Compartmentalized Consensus: Agreeing With High Throughput

Anonymous Author(s)
Submission Id: 705

Abstract

Dozens of state machine replication protocols have been invented that use a myriad of sophisticated techniques to achieve high throughput (e.g., multi-leader execution, generalization). These techniques are often complex and protocol-specific. In this paper, we present a simple and generally applicable technique, dubbed compartmentalization, that revolves around decoupling and scaling. We compartmentalize three existing protocols—MultiPaxos, Mencius, and S-Paxos—increasing their throughput by as much as 8.3×. In doing so, we debunk the myth that simple state machine replication protocols like MultiPaxos cannot achieve high throughput. Ironically, we find that after compartmentalization, some sophisticated techniques designed to improve throughput can actually become throughput bottlenecks.

1 Introduction

Machines can fail. State machine replication protocols like MultiPaxos [22] and Raft [32] mask these failures by executing multiple copies, or replicas, of a state machine at once. Today, state machine replication is ubiquitous. It is hard to find a strongly consistent distributed system that *does not* use some form of state machine replication [1–6, 10, 14–16, 20, 39].

Due to their importance, dozens of state machine replication protocols have been invented over the last several decades. These protocols use a number of sophisticated techniques like multi-leader execution [9, 29, 30], speculative execution [26, 33, 34], generalization [23, 38], fast paths [24], and flexible quorums [19, 31] to achieve high throughput, low latency, or both.

These techniques introduce a considerable amount of complexity. MultiPaxos, one of the oldest and most popular state machine replication protocols, is notoriously difficult to understand, and more modern protocols that employ these sophisticated techniques are *more* complicated. But if we want to squeeze out every drop of performance from our state machine replication protocols, complexity is the price we have to pay, right?

Actually, no. In this paper, we present **a simple and generally applicable technique to increase the throughput of a state machine replication protocol**. Our technique can be applied to many existing state machine replication protocols. In this paper, we apply it to three: MultiPaxos [22], Mencius [29], and S-Paxos [13]. Using our technique we are able to increase MultiPaxos' throughput from 25,000 commands commands per second to 200,000 commands per second without batching (an 8× improvement) and from 200,000 commands per second to 900,000 commands per second with batching (a 4.5× improvement). We achieve similar improvements for Mencius and S-Paxos.

Our technique, which we call **compartmentalization**, revolves around two simple ideas: decoupling and scaling. The insight is that components in existing replication protocols often implement multiple independent functionalities. This makes the protocols difficult to scale. With compartmentalization, we first **decouple** bottleneck components into a number of subcomponents, with each subcomponent implementing a single piece of functionality. We then **scale** up the subcomponents that can benefit from scaling.

Take MultiPaxos as an example. A MultiPaxos leader is traditionally responsible for sequencing commands into a total order *and* for communicating with acceptors to reach consensus on individual commands. Sequencing commands is fundamentally unscalable; it must be done by a single leader. Communicating with acceptors however, is embarrassingly parallel. With compartmentalization, we disentangle the leader's responsibilities and divide the leader in two. Sequencing is performed by a single node, while communicating with acceptors is distributed across multiple nodes.

It is widely believed that simple state machine replication protocols like MultiPaxos have low throughput. "The leader in Paxos is a bottleneck that limits throughput" [29]. "It impairs scalability by placing a disproportionately high load on the master, which must process more messages than the other replicas" [30]. "In practice, [MultiPaxos'] performance is tied to the performance of the leader" [9]. Our technique of compartmentalization shows that this is a myth.

Simple replication protocols *can* achieve as high or higher throughput than more complex protocols that leverage sophisticated techniques such as multi-leader execution, generalization, and fast paths. In fact, these sophisticated techniques can sometimes *hurt* throughput instead of helping it. Complex protocols can be more difficult to decouple and scale, and sophisticated techniques can become a CPU bottleneck. For example, EPaxos [30] replicas and Caesar [9] replicas execute directed graphs of state machine commands in reverse topological order. We find that this graph processing becomes a CPU bottleneck that prevents the protocols from achieving the same throughput as our compartmentalized variant of MultiPaxos.

In summary, we present the following contributions.

- We present a straightforward and generally applicable technique of decoupling and scaling, called compartmentalization, that can increase the throughput of many existing state machine replication protocols.

- We show how to compartmentalize MultiPaxos, Mencius, and S-Paxos. Doing so, we increase MultiPaxos' throughput 8× without batching and 4.5× with batching, and achieve similar speedups for the other two protocols.
- We rebut the widely believed claim that simple state machine replication protocols like MultiPaxos cannot achieve high throughput. We show that our technique enables simple protocols to achieve higher throughput than more complex protocols that were specially designed for high throughput

2 Background

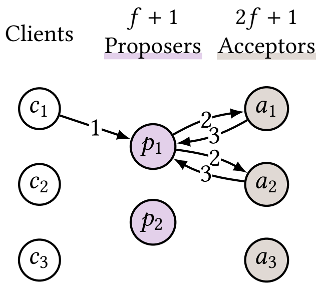
2.1 System Model

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 cannot act maliciously; i.e., we do not consider Byzantine failures. Every protocol discussed in this paper assumes that at most f machines can fail for some parameter f.

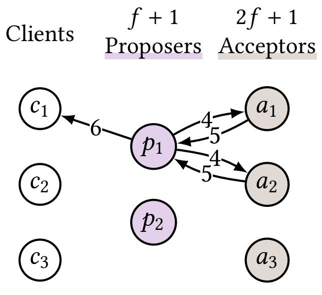
2.2 Paxos

Consensus is the act of choosing a single value among a set of proposed values. **Paxos** [25] is the de facto standard consensus protocol. We assume the reader is familiar with Paxos, but we pause to review the parts that are most important to understand for the rest of this paper.

A Paxos deployment that tolerates f faults consists of $f+1$ **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, p sends Phase 1a messages to at least a majority of the $2f+1$ acceptors. When an acceptor receives a Phase 1a message, it replies with a Phase 1b message. When the leader receives Phase 1b messages from a majority of the acceptors, it begins Phase 2. Phase 1 is illustrated in Figure 1a.



(a) Phase 1



(b) Phase 2

Figure 1. Paxos communication diagram.

Phase 2 mirrors phase 1. The leader sends Phase 2a messages to the acceptors, and the acceptors respond with Phase 2b messages. Upon receiving Phase 2b messages from a majority of the acceptors, the proposed value is considered chosen, and the leader responds to the client informing it of the chosen value. Phase 2 is illustrated in Figure 1b.

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 prefix 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.

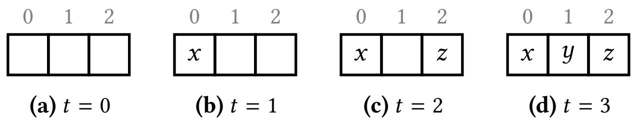


Figure 2. At time $t = 0$, no state machine commands are chosen. At time $t = 1$ command x is chosen in slot 0. At times $t = 2$ and $t = 3$, commands z and y are chosen in slots 2 and 1. All state machines execute commands x, y, z in log order.

MultiPaxos is one of the simplest and 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 commands in the log one slot at a time. A MultiPaxos deployment that tolerates f faults consists of at least $f+1$ proposers and $2f+1$ acceptors (like Paxos) as well as at least $f+1$ replicas, as illustrated in Figure 3. Typically, MultiPaxos is deployed with $2f+1$ servers, with each server hosting a proposer, an acceptor, and a replica.

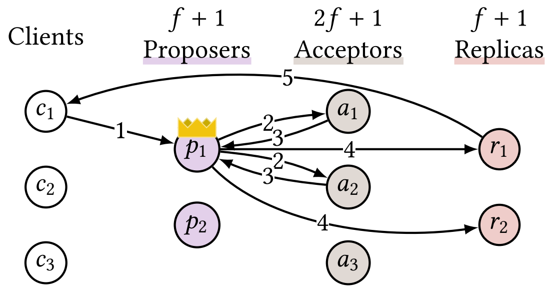


Figure 3. An example execution of MultiPaxos. The leader is adorned with a crown.

Initially, one of the proposers is elected leader and runs Phase 1 of Paxos for every single log entry. When a client wants to propose a state machine command, it sends the command to the leader. The leader assigns the command a log entry i and then runs Phase 2 of Paxos to get the value chosen in entry i. 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. Once the leader learns that a command has been chosen in a given log entry, it informs the replicas. Replicas insert chosen commands into their logs and execute the logs in prefix order.

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. For example, log entries can be round-robin partitioned across the replicas. With n replicas for example, replica r_i returns results for log entries j where $j \bmod n \equiv i$.

3 MultiPaxos

In this section, we show how to increase MultiPaxos' throughput using our technique of compartmentalization. We begin with the basic MultiPaxos protocol that achieves a throughput of 25,000 commands per second and repeatedly apply our technique until we get a protocol capable of processing 200,000 commands per second. We then introduce batching and again decouple and scale to get a protocol that can process 900,000 commands per second. Full experimental details are given in Section 7.

In addition to describing *how* to decouple and scale MultiPaxos, we also describe *why*. In particular, we provide a number of generally applicable heuristics that protocol developers can use to reason about how to best compartmentalize their own protocols.

3.1 Step 1: Proxy Leaders

MultiPaxos' throughput is bottlenecked by the leader. 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 $f + 1$ Phase 2a messages to the acceptors, receive at least $f + 1$ Phase 2b messages from the acceptors, and send at least $f + 1$ messages to the replicas. Thus, the leader sends and receives a total of at least $3f + 4$ messages. 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.

To alleviate this bottleneck, we first **decouple** the leader. To do so, we note that a MultiPaxos leader has two jobs. First, it sequences commands by assigning each command a log entry. Log entry 0, then 1, then 2, and so on. Second, it sends Phase 2a messages, collects Phase 2b 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 chosen commands to the replicas.

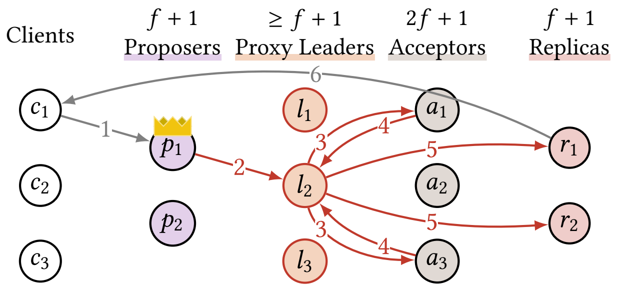


Figure 4. An example execution of MultiPaxos with three proxy leaders. Throughput the paper, nodes and messages that were not present in previous iterations of the protocol are highlighted in red.

More concretely, when a leader receives a command x from a client, it assigns the command x a log entry i and then forms a Phase 2a message that includes x and i. The leader does *not* send the Phase 2a message to the acceptors. Instead, it sends the Phase 2a message to a randomly selected proxy leader. Note 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 Phase 2a message, a proxy leader broadcasts it to the acceptors, gathers a quorum of $f + 1$ Phase 2b responses, and notifies the replicas of the chosen value. 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.

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 question, we **scale**.

Note that the proxy leaders are embarrassingly parallel. They operate independently from one another. Moreover, because the leader distributes load among the proxy leaders equally, 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. [1]

[1]Note that increasing the number of machines does not affect fault tolerance, but it does decrease the expected time to f failures.

Decoupling the leader into separate proxy leaders abides by the following two heuristics.

Heuristic 1. Decouple control flow from data flow.

Heuristic 2. Decouple unscalable nodes from scalable ones.

First, every state machine replication protocol involves control flow (e.g., sequencing commands) and data flow (e.g., broadcasting messages, collecting responses). Decoupling control flow from data flow is a well established technique [17], but it is not often applied to state machine replication protocols.

Second, state machine replication protocols often involve fundamentally unscalable components. For example, the MultiPaxos leader assigns contiguous increasing ids to commands, something that has been proven to be fundamentally difficult to distribute [7, 18]. If we co-locate a scalable component (e.g., proxy leaders) with a centralized component (e.g., a leader), we prevent ourselves from scaling up the components that can benefit from scaling.

3.2 Step 2: Multiple Acceptor Groups

After decoupling and scaling the leader, it is possible that the acceptors are the bottleneck. Surprisingly, we can eliminate the acceptors as a bottleneck by **scaling**. It is very 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" [41]. The argument is that adding more acceptors increases the number of messages that have to be sent and received and that this *hurts* throughput instead of helping it.

While it is true that *naively* adding more acceptors decreases throughput, we can scale the number of acceptors in a meaningful way by taking advantage of the following insight. A single log entry requires a fixed set of acceptors, but different log entries are free to have different sets of acceptors. After two commands have been sequenced by the leader and assigned different log entries, they can be chosen completely independently from one another. The acceptors used to choose the first command can be disjoint from the acceptors used to chose the second. In other words, acceptors cannot be implemented with *intra-command* parallelism but can be implemented with *inter-command* parallelism.

We take advantage of this intuition by introducing multiple **acceptor groups**, with each acceptor group having $2f + 1$ acceptors. This is illustrated in Figure 5.

Log entries are round-robin partitioned among the acceptor groups. Given n acceptor groups, when a proxy leader receives a Phase 2a message for slot s, it contacts acceptor group i where $i \bmod n \equiv s$. Moreover, if the leader fails and a new leader is elected, the new leader runs Phase 1 by contacting all the acceptor groups. Besides this, the protocol remains unchanged.

As with the proxy leaders, acceptor groups are embarrassingly parallel. We can trivially increase the number of acceptor groups until they are no longer the throughput bottleneck.

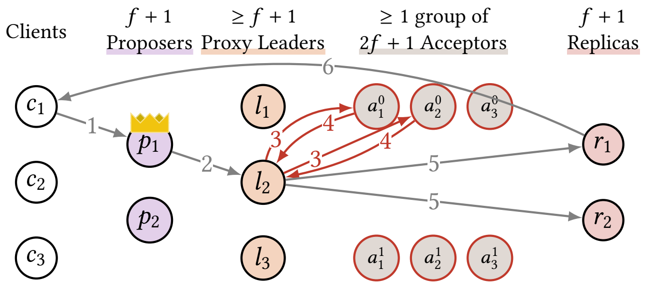


Figure 5. A MultiPaxos execution with two acceptor groups.

Acceptor groups introduce another heuristic.

Heuristic 3. Process independent commands independently.

Recognizing that independent commands can be processed by independent acceptor groups allows us to scale up something that was previously thought to be unscalable. This idea has been applied to replication protocols before (e.g., generalization [23, 38]) but often not to its fullest extent.

3.3 Step 3: Scaling Replicas

After decoupling and scaling the leader and the acceptors, it is possible that the replicas are the bottleneck. Certain aspects of the replicas do not scale. For example, every replica must receive and execute every state machine command. This is unavoidable, and adding more replicas does not reduce this overhead.

However, recall that for every state machine command, 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 increase 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. This is illustrated in Figure 6.

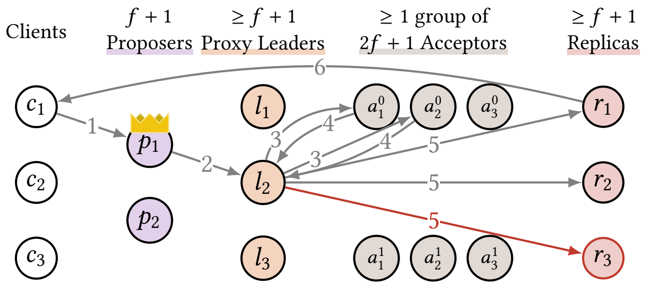


Figure 6. An execution of MultiPaxos with three replicas as opposed to the minimum required two (for $f = 1$).

We call this version of MultiPaxos—with proxy leaders, multiple acceptor groups, and an increased number replicas—**Compartmentalized MultiPaxos**. Compartmentalized MultiPaxos can process 200,000 commands per second, 8× more than MultiPaxos without compartmentalization. Moreover, note that unlike other protocol optimizations (e.g., multi-leader execution, generalization, speculative execution), compartmentalization does not add much complexity to the protocol. The core of the protocol remains unchanged.

3.4 Step 4: Batching

All state machine replication protocols, including MultiPaxos, can take advantage of batching to increase throughput, as illustrated in Figure 7. As is standard [35, 37], the leader collects state machine commands from clients and places them in batches. The rest of the protocol remains relatively unchanged, with batches of commands replacing commands wherever needed. 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.

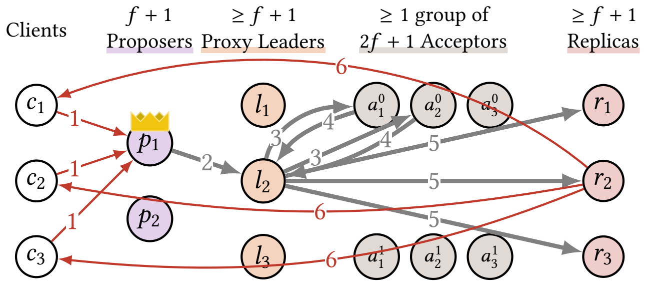


Figure 7. An example execution of MultiPaxos with batching. Messages that contain a batch of commands, rather than a single command, are drawn thicker. Note how replica r_2 has to send multiple messages after executing a batch of commands.

3.5 Step 5: Batchers

Batching increases throughput by amortizing the communication and computation cost of processing a command. Take the proxy leaders for example. Without batching, a proxy leader has to process a total of at least $3f+4$ messages *per command*. With batching, however, a proxy leader has to process $3f+4$ messages *per batch*. The communication overhead is linear in the number of batches, rather than the number of commands. With batches of size 100, for example, the number of messages processed per command decreases by a factor of 100.

Ensuring that communication and computation costs increase linearly with the number of batches rather than the number of commands is essential to maximize the throughput gains that batching affords. However, refer again to Figure 7 and note that the leader does not quite achieve this. To process a single batch of n commands, the leader has to receive n messages and send one message. Its communication cost is linear in the number of commands rather than the number of batches. Ideally, it would only have to receive one message and send one message. This makes the leader a potential throughput bottleneck.

To remove the bottleneck, we apply Heuristic 1 and Heuristic 2 and **decouple** the leader. The leader has two responsibilities: forming batches and sequencing batches. We decouple the two responsibilities by introducing a set of at least $f+1$ **batchers**, as illustrated in Figure 8. The batchers are responsible for receiving commands from clients and forming batches, while the leader is responsible for sequencing batches.

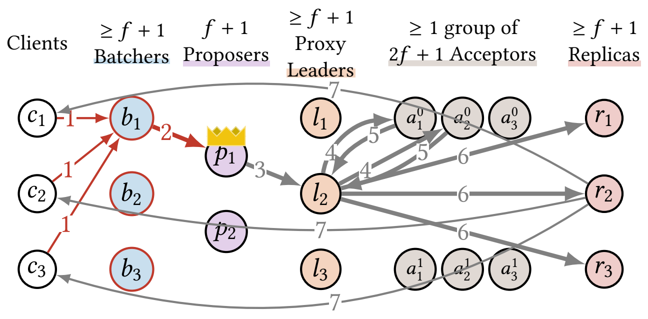


Figure 8. An execution of MultiPaxos with batchers.

More concretely, when a client wants to propose a state machine command, it sends the command to a randomly selected batcher. 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. When the leader receives a batch of commands, it assigns it a log entry, forms a Phase 2a message, and sends the Phase 2a message to a proxy leader. 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. Moreover, as with proxy leaders and acceptor groups, we can **scale** the number of batchers until they are not a throughput bottleneck.

3.6 Step 6: Unbatchers

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.

To solve this, we again apply our heuristics and **decouple** the replicas. We introduce a set of at least $f + 1$ **unbatchers**, as illustrated in Figure 9. The replicas are responsible for executing 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. Upon receiving a result batch, an unbatcher sends the results back to the clients. Also note that we can again **scale** the unbatchers until they are not a throughput bottleneck.

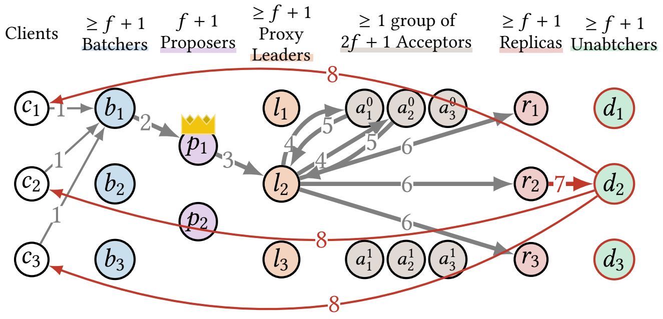


Figure 9. An execution of MultiPaxos with proxy replicas.

Note that batchers and unbatchers are duals. Batchers collect state machine commands into batches and send them into the protocol. Unbatchers receive result batches from the protocol and split them up into individual results. Together, they ensure the protocol can take full advantage of batching.

With batching, batchers, and unbatchers, Compartmentalized MultiPaxos is able to process 900,000 commands per second.

4 Mencius

In this section, we show how to compartmentalize a more complex protocol, Mencius [29].

4.1 Background

As discussed previously, the MultiPaxos leader (without decoupling and scaling) is a throughput bottleneck because all commands go through the leader and because the leader performs disproportionately more work per command than the acceptors or replicas. Mencius is a MultiPaxos variant that attempts to eliminate this bottleneck by using more than one leader.

Rather than having a single leader sequence all commands in the log, Mencius round-robin partitions the log among multiple leaders. For example, consider the scenario with three leaders l_1, l_2, and l_3 illustrated in Figure 10. Leader l_1 gets commands chosen in slots 0, 3, 6, etc.; leader l_2 gets commands chosen in slots 1, 4, 7, etc.; and leader l_3 gets commands chosen in slots 2, 5, 8, etc.

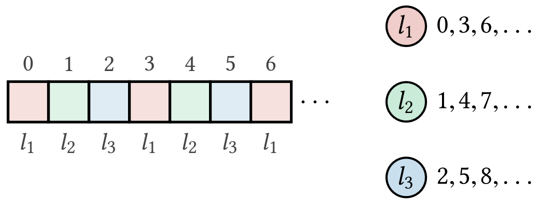


Figure 10. A Mencius log round robin partitioned among three leaders.

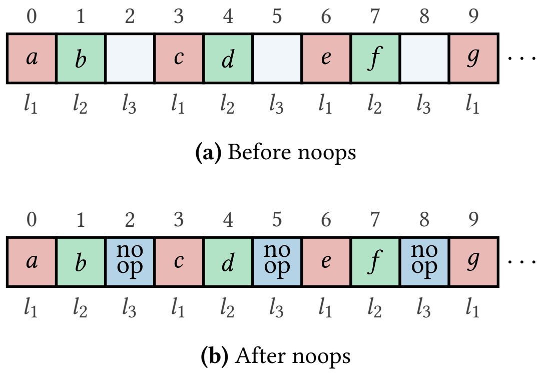


Figure 11. An example of using noops to deal with a slow leader. Leader l_3 is slower than leaders l_1 and l_2, so the log has holes in l_3's slots. l_3 fills its holes with noops to allow commands in the log to be executed.

Having multiple leaders works well when all the leaders process commands at the exact same rate. However, if one of the leaders is slower than the others, then holes start appearing in the log entries owned by the slow leader. This is illustrated in Figure 11a. Figure 11a depicts a Mencius log partitioned across three leaders. Leaders l_1 and l_2 have both gotten a few commands chosen (e.g., a in slot 0, b in slot 1, etc.), but leader l_3 is lagging behind and has not gotten any commands chosen yet. Replicas execute commands in log order, so they are unable to execute all of the chosen commands until l_3 gets commands chosen in its vacant log entries.

If a leader detects that it is lagging behind, then it fills its vacant log entries with a sequence of noops. A **noop** is a distinguished command that does not affect the state of the replicated state machine. In Figure 11b, we see that l_3 fills its vacant log entries with noops. This allows the replicas to execute all of the chosen commands.

More concretely, a Mencius deployment that tolerates f faults is implemented with $2f + 1$ **servers**, as illustrated in Figure 12. Roughly speaking, every Mencius server plays the role of a MultiPaxos leader, acceptor, and replica.

When a client wants to propose a state machine command x, it sends x to any of the servers. Upon receiving command x, a server s_l plays the role of a leader. It assigns the command x a slot i and sends a Phase 2a message to the other servers

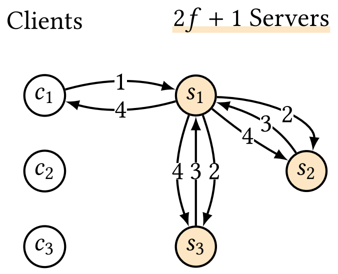


Figure 12. An example execution of Mencius.

that includes x and i. Upon receiving a Phase 2a message, a server s_a plays the role of an acceptor and replies with a Phase 2b message.

In addition, s_a uses i to determine if it is lagging behind s_l. If it is, then it sends a SKIP message along with the Phase 2b message. The SKIP message informs the other servers to choose a noop in every slot owned by s_a up to slot i. For example, if a server s_a's next available slot is slot 10 and it receives a Phase 2a message for slot 100, then it broadcasts a SKIP message informing the other servers to place noops in all of the slots between slots 10 and 100 that are owned by server s_a. Mencius leverages a protocol called Coordinated Paxos to ensure noops are chosen correctly. We refer to the reader to [29] for details.

Upon receiving Phase 2b messages for command x from a majority of the servers, server s_l deems the command x chosen. It informs the other servers that the command has been chosen and also sends the result of executing x back to the client.

4.2 Compartmentalization

Mencius uses multiple leaders to avoid being bottlenecked by a single leader. However, despite this, Mencius still does not achieve optimal throughput. Part of the problem is that every Mencius server plays three roles, that of a leader, an acceptor, and a replica. Because of this, a server has to send and receive a total of roughly $3f + 5$ messages for every command that it leads *and also* has to send and receive messages acking other servers as they simultaneously choose commands.

We can solve this problem by **decoupling** the servers. Instead of deploying a set of heavily loaded servers, we instead view Mencius as a MultiPaxos variant and deploy it as a set of proposers, a set of acceptors, and set of replicas. This is illustrated in Figure 13.

Now, Mencius is equivalent to MultiPaxos with the following key differences. First, every proposer is a leader, with the log round-robin partitioned among all the proposers. If a client wants to propose a command, it can send it to any of the proposers. Second, the proposers periodically broadcast their next available slots to one another. Every server uses this information to gauge whether it is lagging behind. If it is, it chooses noops in its vacant slots, as described above.

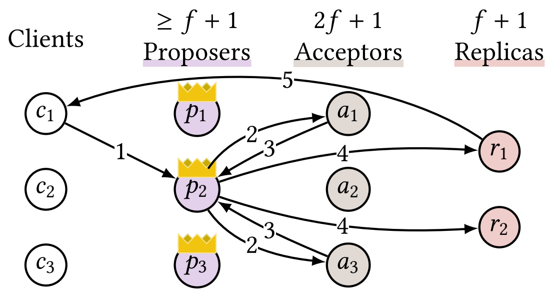


Figure 13. An example execution of decoupled Mencius. Note that every proposer is a leader.

This decoupled Mencius is a step in the right direction, but it shares many of the problems that MultiPaxos faced. The proposers are responsible for both sequencing commands and for coordinating with acceptors; we have a single unscalable group of acceptors; and we are deploying too few replicas. Thankfully, we can apply our heuristics and compartmentalize Mencius in exactly the same way as MultiPaxos by leveraging proxy leaders, multiple acceptor groups, and more replicas. This is illustrated in Figure 14.

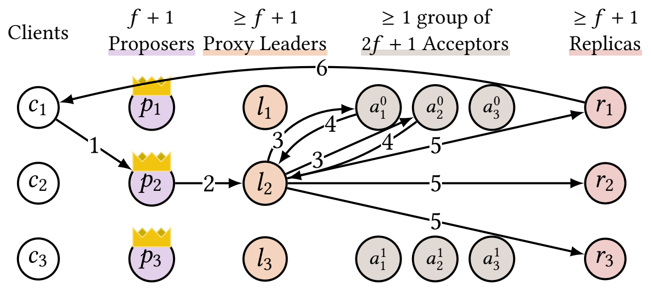


Figure 14. An execution of Mencius with proxy leaders, multiple acceptor groups, and an increased number of replicas.

This protocol, called Compartmentalized Mencius, shares all of the advantages of Compartmentalized MultiPaxos. Proxy leaders and acceptors both trivially scale so are not bottlenecks, while leaders and replicas have been pared down to their essential responsibilities of sequencing and executing commands respectively. Moreover, because Mencius allows us to deploy multiple leaders, we can also increase the number of leaders until they are no longer a bottleneck. To support batching, we can also introduce batchers and unbatchers like we did with MultiPaxos.

Without compartmentalization, Mencius can process 30,000 commands per second without batching and 200,000 with batching. Compartmentalized Mencius can process 250,000 commands per second without batching and 850,000 commands per second with batching.

5 S-Paxos

In this section, we show how to compartmentalize S-Paxos [13], a MultiPaxos variant that decouples command dissemination from command ordering.

5.1 Background

S-Paxos is a MultiPaxos variant that, like Mencius, aims to avoid being bottlenecked by a single leader. Recall that when a MultiPaxos leader receives a state machine command x from a client, it broadcasts a Phase 2a message to the acceptors that includes the command x. If the leader receives a state machine command that is large (in terms of bytes) or receives a large batch of modestly sized commands, the overheads of disseminating the commands begin to dominate the cost of the protocol, exacerbating the fact that command disseminating is performed solely by the leader.

S-Paxos avoids this by applying Heuristic 1. It decouples command dissemination from command sequencing—separating control from from data flow—and distributes command dissemination across all nodes. More concretely, an S-Paxos deployment that tolerates f faults consists of $2f + 1$ servers, as illustrated in Figure 15. Every server plays the role of a MultiPaxos proposer, acceptor, and replica. It also plays the role of a **disseminator** and **stabilizer**, two roles that will become clear momentarily.

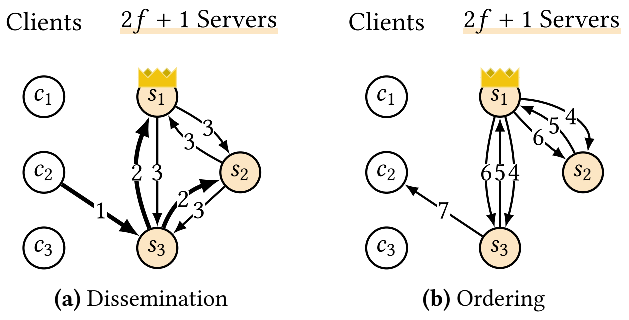


Figure 15. An example execution of S-Paxos. Messages that include client commands (as opposed to ids) are bolded.

When a client wants to propose a state machine command x, it sends x to any of the servers. Upon receiving a command from a client, a server plays the part of a disseminator. It assigns the command a globally unique id id_x and begins a **dissemination phase** with the goal of persisting the command and its id on at least a majority of the servers. This is shown in Figure 15a. The server broadcasts x and id_x to the other servers. Upon receiving x and id_x, a server plays the role of a stabilizer and stores the pair in memory. It then broadcasts an acknowledgement to all servers. The acknowledgement contains id_x but not x.

One of the servers is the MultiPaxos leader. Upon receiving acknowledgements for id_x from a majority of the servers, the leader knows the command is stable. It then uses the id id_x as a proxy for the corresponding command x and runs the MultiPaxos protocol as usual (i.e. broadcasting Phase 2a messages, receiving Phase 2b messages, and notifying the other servers when a command id has been chosen) as shown in Figure 15b. Thus, while MultiPaxos agrees on a log of *commands*, S-Paxos agrees on a log of *command ids*.

The S-Paxos leader, like the MultiPaxos leader, is responsible for ordering command ids and getting them chosen. But, the responsibility of disseminating commands is shared by all the servers.

5.2 Compartmentalization

We compartmentalize S-Paxos similar to how we compartmentalize MultiPaxos and Mencius. First, we **decouple** servers into a set of at least $f + 1$ disseminators, a set of $2f + 1$ stabilizers, a set of proposers, a set of acceptors, and a set of replicas. This is illustrated in Figure 16. To propose a command x, a client sends it to any of the disseminators. Upon receiving x, a disseminator persists the command and its id id_x on at least a majority of (and typically all of) the stabilizers. It then forwards the id to the leader. The leader gets the id chosen in a particular log entry and informs one of the stabilizers. Upon receiving id_x from the leader, the stabilizer fetches x from the other stabilizers if it has not previously received it. The stabilizer then informs the replicas that x has been chosen. Replicas execute commands in prefix order and reply to clients as usual.

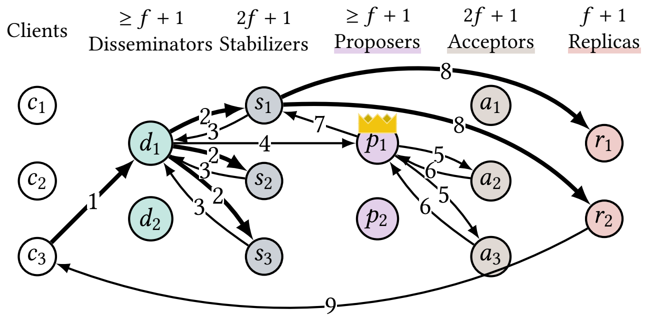


Figure 16. An example execution of decoupled S-Paxos. Messages that include client commands (as opposed to ids) are bolded. Note that the MultiPaxos leader does not send or receive any messages that include a command, only messages that include command ids.

Though S-Paxos relieves the MultiPaxos leader of its duty to broadcast commands, the leader still has to broadcast command ids. In other words, the leader is no longer a bottleneck on the data path but is still a bottleneck on the control path. Moreover, disseminators and stabilizers are potential bottlenecks. We can resolve these issues by compartmentalizing S-Paxos similar to how we compartmentalized MultiPaxos. We introduce proxy leaders, multiple acceptor groups, and more replicas. Moreover, we can trivially scale up the number of disseminators and can deploy multiple disseminator groups similar to how we deploy multiple acceptor groups. This is illustrated in Figure 17. To support batching, we can again introduce batchers and unbatchers.

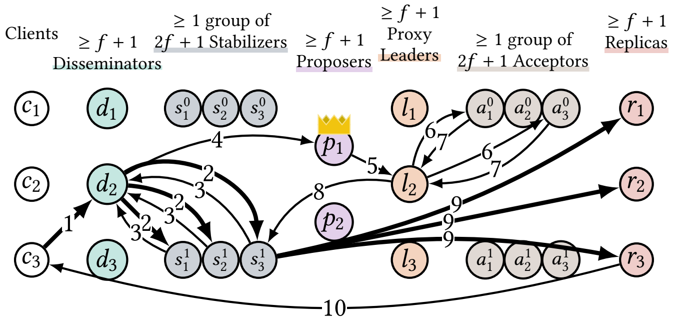


Figure 17. An example execution of S-Paxos with multiple stabilizer groups, proxy leaders, multiple acceptor groups, and an increased number of replicas. Messages that include client commands (as opposed to ids) are bolded.

6 Discussion

In this section, we clarify the novelty of the paper and share some insights on protocol complexity.

6.1 Novelty

Decoupling and scaling are classic systems techniques, and the heuristics described in this paper are already well known. We do not claim to have invented these. We are not even the first to apply these techniques or heuristics to state machine replication protocols. Both Mencius and S-Paxos, for example, use decoupling to try and prevent the leader from becoming a throughput bottleneck.

The novelty of this paper is the depth and breadth with which we investigate compartmentalizing state machine replication protocols. In terms of **depth**, we propose novel ways to decouple and scale replication protocols. We introduce proxy leaders to pare down the leader. We use multiple acceptor groups to debunk the myth that acceptors do not scale. We show how to take full advantage of batching by using batchers and unbatchers. While existing protocols, like Mencius and S-Paxos, perform some forms of decoupling and scaling, by applying our techniques, we increase their throughput by as much as 8×.

In terms of **breadth**, we demonstrate how widely applicable decoupling and scaling are. In this paper, we focus on three state machine replication protocols—MultiPaxos, Mencius, and S-Paxos—but many other state machine replication protocols can be compartmentalized as well. Protocols like Viewstamped Replication [27], Raft [32], Flexible Paxos [19], and DPaxos [31] that are very similar to MultiPaxos, for example, can leverage the techniques described in this paper without much modification. We can also apply the techniques to more sophisticated protocols like Generalized Paxos [23], EPaxos [30], and Caesar [9]. Bipartisan Paxos [8], for example, is a partially compartmentalized version of EPaxos.

6.2 The Curse of Complexity

The depth and breadth of our investigation also debunks the commonly held belief that simple state machine replication protocols like MultiPaxos cannot achieve high throughput. Our implementation of Compartmentalized MultiPaxos achieves a throughput of 200,000 commands per second without batching. This is roughly 4× higher than the reported throughput of EPaxos and Caesar, two protocols that aim to increase the throughput of MultiPaxos by using multiple leaders, fast paths, and generalized execution [9, 30].

Ironically, these sophisticated techniques that aim to eliminate throughput bottlenecks can actually *become* the throughput bottleneck! Take EPaxos as an example. EPaxos can be partially compartmentalized to increase its throughput. This was done in [8], where BPaxos, a compartmentalized variant of EPaxos, achieves a throughput twice that of EPaxos. However, EPaxos cannot easily achieve higher throughput because the techniques it uses become CPU bottlenecks.

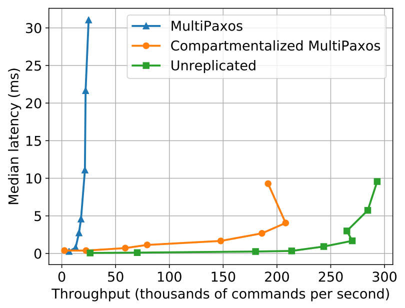
For example, an EPaxos replica executes a directed graph of commands in reverse topological order, one strongly connected component at a time. This requires replicas to execute a computationally expensive graph algorithm (e.g. Tarjan's algorithm). Contrast this with a MultiPaxos replica which instead executes commands by scanning over a log (read array) of commands. Empirically, this graph execution was the throughput bottleneck for the BPaxos implementation presented in [8]. Moreover, while it is not *impossible* to scale this graph traversal using some sort of distributed graph algorithm, it is hard to imagine that it could be optimized enough to outperform the simple scan of a contiguous array that MultiPaxos replicas perform.

7 Evaluation

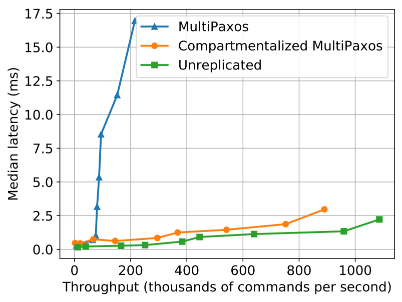
7.1 MultiPaxos Latency-Throughput

Experiment Description We implemented MultiPaxos, Compartmentalized MultiPaxos, and an unreplicated state machine in Scala using the Netty networking library. MultiPaxos employs $2f + 1$ machines with each machine playing the role of a MultiPaxos proposer, acceptor, and replica. Compartmentalized MultiPaxos is the fully compartmentalized protocol described in Section 3.

The unreplicated state machine is implemented on a single process on a single server. Clients send commands directly to the state machine. Upon receiving a command, the state machine executes it immediately and 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



(a) Without batching



(b) With batching

Protocol	Batched	Number of clients		
		1	10	100
▲ Multi…	no	0.27	0.78	4.58
● Comp…	no	0.38	0.38	1.13
■ Unrepl…	no	0.07	0.11	0.34
▲ Multi…	yes	0.29	0.39	1.03
● Comp…	yes	0.47	0.44	0.62
■ Unrepl…	yes	0.16	0.21	0.32

(c) Median latency (ms) from (a) and (b)

Figure 18. The latency and throughput of MultiPaxos, Compartmentalized MultiPaxos, and an unreplicated state machine with and without batching for 1, 10, 50, 100, 300, 600, 1000, 2000, and 4000 clients.

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 1, 10, 50, 100, 300, 600, 1000, 2000, and 4000 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. Note that multiple clients are run within a single process, so the number of physical client processes can be significantly less than the number of logical clients.

We note the following details about our experiment.

- We deploy the protocols with and without batching. For a given number of clients, the batch size is set empirically to optimize throughput.
- To fairly compare the unreplicated state machine with Compartmentalized MultiPaxos, we deploy the unreplicated state machine with a set of batchers and unbatchers (when batching is enabled).
- We deploy all protocols with $f = 1$. We deploy Compartmentalized MultiPaxos with two proposers, three acceptor groups, and five replicas. Without batching, Compartmentalized MultiPaxos uses 30 proxy leaders. With batching, it uses between 2 and 8 batchers, 23 proxy leaders and 10 unbatchers. We deploy the unreplicated state machine with 10 batchers and 10 unbatchers. Note that Compartmentalized MultiPaxos uses more machines than MultiPaxos because it has been scaled up.
- All three protocols use a "noop" state machine. State machine commands are zero bytes, executing a command is a noop, and the result of executing a command is also zero bytes. This allows us to study the effects of compartmentalization in isolation. As command execution time increases, the effects of compartmentalization (and any other protocol optimization) become less and less pronounced. In the limit, if an application is bottlenecked by command execution, then protocol optimizations are moot.
- We deploy the three protocols on AWS using a set of m5.2xlarge 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 [30].

Results The results of the experiment are shown in Figure 18. In Figure 18a, we see the median latency and throughput of the three protocols without batching. MultiPaxos is able to process at most 25,000 commands per second. Compartmentalized MultiPaxos is able to process roughly 200,000 commands per second, an 8× throughput improvement. Moreover, at peak throughput, MultiPaxos' median latency is three times that of Compartmentalized MultiPaxos.

The unreplicated state machine outperforms both protocols. It achieves a peak throughput of roughly 300,000 commands per second with latencies similar to Compartmentalized MultiPaxos. Compartmentalized MultiPaxos underperforms the unreplicated state machine because—despite decoupling the leader as much as possible—the single leader remains a throughput bottleneck.

The median latency and throughput of the three protocols with batching is shown in Figure 18b. With 4,000 clients, MultiPaxos, Compartmentalized MultiPaxos, and the unreplicated state machine achieve 200,000, 900,000, and 1,100,000 commands per second respectively. With batching, Compartmentalized MultiPaxos comes closer to matching the throughput of the unreplicated state machine since batching amortizes the overheads of the leader.

Figure 18c shows the median latency of the three protocols when subjected to load generated by 1, 10, and 100 clients. We refer to the number of network delays that a client must wait between proposing a command x and receiving the result of executing x as the **commit delay**. Referring to Figure 6, we see that Compartmentalized MultiPaxos has a commit delay of six, while MultiPaxos has a commit delay of only four. With only one client, this smaller commit delay translates directly to lower latency. MultiPaxos achieves a median latency of 0.27 milliseconds compared to Compartmentalized MultiPaxos' 0.38 milliseconds. However, with fast networks and moderate to heavy loads, queueing times (rather than network traversals) become the determining factor of commit latency. With as few as 10 or 100 clients, Compartmentalized MultiPaxos is able to achieve lower latency than MultiPaxos. We note though that this result is particular to our deployment within a single data center. For geo-replicated protocols deployed on the WAN, commit delay is the determining factor of commit latency. Compartmentalized protocols are not a good fit for this scenario.

Also note that the primary goal of this experiment is to measure the relative speedups that compartmentalization provides. While the absolute performance numbers that we achieve are important, they are not our primary focus. For example, this experiment is not meant to demonstrate that Compartmentalized MultiPaxos is the world's fastest state machine replication protocol (though it is fast). Rather, it is designed to demonstrate that compartmentalization yields significant speedups.

7.2 Mencius Latency-Throughput

We repeat the experiment above but with Mencius and Compartmentalized Mencius. Compartmentalized Mencius uses the same number of machines as Compartmentalized MultiPaxos, except it uses three proposers instead of two.

The results are shown in Figure 19. Without batching, Mencius can process roughly 30,000 commands per second. Compartmentalized Mencius can process roughly 250,000 commands per second, an 8.3× improvement. Compartmentalized Mencius outperforms Compartmentalized MultiPaxos and comes close to matching the performance of the unreplicated state machine by avoiding the single leader bottleneck. With batching, Mencius and Compartmentalized Mencius achieve peak throughputs of nearly 200,000 and 850,000 commands per second respectively, a 4.25× improvement. The latencies reported in Figure 18c confirm that Compartmentalized Mencius has higher latency than Mencius under low load but lower latency under moderate to heavy load.

7.3 S-Paxos Latency-Throughput

We repeat the experiments above with S-Paxos and Compartmentalized S-Paxos. Without batching, Compartmentalized S-Paxos achieves a peak throughput of 180,000 commands per second compared to S-Paxos' throughput of 22,000 (an 8.2× improvement). With batching, Compartmentalized S-Paxos achieves a peak throughput of 750,000 commands per second compared to S-Paxos' throughput of 180,000 (a 4.16× improvement). Note that our implementation of S-Paxos is not as optimized as our other two implementations, so its absolute performance is lower. As noted above though, demonstrating absolute performance is a secondary goal to demonstrating relative speedups.

7.4 MultiPaxos Ablation Study

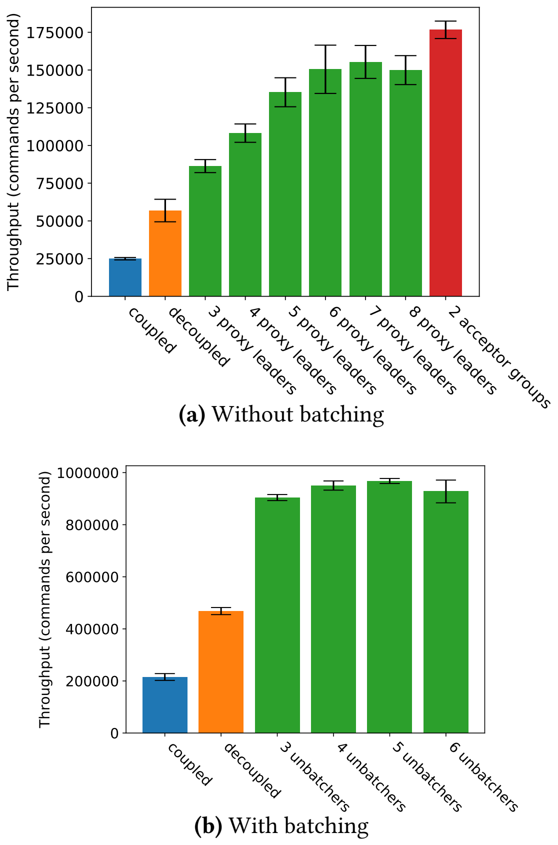
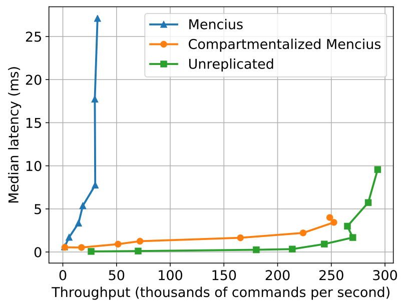


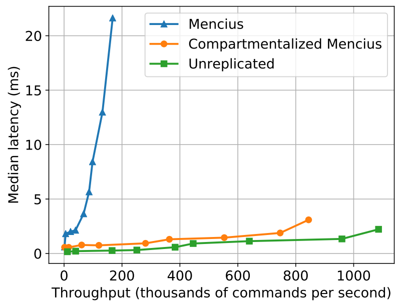
Figure 20. An ablation study. Standard deviations are shown using error bars.

Experiment Description The previous experiments confirm that compartmentalization can increase the throughput of a state machine protocol by as much as 8.3× without batching and by as much as 4.5× with batching. We now perform an ablation study to pinpoint where these throughput improvements come from. In particular, we begin with MultiPaxos (both batched and unbatched) and repeatedly perform the optimizations described in Section 3, reporting the peak throughput of the protocol after each optimization when subjected to load generated by 1000 clients (unbatched) or 4000 clients (batched). All the details of this experiment are the same as the previous experiments unless otherwise noted.

Results The unbatched ablation study results are shown in Figure 20a. MultiPaxos achieves a throughput of roughly 25,000 commands per second. If we decouple the protocol—separating the proposers, acceptors, and replicas—and introduce proxy leaders, we achieve a throughput of roughly 55,000 commands per second. This decoupled MultiPaxos uses the bare minimum number of proposers (2), proxy leaders (2), acceptors (3), and replicas (2). As we scale up the



(a) Without batching



(b) With batching

Protocol	Batched	Number of clients 1	10	100
▲ Mencius	no	0.59	1.69	5.36
● Comp...	no	0.54	0.52	0.92
■ Unrepl...	no	0.07	0.11	0.34
▲ Mencius	yes	0.59	1.81	2.13
● Comp...	yes	0.57	0.57	0.75
■ Unrepl...	yes	0.16	0.21	0.32

(c) Median latency (ms) from (a) and (b)

Figure 19. The latency and throughput of Mencius, Compartmentalized Mencius, and an unreplicated state machine with and without batching for 1, 10, 50, 100, 300, 600, 1000, 2000, and 4000 clients.

number of proxy leaders from 2 to 8, the throughput of the protocol increases until it plateaus at roughly 150,000 commands per second. If we then introduce an additional acceptor group, the throughput jumps to roughly 175,000 commands per second. In this experiment, the two replicas are not a bottleneck, so increasing the number of replicas does not increase the throughput. However, if state machine results were larger in size, the communication overheads of replying to clients would increase, and the protocol would benefit from more replicas.

The batched ablation study results are shown in Figure 20a. Decoupling MultiPaxos and introducing two batchers and two unbatchers increases the throughput of the protocol from 200,000 commands per second to 500,000 commands per second. Increasing the number of unbatchers increases the throughput again to a plateau of roughly 900,000 commands per second. For this experiment, two batchers are sufficient to handle the clients' load. With more clients and a larger load, more batchers would be needed to maximize throughput.

8 Related Work

Bipartisan Paxos [8] presents Bipartisan Paxos (BPaxos), a compartmentalized version of EPaxos with double the throughput and with a simpler design. [8] also discusses decoupling and scaling, but it focuses solely on applying the techniques to EPaxos. This paper demonstrates how to apply the techniques to a wider range of protocols.

Scalable Agreement In [21], Kapritsos et al. present a protocol similar to our Compartmentalized Mencius. 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 by decoupling leaders and acceptors, introducing proxy leaders, introducing the possibility of having multiple acceptor groups per cluster, and describing how to implement batching efficiently.

NoPaxos NoPaxos [26] is a Viewstamped Replication [27] variant that depends on an ordered unreliable multicast (OUM) layer. Each client sends commands to a centralized sequencer that is implemented on a network switch. The sequencer assigns increasing ids to the commands and broadcasts them to a set of replicas. The replicas speculatively execute commands and reply to clients. In this paper, we describe how to use proxy leaders to avoid having a centralized leader. NoPaxos' on-switch sequencer is a hardware based alternative to avoid the bottleneck.

A Family of Leaderless Generalized Protocols In [28], Losa et al. propose a template that can be used to implement state machine replication protocols that are both leaderless and generalized. The goal of this modularization is to unify existing protocols like EPaxos [30], and Caesar [9]. However, the modularity also introduces decoupling which can lead to performance gains. This is realized by BPaxos [8].

Multithreaded Replication [36] and [11] both propose multithreaded state machine replication protocols. The protocol in [36] is implemented using a combination of actors and the SEDA architecture [40]. [11] argues for a Mencius-like approach in which each thread has complete functionality (receiving, sequencing, and sending), with slots round-robin partitioned across threads. 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.

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. However, if we are able to divide the state of the state machine into independent shards, then we can further scale the protocols. For example, in [12], Bezerra et al. discuss how state machine replication protocols can take advantage of sharding.

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.]. CockroachDB Replication Layer. https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html. Accessed: 2019-10-21.

[3] [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.

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

[5] [n.d.]. Neo4j Enterprise Cluster Basics. https://www.graphgrid.com/neo4j-enterprise-cluster-basics/. Accessed: 2019-10-21.

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

[7] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach.. In *CIDR*. Citeseer, 249–260.

[8] Anonymous. Under review. Bipartisan Paxos: A Modular State Machine Replication Protocol. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*.

[9] 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.

[10] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, Vol. 11. 223–234.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. (2003).

[18] Joseph M Hellerstein and Peter Alvaro. 2019. Keeping CALM: When Distributed Consistency is Easy. *arXiv preprint arXiv:1901.01930* (2019).

[19] 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

[20] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8. Boston, MA, USA.

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

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

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

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

[25] Leslie Lamport. 2006. Lower bounds for asynchronous consensus. *Distributed Computing* 19, 2 (2006), 104–125.

[26] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 467–483.

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

[28] Giuliano Losa, Sebastiano Peluso, and Binoy Ravindran. 2016. Brief announcement: A family of leaderless generalized-consensus algorithms. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. ACM, 345–347.

[29] 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.

[30] 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.

[31] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1221–1236.

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

[33] Seo Jin Park and John Ousterhout. 2019. Exploiting commutativity for practical fast replication. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 47–64.

[34] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks.. In *NSDI*. 43–57.

[35] 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.

[36] 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.

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

[38] Pierre Sutra and Marc Shapiro. 2011. Fast genuine generalized consensus. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE, 255–264.

[39] 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.

[40] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 230–243.

[41] 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.