PRaft: Building Provably Safe Distributed Systems using Model Checking

Eliot Han, Andrew Sun, Alex Yeo, James Zhang
UC Berkeley
eliothan,andrewsun,alexsyeo,james.zhang@berkeley.edu

ABSTRACT

Distributed systems are hard to build. Existing validation methods fall under proof-based verification of the high-level description and black box testing of the implementation. We introduce the use of P, a DSL for writing specifications which compile into executable production code, to build a distributed system. We develop the Raft protocol in P and extend P to interface the generated implementation with a key-value store application for proof of concept.

1 INTRODUCTION

Building distributed systems is hard. Systems are composed of multiple independent components running in parallel, with asynchronous execution and constant message passing, contributing to an innumerable number of possible traces. Coding such systems requires immense time and effort while still leading to inevitable, costly bugs. On July 15 2015, the New York Stock Exchange shut down, United Airlines flights were grounded, and the Wall Street Journal went offline—all due to distributed systems going haywire.[1]

Developing and testing distributed systems is often a two-phase effort. Protocol descriptions are constructed in high-level specification languages such as TLA+ to mathematically capture the properties and semantics of the system. These formal specifications are used for proof-based verification, in which correctness under user-defined invariants can be shown using theorem provers. Creating the invariants for these theorem provers is a time-consuming task, and developing in these languages is far different than developing in the languages that software engineers are familiar with (eg. C, Java, Python).

Given a formally proven protocol description, engineers must translate the specification into a concrete implementation. This is notoriously hard; as the creators of Chubby noted, "there are significant gaps between the description... and the needs of a real-world system."[2] These gaps lead to the introduction of subtle errors. Systematic testing of the implementation is then required to fix these bugs. While it is possible to prove the implementation simulates the model, these proofs are difficult to write and understand. As a result, they are often not kept in sync with changes made to the implementation. The common theme is that it is hard to create an implementation that matches the formal specification. When teams at Amazon learned TLA+, "engineers usually ask, 'How do we know that the executable code correctly implements the verified design?' The answer is we do not know."[13] Yet, teams at Amazon working on distributed systems like S3 and Dynamo still utilize formal methods as they have been shown to be uniquely valuable.

A promising alternative is writing protocol specifications in P, a DSL that allows for both implementation and formal verification, alleviating this issue altogether.

The rest of the paper is organized as follows: Section 2 provides background on the P language and the Raft distributed consensus protocol to help contextualize the system that we built. Section 3 explores related works in the field. Section 4 provides our metrics for success. Section 5 discusses implementation details of PRaft. Section 6 delves into challenges we encountered with respect to validation, including handling time in P, checking invariants, and failure injection. Section 7 discusses the executable code, including how we interfaced it with the P runtime and an open source go application. Section 8 goes over our results, including various measurements taken. In Section 9 we note possible areas for future work. Finally, we present our conclusion in Section 10 and offer acknowledgements in Section 11.

2 BACKGROUND

2.1 P

P is a DSL for writing asynchronous, event-driven code.

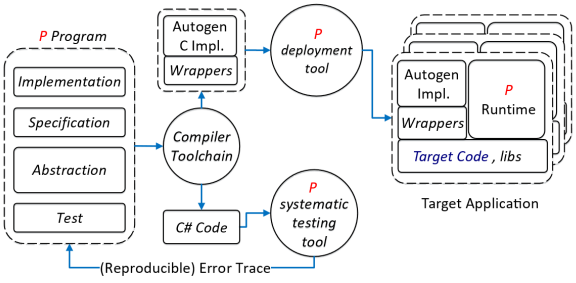

Figure 1: P Architecture & Programming Framework

P allows the programmer to specify the system as a collection of interacting state machines, which communicate with each other using events. The semantics are interpretable by a model checker, and alternatively can be generated into executable C or C# code. This unifies the two-phase development process, and offers a tool to close the gap between theory and practice.

In practice, writing a native P program requires the development of four separate components. The implementation is a high-level model of the program composed of interacting state machines.

Communication is modelled by events passed between state machines; events are predefined with typed payload values. All machines are initialized in the *start state*. Memory management is based on linear typing and C++ style unique pointers, and is automatically handled during compilation into executable code.

```
// P PingPong program from official wiki
event PING assert 1: machine;
event PONG assert 1;
event SUCCESS;

main machine Client {
    var server: machine;

    start state Init {
        entry {
           server = new Server();
           raise SUCCESS;
        }
        on SUCCESS goto SendPing;
    }
    state SendPing {
        entry {
             send server, PING, this;
             raise SUCCESS;
          }
        on SUCCESS goto WaitPong;
    }
    state WaitPong {
        on PONG goto SendPing;
    }
}

machine Server {
    start state WaitPing {
        on PING goto SendPong;
    }
    state SendPong {
         entry (payload: machine) {
             send payload, PONG;
             raise SUCCESS;
         }
        on SUCCESS goto WaitPing;
    }
}
```

Monitors are special state machines that allow programmers to check invariants in the specification. Invariants are defined as assert statements within event handlers. This is possible because specifications can observe the triggering of events by any other state machine in the system. The model checker then checks these invariants when running through the state space of the program.

```
// PRaft Safety Monitor Excerpt
spec SafetyMonitor observes M_LogAppend,
    M_NotifyLeaderElected, M_LeaderCommitted
{
   var CurrentTerm: int;
    ...
    state Monitoring
   {
      on M_NotifyLeaderElected do (payload: (Term:
          int, Logs: seq[Log])) {
         ProcessLeaderElected(payload.Term,
             payload.Logs);
      }
      ...
   }
}
```

Finally, our program will require interaction with the external environment. We use the *ghost machine* feature of P to build an abstraction of the environment for our Raft implementation to interact with during validation, and test drivers to simulate various classes of failure scenarios. These ghost machines can be excluded from compilation into C code, and are used only by the model checker in the generated C# code.

2.2 RAFT

Raft is a distributed consensus protocol for maintaining a replicated log. It has three distinct processes: leader election, log replication, and cluster membership editing. It was developed as a response to Paxos, a distributed consensus algorithm designed by Leslie Lamport. Paxos is infamous for its complexity and difficulty to implement. Thus, the creators of Raft sought to address these deficiencies by designing a new protocol that could make the same guarantees as Paxos while being more understandable and feasible to implement; in fact, the authors state that the primary goal of the algorithm is its understandability itself.[11]

Raft uses a stronger form of leadership compared to other consensus protocols. Log entries are only allowed to flow from the leader to other servers. Leaders are elected using randomized timers and, once elected, send out heartbeats to maintain leadership. Membership changes are implemented

using a joint consensus approach. There are two kinds of RPCs in Raft: AppendEntries and RequestVote. The heartbeats that leaders periodically send out are AppendEntries RPCs with empty bodies. At all times, Raft guarantees the following properties:

(1) Election safety: There is only one leader elected for a given term. A term is a phase between elections and is valid until a leader fails.
(2) Leader Append-Only: A leader will never overwrite or delete entries. It can only append new entries to the log.
(3) Log matching: If two logs contain an entry with the same index and term, they are identical for all entries from the beginning and up to that index.
(4) Leader completeness: If a log entry has been committed in a given term, then that particular entry will be present in the logs of the leaders for all subsequent terms.

Raft servers have 3 possible states: follower, candidate and leader. When servers start up, they begin as followers and will remain as followers as long as they receive valid RPCs from a leader or candidate. If a follower does not receive any request for a given time duration, it times out and will move to the candidate state and begin an election. This election timeout is randomly set to some interval within 150-300ms.

In the candidate state, servers will increment their term and attempt to get a majority of votes. Followers will vote for a candidate if the candidate's term is greater than its current term. If too many followers become candidates and none of the candidates can secure a majority, the candidates will timeout within some randomized interval and start a new election.

When a leader is elected, it sends empty AppendEntries RPC calls to each server and repeats during idle periods to prevent election timeouts. On receiving a command from a client, the leader appends the entry to its local log then issues AppendEntries RPCs. If followers are unable to respond, the leader retries indefinitely until all followers eventually store log entries. The leader can reply to the client once the entry has been safely replicated on a majority of machines. This means that the entry is *committed*. Committing a log entry also ensures that all preceding entries in the leader's log are also committed.

Cluster configuration changes are propagated throughout the cluster as special log entries. Servers run on the newest configuration found in their log. For the change to be safe, there must be no point during the transition where two leaders can be elected for the same term. In the original paper, this safety is ensured by a joint consensus approach: Before the new configuration is committed by the leader, we may still have a majority of the servers on the older

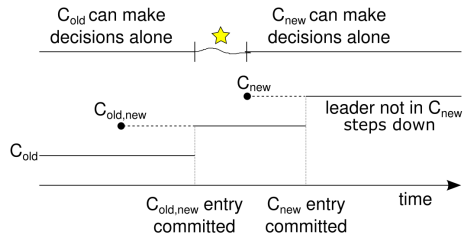

Figure 2: Configuration Change Timeline

configuration. So during that starred period, any server from either configuration may serve as leader, and agreement for elections and log consensus requires separate majorities from both old *and* new configurations. We will discuss a simpler single-server cluster membership change approach in Section 6. For more detail on Raft, we refer readers to the original text.

3 RELATED WORKS

There are several existing efforts for developing provably correct distributed systems. Notable examples are TLA+ and Alloy, formal specification languages describing the set of all possible execution traces of a system.[5,6] These languages allow designers to abstract software programs into a logical specification, which can then be checked for correctness properties over all possible execution traces. This has been used to successfully discover errors in each of the 7 invariants of the Chord protocol and implement a formally correct version of the protocol.[7]

The main issue with these approaches is the unfamiliar syntax and idioms of these languages. Developers have a high barrier of entry in converting code into formal logic. While the abstract model can be shown to be provably safe, the executable implementation is manually coded by developers and is not guaranteed to be a bisimulation of the original model. In contrast, P is a DSL that allows developers to write their program in C-style code, which then compiles into runnable C# code that is formally verified and immediately deployable.

There have been other recent efforts to unify the two phases of distrbuted systems development. Verdi is a framework for writing executable, formally verifiable systems. Verdi requires the programmer to write a specification in Coq (a formal verification langauge) in addition to the state machine implementation required by both Verdi and P.[10] Verification assumes idealized network semantics: packets are delivered exactly once, and there are no node failures. In contrast, we show in this paper an example of using P for verification under realistic network conditions, including message drops and failure injections. Generated code is in

OCaml, which has a far less developed ecosystem than C or C#.[10]

LNT is a process language with formal semantics that has been used to automatically generate a Raft implementation.[9] This language is limited in scope, and cannot compile more elaborate data types such as tuples, lists, or arrays. Validation within the scope of naturals, integers, and enumerated types is done by theorem proving rather than systematic testing, which is the method of choice in P.

4 METRICS FOR SUCCESS

Our goal was to implement the Raft protocol in P, with all three primary features of Raft including leader election, log replication, and cluster membership editing.

We interface the C code generated by our P implementation with a key-value store application with the intent to demonstrate P's utility. We also compare PRaft against an open-source Raft implementation. We aim to show that P offers a way to implement performant distributed systems that are also formally verified to greatly aid the programmer in finding bugs that are difficult or nearly impossible to find.

5 IMPLEMENTING RAFT PROTOCOL IN P

5.1 Overview of State Machines

We defined four different state machines to model our system: Server, Client, ClusterManager, and WallclockTimer. The Server machine represents a node in the Raft protocol. The Client machine represents a client that sends requests to the Raft cluster. The ClusterManager machine serves as an abstraction that keeps track of various state in the system and configures the Server machines upon initialization. The WallclockTimer machine serves as a logical abstraction of time. The significance of the WallclockTimer will be discussed in greater detail in section 6.1. The following section describes the Server machine, including its various states and transitions between these states, which are depicted in Figure 2.

5.2 The Server Machine

We designed the server state machine to have four possible states: Init, Follower, Candidate, and Leader. This is to emulate the possible states of a node in the Raft protocol, with the addition of the Init state which is the first state a P machine enters by default.

Upon entry into the Init state, a server machine initializes its various state variables such as its current term, log, and commit index. Once it receives an SConfigureEvent from the ClusterManager machine, it transitions to the Follower state. SConfigureEvent includes a payload containing a list of the other server machines in the system as well as the Cluster Manager itself, enabling the configured server to refer to these other machines throughout execution.

Upon entry into the Follower state, a server resets its election timer and is available to respond to any vote requests and append entry requests. If a follower receives a client request, it will forward the request to the current leader. (If a leader does not exist, it redirects client requests to the Cluster Manager.) An optimization followers also implement is to ignore VoteRequests if there is a current leader. This helps avoid disruptions from removed servers.[11] If the server's election timer times out (modeled by receiving a Tick event such that its total number of ticks received since its timer was last reset hits a randomized threshold), then the server transitions to the Candidate state.

When becoming a Candidate, a server machine starts the election process by broadcasting vote requests to the other server machines through VoteRequest events. Just like in the Follower state, the machine redirects any received client requests to the Cluster Manager or leader, and can also react to VoteRequest events from other server machines. For example, if a Candidate server receives a VoteRequest event from another server machine with a greater term number than its own, it reverts back to the Follower state as specified by the Raft protocol. Upon receiving successful VoteResponse events from a majority of the other server machines, the server transitions to the Leader state.

Upon becoming a Leader, a server sends a NotifyLeaderUpdate event to the ClusterManager machine, which updates its knowledge of the current leader. The Leader then sends heartbeats to followers in the form of AppendEntriesRequest events with empty payloads. Additionally, the Leader processes incoming client requests by adding those requests to its own log. These log additions will then be piggybacked on the payload of the next heartbeat. Follower servers that receive these requests with non-empty payloads will respond to the leader with AppendEntriesResponse events. Depending on whether or not a majority of these responses are successful, the leader is able to tell whether or not it should commit the request. If the leader happens to receive a VoteRequest or AppendEntriesRequest whose term is greater than its own, it knows that it is behind and transitions to the Follower state.

5.3 Membership Changes

The original Raft paper espoused a joint consensus approach to cluster membership changes. In theory, this ensured the correctness of arbitrary cluster configuration changes. However, the complexity of implementing two separate majorities and correctly switching leaders led to a buggy implementation in practice.

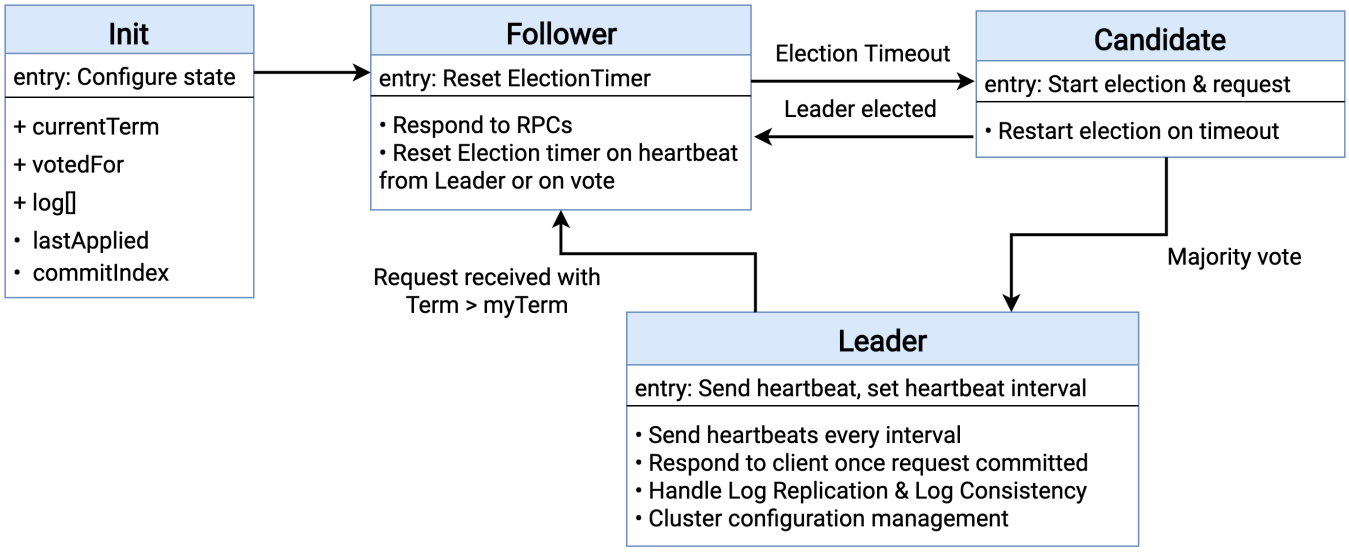

Figure 3: Server State Transition Diagram

Since the literature is not clear on the coding of this approach, we instead used the single-server membership change protocol described in the Raft author's dissertation. While this only allows for the removal or addition of a single server at a time, it is much simpler to prove correctness for and implement: Correctness is ensured since any majority

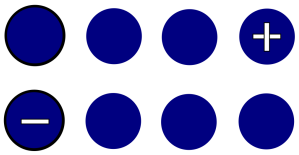

Figure 4: Single Membership Change

requires a majority of both new and old servers; it is impossible for two leaders to be elected in the same term. The main challenge in P is storing and parsing the configurations: due to static typing, a log can only store one type of data.

We solved this problem by splitting the log into a *KVLog* storing string $< K, V >$ pairs and a *ConfigLog* holding server configurations. The ConfigLog is abstracted away from the user, and indices are automatically mapped to the correct log through event handlers and storing of indices in pairs.

5.4 PRaft Architecture

We now briefly discuss the entire architecture of PRaft. As mentioned earlier, there are four distinct machine types in the system: Client, ClusterManager, Server, and WallclockTimer (denoted 'Logical Timer' in Figure 3).

The Client sends requests to the Cluster Manager, which responds to the Client when these requests have been committed by the Server machines. Upon receiving a response event, the Client machine can handle this response accordingly under the assumption that the request payload has been committed. Applications can be built on top of this Client-to-Cluster Manager interface; for example, the key-value store application used for our performance metrics sends key-value requests to the system using this Client abstraction.

The Cluster Manager serves as the intermediary between the Client and Servers. It is worth noting that the Cluster Manager is not a feature of the Raft protocol but rather an abstraction unique to our PRaft implementation which is useful for synchronizing information such as the state of the current Leader and Logical Timer. In real applications of Raft, the core nodes running the algorithm will not be directly available to clients from the outside world. Some form of load balancer or proxy will sit in front of the core nodes to prevent tampering and validate client requests. The Cluster Manager can be viewed as an abstraction of this client facing layer. Another way to think of the Cluster Manager machine is as a passive learner from Paxos. Learners do not participate in the core consensus algorithm and simply learn the decided value and can relay that information to clients.

Depending on whether or not the Cluster Manager has a reference to a current Leader, it can be in one of two states: Available or Unavailable. In the Available state, the Cluster Manager forwards requests from the Client to the Leader server. Otherwise, the Cluster Manager is in the Unavailable state and will defer request events until a new server is

elected as a Leader. Deferred client request events are not dropped but are simply re-added to the event queue and will be handled at a later time. A server sends a NotifyLeaderUpdate event to the Cluster Manager upon becoming Leader; it is this event that signals the Cluster Manager to transition from the Unavailable state to the Available state. Note that even though the Cluster Manager is in the Available state and holds a reference to the leader server, this machine could be in a failed state. In this case, an election timeout will occur among at least one of the other servers and a new leader will be elected. The Cluster Manager can then update its current leader and resend the uncommitted client request.

In the next section, we will now turn our attention to the Logical Timer, discussing in detail the problem it solved, how it was implemented, and some challenges it presented with respect to performance.

6 VALIDATION CHALLENGES

6.1 Handling Time in P

Timing in Raft is critical for liveness. For the system to make progress towards committing log entries, Raft must elect and maintain a steady leader. The Raft paper gives the following inequality:

$$broadcastTime << electionTimeout << MTBF$$

BroadcastTime is the average time it takes a server to send RPCs in parallel to every other server in the cluster and receive their responses while electionTimeout is the election timeout period for follower state servers. MTBF is the average time between failures for a single server. If these properties aren't met, there will be elections happening constantly as the leader server will not be able to send a heartbeat in time to the other servers.

The approach to Timers we found in the P codebase is to use a timer machine that represents an abstraction of the OS timer. This machine has no real notion of time and is simply used to model race conditions between the arrival of events from the client and the elapse of the timer. For instance, once the timer has started and the timer's client wants to cancel it, the P timer will nondeterministically decide to either return a CANCEL_SUCCESS event to the client or a CANCEL_FAILURE and TIMEOUT. This second case models the instance when the timer fires before the CANCEL event arrives. This Timer machine is useful to the model checker by capturing various different race conditions that can occur between the OS and other components of the system. For actual execution, the system's interactions with the Timer machine must be replaced with hand-written C code to actually use an OS timer.

Due to Raft's heavy reliance on timing for liveness, we were unable to get the default P Timer machine to work for model checking. Elections would constantly occur during verification and the actual log would made no progress. To get around this issue, we created a logical timer machine called the WallclockTimer. The high level idea behind this machine was to take advantage of the P scheduler to introduce a logical notion of time. During model checking, the various state machines in the system coexist but execution of the code for these machines is serialized; this allows traces to be easily reproduced given the random seed. The scheduler randomly picks a machine that has available actions to execute (unprocessed events in its event queue or a currently running handler for an event). Our WallclockTimer machine uses the number of times the scheduler allows it to run to capture the $broadcastTime << electionTimeout$ inequality.

The WallclockTimer holds a reference to all the server machines in the cluster and when it is allowed to run by the P scheduler, the machine sends Tick events to all the servers. Once this is done, the WallclockTimer will send a SentAllTicks event to the Cluster Manager and enter an inactive state. When the Cluster Manager is scheduled and is able to dequeue this event, the Cluster Manager sends an event back to the WallclockTimer that when dequeued by the timer, will reactivate it. Servers that receive the Tick event will decrement a TickCounter variable. This variable is set when they transition into a new state. For the Follower and Candidate states, this number is randomly set between 250 and 300. When entering the Leader state, TickCounter is set to 5. Whenever a server machine decrements its TickCounter and the value reaches 0, the server either sends out heartbeats (if in the Leader state) or will "timeout" and enter the Candidate state to start an election. Setting the TickCounter for Followers and Candidates to be much greater than the Leader is what allows the model checker to capture the Raft inequality while still allowing election timeouts to occur in the case of a dead leader. In instances where a server should reset its ElectionTimer, such as when it receives a heartbeat from a Leader, this means we reset the TickCounter back to some number between 250 and 300.

The WallclockTimer serves as the single timer for all machines in our model. Thus, to actually deploy the generated executable code in some real system, each server node will presumably need its own timer. We did not replace the WallclockTimer machine with outside C code in the testing of our generated code. This means that our results contain a lot of overhead related to this machine. The P runtime will spend a lot of cycles running this machine and all the handlers in other machines that this machine's events trigger.

6.2 Checking Invariants

The four Raft invariants listed in 2.2 are checked using a P Monitor. We show an example invariant check below:

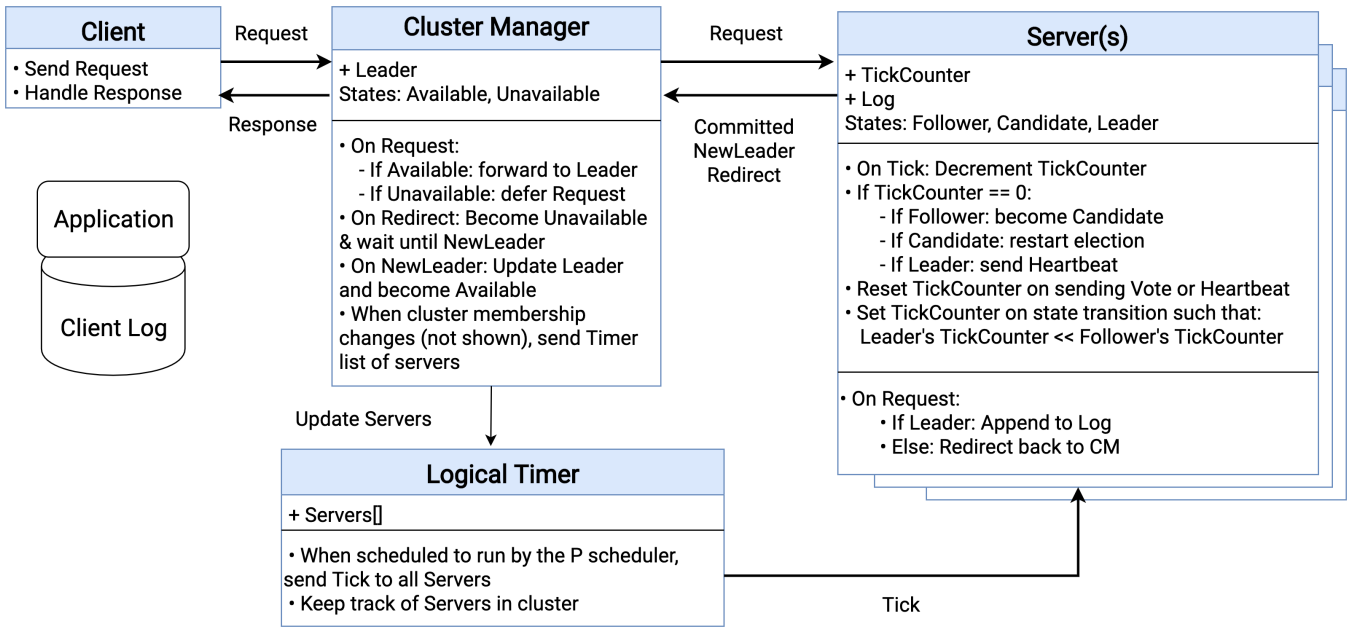

Figure 5: PRaft Architecture

```
fun ProcessLeaderElected(Term: int) {
   // Terms should be increasing
   assert(Term > CurrentTerm);
   CurrentTerm = Term;

   // At most one leader elected in a given term
     assert(!(CurrentTerm in TermsWithLeader));
     TermsWithLeader[CurrentTerm] = true;
     ...
}
```

This special state machine can observe any events that are sent between state machines and can also receive its own events. Our Safety Monitor runs a handler whenever a leader is elected, a log entry is appended, or an entry is committed. It keeps track of the system's current term, a map from each term to that term's elected leader, and the logs for each server in the cluster. By intercepting all these events and keeping track of the state for the whole system, it is easy to add assert statements to express invariants. For instance, when a log entry is appended, the monitor can loop through the logs of every other server to verify the Log Matching property. While this verification process is quite expensive, note that the Safety Monitor is only created for model checking and not for execution.

6.3 Failure Injection

To test a P program in the model checker, a machine must be written which instantiates the Cluster Manager as well as the various server machines. We call this machine the TestDriver. Various tests can be run by the model checker using different TestDriver machines. The TestDriver machine serves as the "Client" during testing and sends Requests to the Cluster Manager and receives Responses. Once a certain number of responses have been received, the TestDriver can stop the model checker. TestDrivers can be used to create different scenarios for the system such as instantiating the cluster with different numbers of servers or deciding when to inject failures. An example test which can then be selected to run in the model checker is depicted below. Note that while the TestDriver initiates the model checking procedure, the monitor is what actually asserts any invariants and detects bugs.

```
test Test0[main = TestDriver0]: assert SafetyMonitor
    in { TestDriver0, ClusterManager, Server,
    WallclockTimer };
```

We've tested for fault tolerance in two ways. One of our TestDrivers sends 1000 client requests but will randomly select a server in the cluster for removal between every request and create a new server to add. This tests for cluster membership changes while the cluster is serving requests. This test is detailed further below in section 9.2. The other test randomly halts server machines at any point during execution. This is achieved through a special FailureInjector

machine that holds references to the server machines. Whenever the scheduler schedules it to run, it may randomly pick a server to send a halt event to. Halt is a special event that terminates a P state machine. The FailureInjector can send up to $(n-1)/2$ halt events which is the maximum number the Raft protocol can tolerate. The model-checker explores all possible interleavings, thus the FailureInjector is exhaustive and will add failures at all possible points. A simplified version of the FailureInjector machine we used is shown below.

```
machine FailureInjector {
   start state Init {
      entry (participants: seq[machine]){
         var i : int;
         i = 0;
         while(i < sizeof(participants)) {
            //nondeterministically true
            if($) {
               send participants[i], halt;
            }
            i = i + 1;
         }
      }
   }
}
```

FailureInjector from P examples library

7 EXECUTABLE CODE

7.1 C/Go Framework with KV Application

We compared our Raft implementation against a preexisting implementation used by an open source key-value store etcd. Etcd is a Raft-backed key-value store used by multiple companies like Docker, OpenTable, and Salesforce, therefore it provided an opportunity for us to compare our implementation against production level code. Etcd also included a simple example of a key-value store within its repository that we modified to use our own Raft implementation.

There were multiple modifications we had to make to the sample key-value store to interface it with PRaft. Since PRaft was single process we had to modify the multi-process key-value store to be multi-threaded. After changing the key-value store to be single process we were then able to launch a thread devoted to running PRaft.

Secondly, since etcd is based in golang, part of our work involved finding a way to make requests from the key-value store into our own C based PRaft implementation. To do this we wrote wrapper functions in C to send desired events to the PRaft implementation, the events would either add machines or send append requests. To sync our key-value store we pulled desired entries of the PRaft log directly from memory and would replay the put requests from there. After defining these wrapper functions, we imported the functions via cgo and called the wrapper function directly from the key-value store application.

7.2 Compiler and Generating Mappings

Interfacing the P generated C code to our go based key-value store required a few changes to the compiler. A major issue for us was accessing variables that are necessary for our wrapper functions in creating new machines and sending events. There was one process definition that was missing from our generated code, but with a few small changes to the compiler, we were able to successfully generate the previously desired definitions.

7.3 P Runtime Library

Hooking up PRaft to the key-value store also required recompilation of the C code during build time for the go application. A lot of the generated code had dependencies in the P Runtime Library, therefore it was necessary for us to include these files into the linking process for compilation. Initially when we attempted to import a static library containing the files of the P Runtime Library, improperly defined variables concerning system dependent definitions caused numerous compilation errors. Therefore, we manually moved all files from the runtime library into our key-value store application and fixed the definitions from within the application.

8 RESULTS

8.1 Unexpected Failures

An unexpected failure that we encountered when interfacing PRaft with the sample key-value store was the fact that the key-value store's communication via network sockets seems to be a limiting factor to performance when the number of servers increases past 20 in a local setting. Therefore, when running the application in high node volume contexts, we would often encounter TCP errors when servers would deem peers to be unresponsive (even when they weren't). As a result, we had to limit our testing environment to 20 servers.

8.2 Measurements and Discussion

To test performance over time we spammed put requests to our KV store application and measured the number of committed entries for each implementation over time. While the etcd implementation demonstrates a pretty constant throughput, PRaft seems to have degraded performance as time goes on. This corresponds to the length of the log, and due to the lack of log checkpoints, PRaft sees a degradation of performance as the log grows to an unbounded length. Etcd does

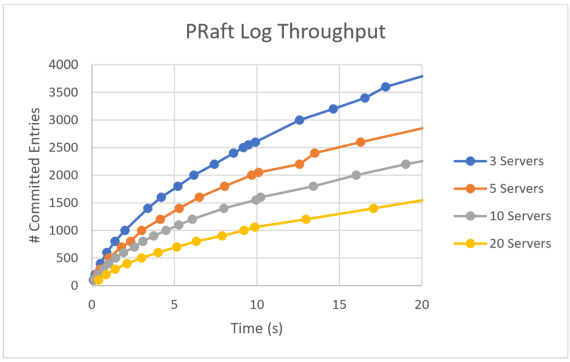

Figure 6: PRaft Log Throughput

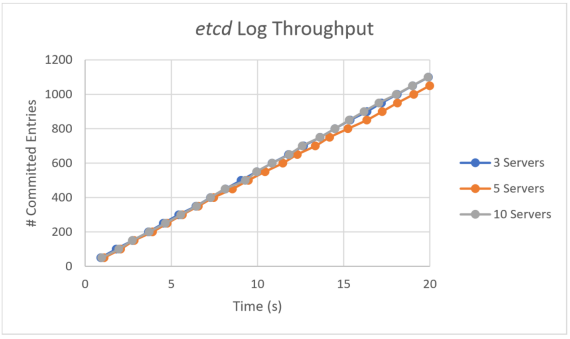

Figure 7: etcd Log Throughput

not deal with this issue due to the fact that it has a native snapshotting feature present in its application.

There's also a degradation in performance in PRaft as the number of servers increase. One possible explanation for this is found in the nature of P being a state transition language. As the number of events grows (which corresponds to the number of servers), the scheduler on the process may schedule things non-optimally, leading to some servers falling behind while other get ahead. Therefore, when a leader is waiting for a response from followers to complete a commit, it may take longer than necessary when members that haven't voted yet are starved by the scheduler.

Despite these performance degradations, PRaft still performs several magnitudes quicker when compared to the etcd implementation. As a side effect of snapshotting, etcd flushes its log to disk every once in a while: the extra IO operations are expensive and could reduce the throughput of the system.

Our second test verifies election correctness and time over cluster size. In PRaft, elections are still triggered by followers, but an extra step is required to update the cluster manager cache with the new leader. We thus initialize the cluster manager, then record the average time before the election for initial leader completes *and* the cluster manager's cache is

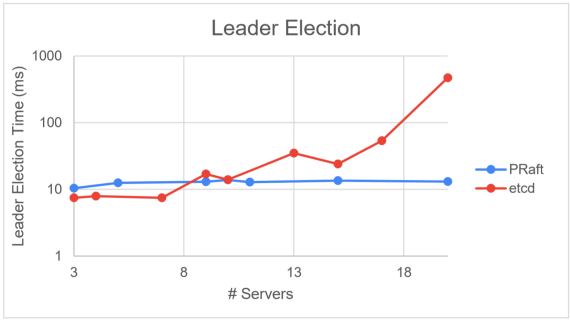

Figure 8: Election Time Comparison

updated. The *etcd* election time is only the time to determine a leader. We see that the election time for PRaft is relatively constant, taking around 10ms. The *etcd* implementation is similar in magnitude for smaller number of servers, but the effect of HTTP communication becomes significant beyond 10 servers. We expect PRaft to have similar performance once HTTP communication between processes is implemented.

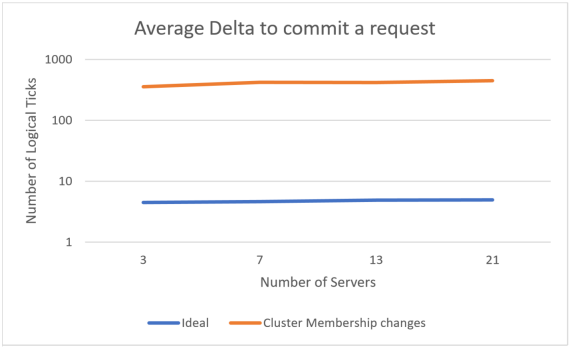

Figure 9: Average Delta to commit a request

Figure 9's experiment is different from the others: this test does not use the generated C executable code but is from a special TestDriver in the model checker. The graph depicts the average delta in units of logical ticks to commit a request. To do this, we number every round of Ticks that the WallclockTimer sends out to all the servers. When a client request enters the Cluster Manager, the start tick number is recorded for the request ID. When the request's corresponding Response event is sent back from the Leader to the Cluster Manager, the current tick number is recorded. The delta represents the difference between the end and the start ticks. The blue line shows the ideal case when there are no cluster membership changes or failures in our system; the number of ticks to commit a request only slightly increases from 4 to 5 as the number of servers increase. The orange line shows the test that randomly removes a server and creates a new server to add to the cluster during every request. The

number of ticks to service requests in this case is between 400 to 500. There are two possible scenarios which could explain the large difference between the two lines:
1. If the leader is removed, it will take the cluster at least 250 to 300 logical ticks to elect a new leader (choice of election timeout values described in 7.1)
2. When a new server is added to the system, it will take time in ticks linear to the length of the log for that server to catch up. This could be sped up with the usage of checkpoints which our implementation of Raft lacks.

9 FUTURE WORK

One area of potential improvement for our PRaft implementation could be to implement log compaction, which could lead to performance gains since the state machines would no longer have to maintain logs of unbounded size.

Additional work could be done to improve the existing implementation by exploring alternative approaches to how the timers are defined. For example, in our system the WallclockTimer machine emulates time through continuous Tick events, which worked for the purposes of our implementation but could perhaps be modified to not have as much overhead on the entire system. The WallclockTimer should also be swapped out for an actual OS timer written in C for application execution.

Another area for future work could be to modify our implementation to support multiple processes, run on either the same physical machine or on multiple machines spread across a hardware cluster. This would involve modifying the P runtime functions such as PrtSend to support network requests. After this, one could measure the performance of the P implementation of Raft on a hardware cluster and compare it to the etcd implementation on the same cluster for a more accurate analysis of how the two compare in a real-world setting.

10 CONCLUSION

We have presented PRaft, an implementation of the Raft consensus protocol in the P programming language. We have demonstrated that P can be used to develop and automatically verify Raft with all primary features of the protocol, including leader election, log replication, and cluster membership editing. Furthermore, we have shown through benchmarking against an open source implementation that PRaft displays promising performance results, which, combined with the formal verification of P, demonstrate the practicality of using P to build safe and performant distributed systems.

11 ACKNOWLEDGEMENTS

We would like to thank Professor John Kubiatowicz for general project advising and Ankush Desai for extensive help with the P programming language and project advising.

REFERENCES

[1] Popper, N. The Stock Market Bell Rings, Computers Fail, Wall Street Cringes. New York Times, 2015. http://www.nytimes.com/2015/07/09/business/ dealbook/new-york-stock-exchange-suspends-trading.html.
[2] Chandra, Tushar D., Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective." Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. ACM, 2007.
[3] Ankush Desai, et al. P: Safe asynchronous event-driven programming. In Proceedings of ACM SIGPLAN, 2013.
[4] Ankush Desai, et al. Compositional Programming and Testing of Dynamic Distributed Systems. International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2018.
[5] Daniel Jackson. Alloy: a lightweight object modelling notation, ACM Transactions on Software Engineering and Methodology (TOSEM), v.11 n.2, p.256-290, April 2002
[6] Chris Newcombe, et al. How Amazon web services uses formal methods, Communications of the ACM, v.58 n.4, April 2015
[7] Zave, P. Using lightweight modeling to understand Chord. ACM SIGCOMM Computer Communication Review 42, 2 (Apr. 2012), 49–57.
[8] Padon, Odel et. al. Paxos made EPR: Decidable Reasoning about Distributed Protocols. Proceedings of the ACM on Programming Languages, 2017.
[9] Evrard, Hugues, and Frédéric Lang. "Automatic distributed code generation from formal models of asynchronous concurrent processes." 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, 2015.
[10] Wilcox, James R., et al. "Verdi: a framework for implementing and formally verifying distributed systems." ACM SIGPLAN Notices 50.6 (2015): 357-368.
[11] Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." 2014 USENIX Annual Technical Conference. 2014.
[12] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for Change in a Formal Verification of the Raft Consensus Protocol. Certified Programs and Proofs (CPP), January 2016.
[13] Newcombe, Chris, et al. “How Amazon Web Services Uses Formal Methods.” Communications of the ACM, vol. 58, no. 4, 2015, pp. 66–73., doi:10.1145/2699417.