# Scaling Replicated State Machines with Compartmentalization

Michael Whittaker
UC Berkeley
mjwhittaker@berkeley.edu

Ailidani Ailijiang
Microsoft
aiailiji@microsoft.com

Aleksey Charapko
University of New Hampshire
aleksey.charapko@unh.edu

Murat Demirbas
University at Buffalo
demirbas@buffalo.edu

Neil Giridharan
UC Berkeley
giridhn@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

Heidi Howard
University of Cambridge
hh360@cst.cam.ac.uk

Ion Stoica
UC Berkeley
istoica@berkeley.edu

Adriana Szekeres
VMWare
aszekeres@vmware.com

## ABSTRACT

State machine replication protocols, like MultiPaxos and Raft, are a critical component of many distributed systems and databases. However, these protocols offer relatively low throughput due to several bottlenecked components. Numerous existing protocols fix different bottlenecks in isolation but fall short of a complete solution. When you fix one bottleneck, another arises. In this paper, we introduce compartmentalization, the first comprehensive technique to eliminate state machine replication bottlenecks. Compartmentalization involves decoupling individual bottlenecks into distinct components and scaling these components independently. Compartmentalization has two key strengths. First, compartmentalization leads to strong performance. In this paper, we demonstrate how to compartmentalize MultiPaxos to increase its throughput by 6× on a write-only workload and 16× on a mixed read-write workload. Unlike other approaches, we achieve this performance without the need for specialized hardware. Second, compartmentalization is a technique, not a protocol. Industry practitioners can apply compartmentalization to their protocols incrementally without having to adopt a completely new protocol.

**PVLDB Reference Format:**
Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling Replicated State Machines with Compartmentalization. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol14.html.

## 1 INTRODUCTION

State machine replication protocols are a crucial component of many distributed systems and databases [1–4, 11, 15, 35, 38]. In many state machine replication protocols, a single node has multiple responsibilities. For example, a Raft leader acts as a batcher, a sequencer, a broadcaster, *and* a state machine replica. These overloaded nodes are often a throughput bottleneck, which can be disastrous for systems that rely on state machine replication.

Many databases, for example, rely on state machine replication to replicate large data partitions of tens of gigabytes [2, 34]. These databases require high-throughput state machine replication to handle all the requests in a partition. However, in such systems, it is not uncommon to exceed the throughput budget of a partition. For example, Cosmos DB will split a partition if it experiences high throughput despite being under the storage limit. The split, aside from costing resources, may have additional adverse effects on applications, as Cosmos DB provides strongly consistent transactions only within the partition. Eliminating state machine replication bottlenecks can help avoid such unnecessary partition splits and improve performance, consistency, and resource utilization.

Researchers have studied how to eliminate throughput bottlenecks, often by inventing new state machine replication protocols that eliminate a *single* throughput bottleneck [5, 6, 10, 13, 18, 22, 23, 26, 27, 37, 43]. However, eliminating a *single* bottleneck is not enough to achieve the best possible throughput. When you eliminate one bottleneck, another arises. To achieve the best possible throughput, we have to eliminate *all* of the bottlenecks.

The key to eliminating these throughput bottlenecks is scaling, and thanks to the technological trends surrounding the cloud, scaling up has never been easier or cheaper. Unfortunately, it is widely believed that state machine replication protocols don't scale. After all, the key to scaling is parallelism, but the goal of a state machine replication protocol is to eliminate parallelism by imposing a serial order on a set of concurrently proposed commands.

In this paper, we show that this is not true. State machine replication protocols can scale. Specifically, we analyze the throughput bottlenecks of MultiPaxos and systematically eliminate them using a combination of decoupling and scaling, a technique we call **compartmentalization**. For example, consider the MultiPaxos leader, a notorious throughput bottleneck. The leader has two distinct responsibilities. First, it sequences state machine commands into a log. It puts the first command it receives into the first log entry, the next command into the second log entry, and so on. Second, it broadcasts the commands to the set of MultiPaxos acceptors, receives their


Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

responses, and then broadcasts the commands again to a set of state machine replicas. To compartmentalize the MultiPaxos leader, we first **decouple** these two responsibilities. There's no fundamental reason that the leader has to sequence commands *and* broadcast them. Instead, we have the leader sequence commands and introduce a new set of nodes, called proxy leaders, to broadcast the commands. Second, we **scale** up the number of proxy leaders. We note that broadcasting commands is embarrassingly parallel, so we can increase the number of proxy leaders to avoid them becoming a bottleneck. Note that this scaling wasn't possible when sequencing and broadcasting were coupled on the leader since sequencing is not scalable. Compartmentalization has two key strengths.

**(1) Strong Performance Without Strong Assumptions.** We compartmentalize MultiPaxos and increase its throughput by a factor of 6× on a write-only workload and 16× on a mixed read-write workload. Moreover, we achieve our strong performance without the strong assumptions made by other state machine replication protocols with comparable performance [19, 36, 37, 40, 43]. For example, we do not assume a perfect failure detector, we do not assume the availability of specialized hardware, we do not assume uniform data access patterns, we do not assume clock synchrony, and we do not assume key-partitioned state machines.

**(2) General and Incrementally Adoptable.** Researchers have invented *new* state machine replication protocols to eliminate throughput bottlenecks, but these new protocols are often subtle and complicated. As a result, these sophisticated protocols have been largely ignored by industry due to their high barriers to adoption. Compartmentalization, on the other hand, is not a new protocol. It's a technique that can be systematically applied to existing protocols. Industry practitioners can incrementally apply compartmentalization to their current protocols without having to throw out their battle-tested implementations for something new and untested.

In summary, we present the following contributions

- We characterize all of MultiPaxos' throughput bottlenecks and explain why, historically, it was believed that they could not be scaled.
- We introduce the concept of compartmentalization: a technique to decouple and scale throughput bottlenecks.
- We apply compartmentalization to systematically eliminate MultiPaxos' throughput bottlenecks. In doing so, we debunk the widely held belief that MultiPaxos and similar state machine replication protocols do not scale.

# 2 BACKGROUND

## 2.1 System Model

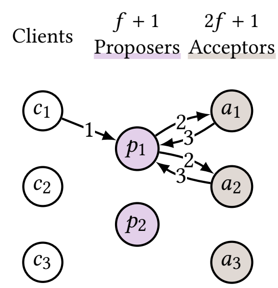
Throughout the paper, we assume an asynchronous network model in which messages can be arbitrarily dropped, delayed, and reordered. We assume machines can fail by crashing but do not act maliciously; i.e., we do not consider Byzantine failures. We assume that machines operate at arbitrary speeds, and we do not assume clock synchronization. Every protocol discussed in this paper assumes that at most $f$ machines will fail for some configurable $f$.

## 2.2 Paxos

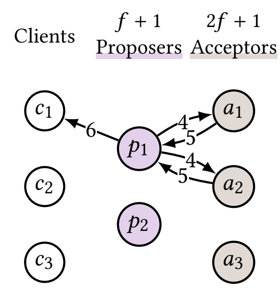
**Consensus** is the act of choosing a single value among a set of proposed values, and **Paxos** [21] is the de facto standard consensus protocol. We assume the reader is familiar with Paxos, but we pause to review the parts of the protocol that are most important to understand for the rest of this paper.

A Paxos deployment that tolerates $f$ faults consists of an arbitrary number of clients, at least $f+1$ **proposers**, and $2f+1$ **acceptors**, as illustrated in Figure 1. When a client wants to propose a value, it sends the value to a proposer $p$. The proposer then initiates a two-phase protocol. In Phase 1, the proposer contacts the acceptors and learns of any values that may have already been chosen. In Phase 2, the proposer proposes a value to the acceptors, and the acceptors vote on whether or not to choose the value. If a value receives votes from a majority of the acceptors, the value is considered chosen.

More concretely, in Phase 1, $p$ sends Phase1a messages to at least a majority of the $2f+1$ acceptors. When an acceptor receives a Phase1a message, it replies with a Phase1b message. When the leader receives Phase1b messages from a majority of the acceptors, it begins Phase 2. In Phase 2, the proposer sends Phase2a$\langle x \rangle$ messages to the acceptors with some value $x$. Upon receiving a Phase2a$\langle x \rangle$ message, an acceptor can either ignore the message, or vote for the value $x$ and return a Phase2b$\langle x \rangle$ message to the proposer. Upon receiving Phase2b$\langle x \rangle$ messages from a majority of the acceptors, the proposed value $x$ is considered chosen.




**(a) Phase 1**



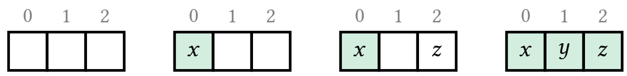

**(b) Phase 2**

**Figure 1: An example execution of Paxos ($f = 1$).**

## 2.3 MultiPaxos

While consensus is the act of choosing a single value, **state machine replication** is the act of choosing a sequence (a.k.a. log) of values. A state machine replication protocol manages a number of copies, or **replicas**, of a deterministic state machine. Over time, the protocol constructs a growing log of state machine commands, and replicas execute the commands in log order. By beginning in the same initial state, and by executing the same commands in the same order, all state machine replicas are kept in sync. This is illustrated in Figure 2.

**MultiPaxos** is one of the most widely used state machine replication protocols. Again, we assume the reader is familiar with MultiPaxos, but we review the most salient bits. MultiPaxos uses one instance of Paxos for every log entry, choosing the command in the $i$th log entry using the $i$th instance of Paxos. A MultiPaxos deployment that tolerates $f$ faults consists of an arbitrary number of clients, at least $f+1$ proposers, and $2f+1$ acceptors (like Paxos), as well as at least $f+1$ replicas, as illustrated in Figure 3.




**(a)** $t = 0$ **(b)** $t = 1$ **(c)** $t = 2$ **(d)** $t = 3$

**Figure 2: At time $t = 0$, no state machine commands are chosen. At time $t = 1$ command $x$ is chosen in slot 0. At times $t = 2$ and $t = 3$, commands $z$ and $y$ are chosen in slots 2 and 1. Executed commands are shaded green. Note that all state machines execute the commands $x$, $y$, $z$ in log order.**

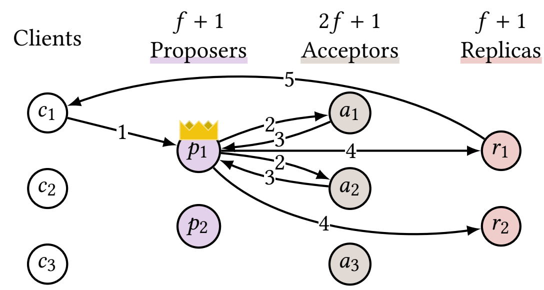



**Figure 3: An example execution of MultiPaxos ($f = 1$). The leader is adorned with a crown.**

Initially, one of the proposers is elected leader and runs Phase 1 of Paxos for every log entry. When a client wants to propose a state machine command $x$, it sends the command to the leader (1). The leader assigns the command a log entry $i$ and then runs Phase 2 of the $i$th Paxos instance to get the value $x$ chosen in entry $i$. That is, the leader sends Phase2A$\langle i, x\rangle$ messages to the acceptors to vote for value $x$ in slot $i$ (2). In the normal case, the acceptors all vote for $x$ in slot $i$ and respond with Phase2B$\langle i, x\rangle$ messages (3). Once the leader learns that a command has been chosen in a given log entry (i.e. once the leader receives Phase2B$\langle i, x\rangle$ messages from a majority of the acceptors), it informs the replicas (4). Replicas insert commands into their logs and execute the logs in prefix order.

Note that the leader assigns log entries to commands in increasing order. The first received command is put in entry 0, the next command in entry 1, the next command in entry 2, and so on. Also note that even though every replica executes every command, for any given state machine command $x$, only one replica needs to send the result of executing $x$ back to the client (5). For example, log entries can be round-robin partitioned across the replicas.

## 2.4 MultiPaxos Doesn't Scale?

It is widely believed that MultiPaxos does not scale. Throughout the paper, we will explain that this is not true. We can scale MultiPaxos, but first it helps to understand why trying to scale MultiPaxos in the straightforward and obvious way does not work. MultiPaxos consists of proposers, acceptors, and replicas. We discuss each.

First, increasing the number of proposers *does not improve performance* because every client must send its requests to the leader regardless of the number proposers. The non-leader replicas are idle and do not contribute to the protocol during normal operation.

Second, increasing the number of acceptors *hurts performance*. To get a value chosen, the leader must contact a majority of the acceptors. When we increase the number of acceptors, we increase the number of acceptors that the leader has to contact. This decreases throughput because the leader—which is the throughput bottleneck—has to send and receive more messages per command. Moreover, every acceptor processes at least half of all commands regardless of the number of acceptors.

Third, increasing the number of replicas *hurts performance*. The leader broadcasts chosen commands to all of the replicas, so when we increase the number of replicas, we increase the load on the leader and decrease MultiPaxos' throughput. Moreover, every replica must execute every state machine command, so increasing the number of replicas does not decrease the replicas' load.

# 3 COMPARTMENTALIZING MULTIPAXOS

We now compartmentalize MultiPaxos. Throughout the paper, we introduce six compartmentalizations, summarized in Table 1. For every compartmentalization, we identify a throughput bottleneck and then explain how to decouple and scale it.

## 3.1 Compartmentalization 1: Proxy Leaders

**Bottleneck:** *leader*
**Decouple:** *command sequencing and broadcasting*
**Scale:** *the number of command broadcasters*

*Bottleneck.* The MultiPaxos leader is a well known throughput bottleneck for the following reason. Refer again to Figure 3. To process a single state machine command from a client, the leader must receive a message from the client, send at least $f + 1$ Phase2A messages to the acceptors, receive at least $f + 1$ Phase2B messages from the acceptors, and send at least $f + 1$ messages to the replicas. In total, the leader sends and receives at least $3f + 4$ messages per command. Every acceptor on the other hand processes only 2 messages, and every replica processes either 1 or 2. Because every state machine command goes through the leader, and because the leader has to perform disproportionately more work than every other component, the leader is the throughput bottleneck.

*Decouple.* To alleviate this bottleneck, we first decouple the leader. To do so, we note that a MultiPaxos leader has two jobs. The first is **sequencing**. The leader sequences commands by assigning each command a log entry. Log entry 0, then 1, then 2, and so on. The second is **broadcasting**. The leader sends Phase2A messages, collects Phase2B responses, and broadcasts chosen values to the replicas. Historically, these two responsibilities have both fallen on the leader, but this is not fundamental. We instead decouple the two responsibilities. We introduce a set of at least $f + 1$ **proxy leaders**, as shown in Figure 4. The leader is responsible for sequencing commands, while the proxy leaders are responsible for getting commands chosen and broadcasting the commands to the replicas.

More concretely, when a leader receives a command $x$ from a client (1), it assigns the command $x$ a log entry $i$ and then forms a Phase2A message that includes $x$ and $i$. The leader does *not* send the Phase2A message to the acceptors. Instead, it sends the Phase2A message to a randomly selected proxy leader (2). Note

**Table 1: A summary of the compartmentalizations presented in this paper.**

| Compartmentalization | Bottleneck | Decouple | Scale |
|---|---|---|---|
| 1 (Section 3.1) | leader | command sequencing and command broadcasting | the number of proxy leaders |
| 2 (Section 3.2) | acceptors | read quorums and write quorums | the number of write quorums |
| 3 (Section 3.3) | replicas | command sequencing and command broadcasting | the number of replicas |
| 4 (Section 3.4) | leader and replicas | read path and write path | the number of read quorums |
| 5 (Section 4.1) | leader | batch formation and batch sequencing | the number of batchers |
| 6 (Section 4.2) | replicas | batch processing and batch replying | the number of unbatchers |

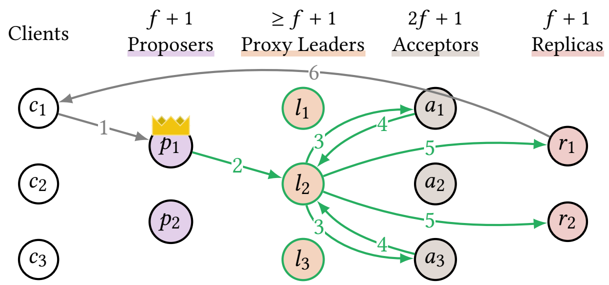



**Figure 4: An example execution of Compartmentalized MultiPaxos with three proxy leaders ($f = 1$). Throughout the paper, nodes and messages that were not present in previous iterations of the protocol are highlighted in green.**

that every command can be sent to a different proxy leader. The leader balances load evenly across all of the proxy leaders. Upon receiving a Phase2A message, a proxy leader broadcasts it to the acceptors (3), gathers a quorum of $f + 1$ Phase2B responses (4), and notifies the replicas of the chosen value (5). All other aspects of the protocol remain unchanged.

Without proxy leaders, the leader processes $3f + 4$ messages per command. With proxy leaders, the leader only processes 2. This makes the leader significantly less of a throughput bottleneck, or potentially eliminates it as the bottleneck entirely.

*Scale.* The leader now processes fewer messages per command, but every proxy leader has to process $3f + 4$ messages. Have we really eliminated the leader as a bottleneck, or have we just moved the bottleneck into the proxy leaders? To answer this, we note that the proxy leaders are embarrassingly parallel. They operate independently from one another. Moreover, the leader distributes load among the proxy leaders equally, so the load on any single proxy leader decreases as we increase the number of proxy leaders. Thus, we can trivially increase the number of proxy leaders until they are no longer a throughput bottleneck.

*Discussion.* Note that decoupling *enables* scaling. As discussed in Section 2.4, we cannot naively increase the number of proposers. Without decoupling, the leader is both a sequencer and broadcaster, so we cannot increase the number of leaders to increase the number of broadcasters because doing so would lead to multiple sequencers, which is not permitted. Only by decoupling the two responsibilities can we scale one without scaling the other.

Also note that the protocol remains tolerant to $f$ faults regardless of the number of machines. However, increasing the number of machines does decrease the expected time to $f$ failures (this is true for every protocol that scales up the number of machines, not just our protocol). We believe that increasing throughput at the expense of a shorter time to $f$ failures is well worth it in practice because failed machines can be replaced with new machines using a reconfiguration protocol [24, 30]. The time required to perform a reconfiguration is many orders of magnitude smaller than the mean time between failures.

## 3.2 Compartmentalization 2: Acceptor Grids

**Bottleneck:** *acceptors*
**Decouple:** *read quorums and write quorums*
**Scale:** *the number of write quorums*

*Bottleneck.* After compartmentalizing the leader, it is possible that the acceptors are the throughput bottleneck. It is widely believed that acceptors do not scale: "using more than $2f + 1$ [acceptors] for $f$ failures is possible but illogical because it requires a larger quorum size with no additional benefit" [42]. As explained in Section 2.4, there are two reasons why naively increasing the number of acceptors is ill-advised.

First, increasing the number of acceptors increases the number of messages that the leader has to send and receive. This increases the load on the leader, and since the leader is the throughput bottleneck, this decreases throughput. This argument no longer applies. With the introduction of proxy leaders, the leader no longer communicates with the acceptors. Increasing the number of acceptors increases the load on every individual proxy leader, but the increased load will not make the proxy leaders a bottleneck because we can always scale them up.

Second, every command must be processed by a majority of the acceptors. Thus, even with a large number of acceptors, every acceptor must process at least half of all state machine commands. This argument still holds.

*Decouple.* We compartmentalize the acceptors by using flexible quorums [18]. MultiPaxos—the vanilla version, not the compartmentalized version—requires $2f + 1$ acceptors, and the leader communicates with $f + 1$ acceptors in both Phase 1 and Phase 2 (a majority of the acceptors). The sets of $f + 1$ acceptors are called **quorums**, and MultiPaxos' correctness relies on the fact that any two quorums intersect. While majority quorums are sufficient for correctness, they are not necessary. MultiPaxos is correct as long as every quorum contacted in Phase 1 (called a **read quorum**) intersects every quorum contacted in Phase 2 (called a **write quorum**). Read quorums do not have to intersect other read quorums, and write quorums do not have to intersect other write quorums.

By decoupling read quorums from write quorums, we can reduce the load on the acceptors by eschewing majority quorums for a more efficient set of quorums. Specifically, we arrange the acceptors into an $r \times w$ rectangular grid, where $r, w \geq f + 1$. Every row forms a read quorum, and every column forms a write quorum ($r$ stands for row and for read). That is, a leader contacts an arbitrary row of acceptors in Phase 1 and an arbitrary column of acceptors for every command in Phase 2. Every row intersects every column, so this is a valid set of quorums.

A $2 \times 3$ acceptor grid is illustrated in Figure 5. There are two read quorums (the rows $\{a_1, a_2, a_3\}$ and $\{a_4, a_5, a_6\}$) and three write quorums (the columns $\{a_1, a_4\}$, $\{a_2, a_5\}$, $\{a_3, a_6\}$). Because there are three write quorums, every acceptor only processes one third of all the commands. This is not possible with majority quorums because with majority quorums, every acceptor processes at least half of all the commands, regardless of the number of acceptors.

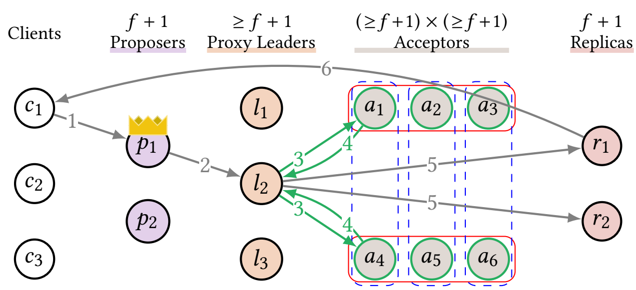



**Figure 5: An execution of Compartmentalized MultiPaxos with a $2 \times 3$ grid of acceptors ($f = 1$). The two read quorums—$\{a_1, a_2, a_3\}$ and $\{a_4, a_5, a_6\}$—are shown in solid red rectangles. The three write quorums—$\{a_1, a_4\}$, $\{a_2, a_5\}$, and $\{a_3, a_6\}$—are shown in dashed blue rectangles.**

*Scale.* With majority quorums, every acceptor has to process at least half of all state machines commands. With grid quorums, every acceptor only has to process $\frac{1}{w}$ of the state machine commands. Thus, we can increase $w$ (i.e. increase the number of columns in the grid) to reduce the load on the acceptors and eliminate them as a throughput bottleneck.

*Discussion.* Note that, like with proxy leaders, decoupling *enables* scaling. With majority quorums, read and write quorums are coupled, so we cannot increase the number of acceptors without also increasing the size of all quorums. Acceptor grids allow us to decouple the number of acceptors from the size of write quorums, allowing us to scale up the acceptors and decrease their load.

Also note that increasing the number of write quorums increases the size of read quorums which increases the number of acceptors that a leader has to contact in Phase 1. We believe this is a worthy trade-off since Phase 2 is executed in the normal case and Phase 1 is only run in the event of a leader failure.

### 3.3 Compartmentalization 3: More Replicas

**Bottleneck:** *replicas*
**Decouple:** *command sequencing and broadcasting*
**Scale:** *the number of replicas*

*Bottleneck.* After compartmentalizing the leader and the acceptors, it is possible that the replicas are the bottleneck. Recall from Section 2.4 that naively scaling the replicas does not work for two reasons. First, every replica must receive and execute every state machine command. This is not actually true, but we leave that for the next compartmentalization. Second, like with the acceptors, increasing the number of replicas increases the load on the leader. Because we have already decoupled sequencing from broadcasting on the leader and introduced proxy leaders, this is no longer true, so we are free to increase the number of replicas. In Figure 6, for example, we show MultiPaxos with three replicas instead of the minimum required two.

*Scale.* If every replica has to execute every command, does increasing the number of replicas decrease their load? Yes. Recall that while every replica has to execute every state machine, only *one* of the replicas has to send the result of executing the command back to the client. Thus, with $n$ replicas, every replica only has to send back results for $\frac{1}{n}$ of the commands. If we scale up the number of replicas, we reduce the number of messages that each replica has to send. This reduces the load on the replicas and helps prevent them from becoming a throughput bottleneck. In Figure 6 for example, with three replicas, every replica only has to reply to one third of all commands. With two replicas, every replica has to reply to half of all commands. In the next compartmentalization, we'll see another major advantage of increasing the number of replicas.

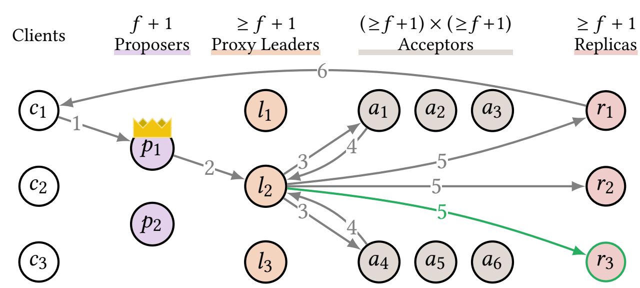



**Figure 6: An example execution of Compartmentalized MultiPaxos with three replicas as opposed to the minimum required two ($f = 1$).**

*Discussion.* Again decoupling *enables* scaling. Without decoupling the leader and introducing proxy leaders, increasing the number of replicas hurts rather than helps performance.

### 3.4 Compartmentalization 4: Leaderless Reads

**Bottleneck:** *leader and replicas*
**Decouple:** *read path and write path*
**Scale:** *the number of read quorums*

*Bottleneck.* We have now compartmentalized the leader, the acceptors, and the replicas. At this point, the bottleneck is in one of two places. Either the leader is still a bottleneck, or the replicas are the bottleneck. Fortunately, we can bypass both bottlenecks with a single compartmentalization.

*Decouple.* We call commands that modify the state of the state machine **writes** and commands that don't modify the state of the state machine **reads**. The leader must process every write because it has to linearize the writes with respect to one another, and *every* replica must process every write because otherwise the replicas' state would diverge (imagine if one replica performs a write but the other replicas don't). However, because reads do not modify the state of the state machine, the leader does not have to linearize them (reads commute), and only a single replica (as opposed to every replica) needs to execute a read.

We take advantage of this observation by decoupling the read path from the write path. Writes are processed as before, but we bypass the leader and perform a read on a single replica by using the idea from Paxos Quorum Reads (PQR) [13]. Specifically, to perform a read, a client sends a PreRead⟨⟩ message to a read quorum of acceptors. Upon receiving a PreRead⟨⟩ message, an acceptor $a_i$ returns a PreReadAck⟨$w_i$⟩ message where $w_i$ is the index of the largest log entry in which the acceptor has voted (i.e. the largest log entry in which the acceptor has sent a Phase2b message). We call this $w_i$ a vote watermark. When the client receives PreReadAck messages from a read quorum of acceptors, it computes $i$ as the maximum of all received vote watermarks. It then sends a Read⟨$x, i$⟩ request to any one of the replicas where $x$ is an arbitrary read (i.e. a command that does not modify the state of the state machine).

When a replica receives a Read⟨$x, i$⟩ request from a client, it waits until it has executed the command in log entry $i$. Recall that replicas execute commands in log order, so if the replica has executed the command in log entry $i$, then it has also executed all of the commands in log entries less than $i$. After the replica has executed the command in log entry $i$, it executes $x$ and returns the result to the client. Note that upon receiving a Read⟨$x, i$⟩ message, a replica may have already executed the log beyond $i$. That is, it may have already executed the commands in log entries $i + 1$, $i + 2$, and so on. This is okay because as long as the replica has executed the command in log entry $i$, it is safe to execute $x$. See our technical report [41] for a proof that this protocol correctly implements linearizable reads.

*Scale.* The decoupled read and write paths are shown in Figure 7. Reads are sent to a row (read quorum) of acceptors, so we can increase the number of rows to decrease the read load on every individual acceptor, eliminating the acceptors as a read bottleneck. Reads are also sent to a single replica, so we can increase the number of replicas to eliminate them as a read bottleneck as well.

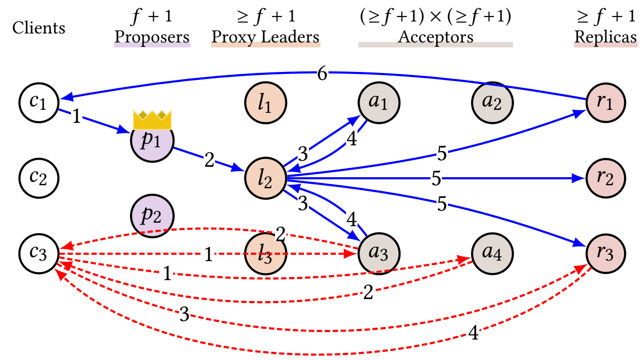



**Figure 7: An example execution of Compartmentalized MultiPaxos' read and write path ($f = 1$) with a $2 \times 2$ acceptor grid. The write path is shown using solid blue lines. The read path is shown using red dashed lines.**

*Discussion.* Note that read-heavy workloads are not a special case. Many workloads are read-heavy [7, 17, 27, 29]. Chubby [11] observes that fewer than 1% of operations are writes, and Spanner [15] observes that fewer than 0.3% of operations are writes.

Also note that increasing the number of columns in an acceptor grid reduces the write load on the acceptors, and increasing the number of rows in an acceptor grid reduces the read load on the acceptors. There is no throughput trade-off between the two. The number of rows and columns can be adjusted independently. Increasing read throughput (by increasing the number of rows) does not decrease write throughput, and vice versa. However, increasing the number of rows does increase the *size* (but not number) of columns, so increasing the number of rows might increase the tail latency of writes, and vice versa.

## 4 BATCHING

All state machine replication protocols, including MultiPaxos, can take advantage of batching to increase throughput. The standard way to implement batching [31, 33] is to have clients send their commands to the leader and to have the leader group the commands together into batches, as shown in Figure 8. The rest of the protocol remains unchanged, with command batches replacing commands. The one notable difference is that replicas now execute one batch of commands at a time, rather than one command at a time. After executing a single command, a replica has to send back a single result to a client, but after executing a batch of commands, a replica has to send a result to every client with a command in the batch.

### 4.1 Compartmentalization 5: Batchers

**Bottleneck:** *leader*
**Decouple:** *batch formation and batch sequencing*
**Scale:** *the number of batchers*

*Bottleneck.* We first discuss write batching and discuss read batching momentarily. Batching increases throughput by amortizing the communication and computation cost of processing a command.

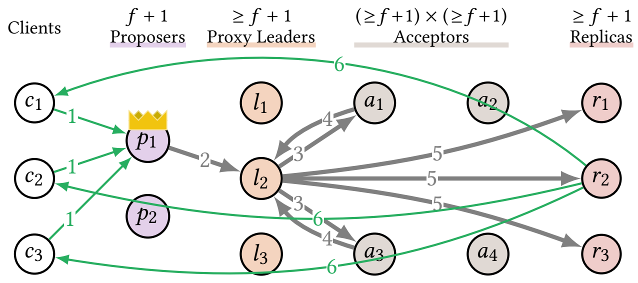



**Figure 8: An example execution of Compartmentalized MultiPaxos with batching ($f = 1$). Messages that contain a batch of commands, rather than a single command, are drawn thicker. Note how replica $r_2$ has to send multiple messages after executing a batch of commands.**

Take the acceptors for example. Without batching, an acceptor processes two messages *per command*. With batching, however, an acceptor only processes two messages *per batch*. The acceptors process fewer messages per command as the batch size increases. With batches of size 10, for example, an acceptor processes 10× fewer messages per command with batching than without.

Refer again to Figure 8. The load on the proxy leaders and the acceptors both decrease as the batch size increases, but this is not the case for the leader or the replicas. We focus first on the leader. To process a single batch of $n$ commands, the leader has to receive $n$ messages and send one message. Unlike the proxy leaders and acceptors, the leader's communication cost is linear in the number of commands rather than the number of batches. This makes the leader a very likely throughput bottleneck.

*Decouple.* The leader has two responsibilities. It forms batches, and it sequences batches. We decouple the two responsibilities by introducing a set of at least $f+1$ **batchers**, as illustrated in Figure 9. The batchers are responsible for forming batches, while the leader is responsible for sequencing batches.

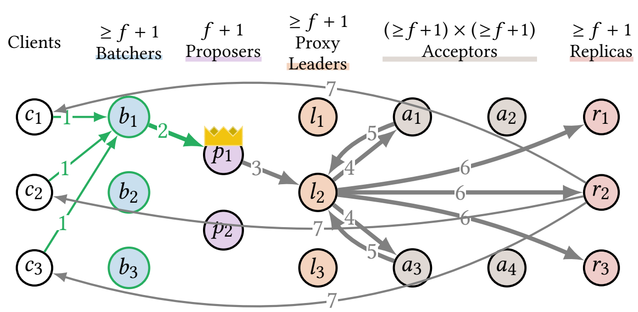



**Figure 9: An example execution of Compartmentalized MultiPaxos with batchers ($f = 1$).**

More concretely, when a client wants to propose a state machine command, it sends the command to a randomly selected batcher (1). After receiving sufficiently many commands from the clients (or after a timeout expires), a batcher places the commands in a batch and forwards it to the leader (2). When the leader receives a batch of commands, it assigns it a log entry, forms a Phase 2a message, and sends the Phase2a message to a proxy leader (3). The rest of the protocol remains unchanged.

Without batchers, the leader has to receive $n$ messages per batch of $n$ commands. With batchers, the leader only has to receive one. This either reduces the load on the bottleneck leader or eliminates it as a bottleneck completely.

*Scale.* The batchers are embarrassingly parallel, so we can increase the number of batchers until they are not a throughput bottleneck.

*Discussion.* Read batching is very similar to write batching. Clients send reads to randomly selected batchers, and batchers group reads together into batches. After a batcher has formed a read batch $X$, it sends a PreRead$\langle\rangle$ message to a read quorum of acceptors, computes the resulting watermark $i$, and sends a Read$\langle X, i\rangle$ request to any one of the replicas.

## 4.2 Compartmentalization 6: Unbatchers

**Bottleneck:** *replicas*
**Decouple:** *batch processing and batch replying*
**Scale:** *the number of unbatchers*

*Bottleneck.* After executing a batch of $n$ commands, a replica has to send $n$ messages back to the $n$ clients. Thus, the replicas (like the leader without batchers) suffer communication overheads linear in the number of commands rather than the number of batches.

*Decouple.* The replicas have two responsibilities. They execute batches of commands, and they send replies to the clients. We decouple these two responsibilities by introducing a set of at least $f+1$ **unbatchers**, as illustrated in Figure 10. The replicas are responsible for executing batches of commands, while the unbatchers are responsible for sending the results of executing the commands back to the clients. Concretely, after executing a batch of commands, a replica forms a batch of results and sends the batch to a randomly selected unbatcher (7). Upon receiving a result batch, an unbatcher sends the results back to the clients (8). This decoupling reduces the load on the replicas.

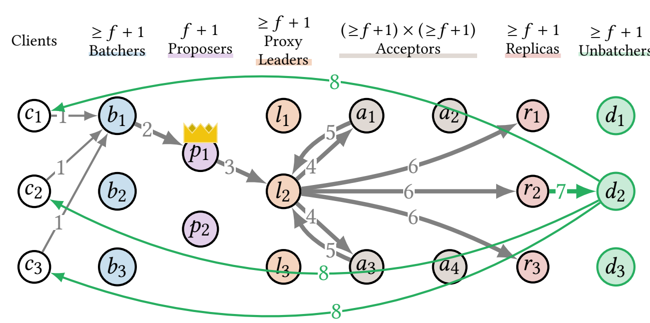



**Figure 10: An example execution of Compartmentalized MultiPaxos with unbatchers ($f = 1$).**

*Scale.* As with batchers, unbatchers are embarrassingly parallel, so we can increase the number of unbatchers until they are not a throughput bottleneck.

*Discussion.* Read unbatching is identical to write unbatching. After executing a batch of reads, a replica forms the corresponding batch of results and sends it to a randomly selected unbatcher.

# 5 FURTHER COMPARTMENTALIZATION

The six compartmentalizations that we've discussed are not exhaustive, and MultiPaxos is not the only state machine replication protocol that can be compartmentalized. Compartmentalization is a generally applicable technique. There are many other compartmentalizations that can be applied to many other protocols.

For example, Mencius [26] is a multi-leader MultiPaxos variant that round-robin partitions log entries between the leaders. S-Paxos [10] is a MultiPaxos variant in which every state machine command is given a unique id and persisted on a set of machines before MultiPaxos is used to order command ids rather than commands themselves. In our technical report [41], we explain how to compartmentalize these two protocols. We compartmentalize Mencius very similarly to how we compartmentalized MultiPaxos. We compartmentalize S-Paxos by introducing new sets of nodes called **disseminators** and **stabilizers** which are analogous to proxy leaders and acceptors but are used to persist commands rather than order them. We are also currently working on compartmentalizing Raft [30] and EPaxos [27]. Due to space constraints, we leave the details to our technical report [41].

# 6 EVALUATION

We begin by measuring the throughput and latency of MultiPaxos with all six of the compartmentalizations described in this paper (Section 6.1). We then perform an ablation study to measure the impact of each compartmentalization (Section 6.2). We conclude by measuring the scalability of reads (Section 6.3) and the skew tolerance of reads (Section 6.4)

## 6.1 Latency-Throughput

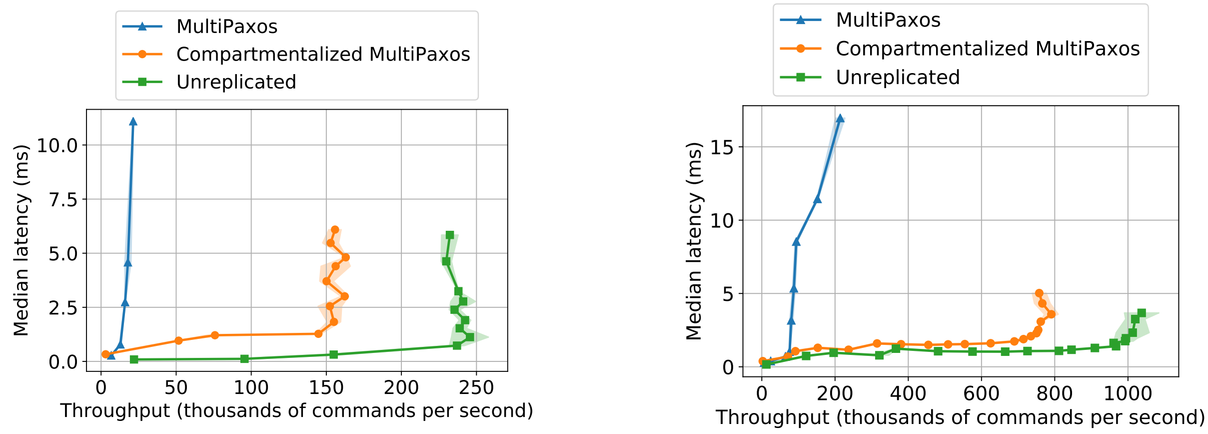
*Experiment Description.* We call MultiPaxos with the six compartmentalizations described in this paper **Compartmentalized MultiPaxos**. We implemented MultiPaxos, Compartmentalized MultiPaxos, and an unreplicated state machine in Scala using the Netty networking library (see github.com/mwhittaker/frankenpaxos). MultiPaxos employs $2f + 1$ machines with each machine playing the role of a MultiPaxos proposer, acceptor, and replica. The unreplicated state machine is implemented as a single process on a single server. Clients send commands directly to the state machine. Upon receiving a command, the state machine executes the command and immediately sends back the result. Note that unlike MultiPaxos and Compartmentalized MultiPaxos, the unreplicated state machine is *not* fault tolerant. If the single server fails, all state is lost and no commands can be executed. Thus, the unreplicated state machine should not be viewed as an apples-to-apples comparison with the other two protocols. Instead, the unreplicated state machine sets an upper bound on attainable performance.

We measure the throughput and median latency of the three protocols under workloads with a varying numbers of clients. Each client issues state machine commands in a closed loop. It waits to receive the result of executing its most recently proposed command before it issues another. All three protocols replicate a key-value store state machine where the keys are integers and the values are 16 byte strings. In this benchmark, all state machine commands are writes. There are no reads.

We deploy the protocols with and without batching for $f = 1$. Without batching, we deploy Compartmentalized MultiPaxos with two proposers, ten proxy leaders, a two by two grid of acceptors, and four replicas. With batching, we deploy two batchers, two proposers, three proxy replicas, a simple majority quorum system of three acceptors, two replicas, and three unbatchers. We deploy the three protocols on AWS using a set of m5.xlarge machines within a single availability zone. All numbers presented are the average of three executions of the benchmark. As is standard, we implement MultiPaxos and Compartmentalized MultiPaxos with thriftiness enabled [27]. For a given number of clients, the batch size is set empirically to optimize throughput. For a fair comparison, we deploy the unreplicated state machine with a set of batchers and unbatchers when batching is enabled.

*Results.* The results of the experiment are shown in Figure 11. The standard deviation of throughput measurements are shown as a shaded region. Without batching, MultiPaxos has a peak throughput of roughly 25,000 commands per second, while Compartmentalized MultiPaxos has a peak throughput of roughly 150,000 commands per second, a 6× increase. The unreplicated state machine outperforms both protocols. It achieves a peak throughput of roughly 250,000 commands per second. Compartmentalized MultiPaxos underperforms the unreplicated state machine because—despite decoupling the leader as much as possible—the single leader remains a throughput bottleneck. All three protocols have millisecond latencies at peak throughput. With batching, MultiPaxos, Compartmentalized MultiPaxos, and the unreplicated state machine have peak throughputs of roughly 200,000, 800,000 and 1,000,000 commands per second respectively.

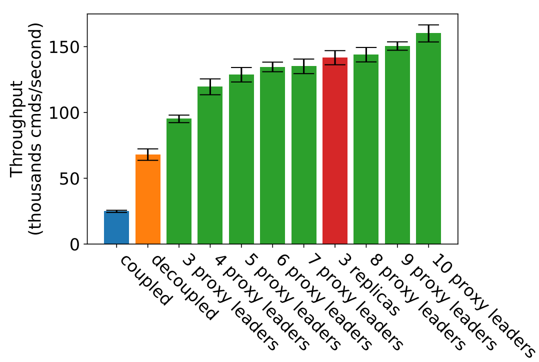
Compartmentalized MultiPaxos uses 6.66× more machines than MultiPaxos. On the surface, this seems like a weakness, but in reality it is a strength. MultiPaxos does not scale, so it is unable to take advantage of more machines. Compartmentalized MultiPaxos, on the other hand, achieves a 6× increase in throughput using 6.66× the number of resources. We scale throughput almost linearly with the number of machines. In fact, with the mixed read-write workloads below, we are able to scale throughput superlinearly with the number of resources. This is because compartmentalization eliminates throughput bottlenecks. With throughput bottlenecks, non-bottlenecked components are underutilized. When we eliminate the bottlenecks, we eliminate underutilization and can increase performance without increasing the number of resources. Moreover, a protocol does not have to be *fully* compartmentalized. We can selectively compartmentalize some but not all throughput bottlenecks to reduce the number of resources needed. In other words, MultiPaxos and Compartmentalized MultiPaxos are not two alternatives, but rather two extremes in a trade-off between throughput and resource usage.



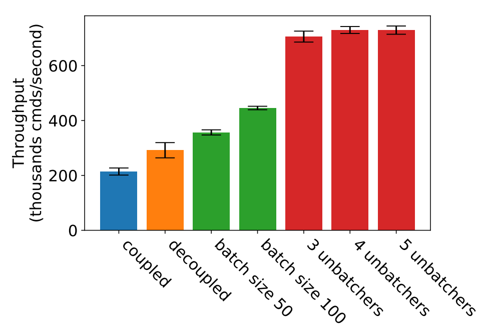

**(a) Without batching**

**(b) With batching**

**Figure 11: The latency and throughput of MultiPaxos, Compartmentalized MultiPaxos, and an unreplicated state machine.**




**(a) Without batching**




**(b) With batching**

**Figure 12: An ablation study. Standard deviations are shown using error bars.**

## 6.2 Ablation Study

*Experiment Description.* We now perform an ablation study to measure the effect of each compartmentalization. In particular, we begin with MultiPaxos and then decouple and scale the protocol according to the six compartmentalizations, measuring peak throughput along the way. Note that we cannot measure the effect of each individual compartmentalization in isolation because decoupling and scaling a component only improves performance if that component is a bottleneck. Thus, to measure the effect of each compartmentalization, we have to apply them all, and we have to apply them in an order that is consistent with the order in which bottlenecks appear. All the details of this experiment are the same as the previous experiment unless otherwise noted.

*Results.* The unbatched ablation study results are shown in Figure 12a. MultiPaxos has a throughput of roughly 25,000 commands per second. When we decouple the protocol and introduce proxy leaders (Section 3.1), we increase the throughput to roughly 70,000 commands per second. This decoupled MultiPaxos uses the bare minimum number of proposers (2), proxy leaders (2), acceptors (3), and replicas (2). We then scale up the number of proxy leaders from 2 to 7. The proxy leaders are the throughput bottleneck, so as we scale them up, the throughput of the protocol increases until it plateaus at roughly 135,000 commands per second. At this point, the proxy leaders are no longer the throughput bottleneck; the replicas are. We introduce an additional replica (Section 3.3), though the throughput does not increase. This is because proxy leaders broadcast commands to all replicas, so introducing a new replica increases the load on the proxy leaders making them the bottleneck again. We then increase the number of proxy leaders to 10 to increase the throughput to roughly 150,000 commands per second. At this point, we determined empirically that the leader was the bottleneck. In this experiment, the acceptors are never the throughput bottleneck, so increasing the number of acceptors does not increase the throughput (Section 3.2). However, this is particular to our write-only workload. In the mixed read-write workloads discussed momentarily, scaling up the number of acceptors is critical for high throughput.

The batched ablation study results are shown in Figure 12b. We decouple MultiPaxos and introduce two batchers and two unbatchers with a batch size of 10 (Section 4.1, Section 4.2). This increases the throughput of the protocol from 200,000 commands per second to 300,000 commands per second. We then increase the batch size to 50 and then to 100. This increases throughput to 500,000 commands per second. We then increase the number of unbatchers to 3 and reach a peak throughput of roughly 800,000 commands per second. For this experiment, two batchers and three unbatchers are sufficient to handle the clients' load. With more clients and a larger load, more batchers would be needed to maximize throughput.

### 6.3 Read Scalability

*Experiment Description.* Thus far, we have looked at write-only workloads. We now measure the throughput of Compartmentalized MultiPaxos under a workload with reads *and* writes. In particular, we measure how the throughput of Compartmentalized MultiPaxos scales as we increase the number of replicas. We deploy Compartmentalized MultiPaxos with and without batching; with 2, 3, 4, 5, and 6 replicas; and with workloads that have 0%, 60%, 90%, and 100% reads. For any given workload and number of replicas, proxy leaders, and acceptors is chosen to maximize throughput. The batch size is 50. In the batched experiments, we do *not* use batchers and unbatchers. Instead, clients form batches of commands themselves. This has no effect on the throughput measurements. We did this only to reduce the number of client machines that we needed to saturate the system. This was not an issue with the write-only workloads because they had significantly lower peak throughputs.

*Results.* The unbatched results are shown in Figure 13a. We also show MultiPaxos' throughput for comparison. MultiPaxos does not distinguish reads and writes, so there is only a single line to compare against. With a 0% read workload, Compartmentalized MultiPaxos has a throughput of roughly 150,000 commands per second, and the protocol does not scale much with the number of replicas. This is consistent with our previous experiments. For workloads with reads and writes, our results confirm two expected trends. First, the higher the fraction of reads, the higher the throughput. Second, the higher the fraction of reads, the better the protocol scales with the number of replicas. With a 100% read workload, for example, Compartmentalized MultiPaxos scales linearly up to a throughput of roughly 650,000 commands per second with 6 replicas. The batched results, shown in Figure 13b, are very similar. With a 100% read workload, Compartmentalized MultiPaxos scales linearly up to a throughput of roughly 17.5 million commands per second.

Our results also show two *counterintuitive* trends. First, a small increase in the fraction of writes can lead to a disproportionately large decrease in throughput. For example, the throughput of the 90% read workload is far less than 90% of the throughput of the 100% read workload. Second, besides the 100% read workload, throughput does *not* scale linearly with the number of replicas. We see that the throughput of the 0%, 60%, and 90% read workloads scale sublinearly with the number of replicas. These results are not an artifact of our protocol; they are fundamental. Any state machine replication protocol where writes are processed by every replica and where reads are processed by a single replica [13, 37, 43] will exhibit these same two performance anomalies.

We can explain this analytically. Assume that we have $n$ replicas; that every replica can process at most $\alpha$ commands per second; and that we have a workload with a $f_w$ fraction of writes and a $f_r = 1 - f_w$ fraction of reads. Let $T$ be peak throughput, measured in commands per second. Then, our protocol has a peak throughput of $f_w T$ writes per second and $f_r T$ reads per second. Writes are processed by *every* replica, so we impose a load of $n f_w T$ writes per second on the replicas. Reads are processed by *a single* replica, so we impose a load of $f_r T$ reads per second on the replicas. The total aggregate throughput of the system is $n\alpha$, so we have $n\alpha = n f_w T + f_r T$. Solving for $T$, we find the peak throughput of our system is

$$\frac{n\alpha}{n f_w + f_r}$$

This formula is plotted in Figure 14 with $\alpha = 100,000$. The limit of our peak throughput as $n$ approaches infinity is $\frac{\alpha}{f_w}$. This explains both of the performance anomalies described above. First, it shows that peak throughput has a $\frac{1}{f_w}$ relationship with the fraction of writes, meaning that a small increase in $f_w$ can have a large impact on peak throughput. For example, if we increase our write fraction from 1% to 2%, our throughput will half. A 1% change in write fraction leads to a 50% reduction in throughput. Second, it shows that throughput does not scale linearly with the number of replicas; it is upper bounded by $\frac{\alpha}{f_w}$. For example, a workload with 50% writes can never achieve more than twice the throughput of a 100% write workload, even with an infinite number of replicas.

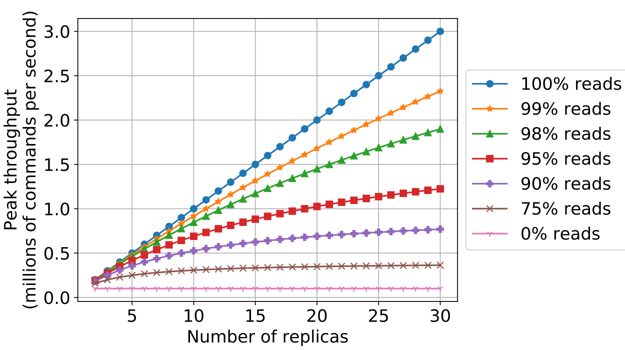
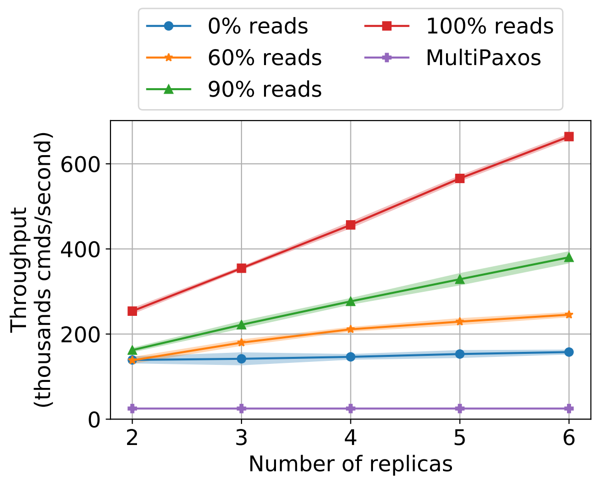



**Figure 14: Analytical throughput vs the number of replicas.**

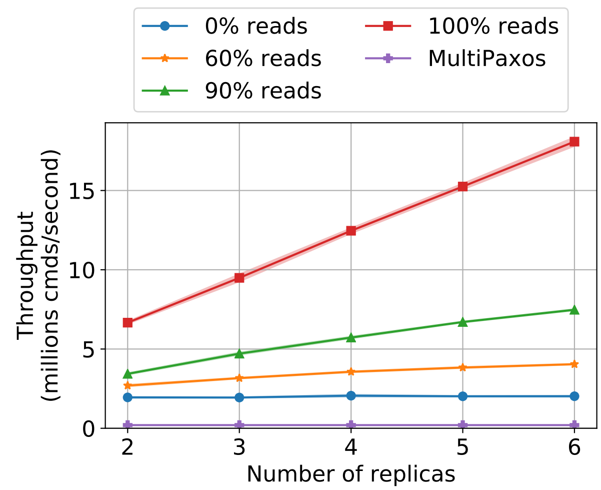
### 6.4 Skew Tolerance

*Experiment Description.* CRAQ [37] is a chain replication [40] variant with scalable reads. A CRAQ deployment consists of at least $f + 1$ nodes arranged in a linked list, or chain. Writes are sent to the head of the chain and propagated node-by-node down the chain from the head to the tail. When the tail receives the write, it sends a write acknowledgement to its predecessor, and this ack is propagated node-by-node backwards through the chain until it reaches the head. Reads are sent to any node. When a node receives a read of key $k$, it checks to see if it has any unacknowledged write to that key. If it doesn't, then it performs the read and replies to the client immediately. If it does, then it forwards the read to the tail of the chain. When the tail receives a read, it executes the read immediately and replies to the client.

We now compare Compartmentalized MultiPaxos with our implementation of CRAQ. In particular, we show that CRAQ (and similar protocols like Harmonia [43]) are sensitive to data skew,




**(a) Unbatched linearizable reads**




**(b) Batched linearizable reads**

**Figure 13: Peak throughput vs the number of replicas**

whereas Compartmentalized MultiPaxos is not. We deploy Compartmentalized MultiPaxos with six replicas and CRAQ with six chain nodes. Both protocols replicate a key-value store with 10,000 keys in the range $1, \ldots, 10,000$. We subject both protocols to the following workload. A client repeatedly flips a weighted coin, and with probability $p$ chooses to read or write to key 0. With probability $1 - p$, it decides to read or write to some other key $2, \ldots, 10,000$ chosen uniformly at random. The client then decides to perform a read with 95% probability and a write with 5% probability. As we vary the value of $p$, we vary the skew of the workload. When $p = 0$, the workload is completely uniform, and when $p = 1$, the workload consists of reads and writes to a single key. This artificial workload allows to study the effect of skew in a simple way without having to understand more complex skewed distributions.

*Results.* The results are shown in Figure 15, with $p$ on the $x$-axis. The throughput of Compartmentalized MultiPaxos is constant; it is independent of $p$. This is expected because Compartmentalized MultiPaxos is completely agnostic to the state machine that it is replicating and is completely unaware of the notion of keyed data. Its performance is only affected by the ratio of reads to writes and is completely unaffected by what data is actually being read or written. CRAQ, on the other hand, is susceptible to skew. As we increase skew from $p = 0$ to $p = 1$, the throughput decreases from roughly 300,000 commands per second to roughly 100,000 commands per second. As we increase $p$, we increase the fraction of reads which are forwarded to the tail. In the extreme, all reads are forwarded to the tail, and the throughput of the protocol is limited to that of a single node (i.e. the tail).

However, with low skew, CRAQ can perform reads in a single round trip to a single chain node. This allows CRAQ to implement reads with lower latency and with fewer nodes than Compartmentalized MultiPaxos. However, we also note that Compartmentalized MultiPaxos outperforms CRAQ in our benchmark even with no skew. This is because every chain node must process four messages per write, whereas Compartmentalized MultiPaxos replicas only have to process two. CRAQ's write latency also increases with the number of chain nodes, creating a hard trade-off between read throughput and write latency. Ultimately, neither protocol is strictly better than the other. For very read-heavy workloads with low-skew, CRAQ will likely outperform Compartmentalized MultiPaxos, and for workloads with more writes or more skew, Compartmentalized MultiPaxos will likely outperform CRAQ.

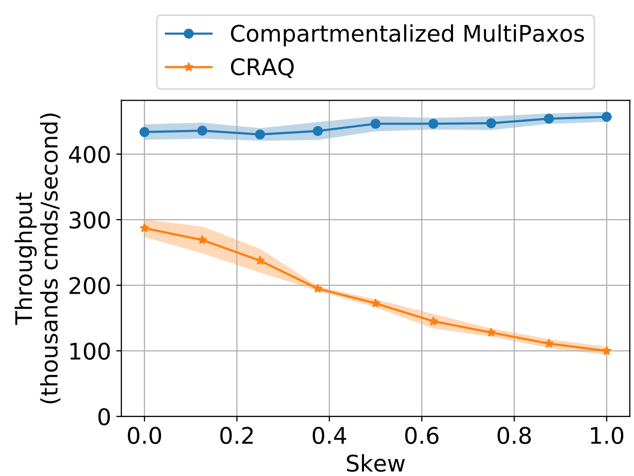



**Figure 15: The effect of skew on Compartmentalized MultiPaxos and CRAQ.**

## 7 RELATED WORK

*MultiPaxos.* Unlike state machine replication protocols like Raft [30] and Viewstamped Replication [25], MultiPaxos [21, 24, 39] is designed with the roles of proposer, acceptor, and replicas logically decoupled. This decoupling alone is not sufficient for MultiPaxos to achieve the best possible throughput, but the decoupling allows for the compartmentalizations described in this paper.

*PigPaxos.* PigPaxos [14] is a MultiPaxos variant that alters the communication flow between the leader and the acceptors to improve scalability and throughput. Similar to compartmentalization, PigPaxos realizes that the leader is doing many different jobs and is a bottleneck in the system. In particular, PigPaxos substitutes direct leader-to-acceptor communication with a relay network. In PigPaxos the leader sends a message to one or more randomly selected relay nodes, and each relay rebroadcasts the leader's message to the peers in its relay-group and waits for some threshold of responses. Once each relay receives enough responses from its peers, it aggregates them into a single message to reply to the leader. The leader selects a new set of random relays for each new message to prevent faulty relays from having a long-term impact on the communication flow. PigPaxos relays are comparable to our proxy leaders, although the relays are simpler and only alter the communication flow. As such, the relays cannot generally take over the other leader roles, such as quorum counting or replying to the clients. Unlike PigPaxos, whose main goal is to grow to larger clusters, compartmentalization is more general and improves throughput under different conditions and situations.

*Chain Replication.* Chain Replication [40] is a state machine replication protocol in which the set of state machine replicas are arranged in a totally ordered chain. Writes are propagated through the chain from head to tail, and reads are serviced exclusively by the tail. Chain Replication has high throughput compared to MultiPaxos because load is more evenly distributed between the replicas, but every replica must process four messages per command, as opposed to two in Compartmentalized MultiPaxos. The tail is also a throughput bottleneck for read-heavy workloads. Finally, Chain Replication is not tolerant to network partitions and is therefore not appropriate in all situations.

*Scalog.* Scalog [16] is a replicated shared log protocol that achieves high throughput using an idea similar to Compartmentalized MultiPaxos' batchers and unbatchers. A client does not send values directly to a centralized leader for sequencing in the log. Instead, the client sends its values to one of a number of batchers. Periodically, the batchers' batches are sealed and assigned an id. This id is then sent to a state machine replication protocol, like MultiPaxos, for sequencing. Scalog is complementary to Compartmentalized MultiPaxos. The state machine replication protocol that Scalog uses can be compartmentalized.

*Scalable Agreement.* In [20], Kapritsos et al. present a protocol similar to Compartmentalized Mencius (as described in our technical report [41]). The protocol round-robin partitions log entries among a set of replica clusters co-located on a fixed set of machines. Every cluster has $2f + 1$ replicas, with every replica playing the role of a Paxos proposer and acceptor. Compartmentalized Mencius extends the protocol with the compartmentalizations described in this paper.

*Multithreaded Replication.* [32] and [8] both propose multithreaded state machine replication protocols. Multithreaded protocols like these are necessarily decoupled and scale within a single machine. This work is complementary to compartmentalization. Compartmentalization works at the protocol level, while multithreading works on the process level. Both can be applied to a single protocol.

*Read Leases.* A common way to optimize reads in MultiPaxos is to grant a lease to the leader [11, 12, 15]. While the leader holds the lease, no other node can become leader. As a result, the leader can perform reads locally without contacting other nodes. Leases assume some degree of clock synchrony, so they are not appropriate in all circumstances. Moreover, the leader is still a read bottleneck. Raft has a similar optimization that does not require any form of clock synchrony, but the leader is still a read bottleneck [30]. With Paxos Quorum Leases [28], any set of nodes—not just the leader—can hold a lease for a set of objects. These lease holders can read the objects locally. Paxos Quorum Leases assume clock synchrony and are a special case of Paxos Quorum Reads [13] in which read quorums consist of any lease holding node and write quorums consist of any majority that includes all the lease holding nodes. Compartmentalization MultiPaxos does not assume clock synchrony and has no read bottlenecks.

*Harmonia.* Harmonia [43] is a family of state machine replication protocols that leverage specialized hardware—specifically, a specialized network switch—to achieve high throughput and low latency. Like CRAQ, Harmonia is sensitive to data skew. It performs extremely well under low contention, but degrades in performance as contention grows. Harmonia also assumes clock synchrony, whereas Compartmentalized MultiPaxos does not. FLAIR [36] is replication protocol that also leverages specialized hardware, similar to Harmonia.

*Sharding.* In this paper, we have discussed state machine replication in its most general form. We have not made any assumptions about the nature of the state machines themselves. Because of this, we are not able to decouple the state machine replicas. Every replica must execute every write. This creates a fundamental throughput limit. However, if we are able to divide the state of the state machine into independent shards, then we can further scale the protocols by sharding the state across groups of replicas. For example, in [9], Bezerra et al. discuss how state machine replication protocols can take advantage of sharding.

## 8 CONCLUSION

In this paper, we analyzed the throughput bottlenecks in state machine replication protocols and demonstrated how to eliminate them using a combination of decoupling and scale, a technique we call compartmentalization. Using compartmentalization, we establish a new baseline for MultiPaxos' performance. We increase the protocol's throughput by a factor of 6× on a write-only workload and 16× on a 90% read workload, all without the need for complex or specialized protocols.

# REFERENCES


[1] [n.d.]. A Brief Introduction of TiDB. https://pingcap.github.io/blog/2017-05-23-perconalive17/. Accessed: 2019-10-21.

[2] [n.d.]. Global data distribution with Azure Cosmos DB - under the hood. https://docs.microsoft.com/en-us/azure/cosmos-db/global-dist-under-the-hood. Accessed: 2019-10-21.

[3] [n.d.]. Lightweight transactions in Cassandra 2.0. https://www.datastax.com/blog/2013/07/lightweight-transactions-cassandra-20. Accessed: 2019-10-21.

[4] [n.d.]. Raft Replication in YugaByte DB. https://www.yugabyte.com/resources/raft-replication-in-yugabyte-db/. Accessed: 2019-10-21.

[5] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2019. WPaxos: Wide Area Network Flexible Consensus. *IEEE Transactions on Parallel and Distributed Systems* (2019).

[6] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2017. Speeding up Consensus by Chasing Fast Decisions. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on.* IEEE, 49–60.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems.* 53–64.

[8] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2015. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Annual Middleware Conference.* ACM, 173–184.

[9] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. 2014. Scalable state-machine replication. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* IEEE, 331–342.

[10] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. 2012. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on.* IEEE, 111–120.

[11] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation.* USENIX Association, 335–350.

[12] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing.* ACM, 398–407.

[13] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2019. Linearizable quorum reads in Paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19).*

[14] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2020. PigPaxos: Devouring the communication bottlenecks in distributed consensus. *arXiv preprint arXiv:2003.07760* (2020).

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[16] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20).* 325–338.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles.* 29–43.

[18] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone (Eds.), Vol. 70. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 25:1–25:14. https://doi.org/10.4230/LIPIcs.OPODIS.2016.25

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* 35–49.

[20] Manos Kapritsos and Flavio Paiva Junqueira. 2010. Scalable Agreement: Toward Ordering as a Service.. In *HotDep.*

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

[22] Leslie Lamport. 2005. Generalized consensus and Paxos. (2005).

[23] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103.

[24] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[25] Barbara Liskov and James Cowling. 2012. Viewstamped replication revisited. (2012).

[26] Yanhua Mao, Flavio P Junqueira, and Keith Marzullo. 2008. Mencius: building efficient replicated state machines for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08).* 369–384.

[27] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* ACM, 358–372.

[28] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2014. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing.* 1–13.

[29] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13).* 385–398.

[30] Diego Ongaro and John K Ousterhout. 2014. In search of an understandable consensus algorithm.. In *USENIX Annual Technical Conference.* 305–319.

[31] Nuno Santos and André Schiper. 2012. Tuning paxos for high-throughput with batching and pipelining. In *International Conference on Distributed Computing and Networking.* Springer, 153–167.

[32] Nuno Santos and André Schiper. 2013. Achieving high-throughput state machine replication in multi-core systems. In *2013 IEEE 33rd International Conference on Distributed Computing Systems.* Ieee, 266–275.

[33] Nuno Santos and André Schiper. 2013. Optimizing Paxos with batching and pipelining. *Theoretical Computer Science* 496 (2013), 170–183.

[34] William Schultz, Tess Avitabile, and Alyson Cabral. 2019. Tunable Consistency in MongoDB. 12, 12 (2019), 2071–2081.

[35] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* ACM, 1493–1509.

[36] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. 2020. FLAIR: Accelerating Reads with Consistency-Aware Network Routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20).* 723–737.

[37] Jeff Terrace and Michael J Freedman. 2009. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads.. In *USENIX Annual Technical Conference.* San Diego, CA, 1–16.

[38] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* ACM, 1–12.

[39] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 42.

[40] Robbert Van Renesse and Fred B Schneider. 2004. Chain Replication for Supporting High Throughput and Availability.. In *OSDI*, Vol. 4.

[41] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2020. Scaling Replicated State Machines with Compartmentalization [Technical Report]. arXiv:2012.15762 [cs.DC]

[42] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. 2018. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 12.

[43] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan RK Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proceedings of the VLDB Endowment* 13, 3 (2019), 376–389.