There is Less Consensus in Egalitarian Parliaments

KATIE LIU, Massachusetts Institute of Technology
KOSI NWABUEZE, Massachusetts Institute of Technology
FREDERICK TANG, Massachusetts Institute of Technology

1 INTRODUCTION

This paper serves as an experience report on implementing the Egalitarian Paxos consensus protocol. Our main objective for this report is to compare the performance of our implementation of EPaxos to Raft, a popular distributed consensus protocol. The benchmarking approach we used in this paper compares execution latency and throughput of both protocols. Our benchmarks contrast with the original paper which focus on commit latency instead of execution latency. Our evaluation shows that our particular implementation of the EPaxos algorithm performs quite poorly compared to our Raft implementation. Our GitHub implementation is available at https://github.com/kosinw/epaxos.

2 DESIGN

The main difference between EPaxos and other consensus protocols is that every replica can act as a leader. If a client requests a command *cmd* for replica L, L will now be the *command leader* for *cmd*. L will assign cmd to an instance, which includes all information about the entry: the command, the instance position (in the format $L.i$ with i being a monotonically increasing number across the lifetime of L), a list of which other instances $L.i$ depends on, etc. It will then initiate the commit protocol for the instance.

The command log in EPaxos for each replica consists of all instances from all command leaders. The log is two dimensional, the first dimension has an index for each replica L, and the second dimension includes all instances which L created.

2.1 Commit Protocol

In order to determine the order of commands to commit and execute, EPaxos utilizes the commit protocol. When a replica L receives a request for a command from a client (at instance number i), it becomes the command leader for that command (γ). It assigns several attributes to γ: dependencies (*deps*), a list of all command instances that interfere with it, and a sequence number (*seq*) to break dependency cycles.

The command leader forwards the command to the rest of the replicas, entering the pre-accept phase. Depending on the number and contents of the replies it receives, the command either takes the *fast path* or the *slow path*.

2.1.1 Phase 1: Pre-Accept Phase. Upon receiving a `PreAccept` message, each replica updates *seq* and *deps* for its copy of the corresponding instance and replies to L with updated *seq* and *deps*. If L receives a fast-path quorum of replies with matching *seq*s and *deps*, the command instance proceeds to the commit phase. When there isn't much contention, this is intended to speed up the commit process. Else, L updates the command's attributes with the maximum seen sequence number and unions the dependencies from all of the replies and sends `Accept` messages to all replicas.

2.1.2 Phase 2: Paxos-Accept Phase. Upon receiving an `Accept` message, each replica updates the status of command $L.i$ and replies. If L receives a majority of successes, it commits γ at instance $L.i$ and sends Commit messages to all replicas.

Authors' addresses: Katie Liu, Massachusetts Institute of Technology, ftang@mit.edu; Kosi Nwabueze, Massachusetts Institute of Technology, kosinw@mit.edu; Frederick Tang, Massachusetts Institute of Technology, ftang@mit.edu.

2.1.3 Phase 3: Commit Phase. Upon receiving a `Commit` message, each replica also commits command $L.i$ and it becomes safe for execution once all of its dependencies have been executed or will be executed at the same time.

2.2 Execute Protocol

To respect dependencies, EPaxos contains an algorithm for ordering execution of operations. Because the dependency graph can have cycles, the algorithm first finds all strongly connected components in the graph. Once found, the algorithm will perform a topological sorting of the SCC graph and execute the SCCs in that order. All instances in the same SCC are executed according to increasing sequence number.

2.3 Explicit Prepare Protocol (EP)

2.3.1 Algorithm. If the command leader for an instance fails or is partitioned away from the majority or crashes before fully committing the instance, it is not clear which replica should take on the workload. Thus, EPaxos introduces a protocol called *Explicit Prepare* for fault tolerance. When a replica R believes the command leader to have failed for instance $L.i$, R will attempt to assume leadership of the instance. It does this first by incrementing the ballot number of the instance (explained below) and sending `Prepare` RPCs to every replica (including itself). Upon receiving a `Prepare` RPC, a replica M will only accept it if the the new ballot number is at least as large the ballot number M has currently recorded for $L.i$, including dependencies, sequence number, and status (pre-accepted, accepted, committed, or executed). If M accepts the `Prepare` RPC, it replies with the current state of $L.i$ in its log. If R receives no rejects and a majority of accepts, it uses the received information to continue the protocol according to an algorithm presented in the original paper. If R received at least one reject, it will stop the Explicit Prepare process.

2.3.2 Ballots. Every instance has an associated ballot number in each replica's log, in the format $b.R$ where b is a number and R is the replica who last modified/requested information about the instance. All RPCs will contain the ballot number in the request arguments. If a replica receives an RPC with a lower ballot number for an instance than it currently has recorded in its log, it rejects the RPC. Ballot numbers are compared first be b then R. This solves the issue if multiple replicas perform EP on the same instance simultaneously, or if the old command leader still believes it is the command leader. The one with the highest ballot number will be the only one who eventually commits. Note that there are edge cases where the old command leader might have already committed but in all these cases the actual committed command will always remain the same.

3 IMPLEMENTATION

3.1 Commit

We implemented an RPC for each phase of the commit protocol, namely `PreAccept`, `Accept`, and `Commit`. Note that each replica maintains an entire two-dimensional log indexed by replica and instance number, representing the complete history of operations.

There were several design decisions we had to make in our implementation of the commit protocol, especially regarding areas that were underspecified in the original paper. In order to determine whether two commands interfere, we chose to take in a function, `interferes`, from the client to perform this detection rather than directly comparing keys (given our goal of implementing a key-value service on top of EPaxos) to allow for future generalizability. We also assume that every instance is dependent on the previous instance in that replica.

The method for making the initial dependency set was not specified in the paper. In order to calculate the attributes for a command at a specific index in a replica, we looped through all

instances in the replica's log backwards in order to find the latest instance that interferes with the command, since that instance's dependency graph would contain all previously interfering instances.

Lastly, replies to stale/old RPCs is not specified. Each time a replica receives an RPC with a lower ballot number, it replies false and rejects the message (just like a term in Raft!). For each round of messages, we spawn a goroutine for each peer. To synchronize the responses, we maintain a condition variable that is broadcast at the end of each goroutine in order to continually check for any rejections. When an RPC times out, we continually retry until receiving a response. Any time an instance receives an RPC and is already in a further phase, we automatically reply success.

3.2 Execute

We chose a modified version of Tarjan's SCC algorithm to create the SCC graph. In the DFS step, we wait for any command we are iterating through to be committed first. An SCC is found in Tarjan's algorithm only after all SCCs that it has directed edges towards (depends on) are found. Thus, immediately after determining each SCC we sort the instances within the SCC and execute them in increasing sequence number order. A goroutine run on each replica will continously run the SCC algorithm over that replica's log.

3.3 Explicit Prepare

The Explicit Prepare protocol given in the paper was ambiguous at certain parts. One mechanism it did not specify was how a replica can decide if a command leader failed. In our Instance struct, upon receiving a command in the log for the first time, we set a Timer field to be equal to the current time. A background goroutine called `ExplicitPreparer` runs on each replica continuously and loops through all non-committed instances in the log. The loop checks if the difference between `time.Now()` and the timer field exceeds a timeout threshold. If it does exceed this threshold, the `ExplicitPreparer` starts Explicit Prepare on the instance. To prevent a replica from preparing an entry that it is currently the command leader for, we set the Timer field to `nil` an instance if a replica is the command leader for that instance. `ExplicitPreparer` will never initiate Explicit Prepare on an instance with a `nil` timer. Note that this creates a problem if a command leader has not finished committing an instance but crashes, since it will not ever redo that command through Explicit Prepare. Thus, upon recovery, all non-committed entries in a replica's log have their timers set to `time.Now()`. Although not specified in the paper, replicas may need to do Explicit Prepare on instances they have not even received before. Below outlines a potential case that may arise.

1. A replica R receives an Accept RPC for instance $L.i$.
2. The RPC includes a dependency set for $L.i$ which contains $L.j$.
3. R has not received any RPC's from L for $L.j$ yet. Before L can send out *any* RPCs for $L.j$, L fail.
4. Because $L.i$ depends on $L.j$, it cannot execute until $L.j$ is executed.
5. However, $L.j$ will never be even committed since no other replica received any RPCs for it.
6. Thus, R actually needs to do EP for $L.j$ even though no replica actually has the command for it.
7. In this special case, a no-op command will be committed. Once this happens, the execution algorithm can execute $L.j$ and then $L.i$

Therefore, each replica must put blank instances into their log if it receives a dependency set that includes that instance but does not have a log entry for it yet. Each of these blank instances will have their Timer field set, so eventually Explicit Prepare will be called even if the command leader

never sent RPCs for that instance to the replica. We made the command −1 for these blank instances to signal that a no-op should be committed.

3.4 Persistence

Although the paper did not mention persistence, implementing it was necessary for crash recovery. The only persistent state we store for each replica is its *nextIndex*, which is what the index of the next command it receives would be, and its 2D log. Similar to Raft, we did this by using a `Persister` object passed in from the client. We call `persist()` every time we update part of a replica's persistent state and `readPersist()` upon restart to recover a replica's persistent state. Every time a replica is restarted, we also replay its entire log to ensure that its state is consistent with the correct state agreed upon by the other replicas.

3.5 Testing

We wrote several tests using Go's testing package to verify the functionality of the individual components of our system such as commit, execute, and Explicit Prepare. In addition, we ported many of the test cases from labs 3B and 3C. We also wrote many of our own tests to test edge cases present in EPaxos which are not present in Raft.

3.6 Key-Value Service

Similar to lab 4, we built a fault-tolerant key-value service on top of EPaxos to serve get, put, and append requests to clients. A key difference is rather than keeping a list of the most recently applied operations, we maintain a set of already applied messages (which takes the form of a hash set of `LogIndexes`, which consists of a replica number and instance index).

4 EVALUATION

Our experimental setup ran on an Apple M2 chip with 16GiB of RAM using a simulated test network forked from `6.5840/labrpc`. For the experiment we compared a key-value service built on top of Raft, called `kvraft`, to an externally equivalent key-value service built on top of EPaxos, called `epaxoskv`. Each experiment runs with five replicas and five clerks for one minute. Each clerk executes append operations continuously in a loop. Our evaluation features five workloads: 0%, 25%, 50%, and 100% conflict rates. Conflict rate describes the probability that two executed operations use the same key. For Raft, conflict rate should not affect the performance meaningfully since the protocol is completely agnostic to the log operations that are being replicated. However, EPaxos must track which log operations interfere with each other. Large amounts of interference increase latency and decrease throughput since execution time becomes desynchronized with commit time in replicas due to the overhead of additional round-trip time delays.

Figure 1 shows an experiment to show basic execution latency in both key-value services. Each bar plot represents the median latency, the error bars show the interquartile range. EPaxos performs poorly compared to Raft even when 0% conflict rate when it comes to this particular workload. As expected, as conflict rate increases, the latency increases as well due to added delay of waiting between commit latency and execution latency at each replica.

Figure 2 shows the results of another latency experiment, this time taking into account wide-area network effects. The clerk code in both key-value services does not take advantage of using replicas that are lowest in terms of latency which hurts the performance significantly for all workloads. We modified the `6.5840/labrpc` code to take into account differences in latency between certain replicas and clerks. If we were to implement the optimization mentioned above, we would expect the lower conflict rate EPaxos workloads to outperform Raft.

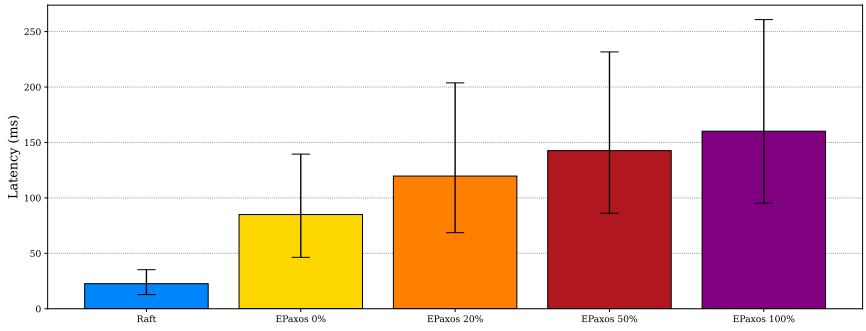

Fig. 1. Comparing clerk-observed execution latency for EPaxos across different contention workloads.

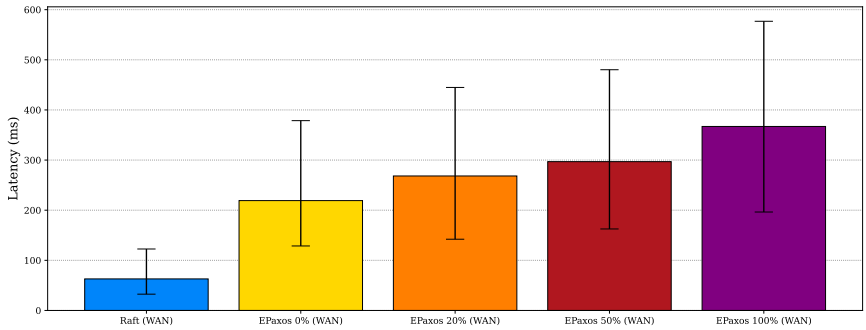

Fig. 2. Comparing clerk-observed execution latency for EPaxos across different contention workloads in a wide-area network simulation.

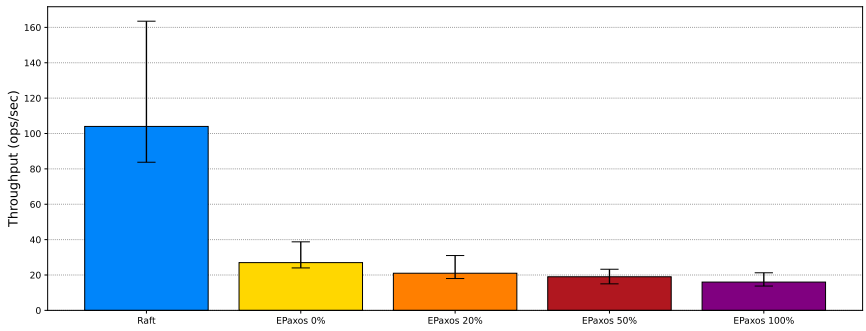

Fig. 3. Comparing clerk-observed execution throughput for EPaxos across different contention workloads.

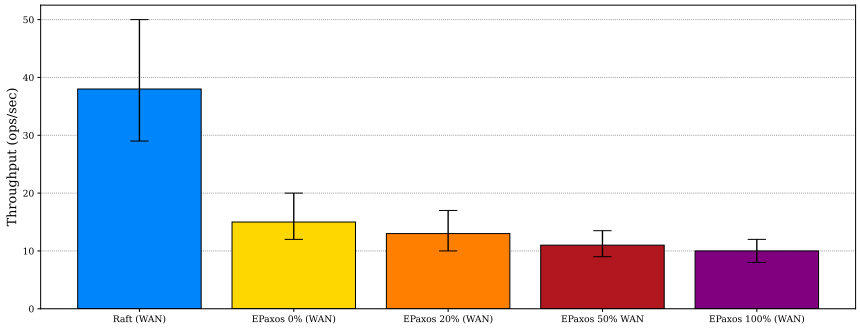

Fig. 4. Comparing clerk-observed execution throughput for EPaxos across different contention workloads in a wide-area network simulation.

Figures 3 and 4 show similar results to the first two figures, but instead measure execution throughput instead of latency.

5 CONCLUSION

Our particular implementation of both protocols shows that Raft performs better than EPaxos.

We noticed during testing with crashes/network partitions that Explicit Prepare drastically increased the latency of executing commands. This is because if one replica decides to EP for a certain instance, likely around the same time (and before that replica can send Prepare messages) all the other replicas will also initiate EP. Thus, many unnecessary RPCs will be sent out, with only one replica actually able to "win" in the end. Compared to Raft, this made EPaxos run much slower for the lab 3 tests.

Of course, our implementation for EPaxos has performance optimizations left on the table. Our implementation experience with EPaxos reveals a very important difference between the EPaxos paper, *There is More Consensus in Egalitarian Parliaments*, and the Raft paper, *In Search of a More Understandable Consensus Protocol*: clarity. Implementing Raft was much easier than EPaxos because far fewer details were up to interpretation. The EPaxos paper was very vague in certain implementation details, especially related to failure recovery and the Explicit Prepare algorithm.