Raft Refloated: Do We Have Consensus?

Heidi Howard Malte Schwarzkopf Anil Madhavapeddy Jon Crowcroft

University of Cambridge Computer Laboratory
first.last@cl.cam.ac.uk

ABSTRACT

The Paxos algorithm is famously difficult to reason about and even more so to implement, despite having been synonymous with distributed consensus for over a decade. The recently proposed Raft protocol lays claim to being a new, understandable consensus algorithm, improving on Paxos without making compromises in performance or correctness.

In this study, we repeat the Raft authors' performance analysis. We developed a clean-slate implementation of the Raft protocol and built an event-driven simulation framework for prototyping it on experimental topologies. We propose several optimizations to the Raft protocol and demonstrate their effectiveness under contention. Finally, we empirically validate the correctness of the Raft protocol invariants and evaluate Raft's understandability claims.

1. INTRODUCTION

Much contemporary systems research is notoriously difficult to reproduce [7], despite taking place in an era of open data, open access and open source. In addition to obstacles common across all areas of Computer Science, reproducing systems research also faces the challenge of having to replicate experimental setups. However, replication of research—or indeed its transfer to production environments—can also be hampered by another factor: the *understandability* of the research described.

A prime example of an area in which reproduction of research suffers from this problem is *distributed consensus*—i.e. protocols that allow nodes in an unreliable distributed system to agree on an ordering of events. The inherent complexity of such protocols hampers their understandability, and complex configuration makes it challenging to replicate experimental setups exactly.

The "gold standard" algorithm in distributed consensus is the Paxos protocol. Lamport's original description of it [14], although highly cited, is notoriously difficult to understand. Moreover, while Lamport went on to sketch approaches to Multi-Paxos based on consecutive runs of Paxos, its under-specification has led to divergent interpretations and implementations. This has led to much work that frames the protocol in simpler terms [15, 28] or optimized it for practical systems [16, 17, 21]. A few implementations of Paxos exist [2, 3] and are used in industry systems, but hardly any implementations are publicly available.

Raft [26] is a new distributed consensus protocol that was designed to address these problems. Its authors argue that Raft is superior to Lamport's Multi-Paxos protocol as it enhances understandability while maintaining performance and correctness. If this is true, practical reproduction of Raft and its performance evaluation ought to be far easier than with Multi-Paxos. Our study in this paper evaluates the claims about Raft made by its designers. Is it indeed easily understandable, and can the encouraging performance and correctness results presented by Ongaro and Ousterhout be independently confirmed?

In the endeavour to answer this question, we re-implemented Raft in a functional programming language (OCaml) and repeat the performance evaluation from the original Raft paper [26] using our independent implementation. Disparity between the original and replicated setups, in combination with lack of detailed descriptions of experimental setups in academic publications, can hamper undertakings like ours. We indeed encountered this problem in our efforts, and document our use of a flexibly configurable simulation environment in response to this challenge.

Finally, we also review our experience of independently reproducing the results of a recent research project that used modern tools and techniques to disseminate its research artifacts. We comment on the impact that the choices of tools and support by the original authors can have on the success of a reproduction effort.

In summary, in this paper we make the following contributions:

1. We build a clean slate implementation of the Raft consensus algorithm (§2), to test the understandability of the protocol (§4.5).

2. We use our experience to describe the protocol in detail in our own words, explicitly dealing with the protocol's subtleties (§3.1).

3. We develop an event-driven distributed systems simulator and test it using our Raft consensus implementation (§3.2; simulator architecture is shown in Figure 1).

4. To calibrate our analysis, we simulate the experiment behind Figure 14 in the original paper and extend the authors' analysis to suggest optimizations to the protocol (§4).

5. To test our implementation's correctness, we model Raft nodes as non-deterministic finite state automata in Statecall Policy Language (SPL) [19, 20] and validate over 100,000 simulations traces against this model (§4.4).

6. We extend this analysis to our trace checker, which uses the simulator's holistic view of the cluster to validate the protocol's safety guarantees (§4.4).

7. We close with a recapitulation of our experience of practically reproducing a recent piece of systems research and contrast our findings with those of previous endeavours (§6).

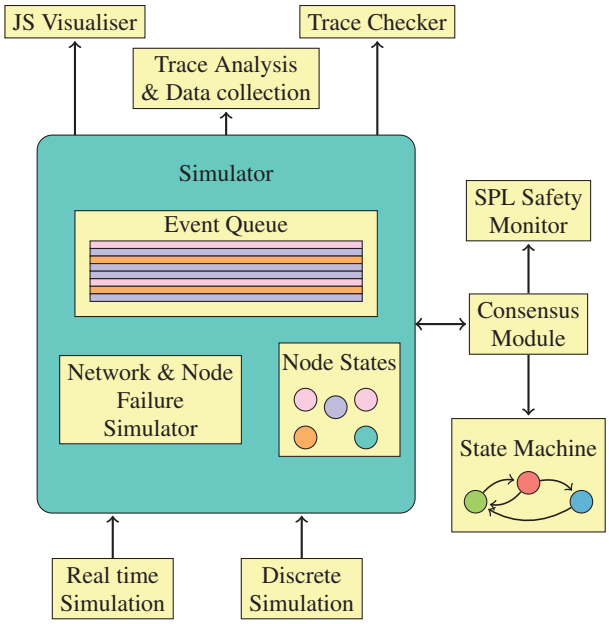


Figure 1: Overall architecture of our simulator.

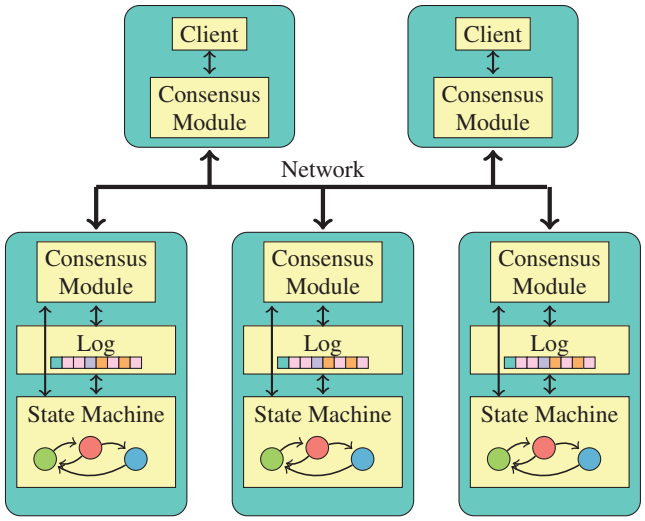


Figure 2: Components of the replicated state machine approach.

When the work started in September 2013, Raft was gaining popularity with developers and a dozen implementations of the protocol in various languages and at different stages of development existed. However, at the time, the Raft paper was yet to be published in a peer-reviewed venue and therefore there was no formal follow-up literature to validate the authors claims.

Nevertheless, the Raft authors had already met many of the requirements for good reproducibility (§5): they provided an open-source reference implementation[1] under the permissive ISC license and made the draft paper available online before formal publication. Furthermore, they also made the teaching material for the protocol, used to analyse the understandability of the algorithm against Multi-Paxos, freely available. All of these factors contributed to the success of our efforts (see §6).

2. BACKGROUND

In this section, we give an informal description of the Raft consensus algorithm. We refer the reader to the original paper [26] for the authors' description of the protocol and the associated correctness proofs. Readers who are familiar with Raft may skip directly to the next section, but should take note of one minor difference in terminology. While Ongaro and Ousterhout use the term *election timer* to refer to the time a follower waits until starting an election and do not name the other timing parameters, we refer to the parameters in a finer-grained fashion as *follower timeout*, *candidate timeout*, *leader timeout* and *client timeout*.

Distributed consensus is typically framed in the context of a replicated state machine (Figure 2), drawing a clear distinction between the *state machine* (a fault-tolerant application), the *replicated log* and the *consensus module* (handled by the consensus protocol like Multi-Paxos or Raft).

This perspective on distributed consensus mimics real-world applications: ZooKeeper [12]—currently the most popular open-source consensus application—and Chubby [3]—a fault-tolerant, distributed locking mechanism used by applications such as the Google Filesystem [9] and BigTable [5]—both use the replicated state machine approach [4].

[1] `http://github.com/logcabin/logcabin`

2.1 Assumptions

Raft consensus operates under the same set of assumptions as Multi-Paxos. The vast literature on Multi-Paxos details techniques to reduce these assumptions in specific contexts, but here we only consider classic Multi-Paxos. Its fundamental assumptions are as follows:

1. the distributed system is asynchronous, i.e. no upper bound exists for the message delays or the time taken to perform computation, and we cannot assume global clock synchronization;
2. network communication between nodes is unreliable, including the possibility of network delay, partitions, packet loss, duplication and re-ordering;
3. Byzantine failures [27] cannot occur; and
4. clients of the application using the protocol must communicate with the cluster via the current leader, and it is their responsibility to determine which node is currently leader.

Furthermore, Raft makes the following implementation assumptions, some of which can be relaxed with further engineering effort:

1. the protocol has access to infinitely large, monotonically increasing values;
2. the state machines running on each node all start in the same state and respond deterministically to client operations;
3. nodes have access to infinite persistent storage that cannot be corrupted, and any write to persistent storage will be completed before crashing (i.e. using write-ahead logging); and
4. nodes are statically configured with a knowledge of all other nodes in the cluster, that is, cluster membership cannot change dynamically.

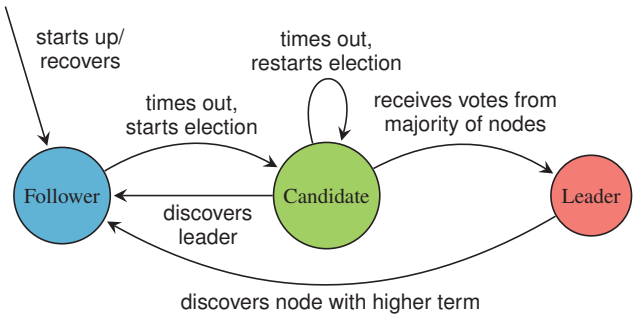


Figure 3: State transition model for Raft leader election.

The latter two points can be relaxed by extending Raft: log compaction [24] helps support Raft on limited storage and a dynamic membership extension [23, 26] has been proposed by Ongaro and Ousterhout. Both of these extensions are beyond the scope of this paper, however.

2.2 Approach

The clients interact with the replicated state machine via *commands*. These commands are given to the consensus module, which determines whether it is possible to commit the command to the replicated state machine and does so if possible. Once a command has been committed, the consensus protocol guarantees that the command is eventually committed on every live state machine and that it is committed in the same order. This provides linearisable semantics to the client: each command from the client appears to execute instantaneously, exactly once, at some point between its invocation and positive response.

In the pursuit of understandability and in contrast to similar algorithms such as Viewstamped Replication [18, 22], Raft uses strong leadership, which extends the ideas of leader-driven consensus by adding the following conditions:

1. all message passing between system nodes is initiated by a leader (or a node attempting to become leader). The protocol specification makes this explicit by modelling communications as RPCs, differentiating between distinctly active or passive node roles;

2. clients are external to the system and must contact the leader directly to communicate with the system; and

3. for a system to be *available*, it is necessary (but not sufficient) for a leader to have been elected. If the system is in the process of electing a leader, it is unavailable, even if all nodes are up.

2.3 Protocol Details

Each node has a consensus module, which is always operating in one of the following modes:

- **Follower:** A passive node which only responds to RPCs and does not initiate any communication.

- **Candidate:** An active node which is attempting to become a Leader. It initiates a request for votes from other nodes (the *RequestVote* RPC).

- **Leader:** An active node which is currently leading the cluster. This node handles requests from clients to interact with the replicated state machine (via *AppendEntries* RPCs).

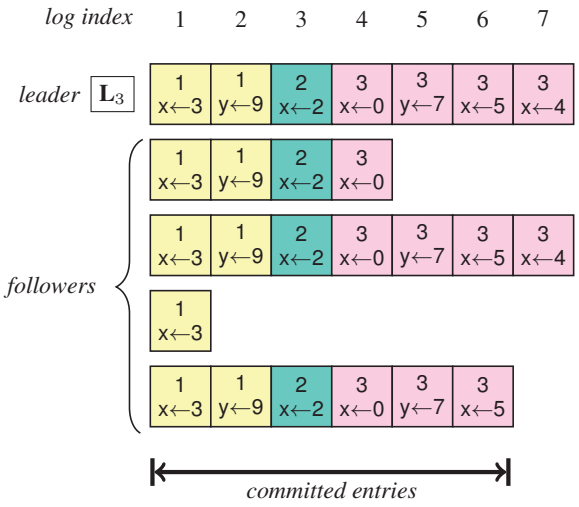


Figure 4: Example logs for a cluster of five nodes. $\boxed{\mathbf{L}_i}$ denotes the leader for term i, while black bars indicate the commit threshold.

Since Raft cannot assume global clock synchronization, global partial ordering on events is achieved with a monotonically increasing value, known as a *term*. Each node stores its perspective of the term in persistent storage. A node's term is only updated when it starts (or restarts) an election, or when it learns from another node that its term is out of date. All messages include the source node's term. The receiving node checks it, with two possible outcomes: if the receiver's term is larger, a negative response is sent, while if the receiver's term is smaller than or equal to the source's, its term is updated before parsing the message.

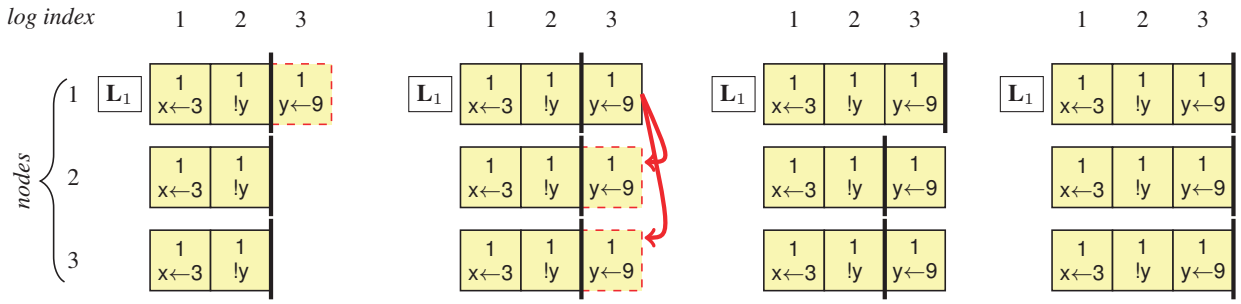
2.3.1 Leader Election

On start-up or recovery from a failure, a node becomes a follower and waits to be contacted by the leader, which broadcasts regular empty *AppendEntries* RPCs. A node operating as a follower will continue to be in follower state indefinitely unless it neither hears from a current leader, or it grants a vote to a candidate (details below). If neither of these occur within its *follower timeout*, the follower node transitions to a candidate.

On becoming a candidate, a node increments its term, votes for itself, starts its *candidate timeout* and sends a *RequestVote* RPC to all other nodes. Figure 3 shows the non-deterministic finite automaton (NFA) for Raft, in which there are three possible outcomes of this election. The candidate either *(i)* receives a strict majority of votes and becomes leader for the term, or *(ii)* fails to receive enough votes and restarts the election, or *(iii)* learns that its term is out of date and steps down. A follower only votes for one node per term. The vote is stored on non-volatile storage and the term increases monotonically such that at most one node becomes leader per term.

2.3.2 Log Replication

Once a node has established itself as a leader, it can service requests for the replicated state machines. Clients contact the leader with commands to be committed. On receipt, the leader assigns a term and index to the command. This uniquely identifies the command in the nodes' logs. The leader then tries to replicate the command to the logs of a strict majority of nodes. If successful, the command is committed, applied to the state machine of the leader



(a) $y \leftarrow 9$ arrives at leader. **(b)** $y \leftarrow 9$ is replicated. **(c)** Leader commits. **(d)** Commit is replicated.

Figure 5: Example of a simple, failure free commit: node 1 commits $y \leftarrow 9$ in term 1 at index 3.

and the result returned to the client.

Figure 4 shows an example set of logs where each row represents the log of a node, each column denotes the index into the log and each log entry contains the command and its associated term (denoted by entry colour). The state machine is a key-value store and the possible commands are *add* (e.g. $x \leftarrow 5$ associates 5 with key x) and *find* (e.g. $!y$ returns the value associated with the key y). In Figure 4, the first node is leader and has committed the first six entries in the log as they have been replicated on a majority of nodes. Nodes 2 and 4 may have failed or had their messages lost/delayed in the network. Therefore, their logs have fallen behind.

Consider a case where no nodes fail and all communication is reliable. We can safely assume that all nodes are in the same term and all logs are identical. The leader broadcasts *AppendEntries* RPCs, which includes the log entry that the leader is trying to replicate. Each node adds the entry to its log and replies with *success*. The leader then applies the command to its state machine, updates its *commit index* and returns the result to the client. In the next *AppendEntries* message, the leader informs all the other nodes of its updated commit index. The nodes apply the command to their state machines and update their commit index in turn. This process is shown in Figure 5.

In this example of a simple commit, all nodes' logs are identical to start with, their commit indexes are 2, and node 1 is the term 1 leader. In Figure 5a, node 1 receives the command $y \leftarrow 9$ from the consensus module and appends it to its log. Node 1 then dispatches *AppendEntries* RPCs to nodes 2 and 3. These are successful, so node 2 and 3 add $y \leftarrow 9$ to their logs, as shown in Figure 5b. In Figure 5c, node 1 hears of this success, it updates its commit index (denoted by a thick black line) from 2 to 3, applies the command to its state machine and replies to the client. A later *AppendEntries* from node 1 updates the commit indexes of nodes 2 and 3 from 2 to 3, as shown in Figure 5d.

Now, consider the case that some messages have been lost or nodes have failed and recovered, leaving some logs incomplete. It is the responsibility of the leader to fix this by replicating its log to all other nodes. When a follower receives an *AppendEntries* RPC, it contains the log index and term associated with the previous entry. If this does not match the last entry in the log, the node sends an *unsuccessful* response to the leader. The leader is now aware that the follower's log is inconsistent and needs to be updated. The leader decrements the previous log index and term associated with that node. The leader keeps dispatching the *AppendEntries* RPC, adding entries to the log until the follower node replies with *success* and is therefore up to date.

Each node keeps its log in persistent storage, including a history of all commands and their associated terms. Each node also has a commit index, which represents the most recent command to be applied to the replicated state machine. When the commit index is updated, the node passes all commands between the new and old commit index to the local application state machine.

2.3.3 *Safety and Extra Conditions*

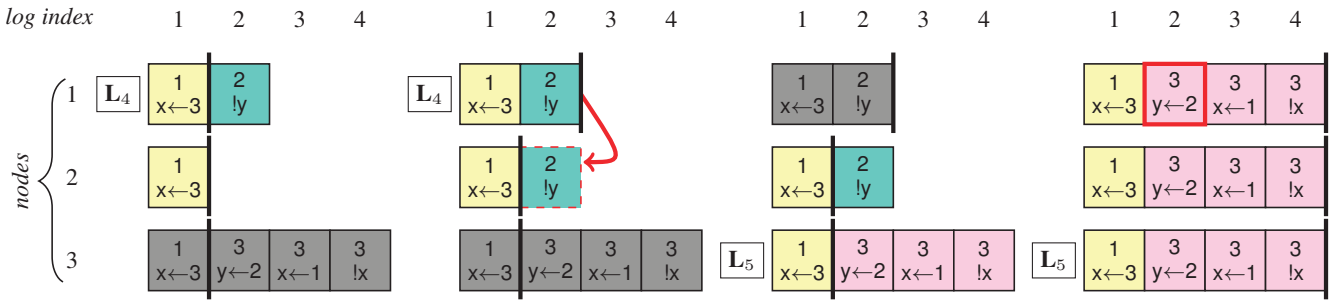
To ensure safety, the above description must be augmented with several extra conditions. Since the leader duplicates its log to all other logs, this log must include all previously committed entries. To achieve this, Raft imposes further constraints on the protocol detailed so far. First, a node will only grant a vote to another node if its log is at least as up-to-date as the other node's (defined as either having a last entry with a higher term or, if terms are equal, a longer log), this is later referred to as the extra condition on leadership.

The leader is responsible for replicating its log to all other nodes, including committing entries from the current and previous terms. In the protocol as described so far, however, it is possible to commit an entry from a previous term, for a node without this entry to be elected leader and to overwrite the already-committed entry.

For instance, in Figure 6a, node 1 is the term 4 leader, node 2 is a follower and node 3 has failed. As leader, node 1 is responsible for replicating its log to the other nodes. In this case, this means replicating $!y$ at index 2 from term 2. Node 1 successfully appends $!y$ to the log of node 2. Figure 6b then shows that since the entry is now replicated onto two of three nodes, node 1 commits $!y$ at index 2 from term 2 by updating its commit index (denoted by the thick black line) from 2 to 3. The leader, node 1, then fails and node 3 recovers. Figure 6c shows the result of the leader election which is held: node 3 wins in term 5 (since the leader's term must be strictly greater than 4). Node 1 quickly recovers and follows the current leader (node 3 in term 5). Since, as before, the leader is responsible for replicating its log to the other nodes, node 3 replicates its log to nodes 1 and 2.

Finally, in Figure 6d, this ends up being inconsistent with $!y$ at index 2, which was already previously committed by node 1. This is an impossible state and node 1 and 3 can now never agree. Raft addresses this problem by restricting the leader to only commit entries from any previous term if the same leader has already successfully replicated an entry from the *current* term. This is later referred to as the "extra condition on commitment".

The Raft protocol provides linearisable semantics [10], guaranteeing that a command is committed between the first time it is dispatched and the first successful response. The protocol, however, does not guarantee a particular interleaving of client requests—but



(a) Node 3 fails and node 1 is leader. **(b)** Node 1 commits $!y$. **(c)** Node 3 is the new leader. **(d)** $!y$ is overwritten at node 1.

Figure 6: Example of Leader Completeness violation, without the extra condition on commitment. The entry $!y$ at index 2, term 2 is committed by node 1 (the leader in term 4) but later overwritten by node 3 (the leader in term 5). Grey logs correspond to failed nodes.

it *does* guarantee that all state machines will see commands in the same order. It also assumes that if a client does not receive a response to a request within its *client timeout* or the response is negative, it will retry the request until successful. To provide linearisable semantics, Raft must ensure that each command is applied to each state machine at most once. To ensure this, each client command has an associated serial number. Each consensus module caches the last serial number processed from each client and each response given. If a consensus module is given the same command twice, then the second time it simply returns the cached response.

The protocol does not allow "false positives", i.e. claiming that a command has been committed when it in fact has not. However, the protocol may give false negatives, claiming that a command has not been committed when in fact it has. To accommodate this, the client semantics specify that each client must retry a command until it has been successfully committed (see above). Each client may have at most one command outstanding at a time and commands must be dispatched in serial number order.

The Raft authors claim that the protocol described provides the guarantees listed below to a distributed consensus implementation.

> **Election Safety:** at most one leader can be elected in a given term.
> **Leader Append-Only:** a leader never overwrites or deletes entries in its log; it only appends new entries.
> **Log Matching:** if two logs contain an entry with the same index and term, then the logs are identical in all entries up to and including the given index.
> **Leader Completeness:** if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
> **State Machine Safety:** if a node has applied a log entry at a given index to its state machine, no other node will ever apply a different log entry for the same index.

We check the correctness of these conditions for our implementation in §4.4.

3. IMPLEMENTATION

While the Raft algorithm is well-described, its utility as an easily understandable and reproducible consensus algorithm can only be tested by experiment. We therefore built a clean-slate implementation of the consensus algorithm from the description provided in the paper and associated teaching materials [25]. Using a modular OCaml-based approach, we: *(i)* separated the Raft protocol's state transitions; and *(ii)* built domain-specific frontends such as a trace checker and simulation framework (see Figure 1).

3.1 Functional Protocol Implementation

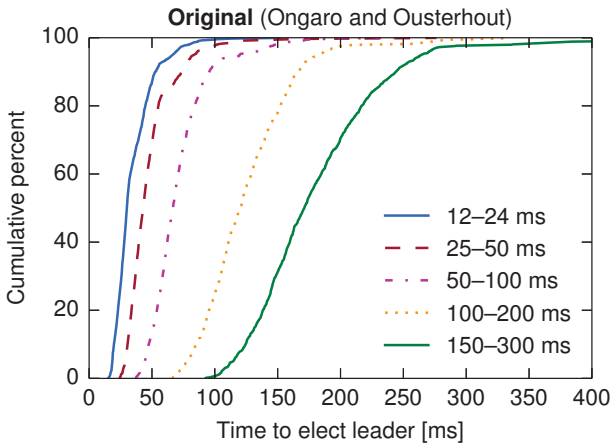
As we are aiming to validate Raft's safety guarantees, we implemented the protocol with a small functional core that is strictly limited to states laid out in the protocol. We chose OCaml as the implementation language for reproduction (as compared to the original implementation's C++) due to its static typing and powerful module system. The core protocol implementation is pure and does not include any side-effecting code, and uses algebraic data types to restrict the behaviour of the protocol to its own safety criteria. Where the static type system was insufficient, we made liberal use of assertion checks to restrict run-time behaviour.

Raft's state transition models (such as Figure 3) are encoded in Statecall Policy Language (SPL) [19, 20]. SPL is a first order imperative language for specifying non-deterministic finite state automata (NFA). We chose to use SPL due to its ability to be compiled to either Promela, for model checking in SPIN, or to OCaml, to act as a safety monitor at run-time. Alternatives to SPL include using systems such as MoDist [31] that model-check protocol implementations directly, or encoding the model directly in Promela, as has been recently done for Paxos [8].

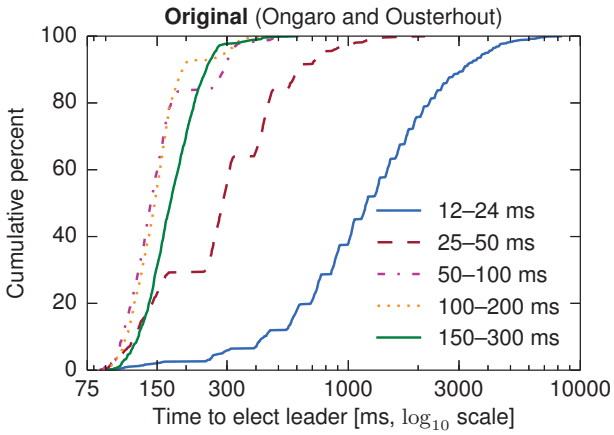
3.2 Event-driven Simulation

In order to evaluate our Raft protocol implementation across a range of diverse network environments, we also built a message-level network simulation framework in OCaml. Beyond evaluating the performance of the protocol, we can also use our simulation traces to catch subtle bugs in our implementation or the protocol specification. Since such issues may only occur very rarely (e.g. once in 10,000 protocol runs), a fast simulator offers the unique opportunity to address them via simulation trace analysis. Furthermore, we can use our simulator's holistic view of the cluster to ensure that all nodes' perspectives of the distributed system are consistent with respect to the protocol. To meet these domain-specific requirements, we designed our own simulation framework, instead of opting for traditional event-driven network simulators like `ns3` or OMNeT++ [30].

We reason about the protocol in terms of events which cause the nodes of the cluster to transition between states in NFAs such as that in Figure 3. These events are all either temporal (e.g. the protocol's timers) or spatial (e.g. receiving RPCs from other nodes or

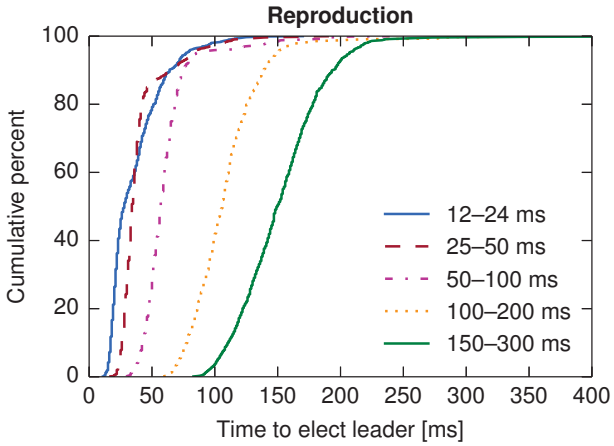


(a) Varying T in follower timeout ranges $(T, 2T)$.

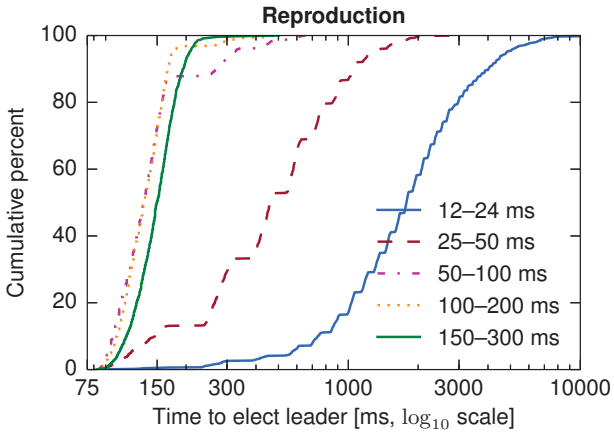


(b) Varying non-determinism in follower timeout.

Figure 7: *Authors' original results:* cumulative distribution function (CDF) of time to elect a leader; follower timeout ranges in legends.



(a) Varying T in follower timeout ranges $(T, 2T)$.



(b) Varying non-determinism in follower timeout.

Figure 8: *Reproduction results:* cumulative distribution function (CDF) of time to elect a leader; follower timeout ranges in legends.

clients). This perspective lends itself to event-driven simulation, where events are held in a global priority queue and applied sequentially to the nodes' states. Each event in the queue holds information as to which node it operates on, the time it operates and the operation itself, which results in new state for the node and new events to be added to the queue.

Our simulator supports two modes of operation, *Discrete Event Simulation* (DES) and *Real-time Event Simulation* (RES). This allows us to vary the trade-off between simulation accuracy and time. Event-driven simulation, particularly DES, allows us to rapidly iterate over the vast state space of simulation parameters to evaluate performance and detect rarely occurring safety violations.

4. RESULTS

We now review the results of the original performance evaluation (§4.1) and compare them against our results (§4.2). We suggest optimizations to the protocol based on explorations using our simulator (§4.3) and evaluate Raft's safety guarantees (§4.4).

4.1 Original Results

The Raft authors' performance analysis considers the time taken to replicate a log entry in a typical case and the worst case time to establish a new leader after a leader failure. For the former, the authors highlight that this will, in the typical case, take one round trip time to half the nodes in the cluster and can be optimized further with batching. For the latter, the authors measure the time taken to elect a new leader after the previous leader fails in a realistic worst case. In the following, we will focus on the latter experiment.

Figure 7 reprints the results from Ongaro and Ousterhout's evaluation of the duration of a Raft leader election.[2] The raw results were not publicly available, but the authors were happy to provide the data for us.

Likewise, the experimental set-up was not described in the paper in sufficient detail to reproduce it, but the authors were happy to provide details on request and they are now documented pub-

[2]Reproduced with permission from the USENIX ATC paper [26, Fig. 15] and Ongaro's PhD thesis [23, Fig. 10.2].

licly [23]. The original experimental set-up used the authors' LogCabin C++11 implementation of Raft, across five idle machines connected via a 1Gb/s Ethernet switch, with an average broadcast time of 15ms.

The authors aimed to generate the worst-case scenario for leader election. Two nodes were ineligible for leadership due to the extra condition on leadership (see §2.3.3). This simulates two of the five nodes having recently recovered from a failure, thus having their log not yet up to date. Hence, they will not receive votes from the other nodes. Furthermore, the setup also encouraged split votes by broadcasting heartbeat *AppendEntries* RPCs before inducing a crash in the leader node.

The follower and candidate timeouts were taken from a uniform distribution, with the bounds shown in the legends in Figure 7. The leader timeout was set to half the lower bound of the follower/candidate timeout. Figure 7 shows cumulative distribution functions (CDFs) of the time between the crash of an old leader and a new leader being established. Figure 7a varies the timeout from T to $2T$, for different values of T. Figure 7b represents the time taken to establish a leader when the follower timeout has varying degrees of non-determinism, with the minimal follower timeout fixed at 150ms.

4.2 Our Simulation Results

We used the information about Ongaro and Ousterhout's experimental setup to calibrate our simulation parameters (such as the packet delay distribution and the number of nodes). The time taken to elect a new leader is sufficiently short that we can assume the absence of node failures during the election. As the original experiments used TCP/IP for RPCs, we did not need to simulate packet loss. Instead, we use a long tail for packet delay.

Our results from simulating the authors' experimental set-up are shown in Figure 8. Each curve represents 10,000 traces from the discrete event simulation running at nanosecond granularity. We observe that our simulation generally took longer to establish a leader than the authors' measurements in the case of high contention. Otherwise, the results are very similar.

By contrast to the results shown, our early results greatly differed from the authors' original results. Our early experiments showed that making some nodes ineligible for leadership due to the log consistency requirements does not actually reflect the worst case. Though reducing the number of eligible nodes increases the average time until the first node times out, it also reduces the contention for leadership. Therefore, it does not in fact simulate the *worst* case (which requires high contention)—hence our initial experiments did not make any of the nodes ineligible for leadership.

Furthermore, since we were simulating the worst case behaviour, we always dispatched *AppendEntries* RPCs from the leader just before crashing, unlike the Raft authors. Since doing so resets the follower timeouts, the authors' results were reduced by $\sim U(0, \frac{T}{2})$, as otherwise *AppendEntries* RPCs are dispatched every $\frac{T}{2}$, where T is the lower limit of the follower timeout range.

Following consultation with the authors, it emerged that we had used different experimental setups. At the time of performing this evaluation, the Raft paper under-specified the experimental setup used, but emphasised that it was a "worst-case scenario". This has since been clarified in the paper before its publication.

After observing that the stepping in the curves changed in comparison with the authors' results, we experimented with different simulation granularities, but observed no significant difference between the levels of granularity tested (with the exception of using millisecond granularity for the 150–151ms case, as expected). Despite this, we chose to proceed with experiments at nanosecond granularity due to a marginal improvement in results and negligible increase in computation time.

	Follower timeout (ms)		
	150–300	150–200	150–155
Original version	10.8	14.6	108.2
Fixed reduction	10.9	15.0	53.0
Exponential backoff	11.1	13.9	87.9
Combined	11.7	14.1	48.6

Table 1: Mean number of packets to elect a leader.

4.3 Proposed Optimizations

We were then able to use our framework to rapidly prototype optimizations to the protocol by virtue of having calibrated our simulation by replicating the authors' original experimental setup. This section details three such examples.

Different Follower and Candidate Timers. The authors' guidance for setting the protocol's timers is summarized below.

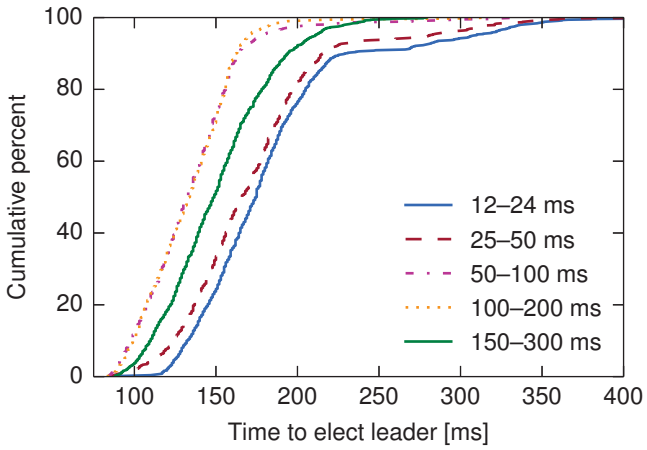
$$\textit{broadcast time} \ll \textit{candidate timeout} \ll \textit{MTBF}$$
$$\textit{candidate timeout} = \textit{follower timeout} \sim U(T, 2T)$$
$$\textit{leader timeout} = \tfrac{T}{2}$$

They suggest T=150ms as a suitable parameter for the protocol and results with this value can be seen in Figure 7. Our hypothesis, however, was that the time taken to elect a leader in a highly contested environment is significantly improved by not simply setting the candidate timer to the same range as the follower timer. As the authors use the same timer range for candidates and followers, they are waiting a minimum of 150ms (and up to twice that) before restarting an election. This is despite the fact that, on average, a node receives all of its responses within 15ms. Figure 9a instead sets the minimum *candidateTimeout* to $\mu + 2\sigma$. Assuming *broadcastTime* $\sim N(\mu, \sigma)$, this is sufficient 95% of the time. We can see that in a highly contested environment (the 150–155ms case), 95% of the time leaders are established within 281ms (Figure 9a), compared to 1330ms without the optimization (Figure 8b).

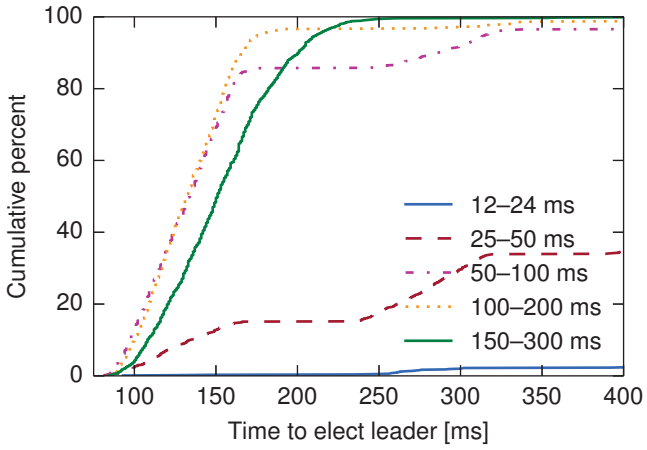
Binary Exponential Backoff. We further hypothesised that we might improve the time taken to elect a leader by introducing a binary exponential backoff for candidates that have been rejected by a majority of replicas. Figure 9b shows the improvement in leader election time gained by enabling binary exponential backoff and Figure 9c shows the effect of combining both optimizations. Both optimizations performed considerably better than the original implementation in isolation, but the combined optimizations performed slightly worse than the fixed reduction optimization alone.

Figure 10 shows the number of packets sent to elect a leader in the 150–200ms case without any optimization. The minimum number of packets is six, representing a single node dispatching *AppendEntries* RPC to the other four nodes and receiving votes from the first two. The plot clusters around multiples of eight: traces under eight represent the first node timing out and winning the election, traces between 8 and 16 represent two nodes timing out before a leader is elected.

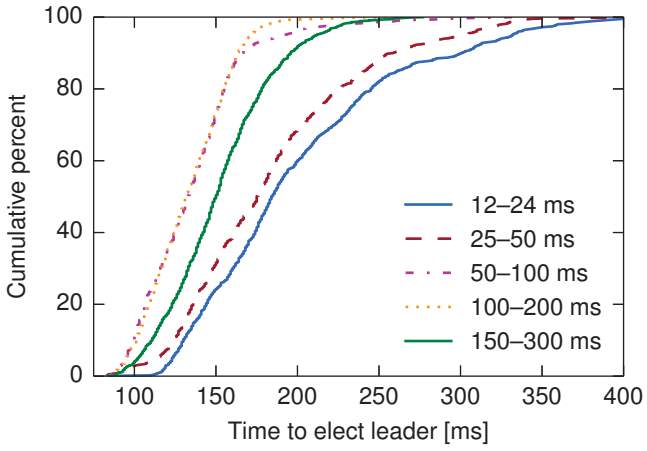
We were concerned that the reduction in time to elect leader might come at the cost of increasing the network load by running additional elections. Table 1 demonstrates that this is not the case: we observe a marginal increase in the low contention cases and a significant reduction in the highly contended cases.



(a) Candidate timeout set to $X \sim U(23, 46)$.



(b) Binary exponential backoff for candidate timeout.



(c) Combining both optimizations from (a) and (b).

Figure 9: Investigating the impact of alternative candidate timeouts, while keeping the range of follower timeouts fixed. The follower timeouts in milliseconds are shown in the legends.

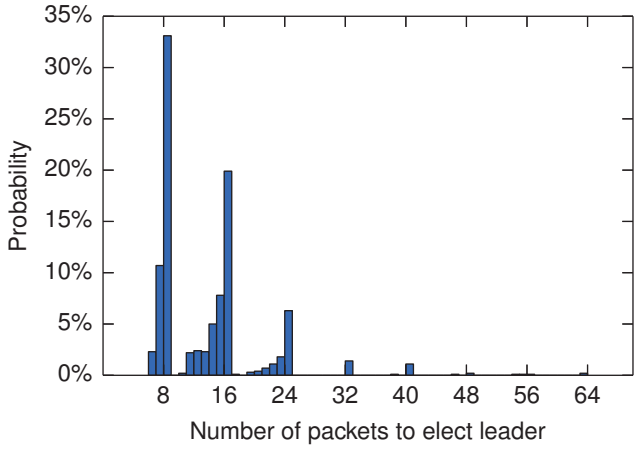


Figure 10: Distribution of the number of packets to elect a leader.

Client Timeout Disaggregation. As expected, the vast majority of client commands were committed with a latency of approximately *BroadcastTime* + *RTT*, where RTT refers to the round trip time between the client and the leader. A small minority of commands took considerably longer, most commonly representing leader failure and subsequent election of a new leader. A significant overhead was observed by clients waiting for responses from failed nodes during leader discovery. Unlike the other RPCs used by Raft, the client commit timer, which is used by clients to retransmit requests to nodes, is much higher than the RTT. This is because the client must wait while the leader replicates the client command to the cluster nodes. Introducing a *ClientCommit* leader acknowledgement would allow us to split this into two distinct timers, where first is just above RTT and handles retransmitting normal *ClientCommit* requests, while the second is used after the client has received a leader acknowledgement and is much higher to allow for time to replicate the request.

4.4 Correctness

We checked each of our thousands of simulation traces for correctness of the Raft safety guarantees using the SPL safety monitor and extensive assertion checking. At no point were the authors' claims about the safety guarantees violated.

However, we did observe permanent livelock in some of our simulation traces, caused by the interaction between the extra condition on commitment (detailed in §2.3.3) and our placement of the client request cache. We recommend that if a client request is blocked by the extra condition on commitment, the leader should create a no-op entry in its log and replicate this across the cluster. We refer the reader to our technical report [11] for a detailed analysis of this issue.

4.5 Understandability

Raft's goal of developing an understandable consensus algorithm addresses a clear need in the distributed systems community. In our experience, Raft's high level ideas were easy to grasp—more so than with Paxos. However, the protocol still has many subtleties, particularly regarding the handling of client requests. The iterative protocol description modularizes and isolates the different aspects of the protocol for understandability by a reader, but this in our experience hinders implementation as it introduces unexpected inter-

action between components (see previous section). As with Paxos, the brevity of the original paper also leaves many implementation decisions to the reader.

Some of these omissions, including detailed documentation of the more subtle aspects of the protocol and an analysis of the authors' design decisions, are rectified in Ongaro's PhD thesis on Raft [23]. However, this only became available towards the end of our effort. Despite this, we believe that Raft has achieved its goal of being a "more understandable" consensus algorithm than Paxos.

5. EXPERIENCE SUMMARY

Simulating experimental setups proved more challenging than expected, as the nature of functional programming and simulation forced us to be highly specific about otherwise implicit parameters. The simulator provides an overwhelming number of configuration parameters and we had great difficulties choosing a suitable statistical distribution for simulating packet delay. Simulation is inherently limited to approximating behaviour.

In addition, our effort was complicated by the regular revisions to the draft of the Raft draft paper. It would have helped us keep track if an explicit "change log" had been available although the original research was still ongoing at the time.

Our approach effectively separates a functional core protocol implementation from language and domain-specific considerations. Event-driven simulation allows us to assess the claims about a system with minimal infrastructure and overhead. Readers can easily reproduce our results: each simulator configuration file offers complete formal documentation of the experiment setup investigated.

Our take-aways from this work are perhaps best summarized as follows:

1. **Modelling protocol state transitions:** the Statecall Policy Language allows researchers to model their protocol and compile this model for validation in the SPIN model checker and to an OCaml runtime safety model, ensuring that the implementation is consistent with the verified model.

2. **Simulation front end:** building systems with interfaces for simulations is worthwhile. Doing so not only aids debugging and testing, but simulation configuration files provide a formal specification of experimental set-ups.

3. **Coordinating a community around a project:** Raft is an excellent example of a research project encouraging reproduction. In particular, Ongaro and Ousterhout's Raft mailing list and the project webpage, with its directory of implementations, resources and talks which can be contributed to directly via `git` pull requests, are highly useful.

More generally, reproducibility of systems research is a topic that receives recurring interest. Collberg *et al.* [7] recently surveyed the reproducibility of 613 papers from top systems venues. They were only able to set up reproduction environments based on the authors' code in fewer than 25% of cases, and consequently proposed a specification sharing syntax for inclusion in paper headers at submission and publication time. Although such specifications would certainly be an improvement over the current situation, the specifications are necessarily static. Yet, reproducibility can change over time: for example, this project benefited from improved reproducibility of Raft as the popularity of the protocol increased.

Broad studies of reproducibility like Collberg *et al.*'s necessarily apply somewhat crude heuristics to publications (as done similarly in efforts by Kovacevic [13] and Vadewalle [29]). By contrast, our work is an in-depth analysis of Raft. In a similar vein, Clark *et al.* performed an in-depth analysis to reproduce the Xen hypervisor paper's results [6]. Both approaches have their own merits and limitations.

Platforms for disseminating research artifacts such as the Executable Paper Grand Challenge[3] have been largely ignored by the wider community. This may be because the community has, in some cases, evolved its own channels for distributing research artifacts and encouraging reproducibility. For example, during the course of this project, a strong community has emerged around Raft, with over 40 different implementations in languages ranging from C++, Erlang and Java to Clojure, Ruby and other OCaml implementations.[4] The most popular implementation, `go-raft`, is the foundation of the service discovery and locking services in CoreOS [1].

6. CONCLUSIONS

We have shown that the Raft consensus algorithm meets its goal of being an understandable, easily implementable consensus protocol. In a well-understood network environment, the Raft protocol behaves admirably, provided that suitable protocol parameters such as follower timeouts and leadership discovery methods have been chosen. Reproducing the performance results proved more challenging than expected, but support from the authors made it possible for us to not only validate the results, but also to recommend a number of optimizations to the protocol.

As our results demonstrate, our simulator is a good approximation to Raft's behaviour. It is also a useful tool for anyone planning to deploy Raft to rapidly evaluate a range of protocol configurations on a specific network environment.

Our source code is available as open-source software under a MIT license at:
`https://github.com/heidi-ann/ocaml-raft` with tag `v1.0`, and can be installed via the OPAM package manager as `raft-sim.1.0`. The datasets used are also available at:
`https://github.com/heidi-ann/ocaml-raft-data`, also tagged as `v1.0`.

There are many future directions for this simulation framework that we hope to develop further. There are many experimental parameters to explore such as packet loss, cluster scale, heterogenous nodes, dynamic network topologies. The Promela automata exported from SPL could also be model checked with more temporal assertions to guide the configuration of the simulation into unexplored parameter spaces.

7. ACKNOWLEDGEMENTS

We would like to thank Ionel Gog, Diego Ongaro, David Sheets and Matthew Huxtable for their feedback on this paper. Some of the authors were supported by Horizon Digital Economy Research, RCUK grant EP/G065802/1, and a portion was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

[3] `http://www.executablepapers.com`
[4] `http://github.com/mfp/oraft`

References

[1] CoreOS website. http://coreos.com. Accessed on 02/09/2014.

[2] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[3] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, 2006.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, 2007.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. BigTable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[6] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews. Xen and the art of repeated research. In *Proceedings of the USENIX Annual Technical Conference*, pages 135–144, 2004.

[7] C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren. Measuring reproducibility in computer systems. Technical report, University of Arizona, 2014.

[8] G. Delzanno, M. Tatarek, and R. Traverso. Model Checking Paxos in Spin. *ArXiv e-prints*, Aug. 2014.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[10] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[11] H. Howard. ARC: Analysis of Raft Consensus. Technical Report UCAM-CL-TR-857, University of Cambridge, Computer Laboratory, July 2014.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC)*, volume 8, pages 145–158, 2010.

[13] J. Kovacevic. How to encourage and publish reproducible research. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 1273–1276, April 2007.

[14] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[15] L. Lamport. Paxos made simple. *ACM SIGACT News 32.4*, pages 18–25vi, 2001.

[16] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[17] L. Lamport and M. Massa. Cheap Paxos. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 307–314, 2004.

[18] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, 2012.

[19] A. Madhavapeddy. *Creating high-performance statically type-safe network applications*. PhD thesis, University of Cambridge, 2006.

[20] A. Madhavapeddy. Combining static model checking with dynamic enforcement using the statecall policy language. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, pages 446–465, 2009.

[21] D. Mazieres. Paxos made practical. http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf. Accessed on 02/09/2014.

[22] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, 1988.

[23] D. Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

[24] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm (extended version). http://ramcloud.stanford.edu/raft.pdf. Accessed on 13/09/2014.

[25] D. Ongaro and J. Ousterhout. Raft: A consensus algorithm for replicated logs (user study). http://www.youtube.com/watch?v=YbZ3zDzDnrw. Accessed on 02/09/2014.

[26] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference*, 2014.

[27] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[28] R. Van Renesse. Paxos made moderately complex. http://www.cs.cornell.edu/courses/cs7412/2011sp/paxos.pdf, 2011. Accessed on 02/09/2014.

[29] P. Vandewalle, J. Kovacevic, and M. Vetterli. Reproducible research in signal processing. *Signal Processing Magazine, IEEE*, 26(3):37–47, 2009.

[30] A. Varga et al. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference*, volume 9, page 185, 2001.

[31] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 213–228, 2009.