Distributed Web Service Framework

Naveen Yamparala, Akash Reddy Gurram
Computer Science and Electrical Engineering
University of Maryland, Baltimore County
rs09174, hr69338@umbc.edu
December 14, 2018

***Abstract*—There are many web services available in the present world. Discovery of these services for the relevant web service among others is a challenge. Another challenge is the balancing of load among all the servers. In this paper, A distributed web service framework is provided which intends to discover the web services and provide an efficient way of balancing the load among the servers. This is being done in such a way that the single points of failures are minimal which gives it the essence of a distributed system. In this paper, the communication among the clients and different servers is done through SOAP protocol. As a result, the best-suited web service can be discovered and routed to the client. Our experimentation shows that the framework is effective for both service discovery and load balancing.**

Keywords*—web services, service discovery, load balancing, distributed systems, simple object access protocol(soap), servers, clients.

I. Introduction

A web service is a service using which communication among different applications is possible. A web service generally utilizes

- XML to tag the data
- SOAP to transfer the message
- WSDL (Web Service Description Language) to describe the availability of a service

A distributed system allows these web services to be accessed from different locations or from different machines. Moreover, a distributed system is resilient to failures or crashes which is not a characteristic of centralized systems. A distributed web service framework is a combination of the above ideas where the web services and provided to a client in a distributed manner. Service discovery is important because it tells the client about all the available services on a web server. It is an automatic process. Load balancer chooses the best available server from all the servers. The server with the least load is deemed the best server for a job. Broadly, there are two ways in which the communication between a client and server is possible. They are using the RESTFul API and the SOAP protocol. A WSDL provides a description of the web services like the parameter type etc. using which a client can communicate with the server. The generation of WSDL for web services will be possible using the SOAP protocol.

II. Design

The design of our distributed system is as in Figure 1 below. There are multiple web servers that provide web services to clients. Clients first discover the services that are available with the help of Service Discovery server.Service Discovery server finds the list of web servers containing the required web service and forwards them to load balancer.Load balancer returns the least loaded web server to the service discovery server. The latter forwards it to the client.

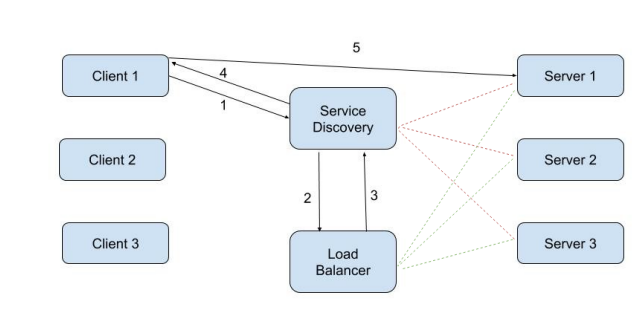

Figure 1. Architecture

The client now contacts the specified web server for the web service. For this to happen, both the Service Discovery server and the Load Balancer must know about the state of the web servers. Any changes in the web servers like the services being offered or the load of that server etc. must be immediately reflected in the load balancer and service discovery. We could have made a design where the load balancer directly forwards the request to the web servers and the request is serviced. But this has a drawback. There is more burden on the service discovery and the load balancer. They have to keep track of all the requests even if it means waiting for a long time. So, we chose the design where the request is forwarded as shown in Figure 1. Using this, service discovery server and load balancer need not keep track of all the requests until they are serviced.

A. Service Discovery

Service discovery is done with a server called Service Discovery Server. In our design, this server runs continuously. For every ten seconds, it pings all the web servers to get the required information. If a server leaves the distributed system or crashes abruptly, the service discovery server updates this information in the very next iteration. By running it every ten seconds, the information about the services and the servers will always be the latest. Apart from the information of the available servers, it also stores the information of all the web services that are being provided. The service discovery server communicates with the load balancer in parallel with other servers. Every request from the service discovery will

be forwarded to the load balancer and the response from load balancer will be sent back to the client via the service discovery.

B. Load Balancer

The main agenda of a load balancer is to split the jobs in such a way that the least loaded site will be always chosen for the client's request. We have used the notion of CPU load for load balancing. The machine on which the server is running will have a CPU load associated with it. We assume that there will be only one server on a machine. In choosing the right server, the load balancer checks the CPU load of the machine on which the server is running. Then, the web server with the least loaded machine will be chosen. If a server is servicing a request, its machine load will obviously be high compared to the machine on which a server without any activity. Once the least loaded server is chosen, the server information is forwarded to the service discovery server. The service discovery server now sends the server information to the client who initially made the request. Now the client directly communicates with the web servers without the interaction of the discovery server and load balancer.

C. Replication for distribution

To achieve the distribution, i.e. to avoid single points of failures, we replicated the service discovery server and the load balancer across multiple machines. In case one of them fails, the other will take over.

III. Assumptions

- We are assuming that the web servers know the addresses of both the service discovery server and the load balancer.
- The clients are familiar with the address of the Service Discovery Server.
- The services provided by the web server are not rapidly changing.
- The service discovery server will contact all the web servers every few seconds for updating its service registry.
- There should be a minimum bandwidth for the servers to accept the requests.
- There is a limit on the number of requests that a server can process.
- There is at least one server available to accept the request at all times.
- The network is reliable and secure.
- There is no change in the topology.

IV. Implementation

To achieve the above design, we implemented it in Python 2.7.15 environment. The server that we chose to deploy all the web servers, service discovery servers, and the load balancer is the Flask server. Flask is a framework supported by Python 2.7.15. For creating the SOAP web services, we used the Spyne framework of Python. For creating the SOAP client requests Suds library is utilized.

A. Web Server

We first created a web server which provides addition, multiplication and a string reversal service. WSDL was successfully generated. Web servers also provide services like request count and server load which is used internally by the load balancer. When the web server is up and running for the first time, it sends a request to the service discovery server and the load balancer to register itself with them.

B. Client

Developed a client which requests the web services. This was achieved using WSDL provided by the web service.

C. Service Discovery

We used a registry to discover and store the information of all the available web services. This is achieved by contacting each web server for every 10 seconds, parsing their WSDLs and updating the service registry. We used the python data structure dictionary to store the servers and their services. This dictionary is a key-value pair where the key is the server URL and the values are a list of methods in that server. In this server, we created two threads. Thread 1 runs continuously to collect the web service information from all the available web servers. Thread 2 contains the web server, i.e. it handles the client requests.

D. Load Balancer

This is identical to the service discovery server but varies in the functionality. The dictionary in load balancer stores the URL of the server as key and its CPU load as value. The two threads perform the same tasks as in the service dictionary. Thread 1 runs continuously to collect the load from all the available web servers. Thread 2 handles requests from the service dictionary server.

V. Testing Strategy

We needed to test the functioning of our framework in 3 areas. Hence, we resorted to three testing strategies.

1. Service Discovery Test
2. Load Balancer Test
3. Scalability and Performance Test

A. Service Discovery Test

To test the functionality and reliability of service discovery, we first created 2 servers with different initial CPU loads. Server 1 with CPU load 10 and server 2 with CPU load 20 and both of them providing the same web services. By default, when requested for a service, the service discovery server pointed to server 1 because it has a lower CPU load. Then we stopped server 1 to see where the next request would be routed. We observed that the service discovery updated its registry as expected and pointed to server 2. By running the server 1 again, it pointed back to server 1 as anticipated. So, this test case proves that service discovery is working as expected.

B. Load Balancer Test

Just like in the service discovery testing, we took 2 servers with different CPU loads, 10 and 20 respectively. For the purpose of testing, we increased the load on each server by one percent for every request it serviced. After some requests, when the CPU load of server 1 becomes greater than the CPU load of server 2, i.e.21, the load balancer sent the request to server 2. Then, we observed that the load balancer pointed to both the servers alternatively for the upcoming requests, making the CPU loads balanced in both the servers(see Figure 2 and 3) Using the same, the load balancer worked for 3 servers. Hence by scaling the servers, the load balancer was working fine.

Figure 2. CPU percentage across servers

C. Scalability and Performance Test

To measure the scalability and performance of our system, we needed some metrics like time to service a request, load on the servers etc. which characterize the web services.

First we tested how the load is balanced across multiple servers for 50, 100 and 150 requests (see Figure 2 and 3)

Secondly, we tested the time taken by each server to service 100 requests (see Figure 4)

Finally, we found the mean, median and variance of the time to service 100 requests. The results are plotted in the graphs below (see Figure 5)

VI. Results

Based on all the above testing strategies, we found the following experimental results.

A. Result 1

The requests served by server 1 is high compared to other servers because its CPU percentage is relatively low in the beginning.

B. Result 2

From the Figure 2, it can be seen that the CPU load is almost equally balanced among all the servers after 100 requests. This shows that the load balancer has done a good job.

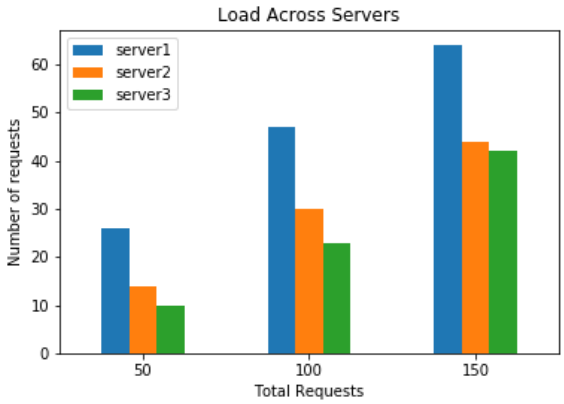

Figure 3. Load across servers

C. Result 3

From the figure below, we see that the time taken to serve 100 requests by a single server is relatively low compared to 2 or more servers. The reason being that it took considerable amount of time to switch between the servers when there is more than one server available. Even though it is not as expected, we can expect better results if the time to service the request is high.

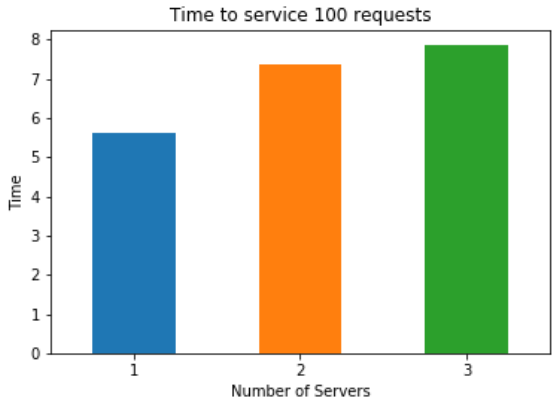

Figure 4. Time to service

D. Result 4

Finally, we plotted a graph of the average, median and variance of the time to service 100 requests. From the graph the mean and median almost coincide. The variance however is constant, meaning there is not much difference in the values. That is the time taken to service 100 requests is more or less equal after experimenting it several times.

VII. Conclusion/Future Work

In this paper, we presented a web service framework which is distributed in nature. We implemented it in python environment and we precisely followed the SOAP design principles. We were able to study and analyze the behavior of our proposed system. One important aspect that fascinated

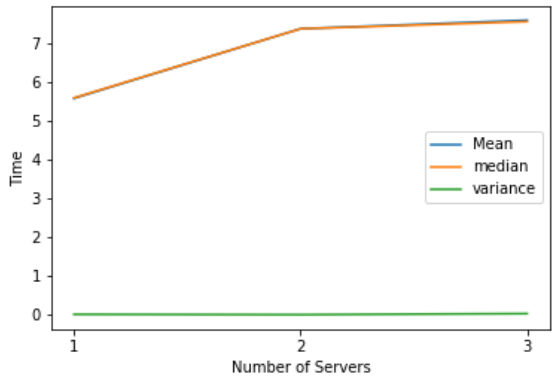

Figure 5. Time to service

us is the load balancer and how it works. The results of the load balancer were quite satisfying but in the future rather than just using the CPU load as criteria, we would like to consider other statistics like available RAM, finding the nearest server using latency of requests as well for load balancing.

ACKNOWLEDGMENT

I would like to thank Dr.Anupam Joshi and TA Oyesh Singh for their guidance. Also, thanks to my teammate for his time and support.