SDPaxos: Building Efficient Semi-Decentralized Geo-replicated State Machines

Hanyu Zhao
Peking University
zhao.hanyu@pku.edu.cn

Quanlu Zhang
Microsoft Research
Quanlu.Zhang@microsoft.com

Zhi Yang
Peking University
yangzhi@pku.edu.cn

Ming Wu
Microsoft Research
miw@microsoft.com

Yafei Dai
Shenzhen Key Lab for Information Technology, School of Electronics and Computer Engineering, Peking University
dyf@pku.edu.cn

ABSTRACT

Existing state machine replication protocols are confronting two major challenges in geo-replication: (1) limited performance caused by load imbalance, and (2) severe performance degradation in heterogeneous environments or under high-contention workloads. This paper presents a new semi-decentralized approach to addressing both the challenges at the same time. Our protocol, SDPaxos, divides the task of a replication protocol into two parts: durably replicating each command across replicas without global order, and ordering all commands to enforce the consistency guarantee. We decentralize the process of replicating commands, which accounts for the largest proportion of load, to provide high performance. In contrast, we centralize the process of ordering commands, which is lightweight but needs a global view, to better stability against heterogeneity or contention. The key novelty lies in that SDPaxos achieves the optimal one-round-trip latency under realistic configurations, despite the two separated steps, replicating and ordering, which are both based on Paxos. We also design a recovery protocol to do rapid failover under failures, and a series of optimizations to boost performance. We show via a prototype implementation the significant advantage of SDPaxos on both throughput and latency, facing different environments and workloads.

CCS CONCEPTS

• Computer systems organization → Availability; Reliability; Software and its engineering → Distributed systems organizing principles;

1 INTRODUCTION

Today’s applications commonly use replication to tolerate server failures and provide highly available services [12, 21]. The replication servers (i.e., the replicas) coordinate via state machine replication protocols like Paxos [26] to make sure they execute the same sequence of commands requested by clients in the same order, hence they can keep in a consistent state.

Meanwhile, the rapid growth of geo-distributed applications [15, 20] has placed two critical demands on replication protocols: high throughput and low (wide-area) latency. Traditional single-leader protocols [27, 38, 39] employ a centralized leader to process all client requests and propose commands (i.e., add commands into the command sequence). This design seriously limits throughput as the leader carries significantly higher load. Moreover, it brings clients substantial latency penalty for sending requests to a remote leader not co-located with the client. In the light of this limitation, a multi-leader protocol would be preferred in geo-replication.

However, existing multi-leader protocols can exhibit poor performance in some undesired but common cases. A typical approach is rotating the leadership among replicas [34], but it makes the system running at the speed of the slowest one (usually called the straggler), which severely degrades performance in heterogeneous environments. Heterogeneity is a fact of life in virtualized data centers such as Amazon EC2 [2], where uncontrollable performance variations caused by co-location of VMs are very common [30, 45]. Heterogeneity also exists in private data centers due to factors like multiple hardware generations, varying network conditions or competition for resources [9, 16, 22, 42].
Another approach is dynamically designating leaders [35], by assuming that most of the commands do not interfere, i.e., will not result in inconsistent state even executed in different orders. If commands interfere, it needs higher computational cost and more round trips of wide-area communication to serialize them, leading to low throughput and high latency. However, services like E-commerce and social network can generate high-contention workload, with many interfering commands on the same object from multiple clients [5, 6, 11, 41]. This problem can be even worse in the wide area: as requests take much longer time to finish, the probability of contention also rises [37].

The root of the inefficiency of existing multi-leader protocols lies in their purely decentralized coordination pattern. Although this decentralization addresses the single-leader bottleneck as every replica can propose commands, replicas still need to agree on a total order on all commands proposed by different replicas to avoid inconsistent state. We show that this process can bring significant overhead without a global view under such decentralization, in heterogeneous environment or under high contention (§2).

Based on this key insight, we propose SDPaxos, a semi-decentralized protocol providing high performance, and strong performance stability against heterogeneity and contention simultaneously. Specifically, we divide the functionality of a replication protocol into two parts: one for durably replicating commands without deciding their execution order, and the other for ordering these commands to generate an execution sequence. SDPaxos, replicating is completely decentralized where every replica can freely propose commands and replicate them to other replicas, which evenly distributes the load among replicas, and enables clients to always contact the nearest one. Meanwhile, we employ one replica to handle ordering in a centralized manner: this global view enables this replica to always order commands appropriately, which eliminates the penalty introduced by heterogeneity and contention.

The idea of using a centralized node to order requests or messages in distributed systems is not original. However, SDPaxos is novel as it, to the best of our knowledge, the first work to apply this pattern to multi-leader state machine replication both correctly and efficiently. That is, SDPaxos not only handles replicating and ordering both based on Paxos for correctness, but also achieves the optimal one-round-trip latency under typical configurations tolerating one or two failures. SDPaxos realizes this by efficiently overlapping replicating and ordering, and designing a fault-tolerance approach that extends vanilla Paxos, to guarantee consistency and linearizability.

In addition, we also fully optimize the centralized ordering to achieve the best performance. We leverage the lightweight nature of ordering, and further implement several performance optimizations for best load-balance. Moreover, we take advantage of the centralized ordering to optimize read operations, in which replicas can directly acquire the latest version of data with no Paxos overhead.

We implemented a prototype of SDPaxos, and compared its performance with typical single-leader (Multi-Paxos [27]) and multi-leader (Mencius [34], EPaxos [35]) protocols. Our experiment results demonstrate that SDPaxos achieves: (1) 1.6× the throughput of Mencius with a straggler, (2) stable performance under different contention degrees and 1.7× the throughput of EPaxos even with a low contention rate of 5%, (3) 6.1× the throughput of Multi-Paxos without straggler or contention, (4) 4.6× the throughput of writes when performing reads, and (5) up to 61% and 99% lower wide-area latency of writes and reads than other protocols.

The key contributions of this paper are summarized as follows.

- We resolve the problem of degraded performance of replication protocols under realistic and complex environments, e.g., heterogeneity and contention, by the semi-decentralization in state machine replication.
- We design SDPaxos, a protocol realizing the semi-decentralization and achieving the one-round-trip latency under typical three/five-replica settings. It efficiently overlaps replicating and ordering with consistency and linearizability guaranteed. It also greatly improves read performance with the semi-decentralized design.
- We build a prototype implementation of SDPaxos, conduct a comprehensive evaluation and demonstrate its high performance.

2 BACKGROUND AND MOTIVATION

State machine replication is a general approach to implementing fault-tolerant services [43]. The service is modeled as a stateful system, and the state is changed by a sequence of commands. A fundamental coordination problem is to ensure every non-faulty replica executes the same sequence of commands in the same order—then all replicas will always reach the same state, given an identical initial state and deterministic service.

The command sequence is produced by a series of instances of consensus, each of which decides a single command in the sequence. An instance of the consensus problem requires a group of possibly faulty processes to agree on a unique value (a command in this context). Upon receiving a client request for a command, a replica will try to have this command chosen in an unused instance, so as to add it into the sequence.

Paxos is a widely-used algorithm to implement each of the consensus instances above. Paxos defines three types of processes—proposer, acceptor, and learner—to propose values, accept values and learn the chosen values (commit the values), respectively. Paxos has two phases. First, in the Prepare phase, a proposer asks a quorum (typically containing a majority) of acceptors whether there is any possibly chosen value. If so, it will have to propose that value; otherwise it could propose a new value requested by client. Then in the Accept phase, the proposer asks the acceptors to accept the proposed value. The value is proposed with a ballot number; an acceptor will accept this value if the ballot is not lower than the highest number the acceptor has ever seen. This rule guarantees only the latest proposal can be accepted, even there are simultaneous ones. The acceptors will notify the learners upon accepting a value. The learners commit the value when it has been chosen, i.e., accepted by a quorum. We refer to a consensus instance implemented by Paxos as a Paxos instance. We say all these instances form a log, and each one is a log slot in the rest of this paper.

To finish the two phases in one round trip, Multi-Paxos [27] designates one replica as the common proposer and learner of all Paxos instances (usually called the leader). The leader can run the Prepare phases for a large number of instances beforehand, after which it can propose a command directly from the Accept phase.
Another approach is **dependency-based ordering**. This approach allows replicas to commit commands bypassing a common leader, and dynamically order them meanwhile, by only partially ordering interfering commands, which won’t lead to inconsistent system state even executed in different orders. For interfering commands, replicas need to coordinate to establish their dependency graph, to decide the execution order. However, the problem is, replicas may see different dependencies simultaneously, which we call conflicts, as none of them can see all commands. As the case in Figure 1(b), there may be a replica seeing a dependency of \( A \rightarrow C \), while another seeing \( B \rightarrow C \) (the two dashed boxes). As a typical protocol using dependency-based ordering, EPaxos [35] needs more round trips and higher CPU consumption, to merge the conflicting commands into a global dependency (i.e., \( (A, B) \rightarrow C \)). Consequently, contention, which brings higher probability of conflicts, unavoidably results in higher latency and lower throughput.

The inefficiency of both the ordering approaches lies in the lack of global information: no replica can precisely know others’ status, i.e., how many commands they have committed or what commands they are committing, when proposing a new command. As a compromise, when ordering the commands, replicas have to assume all replicas have the same speed, or all commands do not interfere, while endure potential impact of straggler or conflict.

The observation above suggests the importance of a global view in ordering commands. However, it is difficult to provide replicas with a global view in multi-leader protocols, while keeping low latency. For example, a similar design in CORFU [8], a distributed log shared by multiple clients, lets a client first ask a centralized sequencer for a position in the log, then write content to this position. This first-ask-then-write pattern introduces at least two round trips, which would be unsatisfactory in the wide area.

In summary, it’s important yet challenging to design a replication protocol that can keep high performance in realistic, changing environments. Next we will present the SDPaxos protocol, and show how it addresses this challenge using the semi-decentralized design.

### 3 SDPAXOS DESIGN

SDPaxos is designed to offer (1) minimized number of round trips for low wide-area latency, (2) (near-)load balance across replicas for high throughput, and (3) strong performance stability against straggler and contention. We will first illustrate its intuition, then elaborate the protocol design, to show how it achieves all these goals.

#### 3.1 Separating Ordering from Replicating

SDPaxos takes two steps to consistently add a command into the log: (1) durably replicating a command across replicas without global order, and (2) assigning this command to an ordered log slot. We illustrate this process in Figure 2. Each replica uses a partially ordered sequence of Paxos instances allocated to it to replicate commands, i.e., \( R_0 \)’s \( A \) and \( C \), \( R_1 \)’s \( B \) and \( R_2 \)’s \( D \) and \( E \). Meanwhile, we assign these commands to the global slots, by designating a replica to each slot, and packing the commands replicated by each replica to the slots belonging to it sequentially. As the case in Figure 2, with \( R_0, R_1, R_2, R_3, R_4 \) designated to the five global slots, we eventually get a totally ordered log of \( A, B, C, D, E \).
We refer to the two steps as replicating and ordering respectively. Separating the two steps as two independent processes enables us to satisfy their different needs at the same time. Replicating is purely decentralized where every replica can freely propose commands. In contrast, to provide ordering with a global view, we employ one specific replica, called the sequencer, to assign commands to global slots in a centralized manner (this single sequencer is not necessary for correctness).

This separation is the source of SDPaxos’s advantages over existing protocols. First, for replicating, the decentralization distributes its load evenly across replicas, and allows clients to always contact the nearest replica in geo-replication. Second, for ordering, the global view of all commands enables the sequencer to order commands optimally—that is, dynamically assign every newly-proposed command to the first unused global slot (unlike Mencius which can leave holes in the log then block fast replicas), and easily serialize all commands without any conflict (unlike EPaxos which can sacrifice performance to solve conflicts). Such an optimal order fundamentally eliminates the impact of straggler or contention. Finally, this separation makes the best trade-off, as the load of replicating is much lighter than that of replicating: the messages for ordering and replicating respectively contain a replica ID and a command, whose size can be several bits versus tens of bytes to KBs or even larger [6]. In §4 we also provide techniques to fully optimize the ordering and further lighten its load, to achieve load-balance even when the commands are small.

For fault tolerance, both the processes of replicating and ordering should be based on Paxos. Correspondingly, we define two types of Paxos instance, the command instance (C-instance), and the order instance (O-instance). Each replica proposes commands in a series of C-instances of its own to produce its partial log. The sequencer proposes replicas’ IDs in O-instances to produce an assignment log. Based on the assignment log, all replicas’ partial logs are finally merged into a global log.

Despite its intuitiveness and effectiveness, realizing this separation is nontrivial. An obvious problem is, how can we complete the two steps in just one round trip? A Paxos instance typically requires a single round trip to be committed; with two separated instances for each command, a trivial protocol design can bring more round trips, especially when all O-instances have a common proposer. Below we show how the SDPaxos protocol efficiently coordinates the two types of instance to solve this problem.

### 3.2 The SDPaxos Protocol

We first use a simple example in Figure 3 to illustrate how to commit a command using C-instance and O-instance. When receiving a client request for a command, replica \( R_0 \) becomes the command leader of this command, picks one of its own C-instance and replicates the command to others (using the C-accept, i.e., Accept phase message of the C-instance). In the meantime, this C-accept also informs the sequencer (\( R_2 \)) to start an O-instance for this command. Then \( R_2 \) proposes \( R_0 \)’s ID in the next (e.g., the jth) O-instance and sends O-accepts to others, to assign this command to the jth global slot. Replicas will then accept these instances and send C-ACKs and O-ACKs to \( R_0 \) (\( R_0 \) acts as the learner of both the instances); \( R_2 \) also sends an O-ACK as it has sent an O-accept to itself. Finally, the two instances are committed after C-ACKs and O-ACKs from a majority are received. \( R_0 \) will broadcast C-commit and O-commit to let others commit these instances.

This is the basic version of SDPaxos which realizes the separation of replicating and ordering using the two types of Paxos instances. It overlaps the two parts by triggering the O-instance with the C-instance message (i.e., the C-accept) sent to the sequencer. This design needs 1.5 round trips to commit a command: since the O-instance starts half a round trip later than the C-instance (if the command leader is not the sequencer), and the O-instance requires one round trip to be committed by Paxos, the overall latency is 1.5 round trips. Below we will present the complete SDPaxos protocol, and show how it further achieves the one-round-trip latency under realistic configurations.

#### The complete protocol

SDPaxos assumes an asynchronous and unordered communication model and tolerates \( F \) non-Byzantine failures given a total of \( N = 2F + 1 \) replicas, which is the strongest guarantee for distributed consensus [18].

Figure 5 shows the pseudocode of SDPaxos without failures. We assume the Prepare phases have been done beforehand. We denote the jth C-instance of \( R_n \) as \( C_{nj} \), and the jth O-instance as \( O_j \). If replica \( R_n \)’s ID is proposed in \( O_j \), then we say \( O_j \) is an O-instance for \( R_n \). Note that we introduce a message type not mentioned in the example above, C-request (line 5). Because replicas can send C-accepts (containing commands) to only a majority to reduce messages, a lightweight C-request can be used to trigger the O-instance, if the C-accept is not sent to the sequencer.
We say a command is \textit{ready} \footnote{We do not use the term “commit a command” in the description of SDPaxos to avoid confusion with “commit an instance.”} when the command leader can safely respond to a client. A command being ready requires the C-instance and enough number (defined below) of O-instances be committed. The conditions of an instance being committed and a command being ready are elaborately defined to reduce latency without breaking correctness guarantee (lines 18 to 31).

In SDPaxos, C-instances are committed strictly through Paxos after being accepted by a majority (line 9), which needs a single round trip. Since the O-instance starts half a round trip later than the C-instance for non-sequencer replicas, we optimize the O-instance to just \textit{half a round trip} to achieve the final one-round-trip latency. The resulting message flow is shown in Figure 4.

**Reducing latency under realistic conditions.** In practice, replica groups tolerating one or two failures (\(N \leq 5\)) are the most commonly used configurations [12, 15], because more replicas inevitably induce much higher consensus overhead. These realistic conditions offer us an opportunity to optimize SDPaxos to achieve the one-round-trip latency. (EPaxos also needs this assumption to offer optimal wide-area latency [35].)

As shown in lines 18 to 22, with three (or more than five) replicas, O-instances are also committed after being accepted by a majority. A command (e.g., the \(i\)th one of a replica) will be ready after at least \(i\) O-instances for this command leader are committed. In three-replica groups, the latency is just a single round trip: as the assigned command leader is the learner of an O-instance, the command leader can commit the O-instance immediately when receiving the O-ACK from the sequencer (the command leader and the sequencer have constituted a majority). This reduces the latency of an O-instance (for a non-sequencer replica) to just half a round trip, leading to one-round overall latency. In groups with more than five replicas, the O-instances still need one round trip, thus the overall latency is \(1.5\) round trips.

In five-replica groups, a command can also be ready in one round trip. Unlike the case of three replicas, an O-instance cannot be accepted by a majority in half a round trip. Instead, we let each non-sequencer replica commit an O-instance, upon receiving the O-ACK from the sequencer (line 29). Here, the O-instance does not rigorously follow Paxos, which raises another problem: if this non-sequencer replica and the sequencer fail, we cannot recover this O-instance simply by Paxos because the other alive replicas may have not seen the O-accept yet. For example in Figure 6, the second O-instance is committed by \(R_1\), but it has not been seen by \(R_2, R_3\) and \(R_4\).

SDPaxos can correctly recover the O-instances of all replicas’ ready commands even in such cases. Figure 6 depicts the intuition of our design. The opportunity is that, we do not need to reduce
the latency of O-instances for the sequencer to half a round trip, because the C-instance and O-instance of the sequencer can start at the same time, without the extra latency to trigger the O-instance. In five-replica groups, O-instances for the sequencer are committed after being accepted by a majority, rather than by only two replicas, which guarantees any committed O-instance for the sequencer must have been seen by at least one non-faulty replica (R2 in the figure). Moreover, the three non-faulty replicas have seen the O-instances for themselves. Therefore, the new sequencer can trace the committed O-instances for these four replicas, i.e., R0, R2, R3, R4 in the figure; but there is still one replica left (R1). The magic we play here is to infer the O-instances for this replica: as we have traced the O-instances of four replicas, the left empty O-instances must belong to the fifth replica. In Figure 6, we retry for R1 in the second O-instance, then all O-instances are correctly recovered. In §3.3 we will show how we leverage this intuition to ensure correctness under failures.

In five-replica groups, a command is ready after all preceding O-instances have been accepted by the command leader (by a majority for the sequencer) (lines 24 to 31). Making all preceding O-instances accepted by the command leader is to avoid putting unready commands before ready ones during recovery (§3.3). This restriction won’t increase latency as the sequencer can piggyback the O-accepts that the command leader has not seen on the latest O-ack sent to it. Similarly, O-ACKs that the sequencer has not seen are piggybacked on the O-ACK sent to it.

Finally, the command in proposed in C-instance Cni will be executed in the jth slot by a replica after the C-instance and O-instance Oj are committed, and all preceding commands are executed—Oj is the ith O-instance for Rn in the assignment log.

We have presented the mainbody of SDPaxos, and demonstrated how we achieve one-round-trip latency under typical settings. Next we will describe the accompanying recover approach for service availability under failures.

### 3.3 Fault Tolerance and Recovery

We design the recovery approach of SDPaxos for full correctness guarantee under failures. Whenever a replica, say, Rn, is suspected to have failed, another replica will initiate the recovery process. The replica will try to recover the C-instances of Rn, and also the O-instances if Rn was the sequencer. By default, we let Rn+1 handle recovery for Rn (note that this is only an implementation choice, and is not necessary for correctness). We will describe the protocols for recovering C-instances and O-instances respectively.

#### Recovery of C-instances.

As C-instances rigorously execute Paxos, after any replica fails, another can recover all of its committed C-instances simply by executing the Prepare phase for them. In each C-instance in which no command is traced, a special no-op command, which keeps the state unchanged, will be proposed. Therefore, even if a C-instance fails but the corresponding O-instance succeeds, all replicas will still execute the same command in this slot (either the previously proposed one, or a no-op), and will not go into inconsistent state or be blocked.

#### Recovery of O-instances.

When the sequencer fails, we need to reelect a new one; then the new sequencer must recover the O-instances to reconstruct the assignment log. We use a view-based approach to ensure correct recovery of O-instances. Each replica stores a view number to identify which view it thinks it is in. Every message is sent with the sender’s view number, and considered valid only if its number is not smaller than the recipient’s.

SDPaxos performs a view change to elect a new sequencer. To start a view change, a replica increments its view number, sends V-request messages to others, and becomes a candidate. V-request also contains Prepare messages of all O-instances (O-prepare[j, bal]) to trace them. On receiving a V-request with a larger view number than its own, a replica increments its view number to that one, and sends a vote to the candidate. The vote also piggybacks the reply to O-prepare, which helps the new sequencer reconstruct the assignment log according to Paxos: for each O-instance, the sequencer picks the value with the highest ballot number among the replies to O-prepare in the votes. Note that replicas only process messages for view change during this period. On receiving votes from a majority, the candidate is elected as the new sequencer, and sends notifications to others.

To ensure linearizability, the new sequencer cannot insert a newly-received command to a slot preceding a ready one [19]. With three or more than five replicas, this can be ensured simply by filling in the “hole”s in the assignment log with no-cl 2 (line 37 in Figure 7)—replicas execute a no-op when encountering a no-cl. We can safely propose no-cl in the holes because these O-instances have not been committed (otherwise at least alive replica will see it).

For five replicas, the “hole”s may be previously allocated to replica Rm and committed (Rm is the replica other than the old sequencer and the alive replicas who vote for the new sequencer). So we need to retry for Rm in these O-instances, until there are as many O-instances for it as its ready commands. We let each voter replica report in its vote the largest-numbered C-instance of each replica it has accepted, then the new sequencer takes the maximum for Rm (C[m] in Figure 7). C[m] implies the maximum number of Rm’s possibly ready commands, because the C-instance of any ready command of Rm must have been seen by one of the voters. Therefore, the sequencer should propose Rm’s ID in the empty O-instances until there are at least C[m] O-instances for Rm (lines 32 to 36). Although the procedure above implicitly assumes Rm also failed (because we need to tolerate two arbitrary failures), it works correctly even if Rm is still alive. A corner case is that, Rm commits an O-instance (e.g., Oj) for it, but the corresponding C-instance (e.g., Cmi) is not committed, then the new sequencer will propose no-cl if it does not see any value in both Oj and Cmi. This case won’t violate consistency or linearizability, because the command in Cmi must have not been executed or ready (Rm will only be blocked at this slot). We allow Rm to accept the no-cl proposed by the new sequencer with a higher ballot number even if Oj has been marked as committed by it, so that it can make progress.

Note that in five-replica groups, the new sequencer should make sure all O-instances have been accepted by a majority in a view change. After all of these complete, the new sequencer sets the

---

2“cl” is the abbreviation of “command leader”.
For five replicas, we focus the proof on the case of view change (ii).
For three or more than five replicas, O-instances are also completed.
Therefore, all the holes in the assignment log must be filled in with no-cl

\[ \text{Input: } C[m]: \text{number of } C\text{-instances of } R_m \text{ accepted by the} \]
\[ \text{majority voters; } O[m]: \text{number of } O\text{-instances for } R_m (R_m \text{is the replica other than the old sequencer and the majority voters in five-replica groups)} \]

\begin{verbatim}
if N == 5 then
  repeat
    O[m] ← 0
  until O[m] ≥ C[m] + 1
  foreach empty O-instance between non-empty ones do
    propose no-cl

Figure 7: Recovery of O-instances.
\end{verbatim}

... O-instance counter to the first empty one in its assignment log, and starts to serve new requests.

3.4 Correctness
SDPaxos guarantees consistency and linearizability of command execution. Here we give a sketched proof, mainly to intuitively explain why SDPaxos is correct. A formal proof can be found in the extended version of this paper [3].

Consistency: At most one command can be executed by different replicas in the same slot of the global log.
Proof sketch. In SDPaxos, C-instances are committed rigorously through Paxos, which ensures that every C-instance can choose at most one command. Since commands in all C-instances are merged into the global log according to the assignment log, it suffices to show that each one command leader in each O-instance in the assignment log.

(i) For three or more than five replicas, O-instances are also committed through Paxos, so there can be at most one command leader in each O-instance.

(ii) For five replicas, we focus on the proof of the case of view change (without sequencer failures, consistency trivially holds as all O-instances will be replicated to all replicas). Specifically, we prove that if some replica \( R \) has executed the command in \( C_{ni} \), then the corresponding and all preceding O-instances can be recovered in a view change (thus the command leader chosen in all these O-instances won’t be changed). Because the last sequencer has made all O-instances accepted by a majority in the last view change, we only need to focus on those proposed in the last view.

1) If \( R \) is one of the majority voters, then all the O-instances to recover can be traced in \( R \)’s vote. 2) If \( R \) is the old sequencer, then all the O-instances to recover can be traced in the votes from the majority voters, as each of these O-instances has been accepted by at least one of them. 3) Otherwise, all these O-instances except those for \( R \) can be traced in the votes from the majority voters. Therefore, all the holes in the assignment log must be previously allocated to \( R \). After filling in the holes with \( R_n \), all O-instances preceding the one of the command in \( C_{ni} \) can be recovered.

Linearizability: If \( \alpha \) and \( \beta \) are commands on the same object, and the request for \( \beta \) arrives at the system after \( \alpha \) is ready, then \( \alpha \) will be executed before \( \beta \).

Proof sketch. Assume \( \alpha \) is proposed in C-instance \( C_{n0} \) of \( R_n \), while \( \beta \) is proposed in C-instance \( C_{mj} \) of \( R_m \). When the sequencer proposes \( R_m \)’s ID in the \( i \)-th O-instance for \( R_m \), there must be more than \( j \) O-instances for \( R_m \) in all preceding ones, because the request for the \( j \)-th command of \( R_m \) (i.e., \( \beta \)) has not arrived at the system yet. Now we prove that after \( \alpha \) is ready, all O-instances preceding the \( i \)-th one for \( R_m \) in the sequencer’s assignment log are always non-empty.

If the sequencer never fails, then this property hold trivially as no O-instances can be lost. Then we prove that this property still holds with a view change.

(i) With three or more than five replicas, the ready condition ensures the new sequencer can trace at least \( i \) O-instances for \( R_n \), with all holes filled in with no-cl.

(ii) With five replicas, if \( R_n \) is one of the three voters or the old sequencer, the new sequencer can also trace at least \( i \) O-instances for \( R_m \) with no preceding hole. Otherwise, the new sequencer knows the number of \( R_n \)’s ready commands is no more than \( C[n] \). After the sequencer proposes \( R_n \)’s ID in the empty O-instances until \( C[n] \) ≤ \( O[n] \), there will be no hole preceding the \( i \)-th O-instance for \( R_n \).

By (i) and (ii), a sequencer can never assign \( \beta \) to a slot preceding the \( i \)-th one allocated to \( R_n \), because these O-instances are all non-empty.

3.5 Optimization for Reads
Besides the protocol presented above, SDPaxos also provides an optimization for operations like reads in storage systems, which only access, but do not change the system state. For such operations, it’s unnecessary to add them into the log via Paxos; all we need is to ensure the freshness of the read data. Fortunately, we can easily know when to read through the sequencer as it can see all the updates on the object to be read.

A read \( R \) on an object \( O \) must be aware of all the updates on \( O \) which are ready before the request of \( R \) is received. We denote the most recent one of those updates as \( W \), and the sequence number of \( W \) in the global log as \( i_E \). Here \( i_E \) implies the earliest safe-to-read time. While it’s relatively difficult to trace \( i_E \) precisely, we can opt for a possibly higher number. We denote the sequence number of the last allocated slot as \( i_G \), the last slot allocated to an update on \( O \) as \( i_A \), when the sequencer receives the request for read from a replica. In fact \( i_G \) is equal to the value of the O-instance counter minus one. Since every committed update must have been seen by the sequencer, we always have \( i_G \leq i_A \leq i_E \). Figure 8 illustrates the temporal relationship of \( i_G \), \( i_A \) and \( i_E \).

A replica sends \( R\text{-request}(O) \) to the sequencer to read \( O \). It’s safe enough for the sequencer to return \( i_G \) or \( i_A \). While \( i_A \) would be better because it eliminates the redundant latency waiting for the execution of irrelevant commands. To enforce this optimization, the sequencer can maintain a mapping called history table in memory from each object to the corresponding \( i_A \). Replicas should also append the name of the object to update to the C-requests. Whenever the sequencer allocates a slot for an update on \( O \), it modifies the value of \( O \) in the history table to the number of that slot. When
the history table is unavailable, e.g., a new sequencer has been just elected, \( i_G \) is also acceptable.

**Sequencer lease.** We use **sequencer lease**, which is similar to “master lease” in some Multi-Paxos systems [13], to authorize the sequencer to directly reply to \( R \)-requests. This is to avoid the case where an old sequencer unaware of a new one replies a stale number. A replica grants a lease to the sequencer making a promise that it will never vote for other replicas in a view change before the lease expires. As long as the sequencer holds unexpired leases granted by a majority of replicas, it can serve reads as previously described. Sequencer lease guarantees safety because it prevents any other replica from being elected as a new sequencer. Thus, there will never be an \( i_E \) larger than the \( i_A \) that the current sequencer knows.

**Cost of reads.** This optimization enables the sequencer to serve reads locally. Non-sequencer replicas handle reads with only two messages, while committing a read through Paxos needs at least \( 2(Q - 1) + (N - 1) = (2Q + N - 3) \) (2 for accept and 1 for commit) messages with a quorum size of \( Q \). Although reads are handled in a centralized way, the sequencer still processes fewer messages for each read than in Paxos-based protocols. If the total number of objects is very large, we can apply this optimization only for some hot items. If the sequencer is carrying high load, we can also deactivate this optimization, as it needs to maintain the history table. Reads in SDPaxos significantly outperform Paxos-based protocols, which is confirmed in our evaluation.

**Message merging.** This optimization is to reduce the number of packets to send and receive between replicas. We merge the messages of \( C \)-instances and \( O \)-instances by lazy-sending \( O \)-instance messages: we first write messages into a buffer, and flush the buffer only when sending \( C \)-instance messages (and \( O \)-accepts to the assigned command leaders) by default. This condition can also be adjusted according to demand.

**Multi-threading.** This optimization is to exploit parallelism in processing the messages of the two types of instances. As the metadata of \( C \)-instances and \( O \)-instances are completely separated, we set two threads to process their messages in parallel, so that they have little impact on each other. We also set another thread dedicated to sending messages, so as to overlap message sending and processing. The optimizations above and the lightweight nature of \( O \)-instance messages together minimize the overhead of the centralized ordering. Note that in the original implementation of other protocols, all messages are sent and processed within a single thread. For fairness, we also add the thread for sending messages into all other protocols.

**Straggler detection.** We deploy a straggler detection to prevent SDPaxos from being impacted by a straggler sequencer. If a sequencer is slow, we can replace it with a view change to get out of the dilemma. Other purely decentralized protocols can hardly benefit from such a detection, because the replicas are totally equivalent and there is nothing to migrate between them.

We implement a low-cost on-demand detection, in which replicas estimate the sequencer’s healthiness according to whether the sequencer can satisfy their requirements for ordering. Specifically, each replica counts the number of \( O \)-accepts for itself received from the sequencer every 500 ms. If the throughput is less than 50% of its requirement for at least 3 times, it considers the sequencer as a straggler. Then it asks others whether they make the same judgment, and if so, it will start a view change to replace the old sequencer. To reduce the probability of misjudgment caused by high load, we also limit the frequency of view change triggered by the detection.

**Thrifty messaging.** In all protocols except Mencius, each replica only sends the Accept phase messages to a quorum including itself, rather than all replicas. The \( O \)-accept for a non-sequencer replica in SDPaxos is only sent to the command leader. These messages will be aggressively resent to other replicas, in case of failure or message loss. This optimization cannot be applied to Mencius, because it needs replies from all replicas to enforce the “skip” mechanism to mitigate the impact of stragglers.

There are still many implementation-level optimization choices for SDPaxos. Below we explore several possible optimizations around the centralized ordering design:

**Sequencer division.** We can divide the responsibility of sequencer to all replicas for complete load balance. For example, in a key-value store, we can partition the key space using approaches like consistent hashing [24], then make each replica order the commands on one partition (commands on different keys can be out-of-order).

**Dedicated sequencer.** In realistic environment, we can use another machine dedicated to playing the role of sequencer, which
We test these protocols with a straggler in a three-replica cluster. We also show their read and write throughput in homogeneous reduction percentage (compared to that without straggler) with a cluster with contention-free workload (§5.3). We then evaluate the wide-area latency of these protocols (§5.4). Finally, we test the results indicate that SDPaxos experiences the least throughput to evaluate their performance in heterogeneous environment. The wide-area latency of these protocols (§5.4). Finally, we test the observe their performance stability against these adverse conditions. Unless otherwise noted, the commands are all writes. The commands of two sizes, small (16 B) and large (1 KB) respectively. We use client requests containing the replica it is co-located with. We use client requests containing commands in one global slot, so the number of O-instances will be divided by $n$. Even a small value of $n$ like 2 or 3 can significantly reduce the load of O-instances.

5 EVALUATION

Experimental setup. We evaluate SDPaxos, Multi-Paxos [27], Mencius [34] and EPaxos [35], on Amazon EC2 m4.large instances. Each EC2 instance has 2 virtual cores and 8GB RAM, running a server or client process on Ubuntu 16.04.

We use close-loop and open-loop clients to measure peak throughput and precise latency, respectively. Close-loop clients wait for the reply to a request to send the next, while open-loop clients don’t. In throughput experiments, each client sends each request to a randomly-chosen replica, or to the leader in Multi-Paxos. In (wide-area) latency experiments, each client only sends requests to the replica it is co-located with. We use client requests containing commands of two sizes, small (16 B) and large (1 KB) respectively. Unless otherwise noted, the commands are all writes.

We first evaluate the throughput of these protocols using heterogeneous cluster (§5.1) and workload with contention (§5.2), to observe their performance stability against these adverse conditions. We also show their read and write throughput in homogeneous cluster with contention-free workload (§5.3). We then evaluate the wide-area latency of these protocols (§5.4). Finally, we test the availability of SDPaxos under failures (§5.5).

5.1 Throughput under Heterogeneity

We test these protocols with a straggler in a three-replica cluster to evaluate their performance in heterogeneous environment. The results indicate that SDPaxos experiences the least throughput reduction percentage (compared to that without straggler) with a non-sequencer straggler, and a relatively high percentage with a sequencer straggler. Moreover, SDPaxos can rapidly replace the sequencer with a normal replica when the current one is slow.

Throughput without straggler detection. We make one replica straggler, by running infinite loop programs contending for CPU on it. The server program acquires roughly one third of CPU time slice compared to that during normal operation. For Multi-Paxos, the sequencer straggler. Moreover, SDPaxos can rapidly replace the sequencer with a normal replica when the current one is slow.

Throughput with a straggler. In (a), the percentage above each bar shows the throughput reduction proportion caused by the straggler. "SDPaxos-S" and "-N" stand for SDPaxos with a sequencer and a non-sequencer straggler respectively. We only show SDPaxos for reads since the read throughput is equal to that of writes for other protocols. In (b), the two dashed lines indicate the average throughput with a sequencer and a non-sequencer straggler respectively.

O-instance batching. Another simple yet effective optimization is batching: a replica’s $n$ commands can be batched in one global slot, so the number of O-instances will be divided by $n$. Even a small value of $n$ like 2 or 3 can significantly reduce the load of O-instances.

With a non-sequencer straggler, SDPaxos always achieves the highest throughput (up to 1.6x and 1.4x compared to Mencius and EPaxos), and also the least reduction proportion among the multi-leader protocols. A non-sequencer straggler can hardly encumber other replicas since they can simply expel the straggler from their quorums (which is also feasible for EPaxos). However, the reduction percentage of writes goes higher when the sequencer is straggler (e.g. 28.27% vs 20.07% with 16B writes). A slow sequencer also causes high throughput reduction of read requests, since all reads are processed by it. Later we will see how the straggler detection rapidly overcomes the problem of a slow sequencer.

Multi-Paxos performs severely worse with a slow leader when the requests are small, i.e., CPU-bound, because all commands are proposed by the leader. With large requests the throughput is nearly unaffected since it has been highly bounded by the bandwidth of the leader. Mencius suffers the most from a straggler among these multi-leader protocols (with the reduction up to 55.61%). Because all replicas have to wait for the straggler’s skips or commits before committing every instance, their throughputs are highly restricted by the straggler. While EPaxos exhibits graceful performance degradation after expelling the straggler from the quorum.

Rapid sequencer shift with straggler detection. As we can see in Figure 9(b), after the sequencer becomes a straggler, it can
be detected within 2 seconds. Then a replica is elected as the new sequencer, and allocates global sequence numbers for the blocked instances (the peak point of the curve). After that, the system exhibits the throughput with a non-sequencer straggler.

5.2 Throughput under Contention

We use increasing conflict rate, i.e., the proportion of updates on the same object, to test the throughput under contention. Figure 10 compares the results of SDPaxos and EPaxos. Multi-Paxos and Mencius are omitted, since their throughputs are unaffected (like SDPaxos).

Conflicts brings higher CPU consumption of EPaxos in tracking dependencies and maintaining dependency graphs. Moreover, the throughput of five replicas is even lower, due to another round trip, i.e., more messages, for interfering commands. When the commands are small, EPaxos’s throughput decreases under higher conflict rate (up to 44%). Note that, even a low rate (e.g., 5%) has visibly reduced the throughput (up to 24%). With large commands, the throughput reduction under contention is lower (with three replicas), since the system is network-bound.

5.3 Throughput in Ideal Case

We test the throughput of these protocols in a local cluster without stragglers using contention-free workload. We also tune the proportion of read requests in the workload to observe the performance variation of these protocols.

As we can see in Figure 11, SDPaxos’s throughput increases significantly as read proportion goes higher, since SDPaxos only costs two messages for each read. In comparison, other protocols show no performance variation as they still need to commit reads using more messages. With small commands, SDPaxos’s reads achieve up to 4.6× throughput of writes: the advantage on number of messages is especially obvious because the main bottleneck of the system is CPU processing messages. When the commands are large, the gap is narrowed (up to 3.2×) because the system consumes more time in sending replies containing large key-values to clients.

We also compare the performance of these protocols when read proportion is 0% (i.e., SDPaxos also commits all commands via the complete Paxos-based protocol). Under high load, Multi-Paxos earliest reaches the throughput limit due to its single-leader bottleneck. However, SDPaxos reaches higher throughput (up to 6.1× that of Multi-Paxos) also with a centralized node, because we have minimized the extra load on the sequencer.

Without stragglers and conflicts, the three multi-leader protocols all exploit the performance of all replicas. Nevertheless, SDPaxos still slightly outperforms the other two protocols when the commands are small due to the separation and paralleled processing of C-instances and O-instances. Mencius sends and processes more messages for every request to enforce the skip mechanism. While EPaxos consumes more CPU resources in tracking and maintaining dependencies, even with this cost reduced to minimum by the conflict-free workload. Large requests also narrow the gap between these protocols since they are all bounded by bandwidth among replicas.

5.4 Latency in the Wide Area

In the wide area, latency is dominated by network communication, which is decided by the number of round trips, and the distance to the replica to contact. The test for Multi-Paxos is omitted because its disadvantage is obvious: client has to communicate with the remote leader, as long as it is not co-located with the leader.

The replicas and clients for wide-area experiments are deployed in California (CA), Oregon (OR), Ohio (OH), Ireland (IRE) and Seoul (SEL). The sequencer of SDPaxos locates in CA. The round-trip times (ping latencies) between these regions are shown in Table 1.

Commit latency. We use the term “commit latency” to refer to the latency to when the replica responds to the client (i.e., when the command is ready in SDPaxos). Figure 12 shows the average commit latencies and 95% CIs of clients in each region. With the optimal number of round trips and quorum size, the only negative factor for SDPaxos’s latency is a sequencer too distant to be contained in

---

**Figure 10:** Throughput with increasing conflict rate.

**Figure 11:** Throughput with no straggler and no conflict. "SD", "Mul", "Men", "E" represent SDPaxos, Multi-Paxos, Mencius and EPaxos respectively. "3R" and "5R" stand for three replicas and five replicas.

**Table 1:** The RTTs between the five regions (ms).
With five replicas, the only exception is the latency of IRE is relatively high (140 ms compared to the theoretically ideal 125 ms), because the sequencer in CA cannot be contained in the quorum of the replica in IRE.

With three or five replicas, EPaxos also has optimal quorum size. However, EPaxos has to handle conflicts with one more round trip with five replicas, which leads to significantly higher latency under conflicts. The average latency increases up to 24% and 60% with conflict rate of 5% and 50%, while the 95% CI increases up to 75% and 165%, respectively.

In Mencius, if some remote replica receives requests in a lower rate, it will increase others’ latencies because they must hear from the slow one to make sure it has committed or skipped all instances. This is essentially the same case as a straggler. Therefore, Mencius’s average latency is higher than SDPaxos and EPaxos in some data centers due to waiting for remote replica’s reply. Moreover, when clients generate requests at different frequencies, replicas in “fast” regions have to wait for replies from “slow” regions more frequently. In “Mencius-imbalanced”, fewer clients in OH and IRE cause the latencies in other sites to be significantly higher, i.e., close to the round-trip times to both the sites.

Read latency. We use the YCSB [14] benchmark to evaluate the read latency under simultaneous accesses to the objects. We use two kinds of R/W ratios, 1:1 and 9:1, and two key distributions, uniform and Zipf (always R/W=1:1 for uniform). For uniform distribution, the key is chosen from 50000 different keys at random. For Zipf, the key is generated by a Zipf generator with an exponent of 0.99, which is the default in YCSB.

For SDPaxos, the theoretical minimum read latency is the actual round trip to the sequencer. As is shown in Figure 13, the average latency is lower than that of other protocols in CA with three replicas, and in OR and OH with five replicas (17% to 99%), because the distance to the sequencer is shorter than that to the nearest majority.

When the the requested object is modified by other replica at the same time, read latencies of all these protocols go higher, as replicas may have to wait for the commit message of the instance after which the read can be done. For SDPaxos, under uniform distribution, the 95% CIs are always nearly equal to the average, since the probability of simultaneous access is low. While the Zipf distribution, under which some popular objects are accessed frequently, increases the worst-case read latency, i.e., higher 95% CI. The worst-case latency is further increased by the 9:1 R/W ratio, i.e. more possibly simultaneous updates. Note that the 95% CI of SDPaxos is nearly always lower than that of other protocols.

For EPaxos, the 95% CI with five replicas is higher than that with three replicas, because conflicts of reads and writes also brings one more round trip. While for Mencius, its latency turns out to be significantly higher than others due to the similar reason explained above: because the read and the corresponding write are both committed in an instance, the latencies of both instances can be nonoptimal, thus the latency to the read is done, i.e., the both instances are committed, can only be even higher.
To evaluate the availability of SDPaxos, we measure the throughput latency (at least two round trips [8, 23]), or stronger assumption in related topics [8, 10, 17, 23]; however, they need either higher latency. In fact, similar communication pattern can be found in some works that overlaps the separated replicating and ordering processes to achieve the one-round-trip latency under realistic configurations while guaranteeing correctness. Comprehensive evaluation shows SDPaxos’s high throughput and low wide-area latency, in ideal, heterogeneous, and high-contention cases. The source code of SDPaxos has been made available at [4].

7 CONCLUSION

We have presented the design and implementation of SDPaxos, a new state machine replication protocol for efficient geo-replication. At its core, SDPaxos proposes the semi-decentralized replication, to provide high performance and strong performance stability against heterogeneity and contention. SDPaxos uses an elaborate protocol design which overlaps the separated replicating and ordering processes to achieve the one-round-trip latency under realistic configurations while guaranteeing correctness. Comprehensive evaluation shows SDPaxos’s high throughput and low wide-area latency, in ideal, heterogeneous, and high-contention cases. The source code of SDPaxos has been made available at [4].

ACKNOWLEDGMENTS

We would like to thank the anonymous SoCC reviewers for their insightful feedback and comments. We thank Prof. Jinyang Li and Prof. Yang Wang for their valuable suggestions on our design. We also thank Prof. Mahesh Balakrishnan for helping us deeply understand the differences between SDPaxos and CORFU and some other related works. This work is supported by National Natural Science Foundation of China (NSFC) under Grant No. 61472009, and Shenzhen Key Fundamental Research Projects under Grant No. JCYJ20151014093505032.


