Multi-Server Network
Semester-1 2018

Team Members
Anubhav Singh <anubhavs@student.unimelb.edu.au>
Ashwin Sajiv <apurushotha@student.unimelb.edu.au>
Sahitya Beru <sberi@student.unimelb.edu.au>
Uttakarsh Tikku <utikku@student.unimelb.edu.au>

1. Introduction

The project deals with building a multi-server system that primarily supports registering and logging in clients, authenticating servers, load balancing clients on the available servers using a redirection mechanism and broadcasting activity objects from one client to all other clients.

One of the challenges we faced in the project was, we had to design a system that made sure that messages received were sent by connections that had the appropriate authority and authentication. Another challenge was that the servers did not broadcast the REGISTER_SUCCESS message, each server had to make assumptions on whether or not a user has been registered based on the number of lock allowed messages that is recognises for a specific username and secret. Finally, the specification also lacks information about the design on how to identify whether a user should be registered based on the number of LOCK_ALLOWED responses received by the server.

In the end, we had successfully overcome these obstacles and built a system that supports client-server and server-server connections, balance load by redirection, process activity-objects, use locks for registration etc.

2. Server Failure Model

The assumption that servers never fail or quit once started is completely unrealistic. Designed with this assumption, the system does not keep any of its data in persistent memory. Hence in case of any failure or shutdown, as would eventually happen realistically, the system is unable to retain any information about other servers or authentication information about users. When the server is restarted, even with the same hostname and port, its state is exactly as if a new server was started in its place. Keeping such data stored on the file system or in the cloud instead of just memory would solve this problem. This restarted server will send a new authenticate message to the remote hostname and port that have been given to it, however, other servers which have specified its hostname and port as their remote connection will not know about it. To solve this, a server can try to make connections to the remote server (the one specified in the command line arguments) after certain time periods.

In the current system, servers do not share any existing information about other servers or users with each other. Users' authentication information is not shared after the lock requests, lock allowed and/or lock denied messages have been shared among the servers. Adding requests for historical data (such as registered users) to the protocol would help solve this problem as well as keep a better-shared state among the servers.

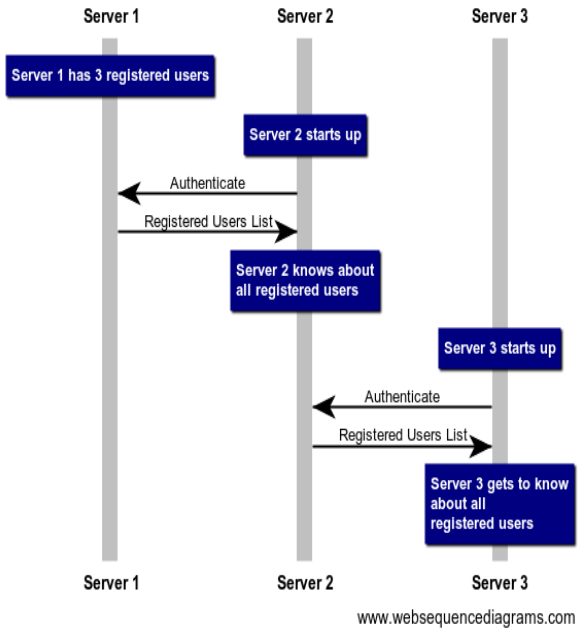

Diagram 1: Sending a registered users list as a reply to a successful authenticate message

If it was assumed that clients may only quit gracefully, then they could be removed from the redirection consideration by the other servers when the quit message is received. The quit message could contain information about one of the servers that this one is directly connected to. Then, all the clients and servers that were directly connected to this one (except the one in the quit message) could instead connect to the one whose information was included in the quit message. The only case where this technique would fail is where the server receives a connection request from a client and sends them a redirect message to the server which has just quit between when it quit and when its quit message was received. Adding a quit message acknowledgement from the server which is mentioned in the quit message would solve even this problem.

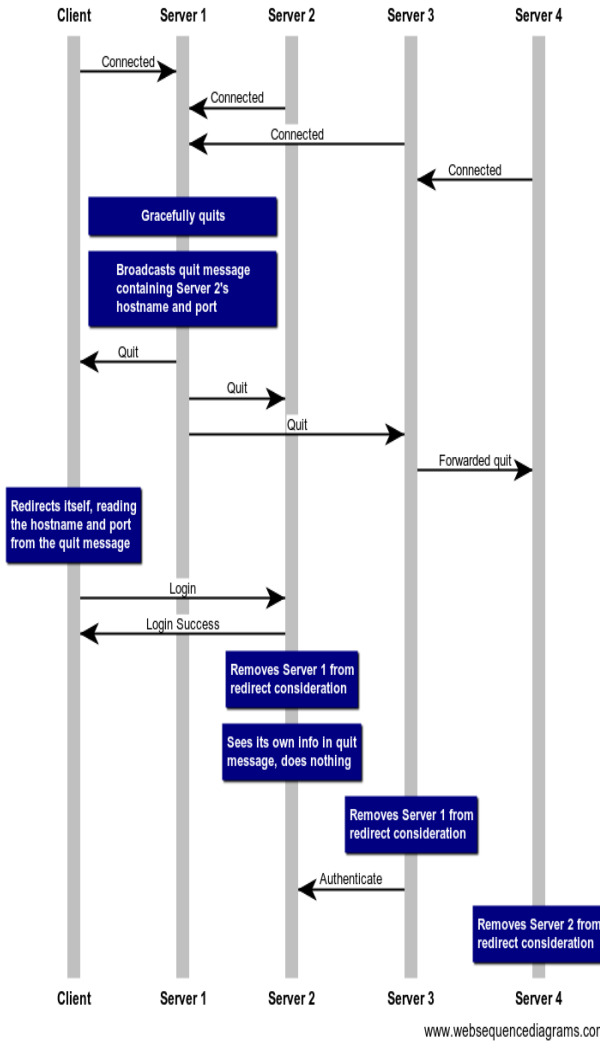

Diagram 2: Gracefully quitting server broadcasts hostname and port of one of its directly connected servers so that clients and other servers can redirect if needed.

If it was assumed that servers may crash at any time without warning, as would be the most realistic case, then any crash could potentially split the entire system into pockets.

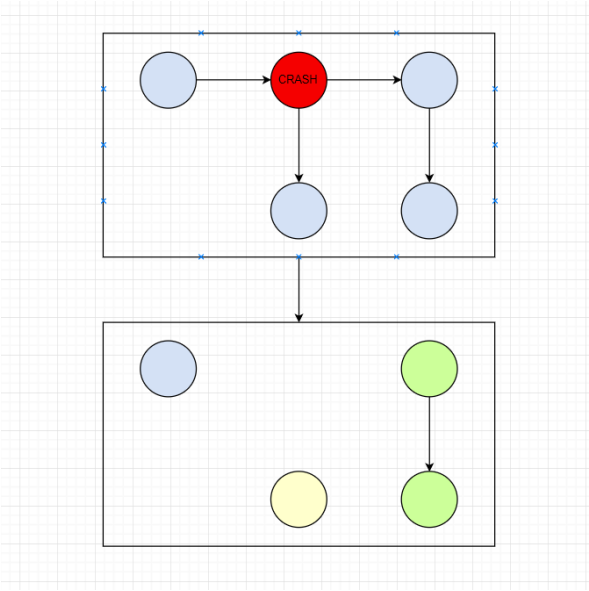

Diagram 3: Creation of pockets of servers instead of one distributed system upon a crash.

The same problem would also occur in the current protocol if a server was to somehow broadcast an invalid message. Then, the servers that it is directly connected to would disconnect with it, leaving the rest of the system in the same state as if it had crashed.
There would be no way to recover the system and reconnect between the different pockets of servers with the current protocol. A solution similar to Diagram 2, sending information about one of the directly connected servers, but in the server announce message, might work as a solution for reconnecting among servers, though the clients directly connected to this server will not have this information.
This problem would also be of much less of an importance if the distributed system had been designed as a mesh instead of a tree. Then, the crash of one server would only affect the users directly connected to it, not the other servers. Then, the only concern that the other servers would have would be to not redirect future users to this (crashed or disconnected) server. To do so, as has been implemented in our system, there could be a time limit after a server has sent a server announce message for which it is considered eligible for redirection. If no server announce was received from a server in that amount of time, it would not be considered eligible for redirection.

3. Concurrency

Concurrency is an important issue to consider in distributed systems. Many issues could come up in distributed systems due to multiple users accessing the same resources at the same time. Below are some of the main problems that may arise while using the given protocol:

Concurrent registration by users with same credentials on two different servers: If two users try to register simultaneously on different servers with the same credentials, this would lead to both the servers requesting locks, if the servers maintain a “locked users” list, both of these lock requests will be denied and they will not be able to register. However, if the “locked users” list is not maintained, this would lead to successful registration of two users on the network. This is a failure of the protocol because we would expect exactly one of the two users to register successfully. An alternative to this would be to maintain the user information in a dedicated user management server which would be the source of truth. So rather than requesting locks from peers, servers could register the users in the registration server.

Load balancing of servers and redirection: The redirect function compares load on the server and it's peers and redirects the incoming client login to another server if need be. This could cause problems in case of concurrent incoming client connections. If two servers have the same load and they receive two more connections each, they will accept one connection each but their load will not be announced for up to five seconds. If another connection comes in at server A, it will redirect it to server B and server B will redirect back to server A. This continues until the load of the two servers are updated. If server A has n connections and server B has n + 1 connections, all incoming connections to server B will be redirected to server A until the new load of server A is announced but server A will accept only two of those connections while other connections will keep redirecting. The alternative to this could be to have more frequent server announce and tolerating higher difference in server loads, which would allow server loads to sync up quicker. But this would increase the network traffic very quickly as the number of servers increase. Another possible alternative is to add a Boolean field to login message saying, “do not redirect” and having a finite number of redirects before the client sets the “do not redirect” field.

Message ordering for concurrent messages is not maintained by the current system. The message ordering would, however, depend upon the closeness of a client to the source of that message in the network. Therefore, the messages may appear out of order to nodes away from the source of that message. This could be compensated using a timestamp in the activity messages.

4. Scalability

The multi-server system uses broadcast throughout the system for certain functionalities including broadcasting activity objects and registering a new user. The following sections include scalability analysis using message complexity for both the use cases.

To provide for the broadcast of activity messages to all other clients, the servers in the system have been designed to broadcast any activity object received from a connected-client to all its connected clients as well as connected servers, each of the connected servers would, in turn, repeat the same process. Hence, in a system with m servers, each with n connected clients, for each activity message, there would be $(m-1)$ server to server messages and $(m.n - 1)$ messages from servers to clients. The message complexity is $(m.n + m - 2)$, which is the total number of messages sent for broadcasting an activity message. The order of growth of message complexity is $m.n$, which is the total number of connected clients. Since the use case requires broadcast of activity message to all connected clients, a minimum of $m.n$ *messages* would be necessary. Hence, it can be concluded that the message complexity of the system cannot be optimized further.

In the second use case, when a client sends the register request to the server, the server in-turn broadcasts a lock request to all the servers and waits for lock allowed or lock denied messages. To broadcast the lock request to all the servers, the server sends the lock request to all the adjacent connected-servers, which in-turn forward the request to their connected-servers (excluding the sender). Each recipient server broadcasts lock denied or lock allowed message to all other servers. If there are m servers in the system, for each register request, there would be 1 messages (register success/ fail) sent by the server to the client, $(m-1)$ lock request messages and $(m-1)^2$ lock allowed/ lock

denied messages between servers. The message complexity is $(m^2 – m + 1)$ which is the total number of messages sent by the system to register a new user. The message complexity is in the order of m^2 , hence a system with a large number of servers would get flooded with lock request, lock allowed and denied messages. This implies that the system would not scale very well with increasing number of servers in the system.

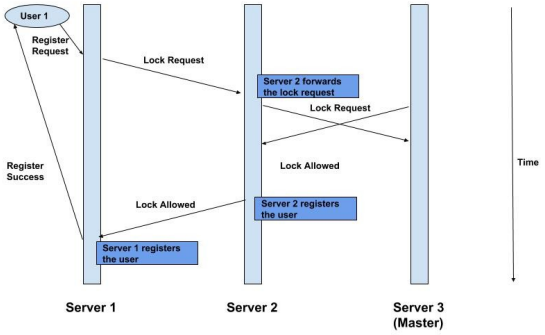

Diagram 4. Proposed protocol for User Registration using a master server for user management

An approach to reducing the message complexity would be to establish a master server for user management *(Diagram 4.)*. When the client sends a register request to a server, it will broadcast a lock request to its connected-servers, then all the servers (except for the master server) in receipt of lock request would forward the lock request to their connected-servers. The master server would then either reply with a lock denied or broadcast lock allowed to all the servers. The lock allowed message from the master server would be used by other servers to register the user information. This will reduce the message complexity for a user registration to a lower value of *1* when the user tries to register with master server directly and its a lock denied, to a highest value of *(2m-2)* when the user sends the register request to a normal server and the lock request has to hop over all the other servers to reach the master server and the reply from master server follows the same route. Furthermore, with this approach we can use the master server to send the registered user list when requested, especially to a new server in the system. This would mitigate the risk of failure of the system when a user is tries to login using a server added after user registration. This is important for scalability as when a new server is added to the system the login requests are redirected to it by all the other servers (because of low load), and since it does not have information about previously registered users it will deny them.

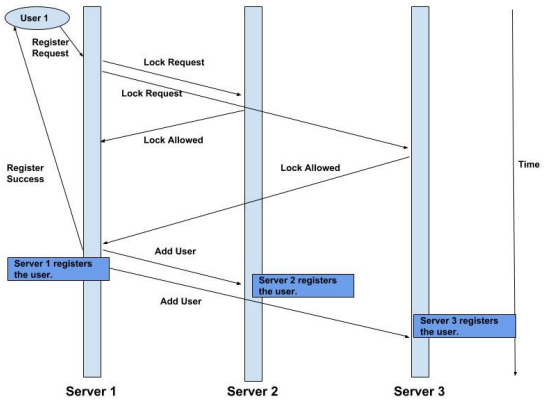

Diagram 5. Proposed Protocol for User Registration considering mesh network

Another possible way to reduce the message complexity would be to have a mesh network design *(Diagram 5.)*. Upon receipt of user registration request the server will broadcast lock request to all other servers, however, the lock allowed/ lock denied response would only be sent to the sending server instead of broadcasting. If the server receives lock allowed from all other servers it would then send a custom add user message to all the servers asking to register the user. With this model, the message complexity would be *(3m-2)* which is also in the order of *m*.