Project 2 Report

Ni Zhang (RIN: 661411846)
Teng Liu (RIN: 661088506)

1. Introduction

In this project, we implemented a simplified distributed Twitter service using the Paxos algorithm to replicate tweet, block, and unblock events. We use C++ socket programming with multithreading to set up five sites. The sites are hosted in Amazon EC2, and connected through SSH.

2. Network Programming

2.1 Architecture

Since the system is fully distributed, every site runs the same code and behaves as a server as well as a client. A site can send messages to, as well as receive messages from, all other sites. We assume that each site knows the IP address and the port number of other sites. This information is provided by a "hosts.txt" file. TCP is used as the transport layer protocol. Five commands are supported in our implementation: "tweet", "view", "block", "unblock", and "viewlog". To execute a command, a user enters a command keyword, followed by an argument. For example, to tweet a message *m*, a user enters "tweet *m*". Tweet, block, and unblock events are replicated at all sites. Once an event happens at a site, the Synod algorithm is executed to determine which slot in the log this even will be written to. All types of events are serialized into strings before sending to other sites, and deserialized to a Paxos message object after receipt.

2.2 Multithreading

Every site can send and receive messages at the same time. This requires at least two threads. Once a site is initialized, two threads are created: one for sending serialized events, one for accepting connections from other sites. When a site needs to send message, the "startTweet" thread will try to connect other active sites, read in the standard input from the terminal, serialize the message, and send the message to the connected sites. When a site receives a connection, a third threads is created to handle the received message. In practice, this naive thread-based method is not scalable. However, since we only have five sites, scalability is not an issue.

2.3 Robustness

Our design is robust even when sites can crash. Each time a site needs to sent something, this site will try to connect other active sites and update the socket file descriptors when necessary. Then, the site will loop through all active socket file descriptors and send the message to the active sites. When a site is down, other sites will simply remove the corresponding socket descriptor, because when the other site tries to connect with this crashed site, the connect() function returns a negative number, which indicates a "connection failure". When a site becomes active again, the

other sites will know this information when they try to send messages to the unblocked sites.

3. Algorithm Implementation

3.1 Classes and Data Structures
There are five classes in our design. "Server", "Event", "Log", "Paxosmsg", and "MyThread". The log is a vector of "Event", and is stored in the disk that can survive crash. We have two data structures, “timeline” and “blocked users” that couldn’t survive crash as required. "timeline" is a list of "Event", which stores the tweet events from other sites who did not block this site. "blockedUsers" is a set of integer, which is the site that this user is forbidden to view the tweets. These data structures are rebuilt after coming back from crash by reading the log.

3.2 Implementation Details
3.2.1 simple synod
The simple synod we implemented exactly follows the algorithm. When one user executes tweet, block or unblock, messages are sent among the sites, including prepare, promise, accept, ack, and commit. After acceptors have received the commit, they will update the log and timeline. If the operation is block and the acceptor is blocked from the proposer, the proposer’s tweet will be removed from the timeline and new tweet it gets from the proposer will not be added to the timeline. When the acceptor is unblocked, the acceptor goes through log again to add back the proposer’s tweets in the timeline.

3.2.2 survive crash
When a site is up, it reads in the log from disk and rebuild its timeline and blocked users information from the log. Then it checks if it has missed any slots by asking other sites what the size of their accVal vectors are. After getting all the answers, the largest size is picked as the limit and then, synod is called for each one of the missing slots.

3.2.3 messaging optimization
We assume that the system is bursty. In order to decrease the number of messages, a distinguished user is chosen for each slot, who can skip the prepare and the promise messages. This distinguished user sends accept message to all acceptors, which results in three rounds of messaging instead of five rounds. The proposal of this distinguished user is always 0. The distinguished user for slot i is the user whose value is written in slot i-1.

4. Conclusion and Possible Extensions

The server handle incoming connections through a naive thread-based procedure. However, this method is not scalable. If there are more sites, we can use the thread-pool method, which is more scalable and manageable.