Distributed Systems
Project II Report

William Rory Kronmiller

December 8, 2015

Introduction

This project implements a variant of the Paxos algorithm to create a distributed log with ordered entries.

The Paxos algorithm ensures a value committed to a distributed system is consistent across all storage nodes. Paxos ensures this by using a multi-step commit process beginning with a Prepare/Promise phase, followed by an Accept/Ack phase, and finally a Commit command. Paxos uses Proposer processes to compete for the right to commit a value and Acceptor processes to mediate the competition. The winning Proposer commits its value and the losing Proposers can also commit the winning value (but only the winning value).

This project applies Paxos to each index of an ordered log in such a way as to enable the creation of a state machine. The state machine, in this case, is a simple calendar program.

In this particular implementation of the project, each log entry corresponds to a self-consistent calendar entry.

High-Level Design

Build Environment

The project was written for the Java 8 run-time (and is incompatible with Java 7 and below), and was developed in Eclipse v4.5, as a Maven project. A Javadoc has been generated, and is in the root of the workspace. The main page can be found at `index.html`.

Packages

The project is divided into a tree of namespaced packages, the root of which is `net.kronmiller.william`. The `.acceptor` package contains the classfiles most relevant to the Paxos Acceptor, and similar rules follow for the other packages. It is important to note that the `.messaging` package contains classes relevant strictly to UDP messages. TCP connections are handled within `.proposer` classes.

Important Class-Files

`Sender.java` and `Receiver.java` contain generic classes responsible for serialization and de-serialization of UDP messages, as well as thread synchronization and queue management, as relates to UDP messaging.

`Acceptor.java` contains the class responsible for most of the logic behind the Paxos Acceptor process, and `Proposer.java` does much the same for the Proposer process.

`LeaderManager.java` contains most of the classes responsible for leader election, as relates to determining the distinguished proposer.

`UserInterface.java` contains the logic driving the text UI, and is more-or-less a modified version of the UI from the WuuBernstein calendar project.

General Control Flow and Synchronization

Overview This system relies primarily on asynchronous messaging over UDP for most communication. The program uses an event-loop model for most message handling such that one can think of the main threads of the system as JavaScript-style programs with static callbacks.

Messaging The UDP messaging, itself, is handled almost entirely by dedicated threads with synchronized queues. When a message is sent to a new node, a persistent thread is spawned to handle all future messages to that node. In a sense, the program implements pseudo-connections over UDP. Typically, there will be a many-to-one relationship between Senders (many) and Receivers (one). As such, each Sender can be treated in a similar manner to that of a TCP socket.

Leader Election Leader election is carried out by way of the Bully Algorithm. Every Proposer has a list of the hostnames of all Proposers in the network. The node's ID is determined by the hostname's position in the list. The ELECT/ACK, COORDINATOR, and leader heartbeat messages are sent on separate TCP ports. Leader election is managed by the LeaderManager class, which uses an event-loop-based flow for message-handling and follows a naive implementation of the Bully Algorithm.

Paxos Events

Proposer The majority of the project's complexity is in the Proposer class, which implements a naive variant of Paxos. On startup, the Proposer will send a PREPARE message for every log entry, starting at the first index. As follows from the Paxos algorithm, if no PROMISE is returned, the Acceptor will increment its proposal number until it receives a PROMISE (note: the proposal numbers for different log indices are independent). Once the Proposer receives a PROMISE, it caches the accNum and accVal in a map associated with a specific Acceptor and log index. Once a Proposer receives a PROMISE from a majority of the Acceptors in the system, it will check if any accVal's are non-null. If all accVal's are null, then the Proposer records $maxIndex = max(maxIndex, logIndex)$ so that the largest index for which a PREPARE was sent and no existing accVal exists becomes the next log entry to write to. If any accVal is not null for a log entry, then the Proposer will proceed to the Accept and Commit phases, even if the value has already been committed. The guarantees provided by Paxos ensure that the same value will always be committed each time.

Note A Proposer will only perform Proposer operations if it has been elected leader. If a Proposer starts up, becomes, leader, loses an election, then resumes being leader, then the Proposer will use PREPARE messages to find the new end of the log, starting from its last known end-of-log.

Acceptor The Acceptor is a relatively simple process, which very naively implements the Paxos algorithm. PREPARE messages with low proposal numbers are ignored. No optimizations are implemented. The Acceptor stores the `maxPrepare`, `accNum`, `accVal`, and committed values in the same file as JSON.

Calendar and Correctness

Each log entry contains a complete calendar. That calendar is constructed using a Calendar class with methods for adding and removing appointments. The add and remove methods include checks to ensure that only non-conflicting appointments can be added to a calendar and only existing appointments can be deleted. As the Calendar class is effectively the only way to create or modify a calendar in the program, each calendar in the system is guaranteed to be self-consistent (contain no conflicting appointments) so long as the logic behind the conflict-checking is valid.

On startup/election, the Lead Proposer performs the aforementioned PEPARE cycle to get the latest calendar and the next empty slot in the log. The latest calendar is cached on the lead proposer, in what amounts to an optimistic UI or a write buffer. Changes from the user interface are immediately reflected in the lead proposer's calendar (assuming the changes are not rejected by the calendar's internal conflict-checking logic). Just because a change shows up in the connected UI's, however, does not mean that a change has been committed to the system. As mentioned before, most of the program is implemented using non-blocking event loops. When the Lead Proposer's cached calendar is modified, so that it is no longer identical to the latest committed calendar, the Lead Proposer marks its cached calendar as "dirty." Once the Lead Proposer reaches a check phase in its event loop, it checks if the cached calendar is "dirty." If "dirty," the Propser sends a PREPARE message and moves through the phases of the Paxos algorithm to commit the changed calendar to the next entry of the log. The cached calendar is then marked as not "dirty".

In theory, a network partition could lead to two Lead Proposers running at the same time. Indeed, the multi-threaded architecture employed by the system means that a Proposer that has recently lost an election could, for a fraction of its main event loop, continue to behave as a Lead Proposer. The result would effectively be a loss of Safety. The two active Proposers could cache different events to their respective calendars. The Paxos algorithm, however, ensures that only one calendar will survive. The Proposer to win the next log slot would have its calendar committed and the Proposer to lose the conflict would, the first time it tried to add an entry to the log, detect that the log had been modified by another Proposer and replace its cached calendar with the latest calendar in the log. This is done using a logMax value that records (in RAM only) the last detected end of log. If the log entry corresponding to logMax is not empty, then the Proposer essentially re-starts its PREPARE cycle from the (erroneous) logMax.

The effect is that a calendar benefits from the Safety guarantee only once it is accepted by a majority of Acceptors. At that point, Paxos ensures that no conflicting calendar will be committed to a log slot. Before a calendar reaches that point, it is essentially a candidate or, more accurately, a buffer. It would be trivial to modify the Proposer class in this project to display only the appointments in the last committed calendar, but the current design allows the UI to display the appointments in the cached calendar, instead.

This makes sense, in part, because conflict resolution is performed against the cached calendar for additions and deletions (by the nature of the Calendar class implementation). The current implementation allows the Proposer to handle multiple requests for changes (or requests for multiple changes) within one Paxos commit. The drawback is that all those changes could be lost if the Proposer crashes or loses in a conflict to another Proposer. An alternative implementation could have required the Proposer to block or buffer requests for changes while committing a calendar and requiring the Proposer to commit after every change. Such a modification would not be totally trivial, but would not require any major changes to the structure of the program.

My thinking was essentially that a request for a calendar change can always be lost, which is equivalent to a request for a change succeeding but having the calendar be lost before it is accepted/committed. The only differences are that, in the current implementation, the UI will show the change as if it had already succeeded and conflict resolution on subsequent requests will be performed as if the change had already succeeded. There is still no possibility for conflicting appointments to enter the same calendar, due to the aforementioned structure of the program. The only practical risk is that a partition will cause all the cached changes made to one of the two partitions to be lost when the system comes back online.

By default, Paxos ensures the CP of CAP. This implementation gives the appearance of AP, but the effective result is the same as CP in that one partition's changes are lost.

Citation

This YouTube video was used as conceptual inspiration for the program design:
`https://www.youtube.com/watch?v=JEpsBg0AO6o`