Distributed Algorithms Project 2

Neil McGlohon
Mark Plagge

December 6th, 2015

1 Introduction

We are attempting to solve the replicated log/dictionary problem using PAXOS with distinguished proposer election protocols. What this attempts to accomplish is that when one node adds something to the collective replicated log, that it is safe to do so. We accomplish this task by having proposals include entire logs/calendars. Upon the receipt of a log/calendar from a proposal, the leader (the arbitrator over what is allowed in the log or not) can compare the proposal number of the proposal If the proposal number is too small, that means that it is not safe to use the proposals version of the log as the new version as it is not guaranteed to include everything in the most recent version of the log.

2 PAXOS: A Closer Look

How this works is that when a node wants to add something to the collective replicated log, he first creates a version of the log with what we wants in it. Then the node sends the message to the leader/distinguished proposer to act on his behalf. If the log is out of date, a result is returned to the initiating node that he failed to add his contents. This is accomplished by the leader: he polls the network and asks "Is this safe/up to date?" and then responds to the initiator whether he has permission or not. This last statement is probably the most important sentence of this report. It holds the meat of the PAXOS algorithm:

The leader asks everyone on the network: "can you guys use this version of the log?", if he receives a majority response of "yes I can promise to use that version of the log" the leader will then initiate an accept broadcast. This is created by taking the value of the largest promise number value from the other nodes and using that in the proposal from here on out. *That is, unless,* the other nodes haven't already promised something to someone else. Otherwise the leader can use the initial value as what will be the new version of the replicated log. He sends an accept message back to all other nodes telling them "hey, this is a good thing, you prepared to commit it?" and wait for a majority of them to respond "yeah, hit me!" to which the leader responds with "commit this proposal to your logs" and everyone rejoices as paxos completed, the leader sends a result to the initiating proposer node "success!" and the process repeats itself ad infinitum.

3 Design

We have several threads on each machine. We have a main thread which spawns a Client. This Client is what spawns additional threads that each contain Proposer, Acceptor/Learner, and Leader. The Leader thread is just a representative object that speaks for the machines behalf in all leader election processes. Additionally we also have server threads that take care of any TCP/UDP communication with other machines. We use python queue.Queue(), a thread-safe queue for the handling of all interprocess communication. Each thread has several queues depending on the nature of the data put in them. They have inboxes and outboxes that the network communication threads monitor to know when to forward messages to the network - Proposers have queues that receive and give input to and from the client such that when the user wants to add/delete an event, the machine's proposer can bundle it up and send it to the designated leader to initiate PAXOS. Because we use these threadsafe queues, the network inbox required a little bit of extra design to get working smoothly. We designed a Grand Central Dispatch in its own thread that constantly monitors the network inbox, it deserializes messages, checks the type of message that it is and depending on said type will place the message in the corresponding actor's inbox (Proposer/Acceptor).

4 Implementation

A few important things to note from our implementation. We use broadcast for the following types of messages: PREPARE, ACCEPT, COMMIT. This means that we don't send to only a majority, we send to everyone and wait to hear back from a majority (with the exception of COMMIT - there is no wait for response here). We made this decision on the case that if we only send to a majority, it only takes one message loss to halt the entire process of PAXOS. We decided that we would like to give PAXOS the best shot that it has at succeeding. We recognize that this could lead to a larger amount of messages sent in a large network, however. All PAXOS network data is exchanged through UDP.

4.1 Leader Election

We use TCP for leader election messages in a Bully algorithm. When a node starts up, it does not know who the leader is so it starts a leader election, it pings all nodes above it with an "ELECTION" message. If they are awake, they respond "OK" to tell the initial node that "Stop your election, I'll handle it from here" and then they'll start their own election, sending "ELECTION" to all nodes above it (which then respond OK if they're awake). If a node doesn't hear "OK" back in a considerable amount of time, he can assume that he is the leader. It then sends "I am the leader!" to all nodes in the network.

5 External Libraries Used

1. We used our Calendar and Calendar Event classes from our previous assignment
2. datetime
3. numpy
4. pickle (python serialization)