**Introduction**

This report focuses on building a multi-server system, broadcasting activity objects among clients. The system allows load balancing of client requests by redirecting clients to other servers, while clients can register and login with a username and a secret. In this system, all messages and activity streams are JSON objects, and the client and server files are both programmed in Java language.

Our team did not meet many problems in the developing process, although the register and lock functions have spent more efforts. An additional problem is the difficulty of understanding the project specification, which includes some unclear explanations.

The system has been developed completely and can work correctly as the requirements on the specification. We have tested our system and made sure that it can work together with other correct implementations.

**Server failure model**

To simplify the project, we are firstly assuming that the servers never fail (no matter expected or unexpected) once started, although this is not the rule in a real-world distributed system. However, no failure does not mean that servers are always performing correctly – corruptions may occur, while dangerous users and network error can also affect messages transmitted. It is possible that correct information does not exist in messages. Under these circumstances, the ‘Invalid\_Message’ protocol is introduced as a solution, which prevents further potential risks to the system.

An invalid message can be sent by both clients and servers. When a server receives the invalid message, it must close the connection, and the sender should close its connection as well. But the closure may bring other potential harm to this distributed system: as we have modelled our servers in a tree architecture, the disconnection of one server will results in failure of the whole subtree under it. (Can add a photo here)

If we give up the assumption of never-fail servers and assume a model in which servers may shutdown but never crash, then they have to be connected back again after restart, using the same addresses and port numbers. To ensure the reconnection, another protocol is necessary; it could be another JSON object with the name ‘Server\_Redirect’ (Can add some codes here) with parent server, port and clients information. Before the gracefully shutdown, a server sends redirect information to other servers and all clients, thus the system can find this server after its restart. In this failure, clients connected to the server may lose their lock requests, although they will be able to connect back with same usernames and secrets later; adding a ‘Lock\_Request\_Timeout’ message as a notification will be helpful for them.

A more complicated situation is when the server my crash without any warning unexpectedly. All clients connected to it will lost connection immediately and cannot restart connection as there are no redirect information. One possible solution is broadcasting the ‘Server\_Announce’ message to clients, thereby keeping newest servers list to prove them other ways of reconnection. In addition, the ‘Server\_Announce’ can be used for servers to find way back to the system when their parent servers crash, even though such solution may lead a serve to its child and create a cycle.

**Concurrency issues**

In a complex distributed system with hundred of servers, concurrency problems may become common. Based on a system without server failures, we will discuss these concurrency challenges in the following section.

It can be always found, even in global business system, that two users are trying to register in a system with same username and password (secret in our case) to two servers. On the one hand, during the Lock\_Request processing time, there are possibility that both servers did not receive response before getting the Lock\_Request from each other, and they will both respond Lock\_Allowed to each other. On the other hand, if both servers have responded Lock\_Denied, neither of the two registrations will be successful.

To avoid such issue, we would recommend that servers save the user information in its local storage before sending Lock\_Request message, therefore, Lock\_Request with same credentials will be certainly denied, and only one user can register with that name and secret. This solution contains all servers in a system: once a Lock\_Request is sent to a server, it saves the user details before passing them to further servers. When this request gets denied in other servers and the Lock\_Denied is broadcasted, the server must remove these user details.

Other problems include situations like two users trying to register on a same server with same user information. For two Lock\_Request requests, their user credentials are same, and servers are not different; how can we define one from the other and prevent duplicate registrations? Here we would like to introduce the idea of numbers; in specific, that means adding an ID to each Lock\_Request. With a unique ID, the system can identify requests even if they are from a same server with same usernames and secrets. Moreover, other revisions may apply for concurrency problems. For example, synchronizing all servers with a full list of registered users may also reduce the rate or concurrency registrations. Such arrangement requires each server sending its user list to its child nodes, once the nodes are connected to it.

**Scalability**

In our system, message objects are sent from one server to its connected servers in order. If we do not set restrictions, the worst case in a system (with server number n) is connecting all servers in line, from which we can conclude the complexity of O (n). This complexity can be improved by setting tree restrictions when adding new servers. When we set the max depth of the tree as 5, new server will not be connected to nodes with depth 5; another node with less height will be assigned to it as parent to keep balance of the tree. For such tree structures, the complexity can be reduced to logarithmic of n, depending on the depth of trees.

According CAP theorem, only two of Consistency, Availability and Network partitioning can be satisfied. In our system, we prefer consistency and partitioning than availability. The problem is, our design specification has not mentioned any protocol about new servers getting information from old servers, hence we cannot guarantee consistency. It would be a more complete system, although not perfect, if we resolve this by making new servers synchronized with all others in authentication.

By default, the Server\_Announce message is broadcasted every 5 minutes from a server. If we turn back to see the Lock\_Request mechanism, the protocol will only allow a client to connect after receiving Lock\_Allowed from every server. In a large-scale distributed system, considering these 5 seconds, the time cost can be great and efficiency improvements may be necessary. Because we have designed the system as every sever knows its parent and children, this whole lock procedures can be simplified by assigning the announcement tasks to parent nodes; the server which sends Lock\_Request can then get all server information from the parents only.