Project Report 1 – Distributed Twitter Service

Course – CSCI 6510 – Distributed Systems and Algorithms

Submitted by – Akshay Bhasin (661208294) and Smit Pandit (661684337)

Introduction

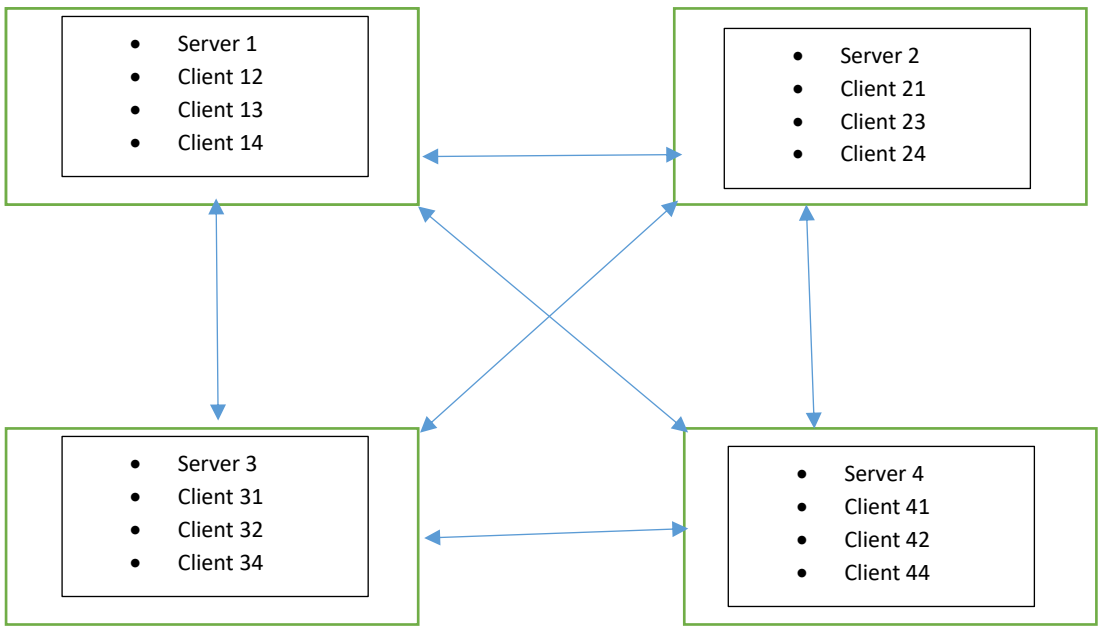
The idea behind this project was to get familiar with the concepts of distributed networks (their implementation using socket programming and multithreading) and implementation of Wu Bernstein algorithm on a Small scaled Twitter Application. We are given a system of 2 – 5 users connected virtually on Amazon EC2 platform. Initially, we have all users following all users. We were required to have four functionalities namely tweet, view, block and unblock. Each user was required to maintain its own copy of the log and dictionary and send it through Wu Bernstein's algorithm at every communication event.

Platform

We used Java for our project. We implemented the networking part using the TCP server and socket protocols. Although the Wu Bernstein algorithm works for message loss, there is no message loss in TCP protocol. Another method to do so was to implement the algorithm in the UDP protocol as it accounts for message loss. In addition, we used GSON library to implement log and dictionary and their storage. We use amazons EC2 to set up our users on virtual machines to communicate with each other.

Architecture

Since there are 2 – 5 users, and they all act as servers, we need to have different client threads connected to users on different sites.



As shown in the about figure, let us suppose we have n=4 users. Thus, each user will have its own server and (n-1) clients. When a user starts, its server thread will spawn up which will continuously listen to clients on the server socket. If any clients connect, it will spawn an individual client thread. The same process is repeated for all the sites. And at the same time, a client thread will spawn up which will keep on trying to listen to other servers. If a connection is made, it will also spawn an individual client threads which will keep on listening to the information on the server side.

Implementation

We first started a main function which spawns both the server thread and the client thread. These threads spawn independent serverClient threads and clientConnection threads respectively for each connection. Also, our main function needs an access to a synchronised array list which is shared between the spawning of serverClient (which holds the independent client connections) and the serverLog function which keeps listening on the server socket in an infinite loop. To remove this synchronisation issue, we implemented the array list using CopyOnWriteArrayList. CopyOnWriteArrayList is a synchronised way of implementing array list which makes different copies of the same list during different operation. This removes the synchronisation issue when serverLog function is adding a client to this list while our tweet function is accessing the list at the same time.

To implement the log and dictionary, we use gson library to parse and read json objects. We have implemented a function called writetolog which takes as input a string, converts it into json objects (events), and writes them onto a log file. Then, we implement a function readfromlog which reads the json objects and converts them into string to be sent across the socket. Furthermore, we have implemented a parsefile class which implements a method to parse a file for the list of users, their ip addresses, port number, and process id. Finally, we have a userinfo class which stores the important information of the current server which is running such as ip address, port number and so on. Since our clients our receiving continuously, we have implemented a log receive which updates our log and the event table.

How to avoid duplicate inserts?

In our implementation, we have maintained a dictionary which is a <Key, Value> pair. The key represents the username, and the value is an ArrayList of the users the current user has already blocked. Now, when an insert(x) or delete(x) operation happens, the dictionary is first checked and the ArrayList is compared to see if a previous block has happened. If it has happened, then we do not perform the insert operation. Otherwise we insert it. On the other hand, in case of delete(x) operation, we check to see if the name is present. If it is NOT, then we do not perform this operation as the user is already unblocked. Otherwise, we perform the delete(x) operation and the user can now view.

Why does it still solve the log and dictionary problem?

The log problem states that for an execution <E, -> >; for every event e, f - > e iff fR belong to L(e). The log problem is satisfied by removing duplicate inserts; an insert(x) operation occurs only if it has not been previously occurred. If it did, and there was not a log for it, it was the first operation. However, if there is a log then the node knows about that event and hence, the log is maintained.

The dictionary problem states that for an execution <E, -> >, for every event e, x belongs to V(e) iff insert(x) ->e and there does not exist an x-delete event g, such that g->e. It solves the dictionary problem as by removing duplicate deletes, it violates the definition that g -> e. Also, by having

duplication inserts, it violates the assumption of our model that every node can have at most 1 unique insert operation x.