Group Report

Introduction

We have built a multi-server system where a number of clients are able to broadcast JSON string activity objects, and we implemented all the functions in specification. When the user intend to register, service firstly check whether he is in the local userlist, if the user is not in the local userlist, it will broadcast lock_request to other servers. For the client who intend to login, the server only check its local userlist. When the user is successfully logged in, a GUI will pop up, it is necessary a previous registration for non "anonymous" users. Then the user can send activity object to all other connected clients inside the tree Multi server network.

There are a lot of challenges to overcome the project specifications. The first step and the most important was to understand the requirements of specifications, because some details are not clearly stated, and because each member sometimes tried to implement a different solution. Under the guidance of the skeleton code and professor and his team supporting us along the way, the framework we built became more and more clear over time. We have also meet other challenges such as using git(understanding the mechanism of git is difficult for members who had not used it before), and implement the lock protocol, since it could be ambiguous in some test cases.

Server Failure Model

(1) Problems in the existing system

The structure of the existing multi-server system is a tree. It means if a server suddenly crashes, all servers connected to it will be separated and disconnected. This will lead to the total failure of this system because these servers cannot work together again. As a result, servers cannot send messages to each other to share information. Also, clients cannot regard these servers as a system. They cannot communicate with each other anymore using these servers.

(2) Approaches to fix this problem

The main principle is to guarantee all working servers can always connect as a tree in all types of conditions.

(3) Deal with server's sudden crash without warning

We can change **Server_Announce** message to handle server's sudden crash. Here are two kinds of **Server_Announce** Messages. For ordinary servers, we add a field named "**parentnode**" to represent its parent node's information. For root server, we add a field named "**firstchildnode**" to indicate its first child server's information. We assume each server record other servers' information in local memory. And Whenever it receives a **Server_Annouce** message, it will update the information of corresponding server.

Server_Announce Message (For ordinary servers)

```
{
"command": "Server_Announce",
"id:" fmnmpp3ai91qb3gc2bvs14g3ue",
"load": 5,
"hostname":"128.150.13.14",
"port":3570,
"parentnode":{
        "id:" dienfufhw91qb3gc2bvs14g3ue",
        "hostname":"128.160.0.56",
        "port":3780
        }
}
```

Figure 1-1 Server_Announce Message for ordinary servers

```
Server_Announce Message (For root server)
{
"command": "Server_Announce",
"id":" fmnmpp3ai91qb3gc2bvs14g3ue",
"load": 5,
"hostname":"128.150.13.14",
"port":3570,
"firstchildnode":{
                "id":" dienfufhw91qb3gc2bvs14g3ue",
                "hostname":"128.160.0.56",
                "port":3780
                }
}
```

Figure 1-2 Server_Announce Message for root server

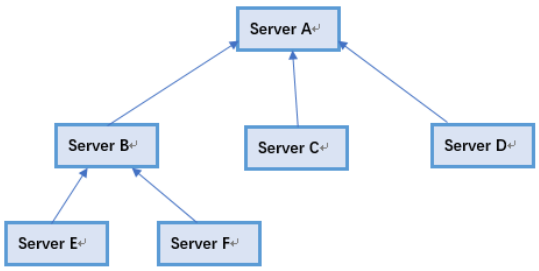

Figure 2-1 A example of distributed server system with tree structure

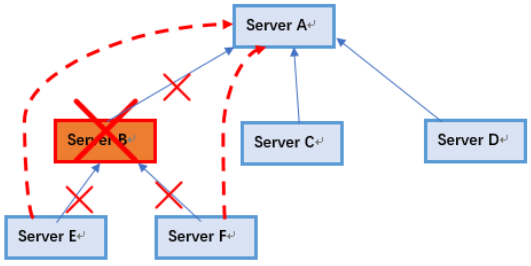

Figure 2-2 When ordinary ServerB crashes without warning

In figure 2-2, whenever server B crashes suddenly, all its child nodes will realize it immediately since they are disconnected to their parent node. These child nodes will search for server B's latest **Server_Announce** and find out its "**parentnode**" field (Server A). All child nodes of server B will immediately reconnect to Server A. Thus, all servers form a tree structure again.

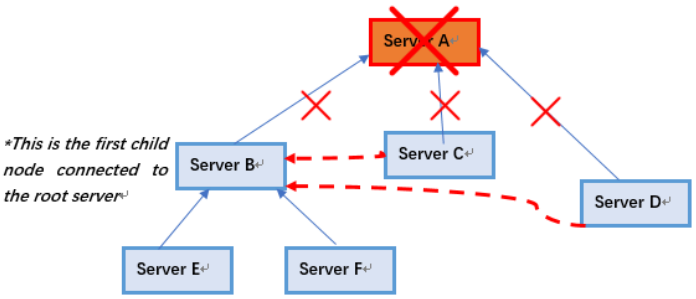

Figure 2-3 When root ServerA crashes without warning

Whenever Server A (root server) crashes, all its child nodes will do the same things as above. The difference is that they will find out "**firstchildnode**" field (Server B) and reconnect to Server B. Now Server B functions as a new root server.

(4) Deal with server's quitting gracefully

If a server wants to quit gracefully, it will broadcast a message to all his child nodes. This message covers a **"redirect_node"** field to tell all child nodes to reconnect to another server.

The following pseudo-code illustrates how Server B can quit gracefully in Figure 3-1.

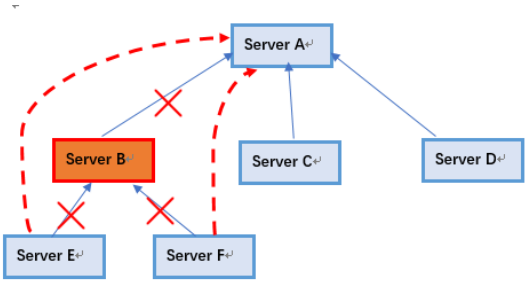

Figure 3-1 When server B wants to quit

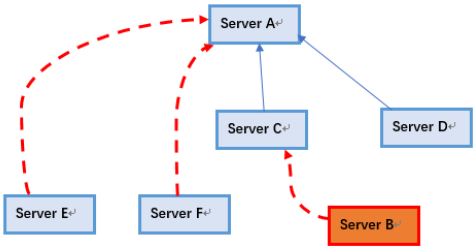

Figure 3-2 When server B wants to restart

Step1

If Server B has child nodes:

Server B broadcasts a **Quit_Request** message to all its child nodes

If Server B is **a root server**:

"redirect_node" will be its first child node's address

Else:

"redirect_node" will be its parent node's address

```
{"command": "Quit_Request",
"id":" fmnmpp3ai91qb3gc2bvs14g3ue",
"hostname":"128.150.13.14",
"port":3570,
"redirect_node":{
        "id":" dienfufhw91qb3gc2bvs14g3ue",
        "hostname":"128.160.0.56",
        "port":3780
        }
}
```

Figure 3-3 An example Quit_Request Message

Step 2

If a child node receives "**Quit_Request**" message:

It disconnects to its original parent node.

It tries to reconnect to the "redirect_node" address.

If it connects to the new parent node successfully:

It will return a **Quit_Allowed** Message

Else:

It will return a **Quit_Denied** Message to Server B

Step 3

If Server B receives all '**Quit_Allowed'** Messages from its child nodes:

Server B will quit gracefully.

Else If Server A receives any '**Quit_Denied**' Message from its child nodes:

Server B will fix its connection issues and broadcast '**Quit_Request'** Message again.

Whenever server A wants to restart, it can connect to any working server in this system

Concurrency Issues

There are several concurrency issues happening during the developing process.

First, data might not be updated on the fly due to the delay of distributed systems. The first example is that the load information of each server, which is the number of current connected clients, is broadcast to every other server at a fix rate. But in reality, the server might be busy dealing with lock requests, activities or replies that it is too busy to broadcast. Thus, even if the server accepts more clients, other servers would not have known this and might still redirect incoming clients to the server and cause the server to be overloaded. One way to solve this is to change the logic of broadcasting. Instead of programming servers to broadcast at a fix rate, the server should collect load status from all other servers right before checking the possibility to redirect. But this still does not solve the problem of inconsistent data, as other servers load status could change rapidly when the server is deciding to which server to redirect. Therefore, the only way to solve this is to block other servers from accepting or disconnecting clients until the server sends redirect message to the client, which is not realistic as disconnection could

be raised from clients and performance could also be largely affected. So, there is a trade-off between availability and consistency under the distributed system where the partition tolerance should be guaranteed. CAP theorem (Andrew Butterfield & Gerard Ekembe Ngondi, 2016) states that a distributed system cannot deliver on all three of the following guarantees: that all system entities see the same view of the overall system state (Consistency); that a response regarding success or failure is delivered to every request (Availability); and that loss of messages or failure of part of the system does not prevent the system as a whole from operating (Partition tolerance). For loading information, what matters is performance rather than consistency, thus implementation of this project tolerants inconsistent data.

Secondly, distributed systems would cause synchronization problems. Same data could be write or read simultaneously by different threads, thus might cause 'dirty data' in memory. A common way of solving this is to use the keyword 'synchronized' for methods that include operations on objects to ensure that objects are processed sequentially.

Thirdly, transaction conflicts might rise. For example, when two clients simultaneously send register request with same username to two different servers, both might register successfully if servers receive 'server allowed' from all other servers as the username does not exist on any server. One way to solve this is to store the username if the server does not have such user locally before broadcasting 'lock request' to other servers. Through this way, replicated usernames will not happen, but the situation that both clients are rejected could occur. one better solution is to allow identical username to be used in distributed system and assign each client with a unique id generated by servers. Another way is to store usernames on one central server with lock on the database table, where performance is trade-off by the achievement of consistency.

Scalability

The scalability is an important factor when designing Distributed Systems, because it aims to handle the increase of users and resources without having a big impact on the current online system. According to Clifford (1994), a single coherent system has three components which affect the implementation of the Distributed system. First, the quantity of users and resources that can be added over time on the network; Second, the geographical distance where the nodes are physically located, and finally, the administrative control which allow the management, support, it or other teams, to easily control the system.

In the architecture used to build this multi-server system, many issues may arise when increasing the tree network by adding either more nodes (resources), or more clients (users). Although we assumed that servers never fail or quit, the current implementation of the message broadcasting between nodes servers relies in a high number of messages flow through the network for handling the data consistency of the system. We acknowledge that message complexity is medium because it is really easy to implement from the development side, nevertheless, it is a bad design solution, for instance, imagine a 1000 server nodes network when one user try to register, the reliability will decreased considerably when adding more servers due to the flow messages number needed for register successfully each user for every server

connected. But not only the reliability, also bad performance issues pop up depending on the geographical location of the nodes. A single poor connection in one server could cause a broadcast failed in its tree network, and even if it does not fail, the latency will increase proportionally to the distance of the nodes.

A possible solution to improve the scalability for this tree architecture is to allow every server to have an IP table of know server connections, and set a timeout when trying to broadcast, if the timeout (for example 2 secs) is reached, the server will try to connect to any other known server to try again. However, this could cause inconsistency data issues that will need to be handle. Finally, another good solution for broadcasting is using indirect communication approaches, for instance, using publisher-subscriber paradigm or distributed message queues could improve both availability and consistency. Indirect communication provides a feasible solution for long distance connected nodes, and it can help with an easier administrative control for the system too; however, the architectural design and development complexity may be higher.

References:

1 Clifford Neuman, B. (1994). Scale in Distributed Systems. Readings in Distributed Computing Systems. IEEE Computer Society Press.

2 Andrew Butterfield, & Gerard Ekembe Ngondi. (2016). A Dictionary of Computer Science (7). Oxford University Press