Improving the Multidecree Paxos Protocol

Ishan Shah
The University of Texas at Austin

Abstract

Multidecree Paxos, or Mutli-Paxos for short, is a popular and well-known protocol for state machine replication that allows crash failures. In this paper, we present and implement four different optimizations to the Multi-Paxos protocol. First, we ensure that only one leader is active most of the time, so that leaders don't contend with each other and are able to make progress. Next, we reduce the state maintained by each acceptor and messages sent from each acceptor to a scout. Third, we colocate leaders and replicas to reduce message traffic and reduce the number of required machines. Finally, we reduce the state maintained by each leader, so that a leader wouldn't have to spawn many commanders each time it becomes active. We provide implementations for each of these optimizations as well as a combination of all four optimizations at `https://github.com/ishanashah/Improving-the-Multidecree-Paxos-Protocol`. We present quantitative evidence for the quality of each optimization.

1 Introduction

A distributed state machine consists of a set of states, a transition function that maps a state and a command to a new state, and an initial state. Using the same initial state, list of commands, and transition function, any distributed state machines must end up with the same state. A replicated state machine consists of many state machines that appear as a single state machine to a client. A replicated state machine is difficult to implement because different replicas may receive commands from clients in different orders. In this case, the states of each replica would diverge and the replicated state machine would no longer be consistent. Multidecree Paxos, or Multi-Paxos for short, is a state machine replication protocol that allows many replicas to agree on an order of commands in the presence of parallel client requests. Furthermore, Multi-Paxos allows the replicated state machines to be both live and consistent in the presence of crashes. As long as a majority of replicas are alive, Multi-Paxos maintains the illusion that the set of replicas appear as a single consistent state machine to every client.

In Multi-Paxos, each replica maintains a sequence of slots that represents the order in which client commands should be executed. To submit a request, a client broadcasts the request to all replicas and waits for a response from a single replica. If replicas can agree on a sequence of commands, replicas can all apply state transitions in the same order, and the set of replicas is guaranteed to be consistent. Each slot is associated with a slot number, and when a replica receives a client request it assigns it to a specific slot number. Since many clients can send requests in parallel to different replicas in different orders, replicas may not agree on which command belongs to which slot. The replicas each propose a command for a specific slot, and Paxos uses the mutli-decree Synod protocol, or Synod for short, to choose a single command for each slot. Once a command is chosen for a slot, it can never be reverted, so the replica can safely apply that command to its own state machine. A replica only applies commands that have been decided by the Synod protocol, so the replicas are guaranteed to be consistent.

Each replica maintains its state, the next slot to propose a command, the next slot that needs to be decided, the set of outstanding requests, the set of outstanding proposals, and the set of decisions. Whenever a replica receives a request, the replica proposes that request in the next available slot. Whenever a replica receives a decision, the replica applies that command to its state machine if it has applied every previous command to its state machine. The Synod protocol is responsible for collecting proposal messages from replicas and broadcasting decision messages back to replicas.

The Synod protocol consists of two types of processes: leaders and acceptors. Leaders receive the proposals from replicas and are responsible for broadcasting decisions back to the replicas. Leaders communicate with acceptors to try to reach consensus on a command. Acceptors use a voting protocol to provide a single decision for a slot from many proposals. When a majority of acceptors vote for a proposal, that proposal becomes decided without requiring input from the rest of the acceptors. Leaders can detect when a command has been decided and broadcast the decision back to the replicas.

Acceptors use the concept of a ballot to keep track of its interactions with leaders. Ballots are totally ordered objects that each have a leader identifier. Acceptors are deterministic state machines that only accept two kinds of commands: phase one commands and phase two commands. Conceptually, phase one commands allow leaders to claim an acceptor. Phase one commands are sent from the leader machine to acceptors and consist of a ballot number. When the ballot number is greater than the acceptor's ballot number, the phase one command succeeds, and the acceptor updates its ballot number and sends back every proposal it has accepted. Phase two commands are sent from the leader machine to acceptors and consist of a ballot number and a proposal. If the acceptor's ballot number hasn't changed since the time the leader had a successful phase one and claimed the acceptor, the phase two command succeeds. In this case the acceptor accepts the leaders proposal. An acceptor can accept any number of proposals for any number of slots, but the Synod protocol guarantees that for a single slot, there can only be at most one proposal that is accepted by a majority of acceptors, and that proposal must be the proposal that is decided. In the event that either a phase one command or a phase two command fails because the leader's ballot number is too small, that means that another leader has claimed the acceptor and the leader must restart from phase one to reclaim the acceptor.

Leaders are responsible for trying to claim acceptors, getting their proposals accepted, detecting when a decision has been made for a command, and broadcasting that command to the replicas. Acceptors on their own are not aware of whether a command has been decided, because an acceptor can accept any number of proposals and does not know whether a majority of acceptors have accepted any of those proposals. Leaders spawn scout threads to attempt to claim acceptors and observe which proposals have already been accepted by different acceptors, and leaders spawn commander threads to get proposals accepted and detect when a decision has been made.

Scout threads send a phase one command to all acceptors using its leader's ballot. If the phase one command fails for any acceptor, the scout aborts and sends back a preempted message to the leader along with that acceptors ballot number. Otherwise, if phase one succeeds for a majority of acceptors, the scout sends back an adopted message to the leader along with the ballot number and the set of all proposals that the majority of acceptors have accepted.

Commander threads send a phase two command to all acceptors using its leader's ballot and a proposal. If the phase two command fails for any acceptor, the commander aborts and sends back a preempted message to the leader along with that acceptors ballot number. Otherwise, if phase two succeeds for a majority of acceptors, the command must be decided. The commander broadcasts the decision to all replicas.

Each leader maintains a ballot number, whether its active or not, and map from slot numbers to proposals. Leaders start with the lowest ballot number, and increase their ballot number each time a scout or commander is preempted. A leader becomes active when its ballot adopted by a majority of acceptors, and becomes inactive when a scout or commander is preempted. The leader's set of proposals only maintains the proposal associated with the highest ballot number for each slot. Leaders start by spawning a scout for the lowest ballot number and then respond to three kinds of messages in a loop: propose, adopted, and preempted. When a leader receives a new proposal from a replica, it adds the proposal to its local set of proposals if it has no proposals for that slot. If the leader is active, the leader also spawns a commander to try to get the proposal accepted. When a ballot is adopted by a majority of acceptors, the leader adds each acceptor's proposal with the highest ballot number in its slot to its local set of proposals. The leader then spawns a commander for all of its proposals. When a scout or commander is preempted, the leader spawns a new scout to attempt to become active again.

We present various optimizations for the Multidecree Paxos Protocol. First, we build a testing environment over the original Multi-Paxos source code to run intensive benchmarks. Next, we modify the environment to measure important stats and features that describe the performance of the protocol. We then implement four different optimizations over Multi-Paxos: leader timeout, accepted state reduction, colocation, and, leader state reduction. Leader timeout uses a backoff protocol to ensure that only one leader is active most of the time to guarantee liveness. Acceptor state reduction reduces the state acceptors need to maintain in relation to accepted proposals and in turn reduces the state acceptors send to scouts in P1b messages. Colocation places leaders and replicas on the same machine as different threads and significantly reduces the number of machines required to run Multi-Decree Paxos. Leader state reduction is enabled by colocation, and reduces the state leaders need to maintain for proposals and the number of commander threads the leader needs to spawn when it becomes active.

2 Related Work

Raft is a popular alternative to the multidecree Paxos protocol. It is much simpler and more understandable than Paxos and performs well in practice. Similar to Multi-Paxos, Raft achieves consensus on a total ordering of commands through an ordered log. Replicated state machines can be in a consistent state by agreeing on each of the client requests in their ordered logs. Replicas in Raft can be in one of three states: leader, follower, or candidate. Raft can only have at most one active leader at a time, and the leader is responsible for replicating its log to followers. Followers respond to log entries sent by the leader, and a log entry is committed when it is present in the logs of a majority of replicas. Candidates are active in the absence of a leader to replace a faulty leader. Raft splits the problem of consensus into independent subproblems: leader election and log replication.

The leader maintains its presence by broadcasting a regular heartbeat message to all replicas. Each follower maintains a randomized timeout to detect when a leader has failed, which is reset when it receives a heartbeat. If a follower reaches its timeout without receiving a heartbeat from a leader, it becomes a candidate. Each replica maintains its term, which indicates the period of time a candidate becomes a leader and stays active. A candidate begins an election by incrementing its term and requesting a vote from all other replicas. A replica can only vote once every term, and if any replica detects that another replica is on a higher term, that replica reverts to the follower state. A replica can only vote for a candidate when the candidate's log is at least as up-to-date as the replica's log. When a candidate receives votes from a majority of replicas it becomes a leader, and when a candidate times out before receiving enough votes it increments its term again and starts a new election.

The leader accepts client requests and broadcasts them to all other replicas. The leader appends each request to its own log entry and keeps track of the which log entries have been successfully replicated on each replica. As long as a replica's log isn't confirmed to be fully replicated, the leader continues to attempt to send log entries to that replica. Once a log entry is confirmed to be replicated on a majority of replicas, the client request is committed and any replica can safely apply the request to its log machine. A committed log entry also implies that every previous entry must also be committed. The leader includes the highest committed log index in future messages to update followers. Inconsistent logs are handled by the leader requesting a follower to delete log entries until the follower's log is consistent.

A candidate cannot win an election unless it has every committed log entry. Candidates can only receive votes from other replicas if its log is at least as up to date as all of those replicas. A log is determined to be more up-to-date if the last index has a higher term, or if the last index has the same term but the log is longer. Since any committed log entry appears on a majority of replicas, the candidate must receive at least one vote from a replica with the committed log entry, which means the candidate must also have the same log entry to win the vote. Thus, when a log entry is committed, the same log entry must be present in every single leader that follows if the original leader crashes.

3 Design and Implementation

3.1 Leader Timeout

Our first optimization is to ensure liveness among highly contested leaders. We ran many trials of the original Multi-Paxos code on our testing environment, and the original protocol rarely made it past 10 client requests. Our testing environment runs Paxos with many concurrent clients, and the original Multi-Paxos protocol is very poorly equipped to deal with many requests. The problem is that leaders can only make progress in two phases: they must first get their ballot adopted by a majority of acceptors and then get their proposals accepted before any of the acceptors adopt a new ballot. If any leader gets preempted during these two phases it must start over again. In the presence of multiple competing leaders, leaders can constantly preempt each other and never make progress. After leader A succeeds in phase one, leader B can get its ballot adopted by the acceptors. Then when leader A attempts phase two, it fails, retries phase one, and gets its ballot adopted by the acceptors. Then when leader B attempts phase two, it fails, and the cycle continues. In our highly concurrent testing environment, contesting leaders fell into this cycle every time, so it is impossible to evaluate the original Multi-Paxos implementation.

Our solution to ensure liveness is a backoff protocol to give leaders enough time to get their proposals accepted. We give each leader an floating-point timeout state, initially zero. We also specify two hyper-parameters: *TIMEOUT_INITIAL* and *TIMEOUT_MULTIPLIER*. Whenever a leader is preempted we set its timeout to *TIMEOUT_INITIAL* if it was zero and we multiply it by *TIMEOUT_MULTIPLIER*. *TIMEOUT_INITIAL* and *TIMEOUT_MULTIPLIER* are 0.1 and 2 respectively in our implementation. Whenever a leader's ballot is adopted, we reset its timeout to zero.

In our implementation when many leaders are contested and constantly preempting each other their timeouts exponentially increase. Eventually, the contesting leaders are sleeping for long enough that one leader is able to have its ballot adopted by a majority of acceptors. This leader has its timeout set to zero, and thus is able to launch commanders to get its proposals accepted while the other leaders sleep. If this leader crashes, eventually other leaders will hit their timeouts and attempt to get their ballots adopted. In any case, if leaders that wake up start contesting with each other the backoff protocol puts them back to sleep. Our timeout protocol ensures that only one leader is active most of the time and allows other leaders to become active in case the original leader crashes. Thus, our optimization guarantees liveness even in the presence of many and highly concurrent client requests.

3.2 Acceptor State Reduction

Our second optimization is to reduce the number of proposals that an acceptor has accepted. In the original Multi-Paxos protocol an acceptor keeps track of every single proposal it has ever accepted. An acceptor can accept many different client requests for the same slot, and can also request the same client request on the same slot many times with different ballot numbers. The overhead is especially noticeable when an acceptor sends a phase one b message to a scout. An acceptor must send the set of all proposals it has ever accepted in every single phase one b message. Phase one b messages are the only messages that carry more than a constant amount of state over the network.

Our solution to reduce acceptor state is to only maintain the most recently accepted proposal for each slot. Instead of maintaining many proposals with different requests or ballot numbers, the acceptor only maintains at most one proposal for each slot. The state maintained by the acceptor and sent in phase b messages are greatly reduced.

Our solution is safe because an acceptor's accepted proposals are only visible through phase one b messages to scouts. If the reduced state in phase one b messages does not affect the behavior of Paxos, then it is safe to apply the state reduction to acceptors. The only reason a scout receives the set of accepted proposals is to combine the sets of proposals across a majority of acceptors and send the combination to its leader. For each slot a leader adds each proposal with the highest ballot number to its state and ignores all other proposals. Thus, an acceptor only needs to maintain the proposal with the highest ballot number. Since an acceptor's ballot number is monotonically increasing, the proposal with the highest ballot number is the most recently accepted proposal. Since the effect of an acceptor's messages stay the same after the state reduction, the state reduction does not change the behavior of Paxos.

3.3 Colocation

Our third optimization is to reduce the number of machines required to run Multi-Paxos. To tolerate *f* failures, Multi-Paxos needs *f+1* replicas, *f+1* leaders, and *2f+1* acceptors. In total Multi-Paxos needs *4f+3* machines.

Our solution is to co-locate replicas an leaders. Instead of running replicas and leaders on different machines, we run leaders on the same machine as replicas as a separate independent process. Instead of *4f+3* machines, our implementation only requires *3f+2* machines. Furthermore, instead of having a replica broadcast to all leaders over the network each of its proposals, our solution allows a replica to simply send its proposal to the leader it is co-located with using inter-process communication.

In the original Multi-Paxos protocol, the reason a replica had to broadcast requests to all leaders was because up to *f* leaders could have failed, and a replica cannot know which leaders have failed and which are alive. In colocation since a replica is running on the same machine as a leader, the replica is alive if and only if the leader is alive. A live replica can send a proposal to only its own leader because its own leader must be alive. Even if a replica's machine crashed in the process of the replica sending the proposal to the leader within the machine, since a client broadcasts its requests to all replicas, there must be a live replica/leader machine that can successfully process the request.

3.4 Leader State Reduction

Our fourth optimization is to reduce the number of proposals that a leader must maintain. In Multi-Paxos a leader must maintain a proposal for every slot. The leader can update the proposal for a slot but never delete it. The larger issue is that every time a leader becomes active it needs to spawn a commander thread for every single proposal it has. As the number of slots grows linearly over time, the number of commanders spawned when a leader becomes active grows quadratically. Each commander thread communicates with all acceptors to try to decide its proposal, so the leader's state results in a lot of overhead.

Our solution builds on top of the colocation optimization. We implement a new message type, namely a *slot_out* message, that a replica sends to its local leader every time its *slot_out* state variable is updated. The local leader also maintains a *slot_out* state variable to keep track of the largest known *slot_out* of the local replica. The local leader deletes every proposal associated with a slot less than *slot_out*.

Our optimization doesn't cost any network messages because the *slot_out* message is only sent within a machine. Our optimization reduces the leaders state from the total number of slots to the size of the latest window of commands that haven't been decided. Since the size of the window is constant, the number of commanders spawned stays the same instead of growing quadratically with the number of slots. Our optimization is safe because using our colocation scheme, a leader can only get proposals from its local replica. A replica never sends a proposal to a leader for a slot that it already knows to have been decided.

4 Evaluation

4.1 Testing Environment

The original Multi-Paxos source code does not come with an environment that allows for stress testing the Paxos protocol. In the original code's env.py, you can specify the number of replicas, leaders, acceptors, and requests. eny.py acts as a client and iterates over however many requests are specified. For each request, the client sends that request to exactly one replica, then sleeps for one second, and continues sending the request and sleeping until every replica has received the request. Since the delay between sending any two requests is one second, the entire Paxos protocol easily finishes in time for the next request. We modify the environment so that the client would not wait before sending requests, so we can evaluate the performance of Multi-Paxos in the presence of highly active and parallel clients.

Furthermore, the original code has no mechanism to detect when the Paxos protocol has completed acting on every client request, so even after every request has been delivered and resolved the environment waits until the programmer interrupts it. The original source code does not implement the response message that a replica should send to a client when its request as been decided and executed. We implement the client response message and modify Paxos replicas to send the response message back to the client whenever it executes a client request. We modify the Paxos environment so that it maintains the set of all client requests and the set of all responses received from replicas. After every client request has been sent, our testing environment waits until the set of client requests matches the set of replica responses. Once our testing environment has received every response it needs, it terminates. In essence, we allow the Paxos environment to be able to terminate at the end of a benchmark, making performance evaluation much simpler.

We add features in our testing environment to measure three protocol characteristics: the total time taken, the number of network messages, and the total state in network messages. We can easily measure total time taken by noting the time at the beginning of a benchmark and noting the time again as soon as the protocol terminates. We modify the message-passing interface to keep track of the total number of network messages, and we ignore client messages and messages between two threads on the same machine. We also modify the message-passing interface to keep track of the total state passed on phase one b messages from an acceptor to a scout.

4.2 Methodology

As we mentioned in section 3.1, it is impossible to evaluate the original Multi-Paxos protocol in the presence of concurrent client requests. Many leaders would constantly preempt each other and no leader would make progress. As such, it is difficult to compare our each of our optimizations with the original protocol. Our solution is to implement leader timeout on every protocol we evaluate. Otherwise, any protocol without leader timeout wouldn't complete any benchmarks. We use the original Multi-Paxos protocol with the leader timeout optimization as the baseline for our evaluation.

We evaluate five protocols. Apart from the leader timeout baseline, we evaluate, our baseline with acceptor state reduction, with colocation, with colocation and leader state reduction, and with all four optimizations. Our testing environment tolerates up to 3 failures and uses 4 replicas, 4 leaders, and 7 acceptors. We evaluate the total time taken, the total

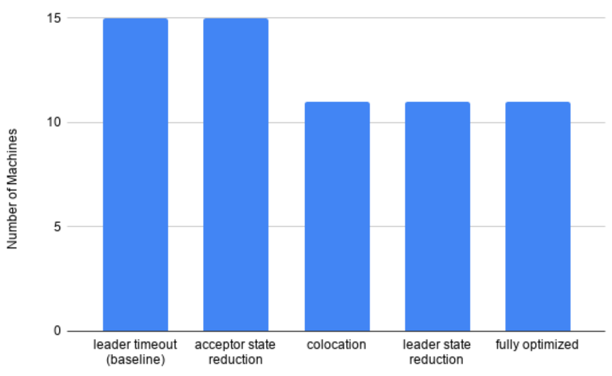

Figure 1: The number of machines each protocol requires.

number of network messages, the total amount of state within network messages, and the number of machines for all five protocols.

4.3 Results

Figure 1 shows the number of machines needed by each protocol. Since our testing environment tolerates 3 failures, leader timeout and acceptor state reduction require $4f+3=15$ machines. Colocation, leader state reduction, and fully optimized only require $3f+2=11$ machines because they colocate leaders on replica machines.

Figure 2 shows the amount of time each protocol takes to complete with respect to the number of client messages. We can see that the baseline tends to take more time to complete and that colocation doesn't affect the time much. We see that both the acceptor state reduction and the leader state reduction optimizations provide significant improvements in time to complete. We see that the fully optimized protocol is noticeably the fastest. Time measurements should be taken with a grain of salt because our testing environment represents each machine as a thread, so network messages are artificially fast. In reality a better indicator for latency would be the number of network messages.

Figure 3 shows the total number of network messages sent during each protocol with respect to the number of client requests. We can see that the baseline sends the largest number of messages. Colocation and acceptor state reduction don't largely impact the number of messages sent, because colocation enables further optimizations and acceptor state reduction only affects the contents of network messages. Leader state reduction has a large impact on network messages because a leader needs to spawn much fewer commanders that communicate with acceptors. Again, we see that the fully optimized Multi-Paxos protocol sends the fewest network messages.

Figure 4 shows the total amount state in phase one b network messages sent during each protocol with respect to the

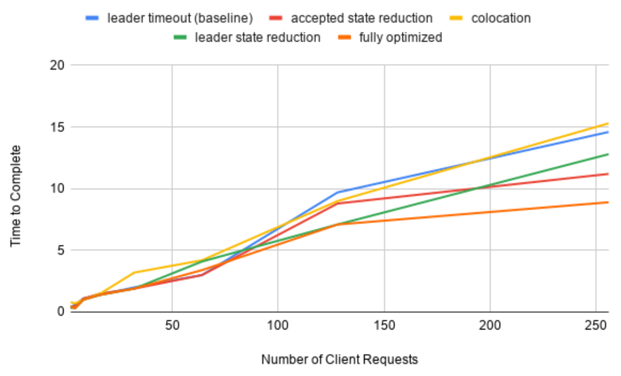

Figure 2: The time to complete for each protocol as a function of the number of client requests.

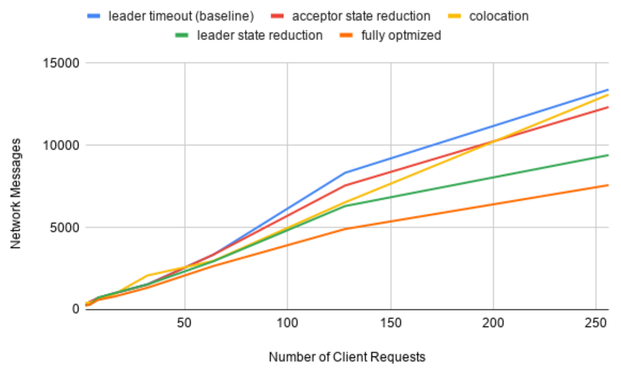

Figure 3: The total number of network messages sent by each protocol as a function of the number of client requests.

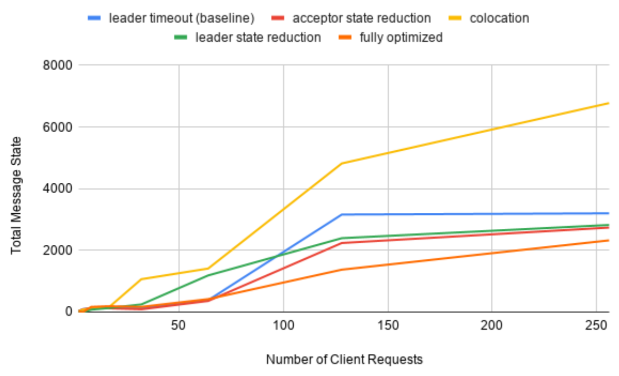

Figure 4: The total state of all phase one b network messages sent by each protocol as a function of the number of client requests.

number of client requests. We only measure the state in phase one b messages because all other messages have a constant amount of state or are messages sent between threads on the same machine. In comparison, phase one b messages have state that grows with the number of slots and are sent over the network. We can see that colocation has the most message state. The reason for this is still a mystery and is left for future work. We see less message state from acceptor state reduction because acceptors maintain less state, so phase 1 b messages from acceptors contain less state. We also see less message state from leader state reduction because the leader spawns less commanders, so less proposals are accepted and acceptor state is reduced indirectly. Finally, we wee that the fully optimized protocol has the least total message state.

5 Conclusion

We have exposed four weaknesses in the Multidecree Paxos protocol, and we have proposed four optimizations to improve the performance of the protocol. Our optimizations enable the protocol to be live in the presence of many concurrent clients, reduce the number of machines required to run the protocol, reduce the amount of time required to respond to many clients, reduce the number of network messages passed within the protocol, and reduce the amount of state passed within network messages. We provide an implementation for each of the four optimizations and a combination of all four optimizations at `https://github.com/ishanashah/Improving-the-Multidecree-Paxos-Protocol`. We implement and provide a robust testing environment capable of evaluating many characteristics of a protocol. We use the testing environment to provide empirical evidence that each of the optimizations as well as a combination of all four optimizations significantly outperform the original Multidecree Paxos protocol according to many factors.