COMP90015
Distributed Systems
Project 1 report

1. Introduction

The aim of the project was to set up a distributed system involving interconnected servers (forming a tree), and clients which could connect to them. Clients would 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), since sound integration is fundamental to aspects of distributed systems such as security. For instance, we needed to ensure servers did not accept messages from clients that would otherwise come from servers (i.e. LOCK_ACCEPT). Implementing and debugging the message locking protocol to specification was not trivial either, 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 robust implementation, however further integration testing would certainly be warranted.

2. Server Failure Model

If a server can be assumed to quit gracefully, we would have an easier time integrating this with the existing system. A graceful quit 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 original server quits. It 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

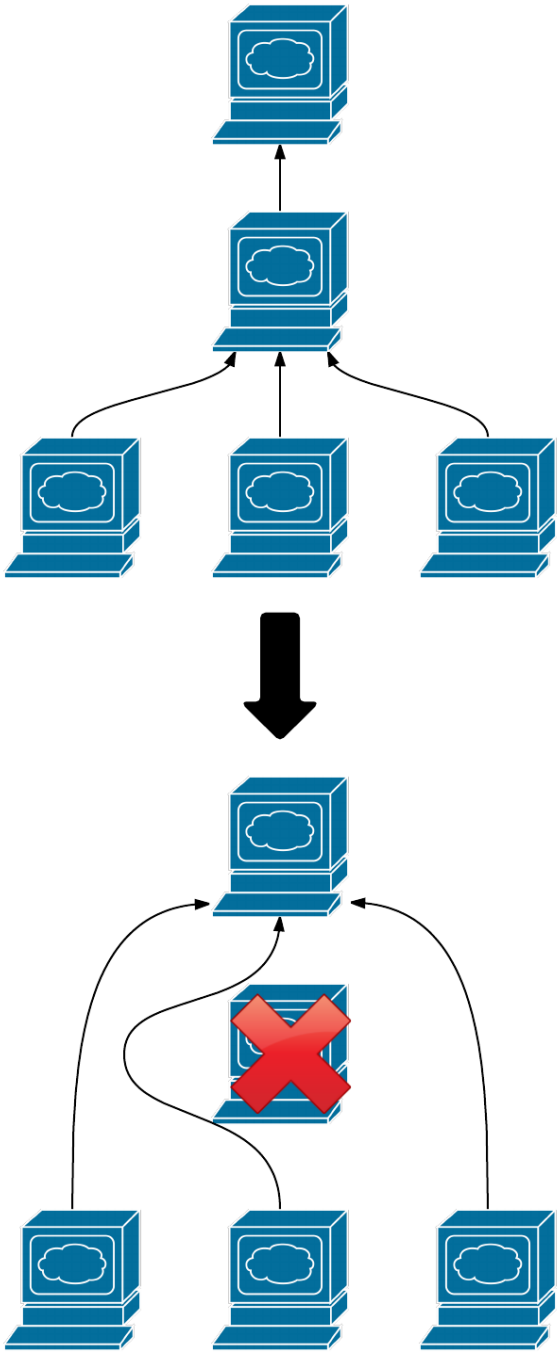

Figure 1. Server tree whereby a server fails gracefully, enabling its clients to connect to its parent.

reached before a remote server could be considered crashed. In this instance we could potentially exploit SERVER_ANNOUNCE data to establish a connection to another server in the network, although because this is broadcasted only periodically there's still a time lapse with which a server may be left “dangling”. Perhaps most reliably, we could have a couple of master servers (at least two for redundancy) which simply maintain the state of the server tree network, which any server can query 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.

An overt issue posed by potential server failure is that our user registration model may be compromised. 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 synchronise their states in the event of server reinstatement.

3. Concurrency Issues

The protocol we implemented is far from flawless under certain edge cases. For instance, if a client attempts to register with a server, the current implementation relies on the server being aware of every other server in the network. This is not guaranteed, given the SERVER_ANNOUNCE delay.

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. There is a time delay here of course. Now, at the same time, on the “other side” of the tree of servers, the other client attempts to register with the same username. This server also propagates a lock request. In due course the lock requests sent will both be denied and both users will receive registration failed commands. However, until this happens there is a short time whereby one client could theoretically *log in* to another server in the network using these new registration credentials (before it has seen a lock denied and removed the user from the local storage). A solution to this would be for every server to record “pending users” in their local storage, and not allow any logins for users on that list. Instead, when they receive a lock accept from every other server for a particular pending user, that user is finally locked in as a full user (with login capabilities).

Concurrency issues may be particularly exacerbated when we consider the different types of tree structures that can be theoretically formed. If we consider the case of a series of servers which form a “line”, and a LOCK_REQUEST is instigated from one end, it could take in the region of several seconds, if not minutes (depending on the number of servers and latency) to propagate to every server. This would be unacceptable in most applications. In a practical scenario, we would probably have to balance the tree using a similar protocol to that of clients being load balanced.

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.

In the present implementation there are a few cases where all servers will broadcast to *each other*. For instance, 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 (minimum), leading to a quadratic increase in the number of messages as the number of servers increases linearly. 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 a quadratic complexity operation.

Evidently, quadratic “message complexity” is going to be unsustainable for a growing network of servers. We propose a solution again centered around the notion of utilising specialised servers (i.e. master servers). The idea would be that all client connections initiate via the master server, which maintains the network state. Periodically, servers update the master server with their state, to keep things in sync. This would cost $O(n)$ messages per time period, where n is the number of servers. When a client wishes to connect to the network, it will first query the master server, which will determine the server with the lowest load. The master server will then authorize it to connect with that particular server, by supplying it with some sort of key. The client can then connect with that server by supplying the key (at which point the key is verified between the server and the master server). The result is that SERVER_ANNOUNCE between all servers is no longer necessary to enable load balancing.

A similar system could be employed for user authentication in order to remove the need for lock requests and responses. A separate specialised authentication server could handle user registrations and logins, using a similar key authorisation scheme to allow the client to connect to the relevant server and authenticate itself. Naturally one downside of this approach is that these master servers could potentially bottleneck the entire network if the number of requests they are processing gets very high. They are also a single point of failure in their raw form. However in theory, they could be extended to become abstract “services” which are distributed systems in and of themselves. At this point the scheme evidently becomes much more complex, and is certainly beyond the scope of this report.