On the Parallels between Paxos and Raft, and how to Port Optimizations

Zhaoguo Wang[†‡], Changgeng Zhao[*], Shuai Mu[◇], Haibo Chen[†‡], Jinyang Li[*]
† Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
‡ Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
∗ Department of Computer Science, New York University
◇ Department of Computer Science, Stony Brook University

ABSTRACT

In recent years, Raft has surpassed Paxos to become the more popular consensus protocol in the industry. While many researchers have observed the similarities between the two protocols, no one has shown how Raft and Paxos are formally related to each other. In this paper, we present a formal mapping between Raft and Paxos, and use this knowledge to port a certain class of optimizations from Paxos to Raft. In particular, our porting method can automatically generate an optimized protocol specification with guaranteed correctness. As case studies, we port and evaluate two optimizations, Mencius and Paxos Quorum Lease to Raft.

KEYWORDS

Paxos, Raft, optimization porting

ACM Reference Format:
Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, Jinyang Li. 2019. On the Parallels between Paxos and Raft, and how to Port Optimizations. In *2019 ACM Symposium on Principles of Distributed Computing (PODC'19), July 29–August 2, 2019, Toronto, ON, Canada. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3293611.3331595*

1 INTRODUCTION

Consensus protocols enable servers to reach agreement on the sequence of operations to execute despite the failure of some servers and arbitrary network delays. Classic Paxos [18] is one of the oldest and most well-studied consensus protocols. However, in recent years, Raft [31] has gradually overtaken Paxos as the consensus protocol of choice, esp. in the industry. Many researchers have observed that Raft and Paxos bear certain similarities. However, no one has shown how the two protocols are related in the formal sense. In fact, does such a formal relationship exist?

While it may seem like a pedantic endeavor, investigating a formal mapping between Raft and Paxos is meaningful for two reasons. First, making the connection between Raft and Paxos helps deepen our understanding of both protocols. In particular, it allows us to articulate what design decisions have made Raft more understandable or more efficient than Paxos. Second, Paxos is not an isolated protocol but consists of a large family of variants and optimizations as a result of almost two decades of research [16, 7, 25, 22, 29, 28]. These Paxos variants range from reducing latency for wide-area operation, balancing replica load, optimizing for mostly-conflict-free workload, to tolerating Byzantine faults. Knowing how Raft relates to Paxos allows one to port some of these optimizations to Raft without having to reinvent the wheel.

In this paper, we attempt to make a formal connection between Raft and Paxos using refinement mapping. We show that, beyond the broad stroke similarities between the two protocols, Raft differs from Paxos in several subtle details, such as allowing a follower to erase extra entries if its log is longer than the leader. Unfortunately, these differences between the two protocols prevent a direct refinement mapping between them. Therefore, we craft a variant of Raft, called Raft*, which is a refinement of Paxos by removing these superficial differences.

We use the refinement mapping between Raft* and Paxos to port existing ideas in the Paxos literature to the world of Raft. Specifically, we develop an automatic porting method which is able to port a certain class of Paxos optimizations to Raft*. The specific class of optimizations that can be ported automatically are those that do not mutate the original state in Paxos. For these optimizations, we derive the set of rules for applying them to Raft*, such that the resulting protocol automatically refines Raft* and the Paxos optimization, thus guaranteeing correctness. As Raft* is very similar to Raft, the derived protocol contains all Raft properties and is improved by the Paxos optimization. As case studies, we choose two published Paxos optimization, Mencius [25] and Paxos Quorum Lease [28], each of which improves one or more aspects of Paxos in terms of load-balancing and latency. We have ported these two protocols to Raft*.

We evaluate the performance benefits of our Raft* optimizations on Amazon AWS in a setup where data is replicated across several geographically separated data centers. For each optimization, we show that the Raft variant has similar benefits as its Paxos counterparts in the literature.

To summarize, we make the following contributions:

- We reveal the formal relationship between Raft and Paxos by showing a refinement mapping between Paxos and Raft*, a close variant of Raft (Section 3).
- We define the problem of porting optimizations across protocols and develop a methodology for automatically porting a

PODC '19, July 29-August 2, 2019, Toronto, ON, Canada

ACM ISBN 978-1-4503-6217-7/19/07…$15.00
https://doi.org/10.1145/3293611.3331595

restricted class of optimizations from one protocol to another (Section 4)
- We port Mencius and Quorum Lease from Paxos to Raft and provide experimental evaluation of the optimized versions of Raft protocols (Section 6)

Limitations. This paper represents a first attempt at automatically porting optimizations from one family of protocols to another related one. Our method has several limitations that limit its applicability. First, in order to port optimizations, we have to provide a refinement mapping between the two related protocols. As such, we can only automatically port optimizations from Paxos to Raft* instead of directly to Raft. Second, only a certain restricted class of optimizations can be automatically ported. Specifically, such optimizations must not mutate protocol state (Section 4.2). Last, optimization porting is done at the level of protocol specification, not implementation. Programmers still need to implement a derived protocol manually.

2 OVERVIEW

2.1 Background

Paxos [18]. Paxos solves the consensus problem using a bottom-up approach. First, single-decree Paxos is developed to let servers agree on a single value. Then, MultiPaxos builds upon single-decree Paxos to agree on a sequence of operations.

In single-decree Paxos, servers reach consensus via a two-phase protocol. In phase-1, a server picks a globally unique proposal number (called ballot) and sends a Prepare RPC to every server. A server receiving a Prepare replies success if it has not seen a higher ballot. In its reply, the server includes the highest ballot it has ever accepted, or *null* if it has never accepted any ballot. If the proposing server receives at least a majority of successful replies, it goes into the second phase, otherwise, it retries with a higher ballot.

In phase-2, the server picks the value for its ballot and sends it in an Accept RPC to every server. This can be any value (usually the operation the server wants to initiate) if the phase-1 replies do not contain any previously accepted value; Otherwise, it must be the value (an operation initiated by some other server) with the highest ballot in the replies. A server receiving the Accept RPC will accept this value and reply success if it has not seen a higher ballot. If the proposing server receives a majority of successful replies, it considers this value as chosen (or equivalently, *committed*). It could notify other servers of the committed operation immediately or piggyback this information with subsequent communication.

MultiPaxos builds upon single-decree Paxos to agree on a sequence of operations. In particular, MultiPaxos tries to agree on the operation for each position in the sequence using a separate instance of Paxos. MultiPaxos also optimizes performance by allowing concurrent instances and batching the phase-1 messages of these instances. Figure 1 gives the pseudocode of MultiPaxos. In the rest of the paper, Paxos refers to the multi-decree version of Paxos and single-decree Paxos refers to the single-decree version.

Raft [31]. Unlike Paxos' bottom-up approach, Raft solves the consensus problem in a top-down manner, by replicating a log of operations across servers without decomposing into single-decree consensus. The Raft protocol consists of two parts: electing a leader and replicating of log entries by the elected leader. Each server maintains a term number that monotonically increases. For leader election, a candidate server increments its term number and sends RequestVote RPCs to all servers to collect votes for itself to become a leader. Elections are ordered by their corresponding *term* numbers and a node rejects RequestVote if it has already processed a request with a higher term or the same term from a different candidate. Raft also adds another restriction to leader election: a node rejects RequestVote if its log is more recent than the sender's log. A candidate becomes the elected leader for this term if it receives a majority quorum of successful votes on its RequestVote RPCs.

The elected leader batches client operations and replicates them to all other servers (called followers) using the AppendEntries RPC. A server rejects the AppendEntries request if it has seen a higher term than the sender. The AppendEntries RPC also lets the receiver synchronize its log with the sender: the receiver catches up if it is missing entries, and it erases extraneous entries not found in the sender's log. The leader considers the operations in AppendEntries committed if a majority quorum of servers successfully replies. Figure 2 shows the pseudocode of Raft (ignoring the code in blue).

2.2 Our approach

At first glance, Raft bears many similarities to Paxos. Both protocols have two phases. Raft's RequestVote corresponds to Phase1a in Paxos. Both RequestVote and Phase1a are considered successful if a majority of ok replies are received. Afterward, Raft uses AppendEntries to replicate operations to other servers, similar to how Paxos uses Phase2a to disseminate an operation associated with a specific single-decree instance. In both protocols, a server rejects the AppendEntries/Accept request if it has seen a higher term/ballot and the operation is considered committed only when a majority of servers return ok.

Our work uses refinement mapping [1] to formally capture the connection between Paxos and Raft. Refinement mapping is commonly used to prove that a lower-level specification correctly implements a higher-level one. For example, existing work [36] has mapped Paxos, Viewstamp Replication and ZAB to a high-level abstract consensus algorithm instead of directly relating them to each other. By contrast, to shed the light on the relationship between Paxos and Raft, we attempt to provide a refinement mapping between the two protocols directly.

It turns out that there are subtle differences that make it impossible for Raft to have a direct refinement mapping to Paxos. Nevertheless, we can modify Raft slightly to remove these superficial differences and create a close variant protocol (called Raft*). By showing the refinement mapping from Raft* to Paxos, we illuminate the connection between Raft and Paxos.

Beyond providing a deeper understanding of protocols, can the refinement mapping between Raft* and Paxos be used for some practical purposes? Guided by this question, we proceed to explore how to migrate optimizations done for Paxos to work for Raft. Based on the refinement mapping, we develop a method to port certain Paxos optimizations to Raft* such that the resulting protocol's correctness is guaranteed. Specifically, some Paxos optimization only reads but does not mutate Paxos' state variables. The intuition is to port these optimizations by replacing Paxos variables with Raft* variables according to the refinement mapping. For example,

```
1  function Phase1a(s):
2    s.ballot = s.ballot + 1
3    unchosen = smallest unchosen instance id
4    s sends <'prepare', s.ballot, unchosen> to all
5
6  function Phase1b(s):
7    if s receives <'prepare', b, unchosen>
8      && b > s.ballot
9    then
10     s.ballot = b
11     s.phase1Succeeded = false
12     reply <'prepareOK', s.ballot,
13                     instances with id ≥ unchosen>
14
15 function Phase1Succeed(s):
16   if s receives <'prepareOK', b, instances>
17       from f + 1 acceptors with the same b
18     && b == s.ballot
19   then
20     start = smallest unchosen instance id
21     end = largest id of received instances
22     for i in start ... end
23       s.instances[i]= safeEntry(received
24                               instances with id i)
25     s.phase1Succeeded = true
```

(a) Phase 1

```
26 function Phase2a(s, i, v):
27   if s.phase1Succeeded
28     && (s.instances[i].val == v
29           || s.instances[i] == Empty)
30   then
31     send <'accept', i, v, s.ballot> to all
32
33 function Phase2b(s):
34   if s receives <'accept', i, v, b> && b ≥ s.ballot
35   then
36     if b > s.ballot
37     then
38       s.phase1Succeeded = false
39       s.ballot = b
40     s.instances[i].bal = b
41     s.instances[i].val = v
42     reply <'acceptOK', i, v, b>
43
44 function Learn(s):
45     if s receives same <'acceptOK', i, v, b>
46       from f + 1 acceptors
47     then
48       s.instances[i].val = v
49       s.instances[i].bal = b
50       add s.instances[i] to s.chosenSet
```

(b) Phase 2

Figure 1: MultiPaxos

considering an add-on checkpointing mechanism for Paxos which saves both system state and last applied instance id. According to the refinement mapping, the instance id is mapped to the log index. Thus, when porting the checkpoint mechanism to Raft*, we can replace the last applied instance id with the last applied index in the log with guaranteed correctness.

3 CONNECTING RAFT TO PAXOS

Refinement mapping. We use refinement mapping to describe the equivalence between two protocols. Intuitively, in order to show that a protocol $\mathcal{B}$ refines a protocol $\mathcal{A}$, we must show that how each state of in $\mathcal{B}$'s state space can be mapped to some state in $\mathcal{A}$'s state space such that any state transition sequence in $\mathcal{B}$ corresponds to a valid state transition sequence in $\mathcal{A}$ under the mapping. This intuitive definition of refinement mapping suffices for this section. In later sections (Section 4), we present a more formal definition of refinement mapping used for optimization porting.

Why Raft cannot be mapped to Paxos directly. Ideally, we should be able to directly refine Raft to Paxos. Unfortunately, this cannot be done for two reasons. First, Raft forces all servers that accept the leader's AppendEntries to match the leader's log. Therefore, if a follower's log is longer than that of the leader, the follower will erase the extra entries. When mapped to MultiPaxos, such an "erasing" step would correspond to a server deleting a previously accepted value for some Paxos instance, a state transition that would never happen in Paxos. Raft makes the erasing step safe by committing log indexes in order so that its phase-1 exchange ensures that the leader's log contains all potentially committed entries. By contrast, Paxos commits instances out of order. Thus Paxos' proposing server must fetch safe values for different uncommitted instances from other servers and never erase (but only overwrite) accepted values at other servers. Second, the term number in each log entry in Raft cannot be mapped to the accepted ballot number of each instance in Paxos. The reason is Raft's leader never modifies its existing log entries. As a result, a newly elected leader at term t would replicate a previously uncommitted log entry with term $t' < t$ without any change. Such a behavior has no equivalence in Paxos. The proposing server in Paxos always over-writes the accepted instance's ballot number with its current ballot number. Not changing the log entry's term number turns out to have subtle correctness implications such that the Raft paper ([31] Section 5.4.2) has to add an extra rule to prevent the loss of committed values.

Raft*. We modify Raft slightly to create a close variant, called Raft*, for which a refinement mapping to Paxos exists. Figure 2 shows the specification of Raft* in pseudocode (including code in blue). Raft* is identical to Raft, except for two introduced differences, based on the two reasons for why Raft cannot be shown to refine Paxos. First, when responding ok to a candidate's RequestVote, a server includes all the extra entries not present in the candidate's log in its reply (line 68-69). The leader chooses the safe values among its majority quorum of replies to extend its log (line 77-81). An acceptor rejects leader's append request if its log is longer than the leader's (line 110). Second, a ballot field is added to each entry. On appending a new entry, Raft* will change all entries' ballot to be the new entry's term (line 99-100).

Refining Paxos with Raft*. Figure 3 gives the refinement mapping between Raft* and Paxos. Most of them are straightforward. For example, the Raft's entries whose indexes are not greater than commitIndex (committed entries) are mapped to Paxos's instances in the chosenSet (chosen instances). We only explain the nuances mappings. First, most messages in Raft* do not have the same content with its counterparts in Paxos. For instance, requestVote includes lastIndex and lastTerm instead of the smallest id of unchosen instance. This is because, if two logs have the entries with the same

```
52  // s starts an election for the new leader
53  function RequestVote(s):
54    s.currentTerm = s.currentTerm + 1
55    s sends <'requestVote', s.currentTerm, s.lastIndex,
56          s.log[s.lastIndex].term> to all
57
58  // s receives leader election request
59  function ReceiveVote(s):
60    if s receives <'requestVote', term, lastIndex, lastTerm>
61      && term > s.currentTerm
62      && s.log[lastIndex].term < lastTerm
63         || (s.log[lastIndex].term = lastTerm
64            && s.lastIndex ≥ lastIndex)
65    then
66      s.currentTerm = term
67      s.isLeader = false
68      extraEnts = non-empty entries in s.log after lastIndex
69      reply <'requestVoteOK', s.currentTerm, extraEnts>
70
71  // s becomes the new leader in the vote phase
72  function BecomeLeader(s):
73    if s receives <'requestVoteOK', term, ents> from f + 1
74      acceptors with same term && term == s.currentTerm
75    then
76      max = largest index of received entries
77      for i in s.lastIndex + 1 ... max
78        e = safeEntry(received entries of index i)
79        s.log[i].bal = s.currentTerm
80        s.log[i].term = s.currentTerm
81        s.log[i].val = e.val
82      s.isLeader = true
83      s.lastIndex = max
84
85
86
87
88
89
90
```

(a) Phase 1

```
91  // s appends vals to the log from i and
92  // broadcasts all entries after prev to replicas
93  function AppendEntries(s, i, vals, prev):
94    if s.isLeader && i == s.lastIndex + 1 then
95      for each v in vals
96        s.log[s.lastIndex+1].val = v
97        s.log[s.lastIndex+1].term = s.currentTerm
98        s.lastIndex = s.lastIndex + 1
99      for i in prev + 1 ... lastIndex
100       s.log[i].bal = s.currentTerm
101     ents = s.log entries after prev
102     pTerm = s.log[prev].term
103     send <'append', s.currentTerm, prev, pTerm, ents,
          s.commitIndex> to all
104
105 // s receives append request from the leader
106 function ReceiveAppend(s):
107   if s receives <'append', term, prev, pTerm, ents, commit>
108     && term ≥ s.currentTerm
109     && s.log[prev].term == pTerm
110     && s.lastIndex ≤ prev + length(ents)
111   then
112     if term > s.currentTerm then
113       s.isLeader = false
114       s.currentTerm = term
115     s.lastIndex = prev + length(ents)
116     s.commitIndex = max(commit, s.commitIndex)
117     replace entries after s.log[prev] with ents
118     reply <'appendOK', s.currentTerm, s.lastIndex>
119
120 // the leader s learns the committed entries
121 function LeaderLearn(s):
122   if s receives <'appendOK', term, index> from f acceptors with
        the same term
123     && s.isLeader
124     && term == s.currentTerm
125   then
126     minIndex = minimal received index
127     s.commitIndex = max(s.commitIndex, minIndex)
```

(b) Phase 2

Figure 2: Raft*. The code in blue is introduced by Raft*

term at a given index, then these two logs are identical before the given index (log matching property). Furthermore, Raft is able to use the last entry's term (lastTerm) to detect if every entry in a log is more up-to-date. Second, Raft*'s leader directly appends entries into the log in AppendEntries and BecomeLeader. This can be considered as implicitly sending an append to itself, then receiving an appendOk from itself. Both explicit and implicit append/appendOk can imply accept/acceptOk. Last, a Raft*'s function may imply multiple functions in Paxos. For example, AppendEntries implies both Phase2a and Phase2b: the leader first implicitly accepts the entry, then sends it to others. The formal specification and the proof of the refinement mapping can be found in [37].

4 A METHOD FOR PORTING OPTIMIZATION

In this section, we show how to automatically port certain Paxos optimizations to Raft* by leveraging their refinement mapping. First, we define the problem of optimization porting formally (Section 4.1). We then characterize precisely which class of optimization is amicable to our method (Section 4.2). Lastly, we describe the algorithm to port the optimization such that the derived protocol specification is guaranteed to be correct (Section 4.3).

4.1 Problem definition

Specifying a protocol. To automatically port some optimization across protocols, we must describe a protocol and its optimization in a formal way. We specify a protocol as a state machine, which can be defined by its initial state and a set of allowed state transitions. We use the TLA$^+$ language [19] for specifying state machines.

In TLA$^+$, one specifies a protocol by describing the set of allowed state transitions called the *Next* action, represented as a collection of subactions $a_1 \vee a_2 \vee \ldots$ where $\vee$ is the or (disjunction) operator. Each subaction a_i is a formula in conjunctive form with one or more clauses; the clauses specify the state transition's enabling conditions and the next state value. As an example, we consider a key-value store supporting two operations Put(k, v) and Get(k). Figure 4a shows its TLA$^+$ specification. The key-value store's internal state is a hash table (table) where each entry corresponds to a set. Its next-state action (Next) consists of two subactions Put(k,v) and Get(k), for each potential key and value. In Figure 4a, the subaction Put(k, v) is defined to equal ($\triangleq$) to the boolean formula asserting that the hash table entry for key k must contain value v in the next state (line 2). In TLA$^+$, attaching the ' symbol with a variable represents its value in the new state. Hence the formula table'[k]={v} is true only when the hash table entry for key k in the next state equals to

		Raft*	MultiPaxos
variables	Constant	Quorums	Quorums
	per server	currentTerm isLeader entries with index ≤ commitIndex	ballot phase1succeeded chosenSet
	per instance	entry.index entry.val entry.bal	instance.id instance.val instance.bal
	messages	requestVote requestVoteOK (im/ex) append (im/ex) appendOK	prepare prepareOK accept acceptOK
functions		RequestVote	Phase1a
		ReceiveVote	Phase1b
		BecomeLeader	Phase1Succeed Phase2a Phase2b
		AppendEntries	Phase2a Phase2b
		ReceiveAppend	Phase2b
		LeaderLearn	Learn

Figure 3: Mapping between Raft* and MultiPaxos. "im" stands for implicit. "ex" stands for explicit

v. We use the output variable to represent value returned to users, thus `Get(k)` uses the clause `output' = table[k]` to assert the value of the output variable in the new state. In this example of $\mathcal{A}$, there is no subaction involving more than one clause in the conjunctive form because there is no enabling conditions for either Put or Get. We will see more sophisticated subactions in subsequent examples.

Refinement mapping with TLA$^+$. Using TLA$^+$ specifications, we can formally describe the refinement mapping. We use $\mathcal{B} \Rightarrow \mathcal{A}$ to refer that $\mathcal{B}$ has a refinement mapping to $\mathcal{A}$. (e.g., Raft* $\Rightarrow$ Paxos). Let $Var_{\mathcal{A}}$ (or $Var_{\mathcal{B}}$) represent the state variables of $\mathcal{A}$ (or $\mathcal{B}$). To prove $\mathcal{B} \Rightarrow \mathcal{A}$, we need to find some function f that maps $\mathcal{B}$'s state space to $\mathcal{A}$'s, i.e., $Var_{\mathcal{A}} = f(Var_{\mathcal{B}})$. Suppose a_i is some subaction in $\mathcal{A}$, we use the term $\overline{a_i}$ to refer to the conjunctive formula when we substitute $Var_{\mathcal{A}}$ in a_i with $Var_{\mathcal{A}} = f(Var_{\mathcal{B}})$. If $\mathcal{B}$ refines $\mathcal{A}$ under f, then every subaction b_i in $\mathcal{B}$ implies some subaction a_j in $\mathcal{A}$ or a no-op step[1], i.e. $b_i \Rightarrow \overline{a_j} \vee f(Var'_{\mathcal{B}}) = f(Var_{\mathcal{B}})$, where $\Rightarrow$ is the boolean operator for implication.

Figure 4b shows an example protocol $\mathcal{B}$ which stores data in a log. Subaction Write(i, v) stores a value at the end of the log at index i. The conjunctive clause at line 2 ensures the invariant that values are stored in the log continuously. Subaction Read(i) reads the log entry i. Protocol $\mathcal{B}$ in Figure 4b refines protocol $\mathcal{A}$ in Figure 4a under the state mapping that maps the i-th entry of the log to the hash table entry with key $k = i$. The subaction Write in $\mathcal{B}$ implies Put in $\mathcal{A}$, and Read implies Get. The formal proof of the refinement between $\mathcal{A}$ and $\mathcal{B}$ can be found in [37].

Defining the problem of porting optimization. Informally, given two protocols $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{B} \Rightarrow \mathcal{A}$, we would like to adapt some existing optimization on protocol $\mathcal{A}$ to also work for protocol $\mathcal{B}$. More importantly, we require the adaption to follow an algorithmic (instead of manual) procedure that can guarantee the correctness of the resulting protocol.

To state the task formally, we are given some protocol $\mathcal{A}$, its optimized version $\mathcal{A}^\Delta$, and another protocol $\mathcal{B}$ which refines $\mathcal{A}$, all specified in TLA$^+$. Furthermore, we assume that all three protocols $\mathcal{A}$, $\mathcal{A}^\Delta$, $\mathcal{B}$ have been proven correct. The problem of porting an optimization is to automatically derive the TLA$^+$ specification of protocol $\mathcal{B}^\Delta$ such that $\mathcal{B}^\Delta$ improves the performance of $\mathcal{B}$ and is guaranteed to be correct. We note that the goal is to derive the TLA$^+$ specification of an optimized protocol, not to automatically generate the implementation of the optimized protocol.

The optimization porting problem as stated above is quite general. In particular, we do not make any assumptions on the types of correctness proofs given for the protocols, nor on any formal relationships between $\mathcal{A}$ and $\mathcal{A}^\Delta$. However, in order to make the problem trackable, we devise a solution that only applies to a restricted class of optimization, which is described next.

4.2 Non-mutating optimization

Given protocol $\mathcal{A}$ and its optimized variant $\mathcal{A}^\Delta$, we consider the optimization to be the difference between the specification of $\mathcal{A}^\Delta$ and $\mathcal{A}$. In particular, the state variables of $\mathcal{A}^\Delta$ include all state variables of $\mathcal{A}$ and may contain additional variables introduced by the optimization. Each subaction of $\mathcal{A}^\Delta$ can be of three forms:

- *An added subaction.* This is a subaction that has no relationships to existing subactions in $\mathcal{A}$.
- *An unchanged subaction.* This is a subaction that is identical to an existing subaction in $\mathcal{A}$.
- *A modified subaction.* This is a subaction derived from an existing subaction in $\mathcal{A}$ by adding extra conjunctive clauses.[2]

Our proposed method for porting an optimization works for a restricted class of optimizations, which we refer to as *non-mutating optimization*. For an optimization $\mathcal{A}^\Delta$ to be considered as non-mutating, we require that none of its added subactions and none of the added clauses in its modified subactions mutate the state variables of $\mathcal{A}$ ($Var_{\mathcal{A}}$). The subactions are free to mutate the new state variables (Var_Δ) added by $\mathcal{A}^\Delta$.

Figure 4c shows protocol $\mathcal{A}^\Delta$, as an example of non-mutating optimization on the key-value store protocol $\mathcal{A}$. The optimized protocol $\mathcal{A}^\Delta$ adds a new state variable size that tracks how many values have been stored in the hash table. Comparing Figure 4c with Figure 4a, we can see that $\mathcal{A}^\Delta$ adds the new clause (line 4) to existing subaction Put and no new subactions. As the new clause does not modify $\mathcal{A}$'s state (table), $\mathcal{A}^\Delta$ is a non-mutating optimization.

Non-mutating optimization not only allows us to port optimization from $\mathcal{A}$ to $\mathcal{B}$ using the method in Section 4.3, it also has the added benefit that $\mathcal{A}^\Delta$ can be shown to refine $\mathcal{A}$ under the identical state mapping function that ignores the extra state. Therefore, non-mutating optimization always guarantees correctness. By contrast, state-mutating optimization may or may not have a refinement mapping to $\mathcal{A}$, and thus its correctness requires a separate proof.

[1] The no-op step is commonly called a stuttering step [1, 23]

```
1: VARIABLES table, output
2: Put(k, v) ≜ table'[k] = {v}
3: Get(k) ≜ output' = table[k]
4: Init ≜ ∀ k ∈ Nat: table[k] = {}
5: Next ≜ ∃ k ∈ Nat, v∈ Values: Put(k, v) ∨ Get(k)
```

(a) A key-value store ($\mathcal{A}$).

```
1: VARIABLES logs, output
2: Write(i, v) ≜ (i = 0 ∨ logs[i-1] ≠ {})
3:               ∧ logs'[i] = v
4: Read(i) ≜ output' = logs[i]
5: Init ≜ ∀ i ∈ Nat : logs[i] = {}
6: Next ≜ ∃ i ∈ Nat, v ∈ Values: Write(i, v) ∨ Read(i)
```

(b) The protocol $\mathcal{B}$ that stores data in a log. $\mathcal{B}$ refines the key-value store $\mathcal{A}$.

```
1: VARIABLES table, output, size
2: Put(k, v) ≜ table[k] = {}
3:            ∧ table'[k] = {v}
4:            ∧ size' = size + 1
5: Get(k) ≜ output' = table[k]
6: Init ≜ ∀ k ∈ Nat: table[k] = {} ∧ size = 0
7: Next ≜ ∃ k ∈ Nat, v ∈ Values: Put(k, v) ∨ Get(k)
```

(c) The optimized protocol $\mathcal{A}^\Delta$.

```
1: VARIABLES logs, output, size
2: Write(i, v) ≜ logs[i] = {}
3:              ∧ (i = 0 ∨ logs[i-1] ≠ {})
4:              ∧ logs'[i] = {v}
5:              ∧ size' = size + 1
6: Read(i) ≜ output' = logs[i]
7: Init ≜ ∀ k ∈ Nat: logs[i] = {} ∧ size = 0
8: Next ≜ ∃ i∈ Nat, v ∈ Values: Write(i, v) ∨ Read(i)
```

(d) The generated TLA$^+$ specification of $\mathcal{B}^\Delta$.

Figure 4: The TLA$^+$ specifications of the example.

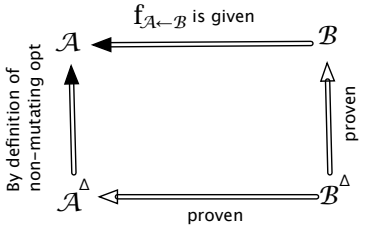


Figure 5: The refinement mappings among given protocols. $\mathcal{A}^\Delta$ is an optimized version of $\mathcal{A}$ using non-mutating optimization. $\mathcal{B}^\Delta$ is the optimized version of $\mathcal{B}$ generated by our method in Section 4.3.

4.3 How to port non-mutating optimization

We only consider the case of porting non-mutating optimizations. Additionally, if the optimization reads the parameters of protocol $\mathcal{A}$, our method also requires a *parameter mapping* from $\mathcal{B}$ to $\mathcal{A}$.

Parameter mapping. Let $P_\mathcal{A}$ and $P_\mathcal{B}$ be the parameter variables of $\mathcal{A}$ and $\mathcal{B}$, respectively. We say $\mathcal{B}$ has a parameter mapping to $\mathcal{A}$ if there exists a function f_{arg} that maps the arguments of subactions in $\mathcal{B}$ to the arguments of subactions in $\mathcal{A}$, i.e. $P_\mathcal{A} = f_{args}(P_\mathcal{B})$.[3] The extra clauses added in a modified subaction in $\mathcal{A}^\Delta$ may use parameter variables. Therefore, the parameter mapping is required in order to correctly translate those clauses to be used in a corresponding subaction in $\mathcal{B}^\Delta$.

Porting the optimization. We are now ready to describe how to transform the specification of $\mathcal{A}^\Delta$ to create $\mathcal{B}^\Delta$, thereby porting the optimization from $\mathcal{A}$ to $\mathcal{B}$. First, we obtain $\mathcal{B}^\Delta$'s state variables as $Var_{\mathcal{B}^\Delta} = Var_\mathcal{B} \cup Var_\Delta$. We also obtain $\mathcal{B}^\Delta$'s initial state ($Init_{\mathcal{B}^\Delta}$) from $Init_\mathcal{B}$ and $Init_{\mathcal{A}^\Delta}$ by replacing every state variable of $\mathcal{A}$ using the state mapping $Var_\mathcal{A} = f(Var_\mathcal{B})$. Next, we generate the subactions of $\mathcal{B}^\Delta$ from each subaction a_i^Δ of $\mathcal{A}^\Delta$ and the no-op step. There are three cases:

Case-1: a_i^Δ is an added subaction. We turn a_i^Δ into a corresponding added subaction b_i^Δ by substituting state variable Var_a with $f(Var_b)$ and keeping Var_Δ unchanged.

Case-2: a_i^Δ is an unchanged subaction which is equal to a^i in $\mathcal{A}$, or the no-op step. There is a set of subactions in $\mathcal{B}$ that imply a^i according to the refinement mapping $f_{\mathcal{B}\to\mathcal{A}}$. We directly add the set of subactions to $\mathcal{B}^\Delta$.

Case-3: a_i^Δ is a modified subaction of a^i in $\mathcal{A}$. Again, there is a set of subactions in $\mathcal{B}$ that imply a^i according to $f_{\mathcal{B}\to\mathcal{A}}$. Suppose b_j is a subaction in the set. We add b_j to $\mathcal{B}^\Delta$ if b_j is not already added (Case-2). Furthermore, we include the extra clauses added by a_i^Δ in b_j by substituting $Var_a = f(Var_b)$ and $P_a = f_{args}(P_b)$.

Correctness. We prove that the generated specification of $\mathcal{B}^\Delta$ is correct. The proof contains two parts. First, we need to show that $\mathcal{B}^\Delta$ correctly incorporates the optimization in $\mathcal{A}^\Delta$. This can be proven by demonstrating that $\mathcal{B}^\Delta$ refines $\mathcal{A}^\Delta$, thus $\mathcal{B}^\Delta$ preserves the invariants introduced by the optimization. Second, we also need to show that $\mathcal{B}^\Delta$ remains correct w.r.t. the original protocol $\mathcal{B}$. This can be proven by demonstrating that $\mathcal{B}^\Delta$ refines $\mathcal{B}$, thus $\mathcal{B}^\Delta$ preserves the invariants of the original protocol $\mathcal{B}$.

As a summary, Figure 5 illustrates the refinement mappings that exist among the four protocols, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}^\Delta$, $\mathcal{B}^\Delta$. Next, we provide proof sketches for the refinement mappings of $\mathcal{B}^\Delta$.

First, we prove that $\mathcal{B}^\Delta$ refines $\mathcal{A}^\Delta$ using the state mapping function that maps the state variables of $\mathcal{B}$ to those of $\mathcal{A}$ according to f and leaves the variables introduced by optimization Δ unchanged. To prove the correctness of this refinement mapping, we must show that $\mathcal{B}^\Delta$'s initial state implies $\text{Init}_{\mathcal{A}^\Delta}$, and $\mathcal{B}^\Delta$'s next-state action ($\text{Next}_{\mathcal{B}^\Delta}$) implies $\text{Next}_{\mathcal{A}^\Delta}$. The former implication is relatively straightforward, so we focus the discussion on the latter.

[2]If the derivation deletes an existing conjunctive clause, then the resulting subaction of $\mathcal{A}^\Delta$ must be viewed as an added subaction instead of a modified one.

[3]To put it more formally, given parameter mapping $P_\mathcal{A} = f_{args}(P_\mathcal{B})$, we use $\overline{Next_\mathcal{A}}$ to refer to the formula after substituting state variables Var_a with $f(Var_b)$ and parameter variables $P_\mathcal{A}$ with $f_{args}(\mathcal{B})$. f_{args} is a valid parameter mapping if $Next_\mathcal{B} \Rightarrow \overline{Next_\mathcal{A}}$.

To show $\text{Next}_{\mathcal{B}^\Delta}$ implies $\text{Next}_{\mathcal{A}^\Delta}$, we show that each $\mathcal{B}^\Delta$'s subaction (b_i^Δ) implies some subaction of $\mathcal{A}^\Delta$ or a no-op step. According to our method, b_i^Δ can be added to $\mathcal{B}^\Delta$ in one of three cases. For case-1 and 2, it is easy to show that b_i^Δ implies a_i^Δ or the no-op step by construction. In case-3, b_i^Δ is constructed from b_i and a subaction a_j^Δ in $\mathcal{A}^\Delta$, such that $a_j^\Delta = a_j \wedge \Delta_{a_j}$, $b_i^\Delta = b_i \wedge \overline{\Delta_{a_j}}$. Δ_{a_j} is the set of extra conjunctive clauses added by the optimization to a_j to form a_j^Δ. $\overline{\Delta_{a_j}}$ is obtained from Δ_{a_j} by substituting variables $Var_a = f_{\mathcal{B}\to\mathcal{A}}(Var_b)$ and parameters P_a by $f_{args}(P_b)$. Because of $b_i \Rightarrow \overline{a_j}$, we have $b_i^\Delta \Rightarrow \overline{a_j} \wedge \overline{\Delta_{a_j}}$ which is equivalent to $b_i^\Delta \Rightarrow \overline{a_j^\Delta}$.

Next, we prove that $\mathcal{B}^\Delta$ refines $\mathcal{B}$, using the state mapping function that simply drops the new variables added by the optimization. To prove $\mathcal{B}^\Delta$ refines $\mathcal{B}$, we argue that $\mathcal{B}^\Delta$ is a non-mutating optimization by analyzing the three cases of our method. We leave the details to [37].

5 PORTING PAXOS OPTIMIZATION TO RAFT*.

The landscape of Paxos variants and optimization. We first study existing Paxos variants and optimizations using the lens of our method. Figure 6 shows the relationship between these protocols. Among them, six protocols belong to the class of non-mutating optimization for Paxos, shown in the double-lined box in Figure 6. Thus, these six optimizations can potentially be ported from Paxos to Raft* using our method. Flexible Paxos [11] relaxes the majority quorum restriction in Paxos to allow differently sized quorums as long as the quorums used in the two phases of Paxos exchange are guaranteed to intersect. As a result, Paxos refines Flexible Paxos but not the other way around. WPaxos [3] is a recently proposed non-mutating optimization on Flexible Paxos. Therefore, our method could also be used to port the optimization of WPaxos to Paxos.

As for the rest of the protocol variants (shown in the left-most box in Figure 6), their relationships to Paxos cannot be captured by refinement mapping. The reasons for the lack of refinement mapping are varied. For example, Fast Paxos changes the quorum size of Paxos to include a super-majority, which prevents a refinement mapping from Fast Paxos to Paxos. However, it also misses state transitions allowed in Paxos, which precludes a refinement mapping from Paxos to Fast Paxos.

Among the six candidate protocols, we choose to port two optimizations as case studies: Paxos Quorum Lease [28] and Mencius [25]. We explain what these optimizations are and how they are ported to Raft*. Our discussion uses pseudocode instead of TLA$^+$ for simplicity. A more formal description (including the refinement mapping, pseudocode, and TLA$^+$) can be found in the extended version of the paper [37].

Paxos Quorum Lease. In Paxos, a strongly consistent read operation is performed by persisting the operation into the log as if it were a write. Paxos Quorum Lease (PQL) [28] introduces an optimization that allows any replica to read locally if the replica holds leases from a quorum of replicas (*quorum-lease*).

Quorum-lease can co-exist perfectly with the quorum in Paxos. Any replica can grant a lease. A replica considers itself holding a quorum-lease if it holds leases from a quorum of replicas. Any lease-quorum must overlap with any Paxos quorum (usually both quorums are majorities of replicas). In Paxos any commit needs to collect from a quorum of acknowledgments, which will intersect with the lease quorum. Therefore, as long as we require every replica in a Paxos quorum to notify their granted lease holders before the replica commits any values, the system is safe—both read and write are consistent.

PQL is a non-mutating variant of Paxos, because all its added and modified subactions do not change the state variables in Paxos. Figure 7 shows the algorithmic changes introduced by PQL. The actions changed are Phase2b and Learn. In Phase2b, a server attaches all leases it granted with the "acceptOk" response (line 143). By collecting the granted leases from a quorum (line 146), a learner can find all servers who hold active leases (line 148), and commits an instance only if it receives "acceptOk" from these holders (line 150). The added actions are Read and LocalRead. When a client issues a "localRead" request, the server will return its local copy if it holds valid leases granted by a quorum, and all committed updates are in the chosen set (line 136-137).

Raft*-PQL. Figure 8 shows the algorithm after applying PQL to Raft*. The code in blue shows the changed part after porting the optimization to Raft*. For a replica to perform a local read, the replica needs to check if two conditions hold. First, the replica must be holding leases from at least $f + 1$ replicas (including itself). Second, the replica needs to wait until commitIndex is greater than the largest index of entries which modify the target record (line 156). This is transformed from PQL where all modifications must be in the chosenSet (line 137).

A replica attaches the lease holders granted by itself in appendOk message, which maps to the acceptOk message. In LeaderLearn, the leader needs to collect the holders of leases attached in the messages and granted by itself (line 165). This is because the f appendOk messages with one extra implicit appendOk message imply $f + 1$ acceptOk messages in Paxos. Thus, collecting leases attached in $f + 1$ messages (line 148) is transformed into collecting the leases from f messages and local granted (the implicit message).

Mencius. Paxos requires all clients requests to be sent to a leader for better throughput. This could lead to unbalanced load between the leader replica and other replicas. When replicas are located in different data centers, non-leader replicas will need at least two wide-area round-trips to commit any requests because requests need to be forwarded to the leader. To address these issues, Mencius [25] partitions the Paxos instances so that each replica serves as the default leader for a distinct subset of instances. With geo-replicas, a client can send its requests to the nearest replica. The replica can commit these requests using those Paxos instances for which it is the default leader. Thus, Mencius can balance the load among all replicas and also reduces wide-area round-trips.

Mencius partitions the instance (log) space in a round-robin way. For example, in a system with three replicas r_1, r_2, r_3, r_1 is the default leader for log entries $(0, 3, 6, \ldots)$, r_2 is the leader for $(1, 4, 7, \ldots)$, and r_3 for $(2, 5, 8, \ldots)$. Mencius separates the execution of a log entry from its commit. The log is executed sequentially. If some default leader has not received any user-submitted operations, it commits skip entries to keep log execution progressing forward. To prevent a crashed replica from delaying the system, the instances belong to that replica can be committed as no-op by other replicas.

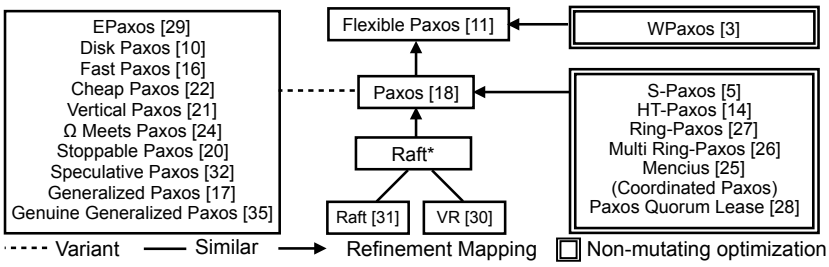


Figure 6: The relationship of different Paxos variants and optimization.

```
128  function Read(c, k):
129    c sends <'localRead', k> to 1 server s
130    if c receives <'ReadReply', v> from s
131    then
132      return v
133
134  function LocalRead(s):
135    if s receives <'localRead', k>
136      && s.validLeasesNum ≥ f + 1
137      && all instances modified k are in chosenSet
138    then
139      s replies <'ReadReply', LocalCopy(k)>
140
141  function Phase2b(s):
142      ...
143      s replies <'acceptOk',..., leases granted by s>
144
145  function Learn(s):
146    if s receives <'acceptOk', i, v, b, s, leases> from f + 1
         acceptors
147    then
148      holderSet = holders of received leases
149
150    if s receives <'acceptOk', ... > from all holders in holderSet
151    then
152      ...
```

Figure 7: Paxos Quorum Lease

```
153  function LocalRead(s):
154    if s receives <'localRead', k>
155      && s.validLeasesNum ≥ f + 1
156      && indexes of entries in s.log modified k ≤ s.commitIndex
157    then
158      ...
159
160  function LeaderLearn(s):
161    if s receives <'appendOK', t, index, holders> from f acceptors
162      && s.isLeader
163      && s.currentTerm == t
164    then
165      holderSet = received holders ∪ holders of leases granted by
           the leader
166    if s receives <'appendOK', ... > from all holders in holderSet
167    then
168      ...
```

Figure 8: Raft*-PQL

These optimizations help Mencius commit and execute requests within 1.5 round-trips on average.

Raft*-Mencius. The complete pseudocode of Raft*-Mencius is included in [37]. We describe some interesting details here. In addition to the Paxos state variables, each replica needs to keep an array of "skip-tags" to track those log entries that can be skipped. When a replica becomes the leader, it needs to collect not only values but also skip-tags from other replicas. Because Phase2b action in Paxos corresponds to many actions (AppendEntries, ReceiveAppend) in Raft*, whatever changes Mencius makes to Phase2b should be applied to these actions as well. As an example, if the newly appended entries are skip entries, they should be marked as executable.

Because Phase2b in Paxos is implied by multiple sub-actions in Raft*, it is possible that manual efforts for porting Mencius may miss some of the actions. For example, if the manual solution only applies changes on Phase2b in Paxos to ReceiveAppend in Raft* and miss AppendEntries, the solution could miss some optimization opportunities or even result in an incorrect protocol.

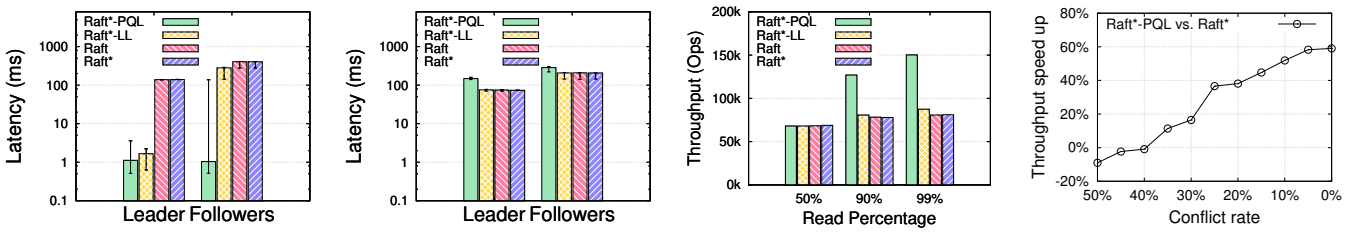
6 EVALUATION

This section shows that the generated algorithms achieve similar optimization effects with their Paxos counterparts [28, 25]. In particular, Raft*-PQL reduces the latency of read requests by performing local read. Raft*-Mencius improves the throughput by balancing the workload across all replicas. Our porting method guarantees the correctness of these derived algorithms. However, the extra effort is still needed for the implementation.

Testbed. The experiments were conducted on Amazon EC2 across 5 different geographical regions: Oregon, Ohio, Ireland, Canada and Seoul. In each region, two m4.xlarge instances are used for the client and server processes respectively. Each instance has 4 virtual CPUs, 16GB memory and one SSD with 750 Mbps bandwidth. The latency across sites varies from 25ms to 292ms.

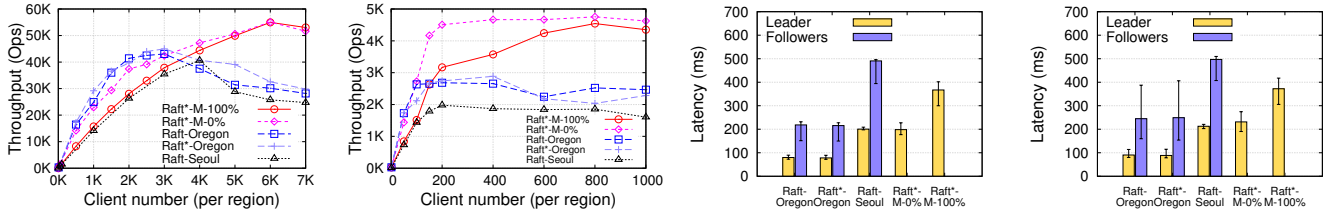
Workload. Our evaluation uses closed-loop clients with a YCSB alike workload: each client issues get or put requests back-to-back. The system is initialized with 100K records. To simulate contention, each client accesses the same popular record at a configured rate (i.e., conflict rate). When not accessing the popular record, the key space is pre-partitioned among the datacenters evenly, and a key is selected from this partition with uniform probability. Raft*-PQL is evaluated with 90% read by default. For Raft*-Mencius, we use a workload with 100% writes. Each trial is run for 50 seconds with 10 seconds for both warm-up and cool-down. Each number reported is the median in 5 trials.

Implementation. The implementation of Raft* is based on a popular industrial Raft codebase—etcd (version c4fc8c09). Etcd has a few important optimizations. First, it has a customized network layer for efficient communication. Second, when a follower receives multiple requests from clients, it forwards them to the leader in a



(a) Read latency (b) Write latency (c) Peak throughput (d) Speedup of Raft*-PQL over Raft

Figure 9: Raft*-PQL vs. LL vs. Raft. Each bar in (a) and (b) represents the 90th percentile latency of the requests with an error bar from the 50th to 99th percentiles. The y-axis is in log scale for (a) and (b).



(a) Throughput - 8B (b) Throughput - 4KB (c) Latency - 8B (d) Latency - 4KB

Figure 10: Raft*-Mencius vs. Raft. Raft*-M-100% and Raft*-M-0% stand for Raft*-Mencius with workload under 100% and 0% conflict rate. Raft-Oregon and Raft-Seoul stand for the leader is in Oregon or Seoul. 8B and 4KB are the request size of the workload.

batch. Such techniques improve system throughput when follower servers accept client requests. In our tests, etcd is 2.4× better in throughput when these optimizations are turned on. We keep these optimizations on in our tests to give etcd extra advantages. Oregon is used as the leader site for etcd which gives it the best result since Oregon has the best network condition.

6.1 Raft*-PQL

We evaluate Raft*-PQL with the same lease parameters in [28]: the lease duration is 2 seconds, is renewed for every 0.5 seconds. We also compare Raft*-PQL with Leader Lease (LL). Here the leader has sole ownership of the lease, so only the leader can process a read request with its local copy. We use 90% read workload with 5% conflict rate by default.

Latency. First, we compare the latency with 50 clients per region. In Raft*-PQL, any server with an active lease is able to conduct local consistent reads, thus 90% of the read requests have only 1ms latency (Figure 9a). In comparison, for LL, only the leader can process read request with similar latency (1.6 ms). Raft*-PQL has 1% read requests have high latency (~137ms) on followers. This is because 5% contention in the workload: followers need to wait for conflicting write to commit before processing the read request. For write latency (Figure 9b) Raft*-PQL is a little bit higher than others, as it needs to wait for leaseholders' acknowledge to commit a write operation, while others can always choose the fastest majority.

Throughput. Figure 9c shows how is the peak throughput affected by reading percentage (50%, 90%, and 99%). Raft, Raft* and LL achieve almost the same peak throughput, as the leader's CPU is the bottleneck, and the saturated leader CPU has the same capability to handle requests for these algorithms. In contrast, Raft*-PQL achieves 1.6× and 1.9× speedup with 90% and 99% reads. The advantage is from conducting the read requests locally. Figure 9d shows how is the throughput speedup affected by the conflict rate. The figure does not show the speedup of Raft*-PQL over Raft*, as they are similar. The speedup increases with the decreasing of conflict rate since all followers can return read requests to the user immediately under a low conflict rate.

6.2 Raft*-Mencius

We use a 100% put workload to measure Raft*-Mencius with 0% and 100% contention, marked as Raft*-Mencius-0% and Raft*-Mencius-100% respectively. To make a fair comparison, we evaluate both the best and worst case scenarios for Raft in the wide area by placing the leader in the nearest (Oregon) and farthest (Seoul) servers to all other regions (Raft-Oregon and Raft-Seoul). We only evaluate Raft* with the leader at Oregon for reference.

Throughput. Figure 10a gives throughput when the system is bounded by the CPU. Raft*-Mencius can achieve a peak throughput of 55K operations per second (ops) since it balances the load among all replicas. In contrast, other systems can reach the peak throughput of 41K ops after their leaders' CPUs are saturated. Figure 10b gives throughput when the system is network bounded. Raft reaches the peak throughput after saturating leader's network bandwidth. Raft-Oregon has 30% higher throughput than Raft-Seoul

as Oregon has higher bandwidth. Raft*-Mencius has 70% higher throughput than Raft-Oregon because it is able to utilize all replicas' network bandwidth. In both figures, with a small number of clients, Raft-Oregon and Raft*-Mencius-0% have better performance than others due to their lower latency.

Latency. Figure 10c and Figure 10d show the latency with 50 clients per region. The leader of Raft-Oregon processes requests with the lowest latency (79ms), as the quorum of Oregon, Ohio, and Canada are closest to each other. In comparison, Raft*-Mencius-100% has much higher 90% percentile latency, while Raft*-Mencius-0% has lower latency because of the different contention levels.

7 RELATED WORK

Elementary consensus protocols. In addition to Raft [31] and Paxos [18], there are many alternative protocols. For example, Viewstamped Replication (VR) [30] was published earlier than Paxos, and ZooKeeper [12] uses ZAB [13]. Our method is also suitable for these protocols. In particular, we can connect these protocols with Paxos by crafting a Raft* similar protocol.

Algorithm comparison. Renesse et al. [36] compared Paxos to VR and ZAB using *refinement mapping*. Lamport[15] discuss the equivalence between Byzantine Paxos and PBFT [7] is discussed. Song et al. [34] identified common traits in the Paxos, Chandra-Toueg [8], and Ben-Or [4] consensus algorithms. Abraham and Malkhi [2] discussed the connections between BFT consensus protocols and block-chain protocols. Compared to these works, we have two notable differences: we use a formal method TLA$^+$ [19] to model the refinement mappings [1]; we have mechanically exported the optimizations from one family of protocols to another.

Paxos variants and optimizations. Figure 6 has shown a number of Paxos variants. Among the non-mutating variants, WPaxos [3] partitions object and use flexible quorums for geo-replication [11]; HT-Paxos [14] and S-Paxos [5] assigns ordering tasks to multiple servers to remove bottlenecks. Ring Paxos [27] and Multi Ring-Paxos [26] partition the workload and achieve better performance. Among the mutating Paxos variants: Cheap Paxos [22] introduces auxiliary servers. Ω meets Paxos [24] elects a stable leader in a weak network environment. NetPaxos [9] adapts Paxos to SDN. Stoppable Paxos [20] is able to perform reconfiguration without slowing down. Additionally, Shraer et al. [33] and Vertical Paxos [21] discusses how to reconfigure a replicated state machine. Disk Paxos [10] achieves consensus in a disk cluster. Fast Paxos [16] and Multi-coordinated Paxos [6] introduce a fast quorum to reach consensus with a single round-trip. Generalized Paxos [17], Genuine Generalized Paxos [35] and EPaxos [29] resolve conflicts because execution. Speculative Paxos [32] introduces speculative execution when messages are delivered in order.

ACKNOWLEDGMENTS

We sincerely thank our anonymous reviewers for their insightful comments. We thank Lamont Nelson for his help and Aurojit Panda for his feedback on this work. This work is supported by the National Key Research & Development Program of China (No. 2016YFB1000104), NSF grant CNS-1409942 and CNS-1514422, and AFOSR FA9550-15-1-0302.

REFERENCES

[1] M. Abadi, and L. Lamport The existence of refinement mappings. *Theoretical Computer Science* 82, 2 (1991).
[2] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman Solida: A cryptocurrency based on reconfigurable byzantine consensus. In *Proc. OPODIS*. 2017.
[3] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar WPaxos: Wide Area Network Flexible Consensus. *arXiv preprint arXiv:1703.08905* (2017).
[4] M. Ben-Or Another advantage of free choice (extended abstract): completely asynchronous agreement protocols. In *Proc. PODC*. Aug. 1983.
[5] M. Biely, Z. Milosevic, N. Santos, and A. Schiper S-Paxos: offloading the leader for high throughput state machine replication. In *Proc. SRDS*. Oct. 2012.
[6] L. J. Camargos, R. M. Schmidt, and F. Pedone Multicoordinated Paxos. In *Proc. PODC*. Aug. 2007.
[7] M. Castro, and B. Liskov Practical byzantine fault tolerance. In *Proc. OSDI*. Feb. 1999.
[8] T. D. Chandra, and S. Toueg Unreliable failure detectors for reliable distributed systems. *JACM* 43, 2 (Mar. 1996).
[9] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 2015.
[10] E. Gafni, and L. Lamport Disk paxos. *DC* (2003).
[11] H. Howard, D. Malkhi, and A. Spiegelman Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696* (2016).
[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX ATC*. June 2010.
[13] F. P. Junqueira, B. C. Reed, and M. Serafini Zab: high-performance broadcast for primary-backup systems. In *Proc. DSN*. June 2011.
[14] V. Kumar, and A. Agarwal HT-Paxos: high throughput state-machine replication protocol for large clustered data centers. *The Scientific World Journal* (2015).
[15] L. Lamport Byzantizing Paxos by refinement. In *Proc. DISC*. July 2011.
[16] L. Lamport Fast Paxos. *DC* 19, 2 (Oct. 2006).
[17] L. Lamport *Generalized consensus and Paxos*. Tech. rep. MSR-TR-2005-33. Microsoft Research, 2005.
[18] L. Lamport Paxos made simple. *SIGACT* 32, 4 (2001).
[19] L. Lamport *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.
[20] L. Lamport, D. Malkhi, and L. Zhou Reconfiguring a state machine. *ACM SIGACT News* (2010).
[21] L. Lamport, D. Malkhi, and L. Zhou Vertical Paxos and primary-backup replication. In *Proc. PODC*. Aug. 2009.
[22] L. Lamport, and M. Massa Cheap Paxos. In *Proc. DSN*. June 2004.
[23] L. Lamport, and S. Merz Auxiliary variables in TLA+. *arXiv preprint arXiv:1703.05121* (2017).
[24] D. Malkhi, F. Oprea, and L. Zhou Ω meets paxos: Leader election and stability without eventual timely links. In *Proc. DISC*. 2005.
[25] Y. Mao, F. P. Junqueira, and K. Marzullo Mencius: building efficient replicated state machines for WANs. In *Proc. OSDI*. Dec. 2008.
[26] P. J. Marandi, M. Primi, and F. Pedone Multi-ring paxos. In *Proc. DSN*. 2012.
[27] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone Ring Paxos: A high-throughput atomic broadcast protocol. In *Proc. DSN*. 2010.
[28] I. Moraru, D. G. Andersen, and M. Kaminsky Paxos quorum leases: Fast reads without sacrificing writes. In *Proc. SoCC*. Nov. 2014.
[29] I. Moraru, D. G. Andersen, and M. Kaminsky There is more consensus in egalitarian parliaments. In *Proc. SOSP*. Nov. 2013.
[30] B. M. Oki, and B. H. Liskov Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. PODC*. June 1988.
[31] D. Ongaro, and J. K. Ousterhout In search of an understandable consensus algorithm. In *Proc. USENIX ATC*. June 2014.
[32] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy Designing distributed systems using approximate synchrony in data center networks. In *Proc. NSDI*. May 2015.
[33] A. Shraer, B. Reed, D. Malkhi, and F. P. Junqueira Dynamic reconfiguration of primary/backup clusters. In *Proc. USENIX ATC*. June 2012.
[34] Y. J. Song, R. van Renesse, F. B. Schneider, and D. Dolev The building blocks of consensus. In *IEEE ICDCN*. Jan. 2008.
[35] P. Sutra, and M. Shapiro Fast genuine generalized consensus. In *Proc. SRDS*. Oct. 2011.
[36] R. Van Renesse, N. Schiper, and F. B. Schneider Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing* 12, 4 (July 2015).
[37] W. Zhaoguo, Z. Changgeng, M. Shuai, C. Haibo, and L. Jinyang On the parallels between Paxos and Raft, and how to port optimizations (Extended Version). *arXiv preprint arXiv:1905.10786* (2019).