COMP90015
Distributed Systems
Project 1 report

Group: PerpetualMotionSquad

Members: Malcolm Karutz – mkarutz
mkarutz@student.unimelb.edu.au

Alexander Zable – azable
azable@student.unimelb.edu.au

Brandon Syiem – bsyiem
bsyiem@student.unimelb.edu.au

Ishesh Gambhir – igambhir
igambhir@student.unimelb.edu.au

1. Introduction

The aim of the project was to set up a distributed system involving interconnected servers (forming a tree), and clients which can connect to them. Clients can log in or register, with the primary goal of broadcasting "activity messages" to each other via the server network. The project posed many challenges. Perhaps the greatest challenge was debugging beyond unit testing (i.e. integration testing). Implementing and debugging load balancing and the registration locking protocol was not trivial, and required substantial prior thought and planning. We figured out that server connections were state machines of sorts, which helped to model our approach to the problem. Overall we feel as though we have a relatively faithful implementation of the specification, however further integration testing would certainly be warranted.

2. Server Failure Model

Handling graceful server crashes could involve extending the JSON protocol to include a SERVER_QUIT message, with which a receving server could knowingly terminate the existing connection. Ultimately a SERVER_QUIT message could contain the address/port of its "parent" server, making reconnection to the network trivial (see Fig. 1). The main edge case here is if the root server quits. In this case the root server could potentially designate one of its child servers as the the new "root", with which the rest of its children would have to connect to.

If a server can quit without warning, this complicates things. The TCP/IP protocol has a timeout period which would have to be reached before a remote server could be considered crashed. Then, we could potentially exploit SERVER_ANNOUNCE data to establish a connection to another server in the network. Although, because this is broadcasted periodically there's still a time lapse with which a server may be left "dangling". Perhaps most reliably, we could have a master server (or a couple for redundancy) which simply maintains the server topology. Any server can query it if it needs to reconnect.

Up until now we have assumed a fail-stop model. If we assume that a server may restart at the same address and port number, we could potentially keep polling it by attempting to reconnect continually. However, given that this is a *may*, we are probably best off connecting to another server in the network as described above, at least in the meantime. We could continue to check for the reinstatement of the crashed server if maintaining the original tree structure is of importance.

Potential server failure may compromise our user registration model. If a server quits, the tree is at least temporarily split into two sections. Hence a LOCK_REQUEST will only propagate within the respective section it is initiated in. Therefore in the case of eventual reinstatement of the entire tree, the local storages will potentially be out of sync. Ideally servers should synchronize their states in the event of server reinstatement/connection.

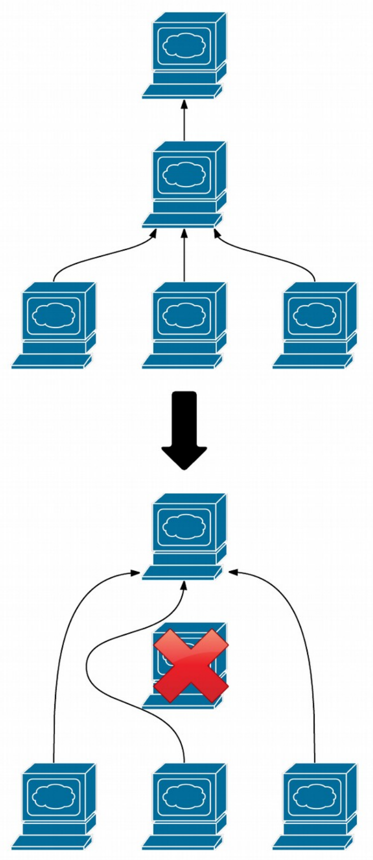
Figure 1. Server tree whereby a server fails gracefully, enabling its clients to connect to its parent.

3. Concurrency Issues

The protocol we implemented is far from flawless under rare edge cases. Let's consider two clients trying to register at the same time, with the same username and different secrets. One connects with a server and sends a registration request. The server it is connected to sends out a lock request that will eventually propagate to every other server after some delay. At the same time, on the "other side" of the tree of servers, the other client attempts to register with the same username but different secret. This server also propagates a lock request. In due course the lock requests sent will cause a clash, and a lock denied will be instigated for one of the username/secret combinations. However, until this happens there is a short time whereby one client could theoretically *log in* to another server which is only aware of *their* LOCK_REQUEST as yet. Thus, one user may be logged in as the other user who successfully registered (see Fig. 2).

A solution would be to keep a separate list of pending users on each server. The idea would be that a lock request adds a user to this list instead (naturally, users on this list cannot log in). When the server that originally got the register request gets a lock allowed from every other server, it could then send a final broadcast (i.e. FINALIZE_REG), which would cause all servers to move the user from the pending list to the registered list.

Another concern is that load balancing as currently implemented may not be able to handle many login attempts because of the delay between SERVER_ANNOUNCE commands. A surge of client connections to a particular server could result in most of the surge being redirected to another single server, which will then redirect most of the remaining surge again (and again). Some clients could experience many redirects before connecting. This issue may be controlled by increasing the redirect tolerance (from 2) so that only when a server has *substantially* less connections do clients get redirected to it.

Concurrency issues such as these may be exacerbated when we consider the pathological case of a series of servers which form a "line". A broadcast could take a very long time to propagate to every server. This would be unacceptable in most applications, and ultimately care needs to be taken when setting up the server tree.

4. Scalability

If we model the server tree as a graph, we can model a broadcast by counting each edge of the graph once (each edge "equivalent" to a single message sent). It is known that a tree with n nodes will have $n - 1$ edges. $n - 1$ messages are required for a single broadcast, making a broadcast an O(n) "message complexity" operation, where n is the number of servers.

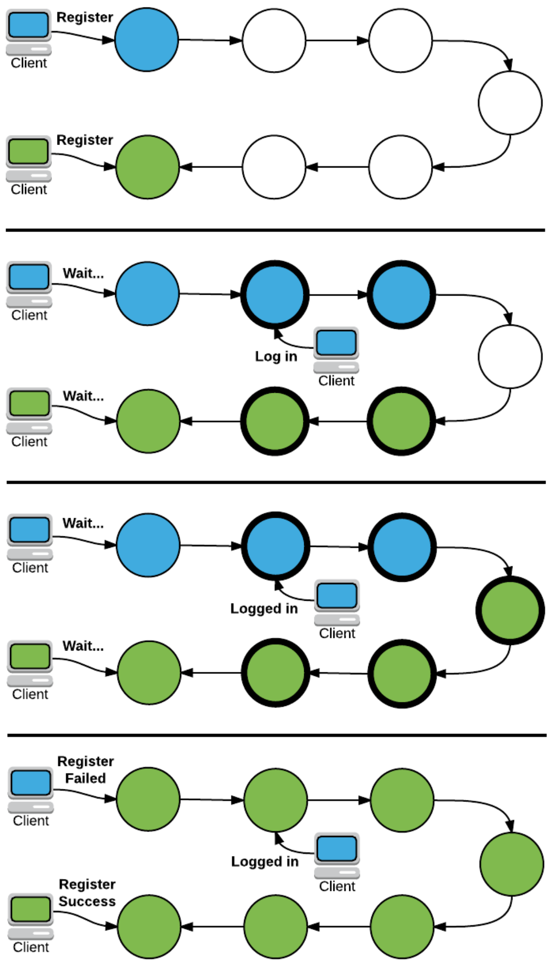

Figure 2. Two clients register the same username, one with the "blue" secret and the other with the "green" secret. The bold outlined servers are vulnerable to being logged into during the registration process with the respectively coloured secret. Note how a third client can theoretically log in with the blue secret, even though eventually the green secret successfully wins the "lock battle", and registers successfully.

Currently there are a few cases where all servers will broadcast to *each other,* such as the SERVER_ANNOUNCE every 5 seconds. Given there are n servers, this means there will be $n * (n - 1)$ messages sent throughout the network every 5 seconds; or in other words, O(n^2) messages per second. Furthermore, if we consider that a lock request propagates to every server, and a LOCK_ALLOW or LOCK_DENIED reply broadcast is given in response, this would also be quadratic message complexity.

Evidently this would result in an unsustainable number of messages as the server network grows. We propose a solution based on a specialised master server which maintains the network state (see Fig. 3). Periodically, servers update the master server with their state, which would be O(n) messages/second. When a client wishes to connect to the network, it first queries the master server, which determines the server with the lowest load. The master server then authorizes it to connect with that particular server (and directs it to that sever).

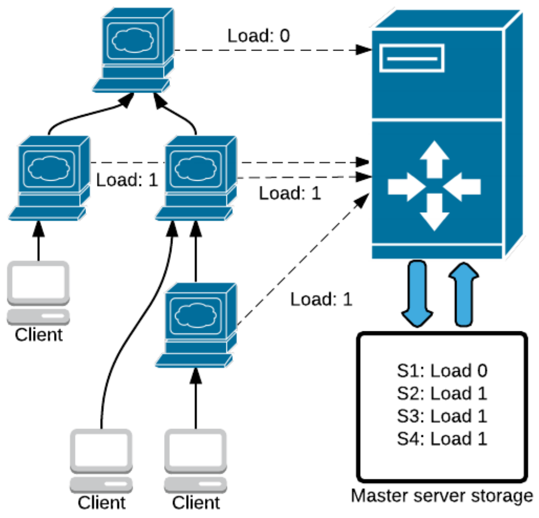

Figure 3. Each server in the network periodically updates the "master server" with its load. In this example a client wishing to connect would be directed to S1.

Therefore, status messages between all servers would no longer be necessary. Although, servers would also need to query the master server to initiate lock requests, unless the fix specified later on is used.

A natural downside of this approach is that a master server could potentially bottleneck the entire network, and would be a single point of failure. However in theory, a master server could be extended to become an abstract “service” which is a distributed system itself.

In terms of user registrations, this could similarly be handled by utilising a specialised user authentication server/serivice. But a more conservative solution would be to better exploit the tree structure of the network. A revised locking protocol could act recursively with servers only relying on lock information from their immediate neighbours, rather than *every* server in the network. For example, when a client attempts to register, rather than waiting for a lock allowed from every other server in the entire network, it could simply wait for a lock allowed from all its immediate neighbours, and this process could propagate recursively. Hence, the overall message complexity for a registration attempt would be O(*n*).