Comp90015 Distributed System Project 1 Report

Group: Legendary

Group members:

- Zhao Peng - zpeng2

zpeng2@student.unimelb.edu.au

- Tiange Wang - tiangew2

tiangew2@student.unimelb.edu.au

- Weizheng Wang - weizhengw

weizhengw@student.unimelb.edu.au

- Yue Yu - y9

y9@student.unimelb.edu.au

1. Introduction

This project is established to build a multi-server system, and it will broadcast activity objects among clients and servers. For the client part: 1. A client could register a username and a secret pair. 2. The application allows clients login and logout with their usernames or secret pair. 3. A client could login and logout without a username or a secret pair, just be anonymous. 4. Clients broadcast arbitrary JSON objects, which is called activities, to all other clients who connected to the network. For the load-balancing connection between clients and servers: the system uses the synchronisation method (TCP socket) to make sure the proper registration and authentication. Thus, the system could prevent the wrong or malicious servers from joining the communication (Server failure model).

The debugging of the distributed system code was a huge challenge when we were editing and testing code. Since the database of each server in the system is not synchronous, we have to test the process of register and lock largely to guarantee the implementation of the functions.

To sum up, we have realised the implementation of primary functions in the specification. However, the multi-server system may encounter some risks when there exists a significant amount of users and servers.

2. Server failure model

The project assumes that servers never crash or quit once started, even if the servers cannot be trusted or the messages which are sent by servers may have been interrupted or not correct, which means the messages sent by servers may contain nothing or wrong information. Besides, in realistic, the servers are possibly crashed (actually this is a normal thing that a server crashed while it is running). Thus, the failure model of servers should be considered in this multi-server system.

Handling the server crashed problem, it is possible to expend the JSON protocol. This process may be called as Server_Quit to make the server terminate the existing connection. If the crashed server is just a 'child' server, it just quit (maybe also sent a warning message). If the crashed server is a parent server, the 'children' of it should connect with the parent server of the crashed server(if the crashed server has a parent server). To achieve this reconnection, the Server_Quit message need contains the port/address of the connecting servers. These possibilities are described in the graph way (Figure 1). Besides, we assumed that when a server crashed and quit, it should send a warning message. If a server can quit without warning, this will be a disaster for the connections. We consider the TCP protocol which has a timeout period. Thus, if the message reaching time is more than the timeout limitation, the server may be deemed to be crashed. Then we potentially send the previous server data to establish another connection with other servers.

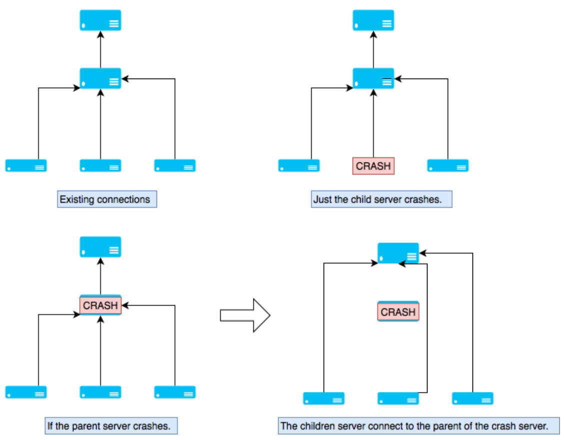
CRASH
Existing connections
Just the child server crashes.
CRASH
If the parent server crashes.
CRASH
The children server connect to the parent of the crash server.

Figure 1. The possibility of a solution when the different kind server crashes.

The failure model that we assumed is a fail-stop model, which means when the server quits or crashes it never restart. If we assume that servers may restart after crashed or quit with the same address and port number, then we could keep polling.

Server failure model may also contain the checking method to pretend any other server interrupt the connections. Once getting the receive messages, the server resent the message, and the receiver checks the two messages whether they are same.

3. Concurrency issues

Through browsing functions in the ControlSolution.java, the concurrency issues of the program may occur in the process of login and register/lock. Since it does not exist issues which caused by modifying and updating data from servers, we primarily paid attention to two aspects. One is the redirect issues caused by vast numbers of login and register requests. And another is registered issues which clients try to register the same username at the same time.

As there exists the delay of broadcast and receiving between different servers, when receiving a large amount of register and login requests, one server will redirect these requests to another vacant server to allow them to connect to the server which has fewer occupancies. Even a client may encounter several redirections in the environment of thousands of servers and clients. Therefore, to avoid this problem, it is necessary to distribute a redirection value to each server. The solution is shown in figure 2; the value serves as a notice that informing other servers the number of current connections of it, such as 50. When the connections are less than 50, this server is allowed to accept the redirections to connect with clients.

Another issue occurs in the register and lock process. Supposed that there are two clients, one called Alice and another one named Bob, try to register the new and same username although their secrets are distinct from server A and B. The server A and B broadcast lock_request to all other servers after Alice and Bob registered. After a period, all the servers except these two broadcasted servers have sent the lock_allow message to other servers. However, these servers are not able to recognise the server where lock_request is from as they just check whether the username is stored in their databases. A and B will allow Alice and Bob to register and store the pair of username and secret in their respective database after they check the username and broadcast lock_allow message to each other. Next, when Alice and Bob try to login, it is confused for the system to verify which secret is unique to the username which results in a confliction.

One possible solution for this circumstance is to store the registered username to a waiting list firstly instead of the local database directly and add the server ID to the lock_request message for other servers to know where the message is from. The server which is the first one to receive all the lock_allow messages is able to allow the client to register and remove the username from the list. Then it broadcast a register_already with ‘username’ to all other servers. The server which is in the process of waiting the lock_allow messages should check this username on the waiting list. If the username is also on its list, the server must remove it and broadcast lock_denied message to the client.

The protocol is not perfect. With the increase of requests, the system should also consider the number of used threads and the balance between the threads and tasks to enhance the system performance.

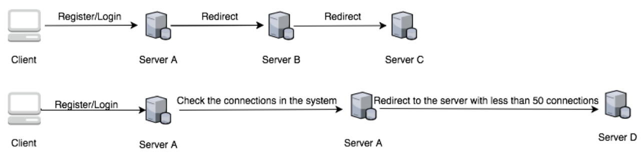

Figure 2. The solution for the redirect issue.

4. Scalability

Scalability is the system’s ability to handle the growth of the number of users. From the project specification, we support multiple servers and client in our system. Thus, we should be handling the case of the number increasing of servers and clients. Initially, we implement the system as only two servers will be connected to each other to send the LOCK_REQUEST. Thus, we considered a tree with multiple servers as the nodes, and in each process, the parent server connected to the child server, and only two servers will talk to each other every time. To compute the message complexity, we have a complexity of O(n) for a single process. For a new user to register, we will first REGISTER; then between the servers, we will use LOCK_REQUEST to check whether the username has already been used and which will respond in a LOCK_ALLOED or LOCK_DENIED to every server in the system. Thus, we can know that the message complexity will be O(n)^3. But we know that in SERVER_ANNOUNCE, it broadcast from every server to every other server once every 5 seconds, hence will result in a message complexity of O(n)^4. The complexity we got is relatively high and may result in low scalability for our system. Thus, we had some discussion and changed our way of implementation for our system. When we have a new user to REGISTER in server A, it sends a LOCK_REQUEST to all the other servers which result in a complexity of O(n). Then every other server checks its own database and responds a LOCK_ALLOW(to the upper connected server) or a LOCK_DENIED the server, when server A wait until it received all the response of LOCK_ALLOW or one LOCK_DENIED, it will make its next execution. From this process, we can get the final complexity is O(n) which remain relatively good scalability.

There are some other ways to improve the scalability. One method of improving the scalability is partitioning, and the most straightforward strategy in partitioning is round robin. From the book “A Fresh Graduate’s Guide to Software Development Tools and Technologies”, we learned that this method is suitable for systems like ours that scan the information in each server sequentially. There are other partitioning strategies such as hash partitioning and range partitioning. Another way of improving the scalability is replication which contains two models – master-slave model and multi-master replication model (Khare et al., 2012).

5. References

Khare, A, Huang, Y., Doan, H., & Kanwal, M. S. (2012). A Fresh Graduate’s Guide to Software Development Tools and Technologies. *Chapter-6: Scalability, School of Computing, National University of Singapore*.