DSA Project 3

Mitesh Kumar (kumarm4), Jason Lam (lamj7)

Project Design and Implementation

Data structures:

The log was stored in a hashtable that maps each log index to the log slot. Each log slot is stored as a mapping from site id to the value that the site has accepted for that log slot. We also store a mapping from each log index to proposer and acceptor states for those log indices.

Stable storage implementation:

The proposer, acceptor, and log state are saved to stable storage each time we need to send a message and the state was modified during the time between the last save and now. We assume that crash failures do not interrupt the save subroutine. We do it this way because we have assumed our algorithm guarantees safety in spite of message loss. Therefore, if we save before sending a message then if we crash before sending the message and before saving, it is as if the message never even got to us in the first place.

Use of sockets:

The proposer, acceptor, and learner each have their own receiver socket bound to their own port. All three share the same socket for outbound messages that is bound to another port, so in total, we use 4 ports.

Use of threads:

The acceptor and learner run on their own separate threads.The proposer runs on the main thread so that the UI can block while the synod algorithm for the next highest log slot is running. The proposer also spawns background threads to run synod to fill log holes in parallel. We utilize go's waitgroup synchronization mechanism to allow the proposer to wait for all log hole fill attempts to terminate before proceeding. We store a temporary thread safe "mailbox", implemented as a map from log indices to go channels, for each log slot that has an active proposer synod thread running. This allows the proposer socket to deliver messages directly to the proposer mailbox of a particular log index, and the proposer synod thread can read it when it is ready to do so. Using multiple threads means we have to utilize mutual exclusion when reading or writing to our hash maps, even if the key itself is guaranteed to be accessed by only one thread due to the underlying map implementation.

Learner Implementation

Description of how you implemented the Learners:

We made each site a learner, (Not the distinguished learner approach). We structured our log as a map[LogIndex] map[ProposalNumber] map[site id] accepted_val. Therefore, our log is just a collection of all of the received **accepted** messages for each log slot. A value is considered by the learner as chosen when a majority of acceptors have accepted that value for the same proposal number. The learner adds an entry to this structure whenever it receives an "**accepted**(acc_num, acc_val)" message from an acceptor. The acceptor sends **accepted** messages to all of the learners whenever it accepts a value

Why you chose this particular method?

We chose this method because when a site crashes, it can recover and retain knowledge of all the accepted values it has received so far. So learning the value will not take as many messages. It is also more straightforward and more effective to implement our devised recovery algorithm with this approach.

Recovery Algorithm

Description of your recovery algorithm:

The recovery algorithm runs when a process starts up. This could be when it has started up for the first time, or if it starts up after crashing. We attempt to load the proposer, learner, and acceptor state from disk. Afterwards, the site's highest learned log index is X. The number of learned log indices is Y. Therefore the number of unfilled log slots is X - Y, assuming we are using 1 indexing. The algorithm first attempts to fill the log slot using the synod algorithm. Our synod algorithm's proposer allows us to use nil as a proposal value. If this is the case, we only run phase 2 of the algorithm if we get at least one promise from an acceptor with a non-nil accept value. Otherwise, we declare that the synod attempt failed. This approach is okay because it is indistinguishable from the scenario in which we do execute phase 2 and all messages simply get lost. Therefore, it is safe to start phase 1 with a nil value without the intention to go onto phase 2. We first fill all of the log holes concurrently by running the synod algorithm with a nil proposal value. If any holes still remain, we try to fill them with a modified recovery protocol where we ask all learners to relay all of the **accepted** messages that they have received for each of our missing log slots. The recovering learner then uses

these **accepted** messages as if the acceptor had sent them directly to us. This guarantees safety because as far as the recovering node knows, the **accepted** messages that were relayed were already in transit to us and are just taking a really long time to arrive. If all holes were successfully filled, then we try proposing a nil value for the next highest log slot until finished recovering. Just as the regular run of synod. If we were able to fill all log holes, we then try to fill the next highest log slot with the same protocol. If that fails, then we complete the recovery process, otherwise we repeat.

Discussion of when/how it guarantees the Paxos safety and liveness properties

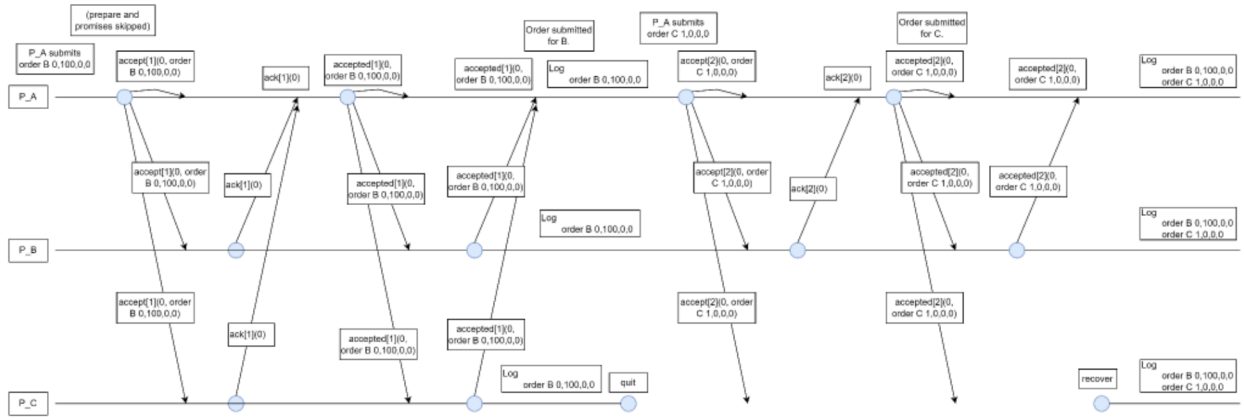
This recovery algorithm does not guarantee liveness, as paxos itself does not guarantee liveness. We can potentially go on forever if we were to utilize unlimited synod attempts, and it could potentially take unlimited attempts to fill the log slot if we keep failing at the prepare or accept phase, because another process is also recovering with the same set of acceptors, and keeps intercepting our proposal numbers with even higher ones. This algorithm would satisfy liveness if only one process makes proposals, that one process never fails, and a majority of the processes are active at all times. Our recovery algorithm's goal is to learn missing log slots that have already been chosen by the system. For accomplishing this goal, the algorithm guarantees liveness when at least one process is alive and reachable that has learned each missing log slot, since it can just relay its own **accepted** messages to the recovering process, because after an exchange of **recovery** and **accepted** messages for each missing log slot, the replica will learn all missing values. The recovery algorithm guarantees safety because from the learner's perspective, relayed accepted messages aren't distinguishable from extremely delayed accepted messages that were sent by the originating replica. We also can guarantee safety for any amount of downed processes, but progress is impossible unless a majority of processes are alive.

Testing:

Order, Crash, Order Again, Recovery:

User Input: (after one order is placed to skip prepare/promise)

order B 0,100,0,0
on site C, quit
order C 1,0,0,0
restart site C and log



This test verifies that orders are still able to be filled despite one crash (wouldn't be able to fill if two sites crashed however). This also verifies that the log is correctly preserved on the recovered site, despite the order being confirmed while the site was crashed. Note, the recovery part is skipped for sake of simplicity.

When testing, we did not set precise message orderings, but we did add a random delay to each message send on the sender side. We have a send helper that gets called on another thread, and the code continues as normal, in order to mimic variable messaging latencies.

We also tested much more hectic scenarios where we start multiple proposals across several sites simultaneously, and crash/recover several sites while they are in the middle of their phase 1 or phase 2 to verify that our stable storage implementation works.

Team Work

Time spent: 20 hours total amongst team members

- Mitesh: Developed working application of assignment, testing, comments, and report
- Jason: Helped with testing, and report