Haoran Ren & Dayuan Tan

CMSC 621 Advanced Operating System

Dr. Anupam Joshi

12/14/2018

Final Project Report – A Distributed System

Abstract

This is the final report of the CMSC 621 course project – implementing a distributed system. In our design, we were aiming to set up a distributed system providing a set of web services with at least five nodes. There should be some centralized nodes that could collect all the web services provided by the system. Each of the web services should be provided by multiple sites at the same time, and client requests should be evenly forwarded to each server. In our final deployment, we set up a system with two central servers and four servers. We combined the service discovery and load balancing mechanisms into each central server. The system also provided automatic failure sensing and correction. The results of testing experiments showed an average response time of 90 milliseconds together with the proof of scalability and reliability.

1. Introduction

Distributed systems, as sets of separated nodes connected with networks, request more configurations than those standalone systems to maintain system internal status. Usually, because of the existence of replicas and concurrent processes acting on those replicas, to keep a healthy

system condition, a distributed system should focus on a lot of tasks including consistency, process scheduling, synchronization, etc. There are several criteria to indicate the functionality of distributed systems, such as availability, computational power, reliability, safety, scalability, and maintainability. Distributed systems with different purposes would come with distinct requirements or various hierarchies of need in those properties. In our project, we introduced mechanisms like load balancing and automatic error correction to hold those natures.

This report is sectioned into 6 parts. Section 2 and 3 illustrate our design and provide an overview of our project. Section 4 goes into details about the project implementation. Section 5 introduces our attempt to blockchain and our implementation of Byzantine Fault Tolerate. Section 6 describes the testing methods and the results. Section 7 concludes our achievement.

2. Design

Based on the project requirement, we first included centralized nodes into our system. The central servers would work as both the service registry and the load balancer. When a server starts, it should report what services it has to the central servers. The central servers would register those services to the service registry as instances of certain types of services on that server. For each service instance, the registry should also record the load on that instance and potential faulty behaviors. If the potential faulty behaviors reach a user defined threshold, the system should mark that instance as unavailable until there is no load on it, and then, remove the instance from the registry. The central servers should also track the load of each server. When a client requests a service, the load balancer should always provide the client with the instance on the minimum-loaded server that provides the requested service. It should also add one to the load of the server

that the instance is located at and remove that load when the service is done. In the cases of all server have no load in the beginning or more than one server have the same minimum load, it should just pick arbitrary one from the possible servers. In the case of initial central server fails, the clients request should be directed to another working central server.

On the client side, when a client requests for a service, firstly, it should send a request to the central server. The central server should give back a service instance that the client could request service from. Then, the client should request service from the target server directly and gets its response. After receiving the response, the client should inform the central server that the job is finished, also, if it was successful or unsuccessful. By doing this, the load balancer could clearly track the load on each server as well as each instance. Furthermore, if the communication between a client and a central server cannot be set up, the system should report the failures to the client.

For number of nodes, we decided that the system should support at least two central servers with mechanism to keep the consistency between them. Also, the system should support at least three servers.

3. Assumptions

We also made several assumptions during the design progress. These assumptions are introduced to simplify the implementation of the system, but without weakening the proof of functionalities.

   a. Different servers should have different names and different services should have different names.

b. A server should not register the same service to the central node when it is currently available in the system.

c. A server should never request to remove a non-registered service.

d. There is no load limit of each server as well as each service instance.

e. There is no dynamically allocation error and out of memory error on all the servers.

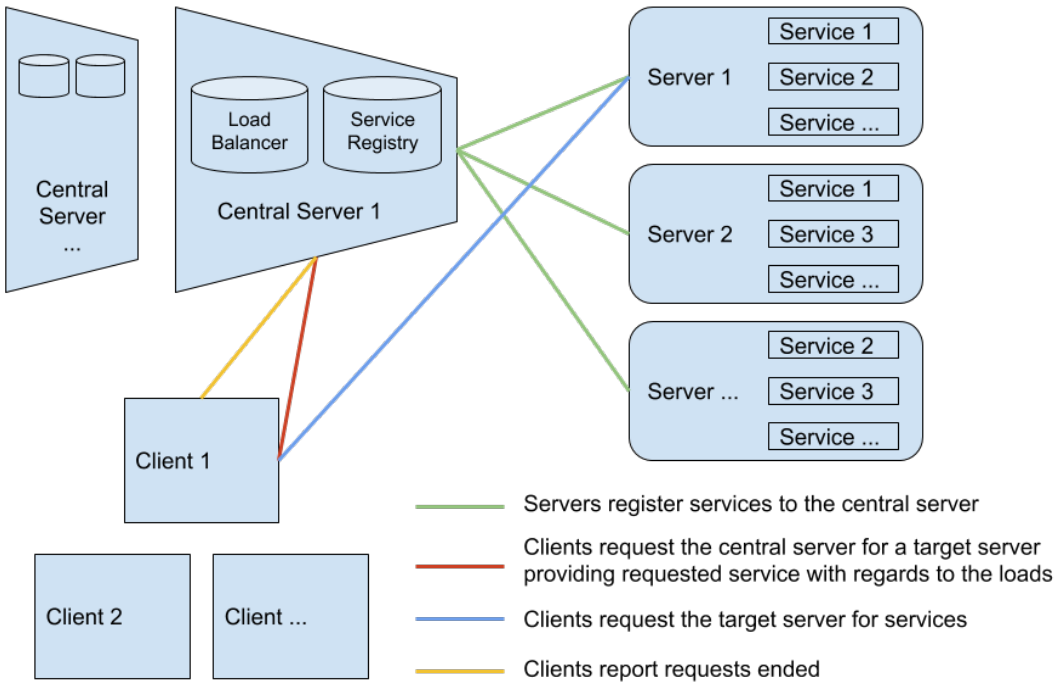

Figure 1: Architecture of the Distributed System.

4. Implementation

We developed our system in Ubuntu 16.04 virtual machines with Eclipse IDE. We used Apache Tomcat servlet engine as server container, and we used Apache Axis2/Java to publish web services. All the coding was done in Java and XML.

The `ServicesList` class was a self-defined class on the central servers to work as the service registry and load balancer. It had the following members:

```
HashMap<String serviceName, HashMap<String serverName, ServiceEntry instance>>
HashMap<String serverName, Integer load>
```

The first map contained key-value pairs of a service name with an instances list of the type of services. The instances list was also a map containing pairs of a server name with a `ServiceEntry` class object which was another self-defined class to describe an instance. The second map was the load list of each server. The `ServicesList` class also included several methods to work as direct program interfaces of the web services provided at the central servers, such as registering or removing a service, handling clients' requests based on server loads and feedbacks if a server provides the service correctly.

The `ServiceEntry` class contained the following members:

```
String url;  int load;  int failure;  int last_failure;  boolean isAvailable;
```

The member `failure` was the number of potential failures that occurred since the last registeration of the service instance. The member `last_failure` was the timestamp of the last faulty action of the service instance. We defined a potential failure as a faulty action that happens after another faulty action within a user-defined period of time. This mechanism would avoid failures cause by network lagging or temporary disconnection. However, if the number of potential failures reach a user-defined limit, we would consider removing that instance from the system. In that case, `isAvailable` was used to block new connection to that service instance and wait for all existing requests to finish before the removing.

To invoke the web services at client side, we used the `invokeBlocking()` method provided by Axis2. If the connection cannot be set up or some other errors occur, that method would throw an `AxisFault`. By capturing such exceptions, we could mark that connection as unsuccessful. If an unsuccessful request was towards a central server, the system should immediately switch to another central server automatically.

5. Blockchain (Byzantine Fault Tolerant)

Inspired by the blockchain technology, we implemented the Byzantine Fault Tolerant to ensure each node of the system has the same record. When a client request is ended, the system would log the request and its related results to a file on the node where the request was launched. All the other nodes would check the latest modification time of that log and compare it with the latest modification times of their local log files. A log file having the most recent latest modification time means that the log file is updated, and other nodes would update their local log files based on the updated one. By doing this, all nodes will have the latest copy.

We also coded up a program that compares log files on different nodes with Byzantine agreement. The program should be executed on one node of a set of four nodes. If only three nodes have the same version, the program would point out the remaining one that needs to be updated. In our test, we manually changed one log file on a node, and the program could discover it immediately. After it is corrected, it reported everyone is same again.

For more details, check the readme file in the Byzantine folder.

6. Experiments & Results

During our experiments, we deployed our system on several virtual machines connected by the same local network. We had one central server and one backup central servers. We coded up three services which just print out who requests what service on which server. Also, they delay the process for a random time between one to five seconds to create simulated processing time and simulated load. We distributed the three services to four servers as shown below.

Server1	Server2	Server3	Server4
Service1 Service2	Service1 Service3	Service2 Service3	Service1 Service2 Service3

Table 1: Services distribution

On the client side, we wrote a program that requests a sequence of random services from the system. Firstly, we tested the performance of the load balancer by launching different numbers of instances of the program simultaneously to simulate multiple client requests at the same time. Figure 2 is the load distribution with clients requesting for random services.

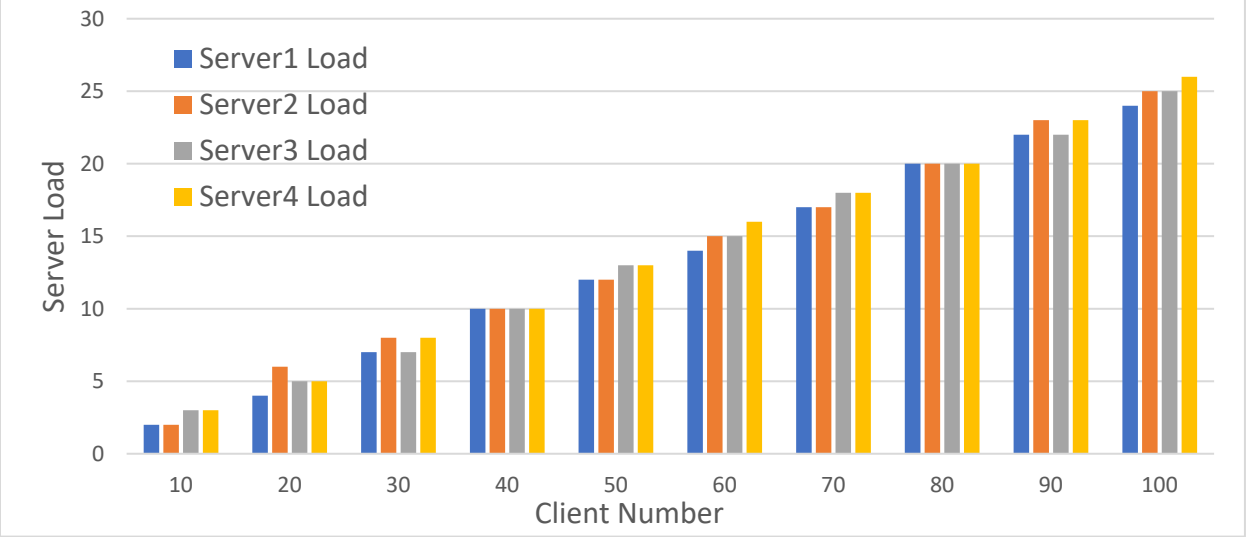

Figure 2: Load distribution on each server with clients requesting random services

We also tested the case that all the clients requesting for the same services. Figure 3 is the results of clients requesting for Service3 at the same time. Notice that Service3 is not provided by Server1.

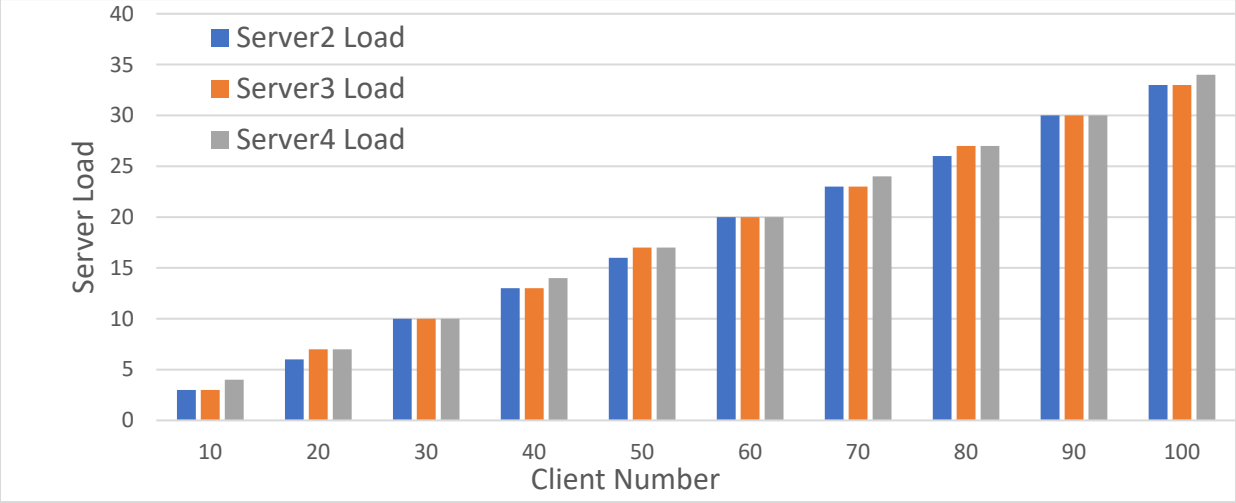

Figure 3: Load distribution on each server with clients requesting Service3

The program could also mark down the response time of all the three communication phases which are initial request to the central server, request to the actual service, and report to the central server. For the second one, we subtracted the simulated processing delay from the whole response time. Table 2 below is one of the experiment results with 15 instances of the random service requests program. The results showed an average full cycle response time around 90 milliseconds which yields a good performance.

Client #	1	2	3	4	5	6	7	8	9	10	11	12	12	14	15
Phase 1	50	51	54	48	53	36	34	47	38	48	40	32	36	55	38
Phase 2	17	13	10	14	13	16	22	24	37	21	34	31	37	36	26
Phase 3	28	32	32	31	28	17	23	16	15	17	13	12	17	13	12
Total	95	96	96	93	94	69	79	87	90	86	87	75	90	104	76

Table 2: Average communication response time in millisecond

Moreover, when we disconnected one server manually from the system, it began to report failures and eventually removed all the services provided by that server. We also tried to disconnect the central server and the system switched to the backup central server automatically. Thus, we proved the system reliability as well as the function of failure sensing and error correction. We also proved the system scalability by having multiple nodes communicating in good manner.

7. Future work & Conclusion

During the development, we discovered several difficulties such as those special circumstances described in the assumptions in Section 4. How to handle those conditions could be a good problem to study in the future. We also foreseen some other good options or modules which would be useful to be added to our system such as security mechanism.

In conclusion, we built a distributed system that fulfills the requirements of the project. Experiments showed that our system had good performance and it was reliable and scalable. We also learnt lessons that are related to the course topics during the work and gained experience in Java coding and web service development.