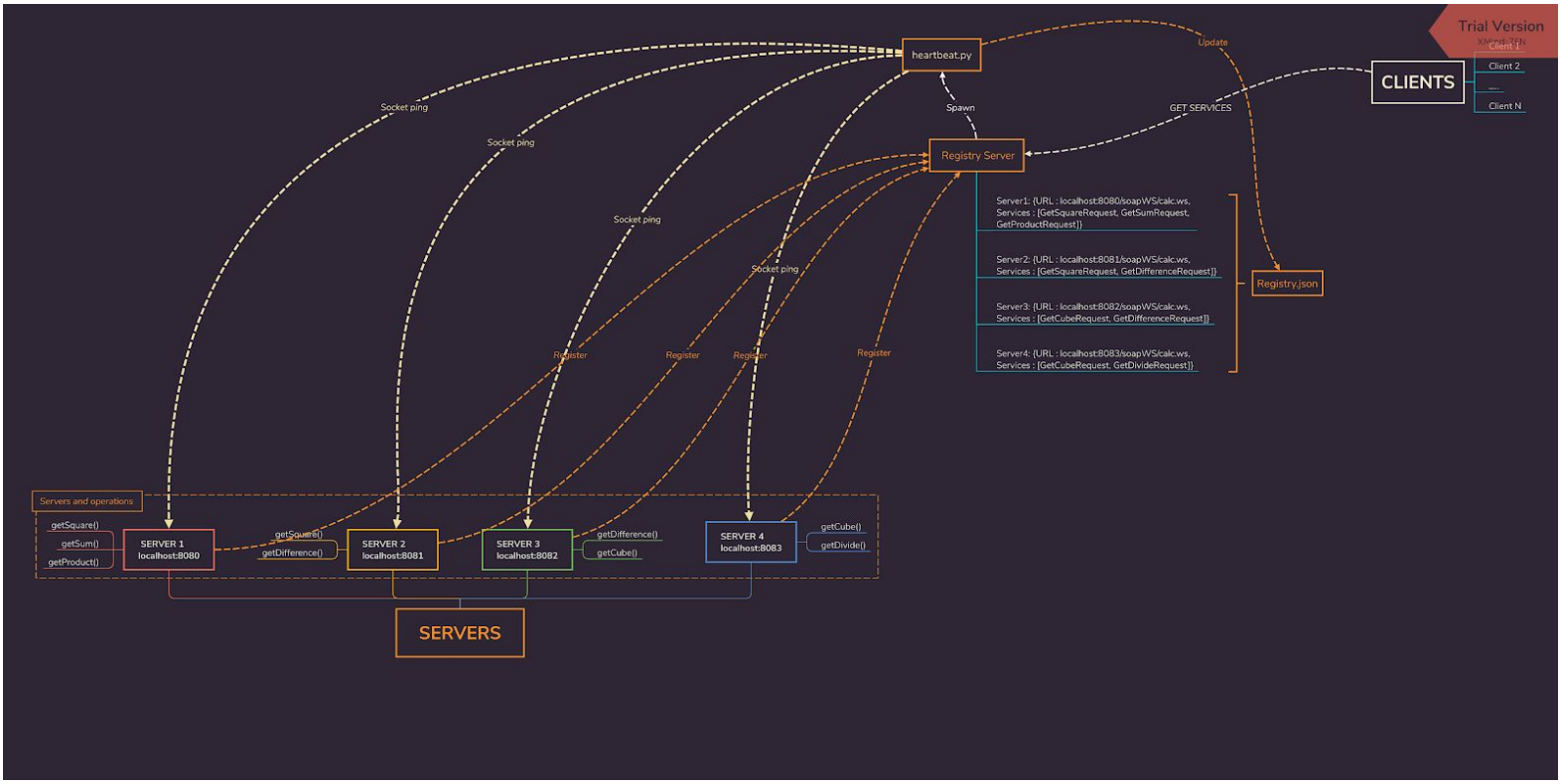

Figure 2. Updated Architecture. (Please refer Architecture.png for a better resolution)

MODULES

Servers

1. There are 4 servers that run on localhost but on different ports. i.e on ports 8080, 8081, 8082, 8083. Doing this simulates running different servers on physically different nodes.
2. Each server runs a SOAP service. The services are created using the Spring Boot Framework. This framework has an embedded Tomcat 7 server on which the server runs.
3. The services are distributed among the four servers such that, there is overlapping of some of the services among two different servers. The intention of this implementation was to make service availability possible. So, if one of the server fails the request can be directed to another server that provides the same service.
4. Each server sends a POST request to the registry server that is running at port 3000. This lets the registry know that the server is alive and adds its entry to the registry.

Registry

1. This is a server that runs on port 3000.
2. It exposes a REST API for both the servers and the potential clients. The documentation provided at the end of the document explains all the services and the underlying operations for the same.
3. The registry server listens for all the requests from the servers and adds their information to the registry.json file. The information includes the URL of the WSDL for each server and the list of services provided by them.
4. It also spawns a python process called ‘heartbeat’ which checks if the servers that provide the server is alive or not.
5. Some of the endpoints that the registry exposes are /getServices , /add, /subtract, /multiply, etc. These are hit by the client side in order to get the list of services that provide the specified operation or to retrieve the WSDL that will be used to generate the client stub to call the SOAP operations.

Heartbeat

1. This is a python process that is used to check if the servers are up and running every 1 second.
2. It accesses the registry.json file which stores the information for each of the servers and tries to establish a socket connection to them.
3. If the socket connections receives a ‘0’ as response (Server running) then it closes the connection and allows other servers to be pinged.

4. If it receives a ‘111’ as a response then it deletes the respective server information from registry.json.

Client

Multiple clients can query the registry using the endpoints that it exposes. The documentation provided below explain the various endpoints. We assume that the client is aware of the location of the registry (localhost:3000).

Testing Strategies

Our architecture includes a copy of the main registry, that can handle the requests if the main registry fail. Let’s assume that the main registry failed, then the client can refer to the copy registry. This client request redirection to the copy registry, on failure of the main registry is handled by the load balancer. By this approach our architecture avoids the single point of failure. We also assume that the architecture does not face multiple failures at the same time.

We plan to distribute the services across the servers in such a way, that every service is at least at two servers. By this implementation we can test for the service discovery and also in load balancing.

Let’s assume that server 1 and server 3 provides the service ‘addition’. Server 1 has a load of 300 and Server 3 has a load of 100. Then the client request is redirected to the server 3. So we plan to pre-load the servers with certain load for testing purposes.

PENDING MODULES

The load balancer will be implemented according to the timeline provided. We plan to implement this like the registry. The information stored in the registry will be updated to include the load of the server. Using this information, the load balancer can perform a simple sort on the loads and select the appropriate server that will be selected to serve the request from the client. Notional loads will be added to the servers as a part of the services that the SOAP servers already offer. For example, the add operation can stall the server for 3 second by adding a sleep. Similarly, the difference operation can stall for 5 seconds, product can take 8 seconds, etc.

This can be tested by overloading a particular server with multiple requests to cause bottlenecks at some targeted servers. Generating multiple clients that can send random requests in a loop is one way to implementing this test case. Another way is to write a test script in postman and observe how many test cases passed.

Registry API Documentation

1. **GET /getServices**
   Returns the JSON data stored in registry.json. This encapsulates all the information about all the servers in JSON format. Refer Archtiecture.png for more information about the data that is stored in the json file.
2. **GET /soapWS/add**
   Returns the information of the servers that provide addition as a service.
3. **GET /soapWS/subtract**
   Returns the information of the servers that provide subtraction as a service.
4. **GET /soapWS/multiply**
   Returns the information of the servers that provide multiplication as a service.
5. **GET /soapWS/divide**
   Returns the information of the servers that provide division as a service.
6. **GET /soapWS/square**
   Returns the information of the servers that provide squaring as a service.
7. **GET /soapWS/cube**
   Returns the information of the servers that provide cubing as a service.
8. **POST /registerService**
   Allows a server to send its information in JSON format to be registered with the registry in registry.json.

References

1. https://www.xmind.net/
2. http://spring.io/projects/spring-boot
3. https://nodejs.org/en/
4. https://youtu.be/mr_2-AWYCoc
5. http://www.springboottutorial.com/creating-soap-web-service-with-spring-boot-web-services-starter