Lucien Christie-Dervaux December 3, 2017
Distributed Systems and Algorithms Project 2

Introduction & Algorithm:
PAXOS is a distributed algorithm to reach consensus on one value among N processes who each propose their value. The system model assumes N/2 + 1 processes stay alive to reach consensus, and that there are no Byzantine Failures.

By applying the basic algorithm over an arbitrary number of rounds of proposals, it results in the multi-PAXOS algorithm, which replicates a log among N sites. All events are ordered in the same way, and eventually all log entries get the correct value by consensus. Building on top of that, we can construct a dictionary by going over the log and adding blocks and unblocks.

The assignment for this project was to develop an implementation of this algorithm that could be launched from N processes, and could communicate via only TCP or UDP. In essence, creating a set of replicating "twitter" nodes, that would live on EC2 AWS instances.

Implementation Logic:
The algorithm I wrote implements paxos for each entry correctly. However, most of the novel work comes from the multi-paxos aspect.

At a high level, each process maintains a maxindex: the highest index that reached consensus that the process knows about. The next event entered in by the user will try to propose its event at log entry Maxindex+1.

Every time we get a commit message with a larger index number, we increment maxindex it to one greater than the index of that commit message, because it indicates that all below it have already been decided. If a round reaches consensus, and the node's proposal value is not chosen, it gives up on its proposal, and places the consensus entry at the index in the log. if If the user wants to send out their tweet, the user must try again on their own. I don't allow for a tweet to preempt the previous tweet's proposal; it must wait until consensus is reached. My asynchronous messaging system model, accounts for any kind of message being received by any node while the above is happening.

System Design:
My algorithm was written in python, using UDP as the message protocol. Design wise, the "front-end" was a constantly running thread that accepted Tweet, Block, Unblock, and View messages, which communicated with a client class, which encapsulated a communicator(middleware), a proposer and acceptors. The latter two could communicate with a storage class, which would accept atomic changes from setter methods and ensure that the log and all other important PAXOS variables would survive crash-failures. Finally, I had a message class, whose objects would be passed between many functions and would be translated to and from JSON so it could be passed into a UDP packet to other processes.

Client:

The Client would accept messages coming in from the communicator middleware, sort it to the proposer or acceptor, and would also send out messages from those two back to the communicator, which would broadcast or send a direct message depending on the type of message.

Proposer & Acceptor:

The Proposer and Acceptor both have recvMessage methods to accept different kinds of messages, as per the algorithm. When those messages were ingested, they would modify the state of each class, atomically, and on disk, and return a new message in response. Those messages sent were Proposals, Promises, AcceptRequests, Accepts, and Commits.

Testing:

I ran a few tests to make sure that the paxos algorithm worked, and it seems functional and robust. Despite using UDP, I haven't yet had a missing packet, and even if there was, the worst case is that there might be an empty log entry among a subset of the processes. However, there is no risk for conflicting entries.

Viewing Command/Blocks

When it comes to sending block messages, they will be sent regardless of who is blocked, unlike the Wuu-Bernstein Algorithm. Instead, they will not be able to be seen on the client's end, by iterating over each tweet, and printing those whose site id isn't blocking your own site id.

Recovery:

On recovery, the block/unblock dictionary is recovered by replaying the log inside the Client, and you can inspect the internals of the algorithm by using the commands (blocks|view_log|data).